PERL PROGRAMOZÁSI NYELV FIZIKUSOKNAK

Tóth Gábor

2003. január 20.

Tartalomjegyzék

1. Bevezetés

1.1. Mit jelent

A Perl az angol Practical Extraction and Report Language (Praktikus kivonat és riport készítő nyelv) rövidítése. Larry Wall alkotta 1986-ban. Fiatal kora ellenére a Perl rendkívül gyorsan elterjedt, számos alkalmazásban (pl. rendszer adminisztráció, hálózatkezelés) szinte nélkülözhetetlenné vált, és ezen kívül is egyre több területen használják sikerrel. A Perl ma is fejlődő nyelv, ez az előadás a Perl 5 alapjait ismerteti.

1.2. Mire jó

- Számok, szövegek, fájlok, könyvtárak manipulálása
- Más programok futtatása, eredményeinek felhasználása
- Hálózatok kezelése

A Perl azonban nem hatékony rendkívül számolásigényes feladatok elvégzésére.

1.3. Miért jó

- ingyenes
- szinte minden operációs rendszeren változtatás nélkül fut
- hasonlít a C-re és a shell script-re
- könnyű hibát keresni
- hatalmas és egyre bővülő modul könyvtár
- nagyon kis részét elég ismerni a nyelvnek egy egyszerű script-hez
- bonyolult feladatokra is alkalmas (objektum orientált)

1.4. Hogyan működik

A Perl interpreter `/usr/bin/perl` értelmezi, lefordítja és végrehajtja a Perl programot. Így a végrehajtás viszonylag hatékony marad, ugyanakkor a forráskód futtatása egyetlen lépésből áll, így rendkívül kényelmes és a hibák is könnyen megtalálhatók.

1.5. Perl filozófia

There is more than one way to do it – Ugyanazt többféleképpen is lehet csinálni.

A jó programozó három tulajdonsága:

- Laziness – lustaság
  azaz olyan programokat ír, amik munkát takarítanak meg, és dokumentálja, hogy ne kelljen kérdésekre válaszolnia
- Impatience – türelmetlenség
  a programnak ki kell találnia, hogy mit akarunk
- Hubris – önhittség
  olyan programokat kell írni, amire mások semmi rosszat nem mondanak

1.6. Perl irodalom és források

- Programming Perl (O'Reilly) – a címlapon egy tevével (Camel book)
- man perl
- http://www.perl.com/perl
- http://www.perl.com/CPAN (Comprehensive Perl Archive Network)
- http://www.perl.org
- http://hermes.elte.hu/~gtoth/Teach/perl.pdf

2. UNIX Operációs rendszer alapok

2.1. Bejelentkezés és jelszó változtatás

A képernyőn látható `login:` mögé írjuk be a felhasználói nevet (pl. student29) az `Enter` billentyűvel lezárva. Általában minden utasítás végén le kell nyomni az `Enter`-t. Ezután `password:` mögé írjuk be a jelszót. Ha rossz felhasználói nevet vagy jelszót adtunk be, akkor újra lehet próbálkozni.

Belépés után az egérrel kattintson a bal alsó sarok közelében lévő terminál ikonra. Ezzel megnyitott egy terminált, amiben a Unix operációs rendszer (shell) fut.

Mindenekelőtt új jelszót kell megadni. Ehhez gépelje be a terminál ablakban a `passwd` utasítást (ismét Enter-rel lezárva), majd írja be a régi jelszót, és utána kétszer egymás után az újat:

```
passwd
Enter your password: regi-jelszo
Enter new password: uj-jelszo
Retype password: uj-jelszo
```

A jelszó 6 és 15 karakter között legyen, és tartalmazzon olyan karaktert, ami nem betű, nem szám, és nem aláhúzás (_).

2.2. Kilépés

A bal alsó sarokban található talp alakú (a Gnome ablakkezelő jelképe) ikonra kattintunk, kiválasztjuk a `logout` pontot a menüből, majd a képernyő közepén megjelenő ablakban a zöld színű `Yes` gombra kattintunk. A gépeket nem szabad kikapcsolni, mert mások is dolgoznak illetve futtatnak rajta!

2.3. Fájlok és könyvtárak kezelése

`ls`	a könyvtárban levő fájlok listázása
`ls *.pl`	a .pl végű fájlok listázása
`ls -l hello.pl`	részletes infomáció a hello.pl fájlról
`touch file1`	file1 megérintése/létrehozása
`cp file1 file2`	file1 másolása file2-be (copy)
`mv file2 file3`	file2 átnevezése file3-ra (move)
`rm file3`	file1 törlése (remove)
`mkdir dir1 dir2`	alkönyvtár(ak) létrehozása (make directory)
`mv file1 dir2 dir1`	file1 és dir2 berakása a dir1 alkönyvtárba
`cd dir1`	belépés a dir1 alkönyvtárba (change directory)
`mv dir2 dir3`	dir2 alkönyvtár átnevezése dir3-ra
`rmdir dir3`	dir3 alkönyvtár eltávolítása (ha üres!)
`cd ..`	kilépés a dir1 alkönyvtárból
`rm -f dir1/*`	dir1 alkönyvtárban lévő fájlok azonnali törlése (forced)
`rm -rf dir1`	dir1 alkönyvtár rekurzív azonnali törlése

2.4. Programok kezelése

A UNIX operációs rendszer egyszerre több programot hajt végre. Ezeket programoknak hívjuk. A programok indítása, megállítása, figyelése:

`emacs akarmi &`	program indítása háttérben
`emacs valami`	program indítása előtérben
`Ctrl-Z`	futó program felfüggesztése
`jobs`	az ablakban futó ill. felfüggesztett programok listája
`bg`	az utoljára felfüggesztett program háttérben futtatása
`kill %2`	háttérben futó job megállítása szám szerint
`kill %emacs`	háttérben futó job megállítása név szerint
`fg`	a háttérben futtatott program előtérbe helyezése
`Ctrl-C`	előtérben futó program megállítása
`ps`	az összes futó program listája, elöl az azonosító (ID)
`kill 14434`	az 14434 azonosítójú program leállítása (ha figyel rá)
`kill -9 14434`	az 14434 azonosítójú program leállítása (ha nem figyel)
`top`	a futó programok listaja CPU idő szerint rendezve (kilépés 'q')

2.5. Egér használata

Általában az egér bal gombját használjuk. Ha több ablak van nyitva, akkor a bal gombbal a megfelelő ablakra kattintva választhatjuk ki az aktív ablakot. Az ablakokat úgy lehet mozgatni, hogy tetejére állítjuk az egeret, majd a bal gombot lenyomva elhúzzuk. Az ablak tetején lévő gombokkal az ablak becsukható, ikonná alakítható, nagyítható, stb.

Egy ablakban található szöveg kijelöléséhez helyezzük az egeret a szöveg elejéhez, majd a bal gombot lenyomva tartva vigyük az egeret a kijelölendő szöveg végére, és engedjük el a bal gombot. Ezután beszúrható a szöveg máshova a középső gombra kattintva. Ez a másolási módszer szövegszerkesztő és a terminál ablakokban illetve ezek között is működik.

2.6. Szöveg szerkesztés

Szöveges állomány (fájl) szerkesztése:

```
emacs hello.pl &
```

A szerkesztőben a nyilakkal és a Page Down, Page Up, Home, End billentyűkkel lehet mozogni, a Delete és Backspace billentyűkkel pedig törölni.

Az emacs a file nevéről illetve az első sorba írt `#!/usr/bin/perl` utasításról felismeri annak típusát, és megfelelően alkalmazkodik hozzá. Néhány hasznos billentyű kombináció:

`Ctrl-A`	ugrás sor elejére
`Ctrl-E`	ugrás sor végére
`Ctrl-D`	a kurzor alatti karakter törlése
`Ctrl-K`	a kurzortól a sor végéig törlés, lehet többször is
`Ctrl-Y`	utoljára törölt sorok visszaírása a kurzor pozíciójánál
`Ctrl-X U`	az utolsó utasítás visszavétele, lehet többször is
`TAB`	a sor megfelelő pozícióba tolása
`ESC Ctrl-Q`	a kurzorral kiválasztott program rész sorainak rendezése

Az `emacs` szövegszerkesztő igen hasznos tulajdonsága, hogy ha egy már létező fájlon változtatunk, akkor kilépés után megmarad az eredeti fájl is egy ~ -vel kiegészítve, pl. `hello.pl~`. További biztonsági elem, hogy szerkesztés közben a pillanatnyi állapotról is készül egy fájl ami # karakterek között van, pl. `#hello.pl#`. Erre akkor lehet szükség, ha az emacs elszáll (pl. `kill` utasítás miatt), vagy a gép leáll (pl. áramkimaradás miatt), vagy a hálózat elromlik, és nem tudunk kilépni a szerkesztőből.

Ha kíváncsiak vagyunk, hogy mit változtattunk az utolsó szerkeztés során, használhatjuk az `xdiff` programot, pl.

```
xdiff hello.pl hello.pl~
```

2.7. Perl program futtatása

Mint a Perl filozófiából tudjuk, ugyanazt többféleképpen is lehet csinálni.

1. A perl interpreter `-e` (execute) paraméterével:

   ```
   perl -e 'print "Hellow world\n"'
   ```

2. A perl interpreternek adott fájl paraméterrel

   ```
   perl hello.pl
   ```

3. Egy `#!/usr/bin/perl` első sort tartalmazó fájl végrehajtásával. A fájlt a

   ```
   chmod +x hello.pl
   ```

utasítással végrehajthatóvá kell tenni. Ezután

```
hello.pl
```

lefuttatja a programot.

3. A Perl nyelv alapjai

3.1. Forráskód formátuma

A kis és nagy betűk különböznek!

Az utasításokat pontosvessző választja el egymástól. A programszöveg tetszőlegesen tördelhető, még idézőjelen belül is (persze akkor a sor törés is része lesz karakterláncnak).

Megjegyzéseket a kettőskereszt (#) karakter mögé lehet írni.

3.2. Változók és értékadás

A változókat nem kell deklarálni, de lehet, illetve bizonyos bonyolultság elérése után érdemes kötelezővé tenni (`use strict`).

A változók nevét egy speciális karakter előzi meg, melyek a változó típusát adják meg.

```
$a = 1.2e-12;              # skalár változó (numerikus)
$b = "valami";             # skalár változó (string)
@c = (1,2,"three");        # lista/tömb
%d = ("small" => "kicsi",  # asszociatív tömb
      "big" => "nagy");    # más néven hash
&read_values();            # eljárás meghívása
```

A speciális kezdő karakter azért jó, mert a változó nevek nem eshetnek egybe az utasítások nevével. Továbbá ugyanaz a név különböző változó típusokra használható, pl. lehet egy tömbünk `@temp` ami hömérsékleteket tartalmaz. Ennek egy elemét, amit éppen használunk a skalár `$temp` változóba tehetjük. Emellett lehet egy `%temp` nevű asszociatív tömbünk, ami megadja, hogy egy adott városban mennyi a hőmérséklet, és egy `&temp` nevű függveny, ami adott koordinátákhoz megadja a hőmérsékletet.

3.3. Skalár változók: `$x`

Az, hogy egy skalár változó szám, karakterlánc, vagy logikai változó, a szövegkörnyezet, más szóval a kontextus dönti el. Ezt a kontextust a változó körüli forráskód adja. Például

```
$n = +10;                          # egész változó
$suly = 7.2e2;                     # valós változó
print "$n ember $suly kg\n";       # karakterlánc (10 ember 720 kg)
```

Mint látható a dupla idézőjelek (double quotes) közötti változókat a Perl behelyettesíti (interpolálja) a string értékükre. Ez az érték a szám szokásos írásmódja, ami nem feltétlenül azonos azzal, ahogy a numerikus értékadásnál a számot megadtuk. Az interpreter a dupla idézőjelek között a \n karaktert is kicseréli/interpolálja a sorvégére. Ez azért hasznos, mert a `print` utasítás nem ír sorvégét. Még számos ilyen speciális karakter van, de ezekkel majd később a 8.6 részben foglalkozunk.

A szimpla idézőjel (single quotes) ezzel szemben nem interpolál:

```
$a = '$100';      # nincs interpoláció
print $a,'\n';    # ($100\n) sorvége nélkül
```

A visszafelé dőlő idézőjelek (backticks) közti utasítást a Perl végrehajtja mintha azt az operációs rendszerben adtuk volna ki és a képernyőre írt szöveget adja vissza (a sorvégekkel együtt!):

```
$ls = `ls`;          # ls utasítás eredménye
print `pwd`,$ls;     # könyvtár fáj1 fájl2 ...
```

A következő példák a kifejezésekkel és a függvényekkel történő értékadást mutatják:

```
$e = exp(1);         # az exponenciális függvény (2.71828182845905)
$x = 3*($e + 1);     # egy egyszerű kifejezés (11.1548454853771)
```

A példából az is kiderül, hogy a Perl nyelvben a számok mindig dupla pontossággal (8 byte) vannak eltárolva. Léteznek olyan Perl modulok, amik tetszőleges hosszúságú egészekkel és valósakkal is tudnak számolni.

A skalár változók logikai változókként is szerepelhetnek, például a feltételes utasításokban. Ilyenkor a következő szabályok érvényesek:

0 értékű szám:	HAMIS
"0" értékű string:	HAMIS
üres string:	HAMIS
definiálatlan változó:	HAMIS
minden egyéb:	IGAZ

3.4. Rendezett tömb/lista: `@x, $x[0], $#x`

A rendezett tömböknek értéket lehet adni a

```
@prim = (2, 3, 5, 7, 11);
```

utasítással. Az így keletkezett tömb elemei 0-tól indexelődnek (hasonlóan a C nyelvhez), és az indexet szögletes zárójelek közé kell írni:

```
print "Az elso es a harmadik prim: $prim[0] $prim[2]\n";
```

Fontos megjegyezni, hogy a tömb egy eleme már skalárként viselkedik, így a $ jelet kell eléje írni. Új tömb úgy is létrehozható, hogy egyese elemeinek értéket adunk:

```
$szo[3] = 'kicsi'; $szo[2] = 'nagy'; print @szo; # nagykicsi
```

Ilyenkor a tömb akkora lesz, amekkora a legnagyobb használt index, és azok az elemek, amik nem kaptak értéket definiálatlanok maradnak, azaz üres stringként illetve 0-ként jelennek meg.

Több dimenziós tömböket a Perl közvetlenül nem támogatja, de lehetőség van tömbök tömbjeit létrehozni, ami hasonlóan használható mint egy több dimenziós tömb. Erről a 15.8 részben lesz szó.

A zárójelben felsorolt változóknak értéket is lehet adni

```
($first, $second, $third, $fourth) = @prim;
```

Ha a bal oldalon több változó van mint a jobb oldalon, akkor a maradék változók definiálatlanok maradnak, míg ha kevesebb, akkor a jobb oldal felesleges változói egyszerűen nem kerülnek felhasználásra.

Az értékadás párhuzamosan történik, így két változó értéke kicserélhető ideiglenes változó bevezetése nélkül:

```
($a, $b) = ($b, $a);
```

A rendezett tömb elemeinek számát kétféleképpen is megkaphatjuk. A `$#some_array` a tömb utolsó elemének indexét adja vissza. Ha a tömb nem létezik, akkor az érték -1. A `@some_array` skalár kontextusban a tömb elemeinek számát adja meg, ha a tömb nem létezik, akkor 0 (azaz ha a tömb létezik, akkor a logikai érték igaz, ha nem, akkor viszont hamis). Szükség esetén a skalár kontextust a `scalar` függvénnyel lehet expliciten megkövetelni. Példák:

```
@a = (1,2,3,4,5);
print "$#a \n";      # =4
print scalar @a,"\n" # =5
$n = @a;             # $n=5
```

3.5. Asszociatív tömb/hash: `%x`, `$x{"y"}`

Az asszociatív tömbök (rövidebb néven hash) indexei karakter láncok (rövidebb nevükön stringek). Az asszociatív tömb elemei nincsenek rendezve, viszont az index alapján a hozzátartozó érték nagyon gyorsan kikereshető. Ez a gyors keresés az úgynevezett hash táblák használatának köszönhető, amely minden stringhez hozzárendel egy egyedi memória címet.

Az értékadás ugyanúgy történik mint a rendezett tömböknél, csak itt az elemeket páronként indexnek és értéknek tekintjük:

```
%english = ('kicsi' => 'small', 'nagy' => 'big');
```

Itt a => helyett lehetne egyszerű vesszőt is írni, azonban az indexek és értékek párosítása így sokkal olvashatóbb. Az asszociatív tömb elemeire a string indexeken keresztül hivatkozunk, amit kapcsos zárójelek közé írunk:

```
print "kicsi angolul = $english{'kicsi'}\n"; # kicsi angolul = small
```

Egy hash elemenként is létrehozható:

```
$magyar{'big'} = 'nagy'; $magyar{'small'}= 'kicsi';
```

4. Műveletek

4.1. Numerikus műveletek: + - * / % **

Ha $a=10; $b=3 akkor

```
$a + $b    # összeadás (=13)
$a - $b    # kivonás (=7)
$a * $b    # szorzás (=30)
$a / $b    # osztás (=3.33333333333333)
$a % $b    # maradékos osztás (=1)
$a ** $b   # hatványozás (=1000)
```

A műveletek prioritása a szokásos, ami zárójelezéssel megváltoztatható. A numerikus műveletek automatikusan a változók numerikus értékét használják.

4.2. String műveletek: . x

Az előző példánál maradva ($a=10; $b=3):

```
$a .  $b   # összefűzés (=103)
$a x $b    # ismétlés (=101010)
```

Az utolsó művelet például arra jó, hogy kiírjunk egy hosszú vonalat:

```
print '-' x 79, "\n"; # ---------------------------------------...
```

4.3. Értékadás művelet: = += *= .= ...

A Perlben minden operátornak, így az értékadásnak is van eredménye, mégpedig ugyanaz, mint amit a baloldal kap:

```
$a=$b=3;              # $b=3; $a=$b;
print "a=$a, b=$b\n"; # a=3, b=3
```

Az értékadás összekombinálható bármelyik numerikus vagy string operátorral:

```
lvalue op= expression  ---> lvalue = lvalue op expression

$a += 3;      # a = a + 3
$b .= "x";    # b = b . "x"
$c *= 2 + 3;  # c = c * (2+3)
```

Ez az írámód rövidebb, hatékonyabb, és némi gyakorlattal még könnyebben is olvasható mint az eredeti.

4.4. Növelés és csökkentés: `++ -`

Mivel nagyon gyakori, hogy valamit `$n = $n+1` vagy rövidebben `$n += 1` formában számlálunk, ezt lehet még rövidebben is írni:

```
$n++;       # $n = $n + 1
++$n;       # $n = $n + 1
```

A két jelölés között csak akkor van különbség, ha a művelet értékét továbbadjuk, ugyanis ha a ++ elöl van, akkor a megnövelt értéket, ha hátul, akkor pedig az eredetit adjuk tovább:

```
$a = 1;
$b = $a++;                  # $b = $a; $a = $a +1; (b=1, a=2)
$b = ++$a;                  # $a = $a +1; $b = $a; (b=3, a=3)
```

A ++ műveletnek van egy *mágikus* tulajdonsága: stringeket is lehet vele inkrementálni, feltéve, hogy a változót előtte string kontextusban használtuk. Például:

```
$betu = "a";
print "Aki $betu-t mond, mondjon ",++$betu,"-t is!\n";
```

A ++ művelet ellenkezője a -, ami egyet kivon:

```
$n=4; print $n--," and ",--$n,"\n"; # (4 and 2)
```

A - művelet stringekre nincs mágikus hatással.

4.5. Logikai műveletek: `and or not xor && || !`

Vannak magas és alacsony prioritású logikai műveletek. Az utóbbiak (`and`, `or`, és `not`) az újabbak (Perl 5) és általában ezeket érdemes használni, mert általában a várakozásnak megfelelő eredményt adnak zárójelek használata nélkül is. A régebbi jelek (`&&`, `||`, és `!`) a C nyelvből származnak, nehezen olvashatóak, és magas prioritásuk néha meglepő eredményt ad.

Ha `$a=5; $b=0` akkor

```
$a && $b    # és            (=0 azaz HAMIS)
$a || $b    # vagy          (=1 azaz IGAZ)
!  $a       # nem           (=0 azaz HAMIS)
$a and $b   # és            (=0 azaz HAMIS)
$a or $b    # vagy          (=1 azaz IGAZ)
not $a      # nem           (=0 azaz HAMIS)
$a xor $b   # kizáró vagy   (=1 azaz IGAZ)
```

A logikai műveleteket *rövid zárlat* műveletnek is nevezik, ugyanis ha a bal oldali argumentum már megadja az eredményt, akkor a jobb oldali argumentumot már nem értékeli ki. Ez rendkívül gyakran használt a következő módon

```
$a or print '$a is not set',"\n";
$a and print '$a is set',"\n";
```

Ha a `$a` változó értéke igaz (azaz nem nulla vagy üres string) akkor a program azt írja ki, hogy '$a is set', egyébként pedig azt, hogy '$a is not set'. Az `or` és az `and` ilyen használata követi a természetes angol nyelv szabályait.

4.6. Összehasonlító operátorok: `< > <= >= == != <=> lt gt le ge eq ne cmp`

Két számot vagy két stringet aritmetikai illetve string operátorokkal hasonlíthatunk össze. Az összehasonlítás eredménye egy igaz vagy hamis logikai érték. Ha pl. `$a="10"; $b="10.00"`, akkor

`$a < $b`	(HAMIS)	$a lt $b	(IGAZ)	kisebb
`$a > $b`	(HAMIS)	$a gt $b	(HAMIS)	nagyobb
`$a <= $b`	(IGAZ)	$a le $b	(IGAZ)	kisebb egyenlő
`$a >= $b`	(IGAZ)	$a ge $b	(HAMIS)	nagyobb egyenlő
`$a == $b`	(IGAZ)	$a eq $b	(HAMIS)	egyenlő
`$a != $b`	(HAMIS)	$a ne $b	(IGAZ)	nem egyenlő
`$a <=> $b`	(0)	$a cmp $b	(-1)	összehasonlítás (-1,0,1)

Az utolsó összehasonlító művelet (`<=>, cmp`) nem logikai, hanem szám értéket ad vissza: -1 ha $a kisebb $b-nél, 0 ha egyenlőek, és 1, ha $a nagyobb $b-nél. Erre a műveletre a `sort` függvénynél lesz majd szükségünk (lásd a 9.3 részt).

5. Kommunikáció

5.1. A perl interpreter paraméterei

A perl interpreter `-e` paraméteréről már volt szó. Ezzel perl utasítások adhatók meg a parancssorban.

A `-n` (no print) paraméter hatására a parancssorban megadott fájl(ok)ból vagy a billentyűzetről (STDIN) beolvasott sorok a `$_` változóba kerülnek (a sorvége karakterrel együtt), és a program utasításai minden sorra végrehajtódnak. Ez az `awk` Unix programhoz hasonló működést eredményez. Például:

```
perl -n -e 'print "On azt irta hogy $_"'
```

Ez az egysoros program beolvassa a billentyűzetre begépelt sorokat, és visszaírja a terminálra az `On azt irta hogy` szöveggel kiegészítve.

A `-p` (print) paraméter ugyanazt csinálja mint a `-n`, sőt még ki is írja a `$_` értékét az utasítások végrehajtása után minden egyes sorra. Ez a `sed` Unix programhoz hasonló működést eredményez. Például:

```
perl -p -e '$_ = "> $_"' level.txt
```

Ez a progran a `level.txt` fajlban lévő sorok elé a '> ' karaktereket teszi és kiírja a képernyőre.

A `-i` (in place) paraméter a `-p` vagy `-n` paraméterekkel együtt lehetővé teszi, hogy a parancssorban szereplő fájl(oka)t megváltoztassuk. Ha a `-i` mögött nincs semmi, akkor a fájlok egyszerűn felülíródnak a Perl program kimenetével. Ha mögé írunk egy stringet (pl. `-i.orig`), akkor az eredeti fájl a stringgel kiegészített névvel megmarad (pl. `level.txt.orig`). Például a

```
perl -p -i~ -e '$_ = "# $_"' prog1.pl prog2.pl
```

eredményeképpen `prog1.pl` és `prog2.pl` fájlok minden sora elé egy `#` karakter kerül, míg az eredeti fájlok a `prog1.pl~ prog2.pl~` néven maradnak meg.

A `-a` (autosplit) paraméter a `-p` vagy `-n` paraméterekkel együtt használatos. A bejövő `$_` sorban lévő egy vagy több szóközzel elválasztott mezőket a `@F` tömbbe (Fields) rakja. Például:

```
perl -na -e 'print "Elso oszlop=$F[0], masodik oszlop=$F[1]\n"'
```

A `-F` (Field separator) paraméterrel megadható, hogy a mezőket mi választja el. Például a `/etc/passwd` fájl, ami a felhasználók főbb adatait tartalmazza (a jelszót éppenséggel nem), az egyes mezőket kettősponttal választja el. A következő egy soros program kiírja, hogy az egyes felhasználók milyen shell-t használnak:

```
perl -F: -na -e 'print "user=$F[0], shell=$F[6]"' /etc/passwd
```

Mind a `-p`, mind a `-n` paraméterek mellett lehetőség van arra, hogy a bejövő sorokat feldolgozó ciklus előtt a BEGIN eljárásban, illetve a feldolgozás után az `END` eljárásban lévő utasításokat végrehajtsuk (ez megegyezik az `awk` program működésével). Például:

```
#!/usr/bin/perl -n
sub BEGIN {
    $n=0;
}
$n++;
sub END {
  print "Beolvasott sorok szama=$n\n";
}
```

A BEGIN és END nevű eljárásoknál a `sub` kiírása nem kötelező. Mivel a nem definiált `$n` változó értéke eleve 0, így a BEGIN eljáras felesleges, azaz a fenti program így is írható:

```
#!/usr/bin/perl -n
$n++; END {print "Beolvasott sorok szama=$n\n";}
```

A `-s` (switch) paraméter lehetővé teszi, hogy a perl programnak átadjunk változókat a parancs sorról. Például legyen a `tst.pl` program

```
#!/usr/bin/perl -s
print "xyz=$xyz\n";
```

Ezt a Perl programot különböző paraméterekkel meghívva a következőt kapjuk:

```
> tst.pl
xyz
> tst.pl -xyz
xyz=1
> tst.pl -xyz=15
```

```
xyz=15
> tst.pl -xyz="X Y and Z"
xyz=X Y and Z
```

Végül a `-w` (warn) paramétert említjük meg, ami bekapcsolja az alapos szintax ellenőrzést. Ezt szinte mindig érdemes használni.

A perl interpreter számos további paraméteréről a `man perlrun` utasítással kaphatunk leírást.

5.2. Feladat: számsor feldolgozása

Írjon egy Perl programot, ami egy fájlból vagy a billentyűzetről számokat olvas be, és mindig visszaírja az addig beolvasott számok átlagát, maximumát és minimumát.

5.3. UNIX kommunikáció

Mivel a Perl sok mindent felhasznál a UNIX operációs rendszerből, érdemes néhány UNIX alapismerettel kezdeni. A UNIX-ban egy program olvashat egy fájlból, vagy a sztenderd input-ról (STDIN), ami alapértelmezésben a billentyűzetet jelenti. Ha megakarjuk spórolni a gépelést, akkor a < jellel elérhetjük, hogy a program egy file-ból olvasson:

```
program < file
```

A program eredménye szintén mehet egy fájlba, vagy a standard output-ra (STDOUT), ami alapértelmezésben a terminál. Ezt szintén átirányíthatjuk egy fájlba:

```
program > file
```

Például egy könyvtárban lévő fájlok listáját a

```
ls > lista
```

utasítással a `lista` fájlba írhatjuk. A fájlba írt szöveget szövegszerkesztővel is nézhetjük, de a `cat` utasítással is kiírhatjuk:

```
cat lista
```

Ha fájl hosszú, akkor a `more` utasítással érdemes megjeleníteni, ami az egyes képernyőoldalaknál megáll, oda-vissza lapozást, sőt keresést is lehetővé tesz.

A `cat` utasítás több fájl kiírását is elvégzi, így több fájl egybe írható a

```
cat file1 file2 file3 > files
```

utasítással. Ha a `files` már létezik, és újabb file-kat akarunk hozzáfűzni, akkor ezt a `>>` (append) utasítással tehetjük meg:

```
cat file4 file5 >> files
```

Az STDIN (STDOUT) nem csak fájlokból (-ba), hanem programokból (-ba) is átirányítható a `|` (pipe) segítségével. Például a

```
ls | more
```

esetében a `ls` utasítás output-ja lesz a `more` utasítás inputja. Sok program is egymás után fűzhető a `|` (pipe = csővezeték) utasítással, például

```
locate perl | grep pm | sort | more
```

ahol a `locate` parancs felsorolja az összes fájlt aminek a könyvtárral együtt vett) nevében a `perl` szó előfordul. Ebből a `grep pm` kiválasztja azokat, amelyek a `pm` stringet tartalmazzák, ezeket a `sort` utasítás ABC sorba rendezi, és ennek az eredményét jeleníti meg a `more` program.

5.4. Perl kommunikáció: `<>, chop, chomp, print, open, close`

A Perl a külvilággal a fájlkezelőn (file handle) keresztül kommunikál. A `print` utasítás alapértelmezésben az `STDOUT` fájlkezelőre ír, azaz a standard outputra. A standard inputhoz alapértelmezésben az `STDIN` fájlkezelő van hozzárendelve. Egy fájlkezelőhöz rendelt fájlból (vagy az STDIN-ből) a sorolvasó operátorral, azaz a kisebb és nagyobb jelek közé írt fájlkezelővel tudunk olvasni:

```
$line = <STDIN>;
```

Ez az utasítás egy sort olvas be a sorvége karakterrel együtt. A következő `<STDIN>` utasítás (skalár kontextusban) a következő sort olvassa be.

Ha nem a standard inputról, hanem egy fájlból akarunk olvasni, akkor először a fájlt meg kell nyitni, azaz hozzárendelni egy fájlt mutatóhoz:

```
open(LEVEL, 'level.txt');
$line = <LEVEL>;
```

Mint látható a fájl mutató egy olyan változó, ami előtt nincs semmilyen speciális karakter. Általában csupa nagy betűt szoktunk használni, bár ez nem kötelező.

Annak sincs akadálya, hogy az egész fájlt beolvassuk. Ehhez a sor olvasó operátort tömb kontextusban kell használni:

```
@lines = <FILE>;
```

Akár egy sort, akár többet olvastunk be, a sorvége karakterek ott lesznek a sorok végen. Ezeket gyakran le kell vágni. A `chop` (levág) operátor egy karaktert vág le, míg a `chomp` csak a sorvége karakter(eke)t vágja le (általában ez \n, de nem UNIX operációs rendszereken más is lehet). Mindkét operátor használható skalárra és tömbre is. Például:

```
$line = <STDIN>; chop($line); # vagy chop($line = <STDIN>);
chomp($sor = <LEVEL>);        # vagy $sor = <LEVEL>; chomp($sor);
chomp(@lines = <LEVEL>);      # vagy @lines=<LEVEL>; chomp(@lines);
```

Fontos megjegyezni, hogy a `chop` függvény az (utoljára) levágott karaktert, míg a `chomp` a levágott sorvégék számát adja vissza és nem a lerövidített stringe(ke)t!.

Természetesen nem csak olvasni, de írni is lehet egy fájlba. Ehhez a fájlt írásra kell megnyitni, és a print utasítással írhatunk bele:

```
open(SZOVEG, '>szoveg.txt');
print SZOVEG '1 computer costs $1000',"\n";
```

A fájl nevét megelőző > jel a UNIX-ból ismert átirányításra utal, azaz a fájlba írunk. Az `open` utasítás néhány további lehetősége:

```
open SZOVEG, 'szoveg.txt';     # olvasás
open SZOVEG, '<szoveg.txt';    # szintén olvasás
open SZOVEG, '>szoveg.txt';    # írás
open SZOVEG, '+<szoveg.txt';   # írás és olvasás
open SZOVEG, '>>szoveg.txt';   # hozzáfűzés
open SZOVEG, '| progi';        # átirányítás programba
open SZOVEG, 'progi |';        # átirányítás programból
```

Egy megnyitott fájl automatikusan bezáródik, ha a Perl program végetér, vagy ha ugyanazt a fájlkezelőt megnyitjuk, de általában helyes a megnyitott fájlokat explicit módon bezárni, amikor már nincs szükség a fájlra (ez megkönnyíti a programszöveg értelmezését is):

```
close SZOVEG;
```

5.5. Fájl tesztelő operátorok

Mielőtt egy fájlt megnyitnánk, vagy megpróbálnánk megnyitni, érdemes meggyőződni róla, hogy létezik-e, milyen típusú, mekkora, stb. Ezeket a műveleteket a fájl tesztelő operátorokkal lehet elvégezni. Ezek az operátorok a Unix shell által használt operátorokhoz hasonlatosan egy '-' jelből, és a közvetlenül mögé írt karakterből állnak. Itt csak a leggyakrabban használt operátorokat soroljuk fel:

`-e`	file exists	fájl létezik
`-r`	readable file	olvasható fájl
`-w`	writable file	írható fájl
`-x`	executable file	végrehajtható fájlt
`-f`	plain file	egyszerű fájl
`-d`	directory	könyvtár
`-l`	symbolic link	szimbolikus link
`-T`	text file	szöveges fájl
`-B`	binary file	bináris fájl
`-A`	access time	fájl kora utolsó hozzáférés óta (napban)
`-C`	change time	fájl kora utolsó változtatás óta (napban)
`-M`	modification time	fájl kora utolsó módosítás óta (napban)
`-s`	file size	fájl mérete (byte-ban)

A felsorolt operátorok a `-s` kivételével igaz értékként 1-t, hamisnak az üres stringet adják vissza.

6. Kontroll utasítások

6.1. Feltételes utasítás: `if, unless`

A feltételes utasítás legegyszerűbb formája

```
if(conditional-expression){
    statement1;
```

```
    statement2;
}
```

A kapcsos zárójelek között álló utasítások akkor hajtódnak végre, ha a gömbölyű zárójelek közti logikai kifejezés értéke IGAZ. Ellentétben a C nyelvvel, a kapcsos zárójeleket nem lehet elhagyni akkor sem, ha csak egy utasítás van köztük.

Az `if` utasítás bonyolultabb formái:

```
if($a){
    print '$a'."=$a is true\n";
}else{
    print '$a'."=$a is false\n";
}
```

Az `else` előtt és mögött is kapcsos zárójel áll, de nem pontosvessző!

Végül az `if` utasítás legbonyolultabb formája:

```
if($a > 0)then{
    print "A is positive\n";
}elsif($a < 0)then{
    print "A is negative\n";
}elsif($a == 0)then{
    print "A is zero\n";
}else{
    print "This cannot happen\n";
    exit;
}
```

Az `unless` utasítás egyszerű `if` utasításhoz hasonló, csak éppen akkor hajtódik végre, ha a zárójelben álló logikai kifejezés hamis:

```
unless($this_is_true){
    print "This is not true\n";
}
```

Az `unless` esetében is használható az `else`, de `elsunless` nem létezik.

6.2. Ciklus utasítások: `while, for, foreach`

A `while` ciklus utasítás addig ismétli a kapcsos zárójelek közti műveleteket, amíg, a zárójelben lévő kifejezés igaz. Például

```
while($line = <FILE>){
    $n++;
    print "$n. line=$line";
}
```

Az értékadás eredménye mindaddig igaz marad, amíg a fájlban vannak sorok, ugyanis a `$line` változó még az üres sorok esetében is tartalmazza a \n sorvége karaktert. Ha a feltétel már kezdetben hamis, például a fájl üres, akkor a ciklus egyszer sem hajtódik végre.

Az `until` ciklus utasítás pontosan ugyanúgy müködik mint a `while`, azzal a különbséggel, hogy a ciklus addig folytátódik, amíg a logikai változó értéke hamis. Ha már kezdetben igaz, akkor az `until` ciklus egyszer sem hajtódik végre.

A `for` ciklus utasítást akkor érdemes használni, ha egy ciklus változóra (általában numerikusra) van szükségünk. Ez a Perl ciklus utasítás megegyezik a C-ben használt `for` utasítással. A zárójelben három utasítás áll pontosvesszökkel elválasztva. Az első a ciklusváltozó inicializálja, a második a ciklus befejezésének feltétele, a harmadik pedig a ciklusváltozó változtatására szolgál. Például egy tömb elemeinek kiírása:

```
@a=(1,4,6,8,-1,7,2,4,0,0);
for($i=0; $i < 10; $i++){
    print "a[$i]=$a[$i]\n";
}
```

A `foreach` ciklus utasításban a ciklus változó egy tömb értékein (és nem az indexén) megy végig. A zárójelek között álló tömb lehet egy változó, vagy egy kifejezés eredménye is. Például

```
@lines = <TEXT>;
foreach $line (@lines){
    $n++;
    if($line eq "\n"){print "Empty line at line $n\n"}
}
```

A tömb elemekeit fel lehet sorolni:

```
foreach $day ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday"){
    print "I have to work on $day\n";
    $work{$day}=1;
}
```

Egy asszociatív tömb kiírására egy szokásos megoldás:

```
foreach $key (sort keys %work){
    print "$key => $work{$key}\n";
}
```

ahol a `keys` függvény a hash kulcsait adja vissza véletlenszerű sorrendben, és ezt a `sort` függvény abc sorba rendezi.

A `foreach` utasításnak számos rövidebb formája létezik. A ciklusváltozó elhagyása esetén a `$_` változó kapja meg a tömb értékeit. Továbbá a `foreach` helyett `for` is írható, mert a szintaxis alapján mindig megkülönböztethető a `for(;;)` utasítástól. Fontos megemlíteni, hogy ha a `foreach` ciklus egy tömb (és nem egy tömb értékű kifejezés) elemein megy végig, akkor a ciklusváltozó megváltoztatása a tömb elemet is megváltoztatja!

A különféle ciklusok természetesen egymásba ágyazhatóak.

6.3. Ciklus kontroll: `last, next, redo`

A ciklus kontroll utasításokat a megismert ciklus utasítások belsejében, a kapcsos zárójelek között használjuk. A `next` utasítás visszaugrik a ciklus elejére, és ha a logikai feltétel értéke megfelelő, akkor folytatja a ciklust. A `redo` utasítás is visszaugrik a ciklus elejére, de a logikai feltételt (és `for` ciklus esetén a ciklus változó növelését) nem hajtja végre. A `last` kontroll utasítás kilép a ciklusból.

A következő program kihagyja az üres sorokat, és ha egy sorban 'END' stringet talál, akkor kilép:

```
while($line=<TEXT>){
    if($line eq "\n"){next}
    if($line eq "END\n"){last}
    print $line;
}
```

Egymásba ágyazott ciklusok esetén a kontrol utasítások alapértelmezésben a legbelső ciklusra vonatkoznak. Ezen úgy módosíthatunk, hogy a ciklusokat címkével látjuk el, és a kontroll utasításoknál ezekre a címkékre hivatkozunk:

```
FILE:foreach $file (@files){
    open(IN,$file);
    LINE:while($line=<IN>){
        chop;
        if($line eq ""){next LINE}
        if($line eq "END"){next FILE}
```

```
        ...
    }
    close IN;
    ...
}
```

Megfelelő címke nevek választásával a program nagyon jól olvasható lesz.

6.4. Blokk: `BLK:{...}`

A címkével és kapcsos zárójelekkel létrehozható egy blokk, ami alapértelmezésben csak egyszer hajtódik végre (a címkét nem kötelező kiírni). Az így létrehozott blokkon belül használhatóak a ciklus kontroll utasítások. Például a C `switch` vagy a Fortran `case` utasításának egy lehetséges megvalósítása címkézett blokk segítségével:

```
SWITCH:{
   if($n==1){print "n is one\n"; last}
   if($n==3){print "n is three\n"; last}
   if($n<0){print "n is negative\n"; last}
   print "n is neither 1 nor 3 or negative\n";
}
```

6.5. Utasítás módosítók: `if, unless, while, until`

Egy egyszerű utasítás végére odaírhatunk egy `if` vagy `unless` feltételt, illetve egy `while` vagy `until` ciklus utasítást. Ilyenkor a zárójelre nincs szükség:

```
next LINE if $line eq "END\n";
exit unless $value eq "CONTINUE";
print $i++,"\n" until $i==10;
print ++$i,"\n" while $i<10;
```

Ha az `until` ciklus utasításban szereplő feltétel eleve igaz, akkor az utasítás egyszer sem hajtódik végre. Ez alól kivétel a `do` utasítás, ami egy (címke nélküli!) blokkot hajt végre, ugyanis ilyenkor az `until` módosító legalább egyszer mindenképpen végrehajtja a `do` utasítást. Például a

```
do {
  $line = <IN>;
} until {$line == "\n"}
```

legalább egy sort olvas az IN fájl mutatóból akkor is, ha `$line` értéke korábban "\n" volt. Ez megfelel a Pascal `repeat until` ciklusának. Fontos megjegyezni, hogy a `do` utasításban szereplő blokknak nincsen címkéje, így a ciklus kontroll utasítások (`next, last, redo`) itt nem használhatóak.

6.6. Feladat: számsorok összehasonlítása

Írjon egy programot, ami két fájlból számokat olvas ki. Ha ezek különbsége meghalad egy bizonyos értéket, akkor kiírja a sor számát, az értékeket, és a különbséget. Az üres sorokat átugorja. Ha egy sorban "END"-t talál, abbahagyja a fájl olvasását. Szól, ha az egyik fájlban több szám van, mint a másikban.

A két fájl nevét a `-s` kapcsoló vagy a `@ARGV` tömb segítségével adhatjuk át.

6.7. Feltételes operátor: `?  :`

A C nyelv feltételes operátora a Perl-ben is megtalálható.

Ez a `condition ? expr_if_true : expr_if_false` formában írt 3 argumentumú operátor a kérdőjel előtt álló `condition` feltétel igaz vagy hamis voltától függően a kettőspont előtt álló `expr_if_true` vagy a kettőspont mögött álló `expr_if_false` kifejezést adja vissza. Példa

```
$max = $a > $b ? $a : $b;
```

6.8. Lista operátor: `..`

A `..` operátor lista kontextusban a két oldalon alló numerikus vagy karakter kifejezések közötti értékek listáját adja vissza. Ez különösen hasznos egyszerű ciklusok írására, vagy tömbök létrehozására:

```
for $i (1..$n){
   print "$i\n";
}
@abc  = ('a'..'z');
@mday = '01'..'31';
```

7. Program egységek

7.1. A főprogram argumentumai: `@ARGV`

A főprogram meghívásánál a parancssorba írt argumentumok a `@ARGV` (argument variables) tömbbe kerülnek bele. (Ellentétben a C-vel, magának a programnak a neve nem szerepel a tömbben, hanem az első elem az első argumentum. A program nevét $$ speciális változó tartalmazza). Ha a `-s` kapcsolót használjuk, akkor ez a programot közvetlenül követő `-kapcsolo` illetve `-kapcsolo=ertek` alakú argumentumokat eltávolítja, és megfelelő változókba teszi.

A `-p` és `-n` kapcsolók lényegében szintén a `@ARGV` tömbön mennek végig, és közelítőleg ezt csinálják

```
foreach $ARGV (@ARGV){
    open(ARGV, $ARGV) or print "Can't open $ARGV\n";
    while($_=<ARGV>){
       ...
    }
}
```

A `-p` és `-n` kapcsolók a fenti kódhoz hasonlóan definiálják az éppen feldolgozott fájl nevét (`$ARGV`) és az `ARGV` fájlkezelőt használják.

A `while(<>)` szintén a `@ARGV` tömbben felsorolt fájlokból illetve az STDIN-ből olvas. A `@ARGV` expliciten is megadható. Íme egy trükkös megoldás arra, hogy egy vagy több fájlt helyben javítsunk ki, de a program túl összetett ahhoz, hogy a `-ni` vagy `-pi` kapcsolók használhatóak legyenek:

```
#!/usr/bin/perl -i
@ARGV = ('tst1','tst2');
while(<>){
  $_ = "> $_";
  print;        # This prints back $_ into the current file
}
```

A program a 'tst1' és 'tst2' fájlok sorai elé '> ' karaktereket ír.

7.2. A program befejezése: `exit, die`

A főprogram általában az utolsó utasítással fejeződik be. A végrehajtás megszakítható az `exit` és a `die` utasításokkal. Az `exit`-nek adható egy numerikus

paraméter, amit a meghívó program (általában a UNIX shell) kap vissza. Ha nem adunk meg paramétert, akkor 0-t ad vissza, ami a sikeres futást jelenti.

A `die` paramétere egy hibaüzenet, ami a sztenderd error (STDERR) fájlkezelőn át íródik ki, valamint kiírja annak a sornak a számát, amiben a `die` utasítás szerepel. Ha éppen fájlt olvasunk, akkor a fájl kezelőből utoljára olvasott sor sorszámát is kiírja. Ha a hibaüzenet végére "\n"-t írunk, akkor csak a hibaüzenet jelenik meg. Példák:

```
open(FILE,'myfile') or die "Cannot open myfile\n";
while(<FILE>){
    chomp;
    die "empty line!" if $_ eq '';
    exit 1 if $_ eq 'ERROR';
    print "$_\n";
}
exit;
```

7.3. Alprogramok (eljárások és függvények): `sub, return`

Az eljárásokat/függvényeket a `sub some_name` után kapcsos zárójelek közé írt utasításokkal adjuk meg. Ez a programban bárhol elhelyezhető, bár általában az alprogramokat a főprogram után szokás írni. Az eljárás meghívásakor `&` jelet használunk az eljárás neve előtt (ez bizonyos esetekben elhagyható), és az argumentumokat az eljárás neve után zárójelek közé írjuk. Az eljáráson belül az átadott argumentumok a speciális `@_` tömbbe kerülnek bele. Ha a meghíváskor az argumentumokat zárójelestül elhagyjuk, akkor a `@_` pillanatnyi értéke adódik tovább, amivel vigyázni kell. Az argumentum listát bezáró zárójeleket csak akkor érdemes elhagyni, ha az eljárás nem használ argumentumokat.

Például:

```
&print_max(1,3);      # =maximum is 3
print_max(3,10,-1);   # =maximum is 10
&print_max();         # =maximum is
&print_max;           # =&print_max(@_); !!!
print_max;            # BAD! use -w to be warned.

sub print_max{
    $max = $_[0];
    foreach (@_) { $max = $_ if $_ > $max};
    print "maximum is $max\n";
}
```

Mint látható, az argumentumok száma tetszőleges. Ha a `@_` tömb elemeit megváltoztatjuk, akkor az argumentumok értéke is változik (persze ilyenkor csak változók adhatók át!):

```
@a=(1,3,-4);
&increase(@a);
print "@a\n";  # =2,4,-3
sub increase{foreach $a (@_){$a++}}
```

Az eljárásból az utolsó utasítás végrehajtásával, vagy a `return` utasítással lehet visszatérni. Az eljárások egyben függvények is. Értékük vagy az utolsó utasítás értéke, vagy a `return` függvény argumentuma:

```
print "max(1,3)=",&max(1,3),"\nmax()=",&max(),"\n";
sub max{
    return "empty array" unless @_;
    $max = $_[0]; foreach (@_) { $max = $_ if $_ > $max};
    $max;  # or return $max
}
```

Természetesen hasonló módon egy tömböt is vissza lehet adni.

Ha argumentumként több tömböt adunk át, akkor azok egyszerűen egymás mögé íródnak, azaz a tömbök méretét is át kell adni:

```
&compare(scalar @a, scalar @b, @a, @b);

sub compare{
  $n1 = $_[0];
  $n2 = $_[1];
  @aa = @_[2..$n1+1];
  @bb = @_[$n1+2..$n1+$n2+1];
  ...
}
```

Látható, hogy ez se nem kényelmes, se nem hatékony megoldás, hiszen a tömbök többször is másolódnak. A kényelmes és hatékony megoldást a tömbmutatókkal lehet megvalósítani (lásd a 15 részt). Hasonlóan több tömb visszaadása is mutatókkal oldható meg célszerűen.

A rekurzió megengedett, azaz az eljárás meghívhatja önmagát. Például íme a klasszikus ám legkevésbé sem hatékony faktoriális program Perl változata:

```
print "fact(10)=",fact(10),"\n"; # => fact(10)=3628800
```

```
sub fact{
    return 1 if $_[0] == 1;
    $_[0]*&fact($_[0]-1);
}
```

7.4. Változók deklarálása: `my`

A nem deklarált változók globálisak, azaz a főprogramból és a függvényekből oda-vissza elérhetőek! Ez kis méretű programoknál kényelmes, de nagyobb programoknál már nem célszerű.

A legegyszerűbb változó deklaráció a `my` utasítással történik:

```
#!/usr/bin/perl

$a = 4;
print "double=",&double($a)," original=$a\n"; #=double=8 original=4
sub double{
  my $a=$_[0]; $a*=2; return $a;
}
```

Ha a `my` nem szerepelne, akkor az eredmény double=8 original=8 lenne!

A deklarálással értéket is lehet adni:

```
my $name = "Gabor";                 # one scalar
my @parents = ("father", "mother"); # one array
my ($a, $b, $c) = (1,2,3);          # more variables require my ( )
my $a, $b, $c;                      # BAD !!! only $a is declared !! use -w
```

A `my` változók érvényessége a legkisebb bezáró blokk végéig terjed.

8. Keresés, illesztés és csere karakterláncban

8.1. Keresés karakterláncban: `index`, `rindex`

Az `index STR, SUBSTR, POS` függvény megkeresi az STR stringben a SUBSTR rész stringet a POS pozíciótól kezdve. Ha POS nem adott, akkor az STR elejétől kezdődik a keresés. Az eredmény az első megtalált SUBSTR kezdőpozíciója

STR-ben (az első karakter indexe 0). Ha STR-ben nem fordul elő SUBSTR, akkor az eredmény -1.

Az `rindex STR, SUBSTR, POS` ugyanúgy működik, mint az `index`, csak visszafelé keres, és POS a maximális pozícióértéket jeleni. Példák:

```
print  index("aba","kabala baba"),   # => 1
       index("aba","kabala baba",2), # => 8
      rindex("aba","kabala baba"),   # => 8
      rindex("aba","kabala baba",7), # => 1
      index("aba","kabala baba", 9); # => -1
```

8.2. Illesztés karakterlánchoz: `m/ /`

Az `STR =~ m/MINTA/` (match) függvény egy string mintát illeszt az STR stringhez. Ha az `STR =~` részt elhagyjuk, akkor az illesztés a `$_` változóhoz történik. Az `m` után szinte bármilyen karakter használható a törtvonal helyett. Ez akkor hasznos, ha a mintában is előfordul a törtvonal. Ha a mintát nyitó karakter zárójel jellegű, akkor a záró karakter ennek párja lesz. Ha a mintát a / határolja, akkor az `m` elhagygató.

Az illesztés eredménye skalár kontextusban igaz (1), vagy hamis (üres string) attól függően, hogy az illesztés sikerült-e. Példák:

```
print "kabala baba" =~ /aba/;   # 1
while(<>){                      # read lines into $_
  last if /END/;                # finish reading if line contains END
  print unless /#/;             # skip lines containing '#'
  if( m{#!/usr/bin/perl}){      # match $_ with a pattern containing /-s
     print "Looks like Perl\n";
  }
}
```

A mintába változókat is beleírhatunk:

```
$lastname="Toth"; $firstname="Gabor";
print "My name\n" if $line =~ /$firstname $lastname/;
```

Az `m/ /` függvény viselkedését módosíthatjuk a mintát bezáró karakter mögé írt módosítókkal:

g	global	összes illesztést keresse meg
i	case insensitive	nagy és kis betűket nem különbözteti meg
m	multipole line	több sor
s	single line	egy sor, akkor is ha van benne \n
o	only compile once	a változókat csak egyszer interpolálja
x	extended expression	szóköz és sorvége nem számít

Például

```
$lastname="toth"; $firstname="gabor";
print "My name\n" if $line =~ /$firstname $lastname/io;
```

8.3. Meta karakterek

A fenti példák megoldhatóak lettek volna az `index` függvénnyel is. A string illesztés azonban sokkal rugalmasabb, ugyanis a mintát normál karakterekből és meta-karakterekből építhetjük fel. A meta-karakterek következőek:

```
\ } ( ) [ { ^ $ * + ? .
```

A meta-karakterek használata. Sor eleje és vége:

```
/^Fredi/ and print "line starts with Fredi\n";   # beginning of line
/Beni$/  and print "line ends with Beni\n";      # end of line
```

Alternatívák:

```
/Fredi|Beni|Irma|Vilma/ and print "Flinstones";  # alternatives
/(Ir|Vil)ma/      and print "Irma or Vilma";     # alternatives
```

Ha meta-karakterekre kellene keresni, akkor egy visszadőlő tört vonalat kell eléírni. Például

```
  $line =~ /Hol a baba\?/;  # back slash is needed to match literal '?'
```

8.4. Ismétlések

```
/subidubidu+/     and print "subidubiduuuu...";  # 1 or more repetitions
/Irma, *Vilma/    and print "Irma,Vilma";        # 0 or more repetitions
```

```
/subidubido{5}/   and print "subidubiduuuuu";    # n repetitions
/subidubido{5,}/  and print "subidubiduuuuu..."; # n repetitions at least
/subidubido{2,4}/ and print "subidubiduu..";     # n to m repetitions
/controll?/       and print "control, controll"; # 0 or 1 times
/tra(la)+/        and print "tralalalala...";    # 1 or more repetitions
```

A nem rögzített számú ismétlődésekből az illesztés a maximális számút igyekszik kiválasztani. Ha az ismétlő jelek mögé (még egy) kérdőjelet írunk, akkor viszont a minimális számút használja.

8.5. Karakter osztályok

Egyes karakterek helyett használhatunk karakter osztályokat:

```
/.*/ and print "this matches anything";       # dot matches except for newline
/[Ff]redi/ and print "Fredi or fredi";        # character class: 2 characters
/[a-zA-Z]/ and print "a letter";              # character class: 2 ranges
/[^a-zA-Z0-9_]/ and print "not alphanumeric"; # inverted character class
/[\b\t\n ]+/ and print "back space, tab, new line or space";
```

8.6. Speciális karakterek és karakterosztályok

Speciális gyakran előforduló karakterek és karakterosztályok:

```
\a     alarm character       chr(7)
\b     backspace             chr(8)  in character classes only!
\t     tab                   chr(9)
\n     newline               chr(10)
\f     form feed             chr(12)
\r     carrige return        chr(13)
\e     escape                chr(27)
\d     digit                 [0-9]
\D     non-digit             [^\d]
\w     word character        [0-9a-zA-Z_]
\W     not a word character  [^\w]
\s     white space character [ \t\n\r\f]
\S     non-white space       [^\s]
```

További speciális karakterek:

```
\c.    control-.             \cD, \cd  => Ctrl-D => chr(4)
```

```
\x..   hexa(..)              \xf => chr(15), \x7f  => chr(127)
\...   octal(...)           \07 => chr(7),  \010  => chr(8)
```

A \b két különböző dolgot is jelent. Karakter osztályokban a backspace karaktert, egyébként viszont a szó határát, azaz egy szó karakter és egy nem szó karakter vagy sor eleje vagy sor vége közti pozíciót. Például, ha meg akarjuk nézni, hogy egy szó előfordul-e, de mint szórészt nem fogadjuk el, akkor ezt így tehetjük meg:

```
print "Line contains 'szo'" if $line =~ /\bszo\b/;
```

8.7. Visszautalás a mintán belül: \1, \2 ...

Ha a mintában egy zárójelben álló rész illeszkedik, akkor arra vissza lehet utalni a \1, \2 ... mintákkal. A zárójeleket a nyitó zárójel alapján számozzuk balról jobbra. Például:

```
print "Repeated word!\n" if /\b(\w+)\b.*\b\1\b/
```

8.8. Az illesztett részek kiolvasása: $&, $`, $', $1, $2, ...

Természetesen kíváncsiak lehetünk arra, hogy az illeszkedésnél mi mivel illeszkedett. Példaul

```
"Where is Fredi and Beni?" =~ /([Ff]redi) +(and)? +(Beni|Vilma)/;
print $&;    # matching part     = 'Fredi and Beni'
print $`;    # part before match = 'Where is '
print $';    # part after match  = '?'
print $1;    # first paren       = 'Fredi'
print $2;    # second paren      = 'and'
print $3;    # third paren       = 'Beni'
print $+;    # last paren        = 'Beni'
```

Ha lista kontextusban használunk egy illesztő operátort, akkor a $1, $2, stb változók egy listába íródnak be. Például a /etc/passwd fájl mezőit, melyek kettősponttal vannak egymástól elválasztva, kiolvashatjuk a következő módon:

```
($user,$passwd) = /^(.*?):(.*?):/
```

Itt a kérdőjelek azért kellenek, hogy a `.*` ne illessze a kettőspontot.

Ha az összes mezőt ki akarjuk olvasni, a `/g` (globális illesztés) módosítóval is megtehetjük:

```
($user,$passwd,$uid,$gid,$name,$home,$shell) = /([^:]+)/g
```

Ez azért működik, mert lista kontextusban a `/g` módosító az összes illesztést visszaadja. Skalár kontextusban a `m//g` újra és újra illeszt az előző illeszkedéstől kezdve és igazat ad amíg illeszkedik a minta, hamisat ha már nem. Például:

```
while(m/([^:]+)/g){print ++$n,". field=$1\n"}
```

kiírja az egyes mezők értékét. Ez addig működik, amíg a stringet meg nem változtatjuk. Azt, hogy éppen hol tartunk az illesztéssel, a `pos` függvénnyel olvashatjuk ki. A `pos` argumentuma az a string, amire az illesztés történt (alapértelmezésben a `$_`), eredménye pedig az illeszkedő minta kezdő pozíciója (hasonlóan mint az `index` függvénynél).

8.9. Egyszeri keresés és negált keresés: `m? ?, !~`

Ha egy string mintának csak az első előfordulására vagyunk kíváncsiak, akkor a törtvonal helyett a kérdőjelekkel határoljuk a string mintát. Ha ezt egy ciklusban meghívjuk, akkor az első sikeres illeszkedésig működik, utána már nem. Például:

```
#!/usr/bin/perl -n
print "Perl code!\n" if ?^!#/usr/bin/perl?;
...
```

csak egyszer írja ki, hogy "Perl code", és további sorokban már nem is illeszti a mintát.

Mivel a string illesztés gyakran áll feltételes utasításban, a `=~` operátor mellett az `!~` operátor is használható, ami igazat (1) ad vissza, ha NEM illeszkedik a minta, és hamisat (üres string) ha illeszkedik. A következő két feltétel equivalens:

```
if($line !~ /^#/){...}
if(not $line =~ /^#/){...}
```

8.10. Összetett példa

Íme egy összetettebb példa, ami „szeret, nem szeret” típusú kijelentéseket keresi meg:

```
($who,$if,$whom)=/(You|I)\s+(do|don't|do +not)?\s*love\s+(me|you|him|her|it)/i;
print "Yes! $who love $whom\n" if $if =~ /do/i or $if =~ /^$/;
print "No! $who do not love $whom\n" if $if =~ /n/;
print "Selfish\n" if $who =~ /I/i and $whom =~ /me/i or
                     $who =~ /$whom/i;
```

8.11. Feladat: valós szám felismerése

Írjon egy Perl programot, amely felismeri, ha egy sorban szerepel egy érvényes valós szám, és ennek értékét egy változóba írja.

8.12. Csere karakterláncban: s///

Az `STR =~ s/MINTA/CSERE/egimosx` (substitute) függvény az `m//` függvényhez hasonlóan a `=~` és `!~` operátorokkal kapcsolható az STR stringhez, vagy pedig annak hiányában az `$_` stringen hajt végre keresést és cserét. A függvény értéke a végrehajtott cserék számát adja meg. A MINTA és a CSERE stringeket határoló karakterek tetszőlegesek, nemcsak a hagyományos törtvonal lehet. Ha zárójel típusú határoló karaktert használunk, akkor abból a MINTA és a CSERE köré is kell egy-egy pár. Az `s` azonban nem hagyható el törtvonal használata esetén sem.

A MINTA ugyanolyan string minta, mint az `m//` függvényben, a CSERE pedig egy tetszőleges string, ami tartalmazhat változókat (többek között a $1, $2, $& változókat is). A CSERE értéke kerül a megtalált MINTA helyére. A módosítók hasonlóan működnek, mint az `m//` függvénynél. Ha a /g (global) módosító szerepel, akkor az összes mintát kicseréli. Az egyetlen új módosító, a /e (evaluate) lehetővé teszi, hogy a CSERE string egy kifejezés legyen, amit kiértékelődik mielőtt a csere végrehajtódna. Példák:

```
s/\r//g;                          # Remove carriage returns from Windows file
$count = s/\blove\b/love/g;       # Count the number of "love"-s in $_
$line =~ s/\blove\b/hate/g;       # Replace "love" with "hate" in $line
s/(\w+)(\s+)(\w+)/$3$2$1/;        # Swap first and second words in $_
s|^#!/usr/bin/perl|#!/bin/perl|; # Change the path to perl
s/([aei]1|[ou][123])/$tex{$1}/ig;# Replace a1 ... U3 with \'a ... \H U
s/\d+/$&*2/eg;                    # Multiply integer numbers by 2
```

8.13. Feladat

Írjon egy olyan utasítást, ami a TAB karaktereket szóközökre cseréli úgy, hogy a stringben a szöveg nem tolödik el.

8.14. Karakterek cseréje és számlálása: tr///

A `STR =~ tr/MIKET/MIRE/cds` függvény a MIKET karaktereket a MIRE karakterekre cseréli ki. A karakterek sorrendje lényeges, azaz a karakterosztályokkal ellentétben itt nem halmazokról, hanem rendezett karakter listákról van szó. A karakterosztályokhoz hasonlóan a felsorolásban használható a kötőjel. A `tr` függvény a kicserélt illetve letörölt karakterek számát adja vissza. Példák:

```
$count = tr/qQ/qQ/;   # count the number of 'q'-s
tr/yzYZ/zyZY/;        # swap 'y' and 'z' in $_
$line =~ tr/a-z/A-Z/; # capitalize $line
tr/\000-\020/ /;      # replace control characters with space
```

Mint az utolsó példa is mutatja, ha a MIRE karakterlista rövidebb, mint a MIKET lista, akkor a MIRE lista automatikusan meghosszabbodik az utolsó karakter megismétlésével. Ezt a viselkedést a /d (delete) módosítóval lehet megváltoztatni, úgy, hogy azok a MIT-ben előforduló karakterek, amiknek nincs párja a MIRE listában törlődnek. A /c (complement) módosító a MIKET karakterek komplementerét veszi, a /s (squash) pedig az STR stringben talált egymás mellett álló MIKET karaktereket egy karakterré vonja össze. Példák:

```
tr/\000-\020//d;      # delete some control characters
$cnt = tr/a-zA-Z//cd; # count and remove all non-letter characters
tr/a-zA-Z_0-9/ /cs;   # squash non-word characters into one space
```

8.15. String manipuláló függvények és speciális karakterek: lc, uc, lcfirst, ucfirst, quotemeta \L, \U, \l, \u, \Q, \E

Kis és nagybetűk közötti váltásra szolgáló függvények:

```
$a = "az ELTE";
print uc($a);          # upper case -> AZ ELTE
print lc($a);          # lower case -> az elte
```

```
print ucfirst($a);     # upper case first character -> Az ELTE
print lcfirst("ELTE"); # lower case first character -> eLTE
```

A `quotemeta` függvény a string mintákban szereplő meta karakterek elé visszafele dőlő törtvonalakat tesz, így a metakarakterek normál karakterekként működnek. Ez akkor fontos, ha a string mintában egy változó szerepel, amiben esetleg előfordulhatnak metakarakterek, de ezeket normál jelentésben kívánjuk illeszteni. Pédául:

```
$find = <>; $pattern = quotemeta($find);
print "Found $find\n" if /$pattern/;
```

A quotemeta függvény biztosítja, hogy ez a program akkor is működik, ha a beolvasott string például $-t, pontot, vagy zárójeleket tartalmaz.

Az eddig felsorolt függvények a stringbe írt speciális karakterekkel is helyettesíthetőek:

```
$a = "az ELTE";
$b = '$3 - $10 or more ...';
print "\U$a\E \L$a\E \u$a \lELTE" # -> AZ ELTE az elte Az ELTE eLTE
print "\Q$b\E\n";                 # -> \$3\ \-\ \$10\ or\ more\ \.\.\.
```

A `\U, \L, \Q` nagy betűre, kis betűre, illetve idézett meta karakterekre vált a string végéig vagy a `\E` speciális karakterig. A `\u, \l` csak a következő karaktert váltja nagy illetve kis betűre.

8.16. Általánosított idézőjelek: q//, qq//, qx//, qw//

Ha az idézőjelek között idézőjelek szerepelnek, azt elég nehézkes leírni. Az általánosított idézőjelek ezt a problémát oldják meg. A `q` a szimpla, a `qq` a dupla, a `qx` (execute) pedig a visszafele dőlő idézőjelet helyettesíti:

```
$tex{a1} = q/\'a/;      # same as '\'."'a"    or   "\\'a"
$quote   = qq/"$text"/; # same as "\"$text\""
$files   = qx/ls *.pl/; # same as `ls *.pl`
```

A `qw` (word) szóközökkel elválasztott szavakat tömbbe rakására szolgál:

```
@words = qw(Adel Bela $10 \help) # same as ('Adel','Bela','$10','\help')
```

8.17. String hossza és darabolása: `length, substr`

A `length STR` függvény az STR string hosszát adja vissza. Ha az STR hiányzik, akkor a $_ hosszát. Ha a string nem létezik vagy üres, akkor 0-t. Például a túl hosszú sorok kiírása:

```
perl -ne 'print if length > 80' level.txt
```

A `substr(STR, POZICIO, HOSSZ, CSERE)` függvény a stringből kivág egy HOSSZ hosszúságú darabot a POZICIO karakterpozíciónál kezdve, és ezt adja vissza függvényértékként. Ha a CSERE argumentum is jelen van, akkor STR érteke is megváltozik, ugyanis a CSERE string a megadott stringdarab kerül. Természetesen ekkor az STR csak változó lehet. Ha a HOSSZ argumentum hiányzik, akkor az STR végéig ér a részstring. Ha a HOSSZ negatív, akkor az STR végétől HOSSZ karaktert kell visszaszámolni. A nulla POZICIO a string legelejét jelenti, negatív pozíciót a string végétől kell venni. Pédák:

```
print substr("vilma",3);      # -> ma
print substr("vilma",-2);     # -> ma
print substr("vilma",1,2);    # -> il
print substr("vilma",0,-1);   # -> vilm
$a = "vilma";
print substr($a,0,1,"pusz");  # -> v
print $a;                     # -> puszilma
substr($line,-1,1,"");        # same as chop($line);
```

9. Rendezett tömbök kezelése

9.1. Konverzió stringek és tömbök között: `split, join`

A `split(/MINTA/, STR, LIMIT)` függvény felvágja az STR stringet a MINTA által meghatározott helyeken legfeljebb LIMIT darabra, és az eredmény egy a string darabokból álló tömb lesz. A string elején álló esetleges üres mezők megmaradnak (kivéve, ha a minta `' '`), a string végén lévő esetleges üres mezők viszont nem kerülnek be a tömbbe.

Ha a LIMIT paraméter hiányzik, akkor annyi darabra vágja fel, ahányra lehet. Ha az STR hiányzik, akkor a $_ változót darabolja fel. Ha a MINTA sincs megadva, akkor alapértelmezésben a $_ változót a szóköz típusú (white space) karakterek által határolt mezőkre vágja fel, ami a `' '` minta szerinti viselkedésnek felel meg.

Példák:

```
@a=split(/:/,":x:yz:t:");    # same as @a=('', 'x', 'yz', 't')
@a=split(/:/,":x:yz:t:",2);  # same as @a=('', 'x:yz:t')
@a=split(/\s+/,"  x  yz ");  # same as @a=('', 'x', 'yz')
@a=split(' '  ,"  x  yz ");  # same as @a=('x', 'yz')   (no empty field!)
open(PASSWD,'/etc/passwd');
($login,$passwd,$rest)=split(/:/,<PASSWD>,3);
```

Ha a MINTA tartalmaz zárójelet, akkor a zárójelhez illeszkedő string darabok is bekerülnek a tömbbe. Ez alkalmas arra, hogy az elválasztókat kulcsként, a mezőket meg értékeknek használva egy asszociatív tömböt definiáljunk:

```
$_=" a=12 b=34.4 c=NaN";
%a=("",split(/\s*(\w+)=/) # %a=(" " => "",'a'=>12, 'b'=>'34.4', 'c'=>'NaN')
```

Mivel az első elválasztó előtt is keletkezik egy üres mező, szükség van egy extra elemre a `split` előtt.

A `join ELVALASZTO, TOMB` függvény egy tömbből csinál stringet. A tömb elemeit az ELVALASZTO stringgel valasztja el. A `join` függvényt gyakran használjuk tömbök olvasható kiírására:

```
@a=(1,2,"NaN"); print join(", ",@a),"\n"; # -> 1, 2, NaN
```

Természetesen az ELVALASZTO egy kozonséges string (vagy string értékű kifejezés) és nem string minta.

9.2. Műveletek tömbökön: `grep, map`

A `grep EXPR, LIST` illetve `grep BLOCK LIST` (nincs vessző!) függvényhívás a LIST tömb összes elemére sorban kiszámítja az EXPR kifejezés, illetve a BLOCK utasítások eredményét, és az igaz eredményeket adó elemekből álló tömböt adja vissza. Az elemek ideiglenesen bekerülnek a `$_` változóba.

A `grep` függvény tehát egy lista elemeiből válogat valamilyen szempont alapján. Ez kicsit hasonlít a UNIX `grep MINTA FILE` utasításához, ami egy fájlnak azon sorait írja ki, melyekre a MINTA illeszkedik, de annál jóval általánosabb. Példák:

```
@comments = grep /^\s*#/, @lines;     # select comment lines starting with '#'
```

```
@comments = grep {/^s*#/} @lines;     # same thing with a block
@dir      = split /\n/, `ls -a`;
@files    = grep {$_!~/^\.\.?$/} @dir # exclude '.' and '..'
```

A `map EXPR, LIST` illetve `map BLOCK LIST` (nincs vessző!) függvényhívás a LIST tömb összes elemére kiszámítja az EXPR kifejezés, illetve a BLOCK utasítások eredményét, és az így kapott értékekből álló listát adja vissza. Akárcsak a `grep` függvénynél, az elemek ideiglenesen bekerülnek a `$_` változóba. Példák:

```
@double = map $_*2, @array;  # double the values in @array using an expression
@ARRAY  = map {uc}  @array;  # switch @array to upper case using a block
```

9.3. Tömbök rendezése: `reverse, sort`

A `reverse LIST` függvény lista kontextusban megfordítja a LIST tömb elemeinek sorrendjét. Skalár kontextusban viszont a tomb elemei egy karakter stringbe kerülnek egymás mögé, és ezt az egész stringet fordítja meg a `reverse` függvény. Egy érdekes alkalmazás, ha egy asszociatív tömböt fordítunk meg, ugyanis ez kicseréli a kulcsokat az értékekkel: Példák:

```
print join ', ', reverse ("egy", "ketto");# -> ketto, egy
print scalar reverse ("egy", "ketto");    # -> ottekyge
%english= ("egy" => "one", "ketto" => "two");
%magyar = reverse %english;               # -> ("two" => ketto", "one => "egy")
```

A `sort LIST`, `sort SUBNAME LIST` illetve `sort BLOCK LIST` (nincs vessző!) függvények a LIST tömb elemeit rendezik. Ha csak egy tömb paraméterrel hívjuk meg, akkor a tömböt ABC sorrendben rendezi. Ha a SUBNAME eljárásnévvel, vagy egy név nélküli eljárással (BLOCK) hívjuk meg, akkor a sorrendet az eljárás eredménye szabja meg, ami `-1, 0, 1` lehet, annak megfelelően, hogy a lokális `$a` változót kisebbnek, egyenlőnek, illetve nagyobbnak tekintjük mint a `$b` változót. Ehhez nagyon jól használhatók a `cmp` és `<=>` operátorok (lásd a 4.6 részt a 18. oldalon). Példák:

```
@sorted = sort @array;             # alphabetic sorting
@sorted = reverse sort @array;     # reverse alphabetic sorting
@sorted = sort {$b cmp $a} @array; # reverse alphabetic sorting again
@sorted = sort {$a <=> $b} @array; # numerical sorting
@sorted = sort {uc($a) cmp uc($b)} @array; # case insensitive sorting
@people = ('Alice', 'Bob', 'Cecil');
%age    = ('Alice'=>10, 'Bob'=>23, 'Cecil'=>2.5);
@sorted = sort {$age{$a} <=> $age{$b}} @people; # sorted by age
```

```
sub byage {$age{$a} <=> $age{$b}}
@sorted = sort byage @people      # sorted by age with SUBNAME
```

9.4. Tömbök indexelése tömbbel

Egy tömböt indexelhetünk egy index tömbbel. Az eredmény egy tömb, aminek az elemeit az index tömb választja ki. Fontos, hogy ilyenkor a @ jelet kell használnunk a tömb neve előtt, hiszen az eredmény tömb, és nem skalár. Példák:

```
@a = (1..10);
print join ",",@a[0..4],"\n";  # =>   1,2,3,4,5
@i = (3,5,7);
print @a[@i];                  # =>   4,6,8
print @a[map 2*$_, 0..$#a/2];  # =>   1,3,5,7,9
```

9.5. Feladat

Olvasson be egy fájlból egy táblázatot, rendezze sorba a harmadik oszlop értéke szerint. Írja ki a táblázat 3., 2. és 4. oszlopát LaTex formátumban, azaz az elemek között álljon & jel, a sorok végén pedig \\. A táblázat előtti, beszúrt, vagy mögötti nem számokból álló sorokat hagyja ki. A bemenetre példa:

```
Atmero   Szakito szilardsag  Suruseg  Minta kora
[mm]      [N]                [kg/m3]   [ev]
0.0011    125                 1100      1
0.0023    432                 3400      2
0.0035    940                 7500      10
A mereseek pontossaga 5\%.
```

9.6. Tömb elemek hozzáadása és elvétele: `push, pop, shift, unshift, splice`

A `push TOMB, LISTA` függvény hozzáadja a LISTA elemeit a TOMB végéhez. A LISTA lehet egy skalár, egy tömb, vagy éppen vesszővel elválasztott skalárok és tömbök is. A függvény értéke a megnövelt tömb elemeinek a száma, bár erre ritkán van szükség. Példák:

```
push @comments, $_ if /^\s*#/; # add a scalar
push @files, '.', '..';        # add two more items
push @a, @b;                   # append @b to the end of @a
```

A `pop TOMB` függvény a `push` ellentéte, azaz a TOMB tömb utolsó elemét adja vissza, és az elveszi a TOMB tömbből. Ha a TOMB üres, akkor a `pop` definiálatlan (hamis) értéket ad vissza. Ha a TOMB argumentum hiányzik, akkor szubrutinokban a @_, a főprogramban a `@ARGV` argumentum lista utolsó elemét adja vissza. Példák:

```
$last_arg = pop;                            # extract last argument
print "$item\n" while $item = pop(@array); # print @array backwards
```

A `shift TOMB` függvény ugyanazt csinálja, mint a `pop`, csak nem az utolsó, hanema a `TOMB` tömb első elemét adja vissza, és törli a tömbből. A `shift` függvényt nagyon gyakran használják az eljárások argumentumainak kiolvasására:

```
print &compare($a,$b);
sub compare
   my $a=shift or die "Missing first argument for sub compare\n";
   my $b=shift or die "Missing second argument for sub compare\n";
   $a <=> $b;
}
```

Az `unshift TOMB, LISTA` függvény a `shift` ellentéte, ugyanazt csinálja mint a `push` csak a LISTA elemeit nem a TOMB végére, hanem az elejére rakja.

Végül a `splice TOMB, KEZDET, HOSSZ, LISTA` eljárás mindazt tudja, amit a fenti 4 függvény, és nagy mértékben hasonlít a `substr` függvényre, csak nem egy string karaktereit, hanem a TOMB elemeit manipulálja. Ha az összes argumentum adott, akkor a TOMB tömb HOSSZ darab elemét a KEZDET indexű elemtől kezdve kicseréli a LISTA tömbre, és a függvény az eltávolított elemeket adja vissza. Ha a LISTA argumentum hiányzik, akkor az elemek egyszerűen törlődnek. Ha HOSSZ negatív, akkor a TOMB végétől visszafele számít. Ha a HOSSZ hiányzik, akkor a KEZDET indextől a TOMB végéig lévő elemeket távolítja el a `splice`. Ha a KEZDET változó negatív, akkor a TOMB végétől számit. Mindezek alapján, a `push, pop, shift, unshift` függvények a következőképpen is írhatóak:

```
push @a, $b;    # same as splice @a, $#a+1, 0, $b;
$b = pop @a;    # same as $b = splice @a, -1;
$b = shift @a;  # same as $b = splice @a, 0, 1;
unshift @a, $b; # same as splice @a, 0, 0, $b;
```

9.7. Feladat

Írjon egy eljárát, ami két tömböt hasonlít össze. A tömbök elemeinek számát a tömb előtti argumentumban adjuk meg. Az eredmény legyen igaz, ha a két tömb hossza megegyezik, és elemeik is megegyeznek, és legyen hamis egyébként.

10. Asszociatív tömbök kezelése

10.1. Asszociatív tömb olvasása: `keys, values`

A `keys HASH` függvény a HASH asszociatív tömb kulcsait adja vissza lényegében véletlenszerű sorrendben. Például egy asszociatív tömb kiírása kulcsok szerint:

```
foreach $key (keys %hash){print "$key => $hash{$key}\n"}
```

A `values HASH` függvény az asszociatív tömbben lévő értékeket adja vissza. Például ha a `%area` az országok méretét adja vissza, akkor az országok összterülete:

```
foreach $area (values %area){$total += $area;}
```

10.2. Asszociatív tömb elemek ellenőrzése és törlése: `exists, delete`

Ha egyszer egy asszociatív tömb elemet használunk, akkor azt csak a `delete HASH_ELEM` függvénnyel lehet törölni, ahol a HASH_ELEM a törlendő elemet tartalmazza. Például egy adatbázisból kitöröljük az elhunytak adatait:

```
foreach $dead (@dead){delete $people{$dead}}
```

Ez nem keverendő össze azzal, hogy a kulcshoz tartozó értéket töröljük például az `undef $people{$dead}` utasítással.

Arról, hogy egy kulcs szerepel-e egy asszociatív tömbben az `exists HASH_ELEM` függvénnyel győződhetünk meg. Ennek értéke igaz ha az elem kulcsa szerepel. Pédául:

```
foreach $name (@names) {
   print "$name is dead" unless exists $people{$name};
}
```

10.3. Asszociatív tömb adatbázishoz csatolása: `dbmopen, dbmclose, each`

Nagyon nagy asszociatív tömbök gyakran nem férnek el a memóriában. Ilyenkor adatbázis fajlt használhatunk. A `dbmopen HASH, DBFILE, MOD` függvény a HASH asszociatív tömböt hozzárendeli a DBFILE nevű adatbázishoz. Az adatbázis maga általában két DBFILE.dir és DBFILE.pag, vagy egy darab DBFILE.db nevű fájlból áll, attól függően, hogy mi az alapértelmezett adatbázis kezelő program. A MODE a fájl írási és olvasási tulajdonságait határozza meg. A második, harmadik és negyedik jegyek a tulajdonosra, a csoportra, és az egyéb felhasználókra vonatkoznak. Az első bit az olvashatóságot, a második az írhatóságot, a harmadik pedig a végrehajthatóságot adja meg.

A `dbmclose HASH` függvény bezárja az adatbázis fájlt, és ezzel a HASH asszociatív tömb is kiürül.

```
dbmopen %szotar, "SZOTAR", 0666;
while(<>){
   ($magyar, $english) = split;
   $szotar{$magyar} = $english;
}
dbmclose %szotar;
```

Az `each HASH` függvény sorba megy egy adatbázis összes kulcs-érték párján, és ezeket a párokat adja vissza egy-egy két elemű tömbben. A párok sorrendje két elemű. Az `each` függvény akkor hasznos, ha nagyon nagy asszociatív tömb elemein akarunk végigmenni. Például keressük meg azokat az angol szavakat, melyek hosszabbak a magyar megfelelőnél:

```
while(($magyar, $english)=each %szotar){
   push(@longer, $english) if length($english)>length($magyar);
}
```

10.4. Feladat

Írjon egy programot, ami a szótár adatbázisból kikeresi a felhasználó által beírt szavakat. Ha a szó nem szerepel a szótárban, azt jelzi. Módosítsa a programot úgy, hogy az egy szöveget szóról-szóra lefordítson, a szótárból hiányzó szavakat hagyja meg az eredeti nyelven.

11. Formázott kiírás

11.1. C típusú formázott írás: `printf, sprintf`

A `printf FAJLKEZELO FORMA,LISTA` utasítás a FAJLKEZELO által reprezentált fájlba írja a LISTA értékeit a FORMA string által megadott formátumban. Ha a FAJLKEZELO hiányzik, akkor az STDOUT-ra ír. A formátum követi a C nyelvben használt alakot. A LISTA egyes elemeinek a FORMA stringben szereplő

`%m.nx`

alakú kifejezések felelnek meg, ahol `m,n` a kiírt érték hosszát definiáló egész számok, míg `x` a típust adja meg:

x	jelentés
c	karakter
d	decimális szám
e	exponenciális formájú valós szám
f	fix tizedes ponttal ábrázolt valós szám
g	kompakt formában írt valós szám
ld	hosszú (long) decimális szám
lo	hosszú oktális szám
lu	hosszú előjel nélküli (unsigned) egész
lx	hosszú hexadecimális szám
o	oktális szám
s	string
u	előjel nélküli (unsigned) egész
x	hexadecimális szám
X	hexadecimális szám nagy betűkkel

Magát a % karaktert a %% írja ki. Példák:

```
printf "Production increased by %4.2f%%\n", 12.3;      # -> ... 12.30%
printf "%s %4.2f%%\n","Production increased by", 12.3; # -> ... 12.30%
printf "Max. speed is %12.4ekm/s\n",2.99e5;            # -> ... 2.9900e+05km/s
printf "Max. speed is %12.1ekm/s\n",2.99e5;            # -> ... 3e+05km/s
printf "Max. speed is %gkm/s\n",2.99e5;                # -> ... 299000km/s
printf "Decimal value %4d\n", 15;                      # -> ...   15
printf "Left adjusted %-4d!\n", 15;                    # -> ... 15  !
printf "Padded with 0 %4.3d\n", 15;                    # -> ...  015
printf "Hexadeximal : %4.3x\n", 15;                    # -> ...  00f
printf "HEXADEXIMAL : %4.3X\n", 15;                    # -> ...  00F
printf "Octal value : %4.3o\n", 15;                    # -> ...  017
```

```
printf "Characters  : %2c%2c%2c%2c\n",80,101,114,108;  # -> ...  P e r l
```

A `sprintf FORMA, LISTA` függvény egy stringet ad vissza, melyben a LISTA értékei szerepelnek a FORMA által meghatározott módon. A függvény így többek között alkalmas arra, hogy egy számot oktális vagy hexadecimális formába alakítsunk:

```
$oct = sprintf(%o,63);   # same as $oct = "77";
$hex = sprintf(%x,255);  # same as $hex = "ff";
```

11.2. Konverziós függvények: `ord, chr, hex, oct, crypt`

A `hex HEXASZAM` függvény kiírja a HEXASZAM string értékét hexadecimálisan értelmezve. Az `oct OKTALISSZAM` függvény pedig egy nyolcas számrendszerben leírt szám értékét adja meg. A `ord CHAR` függvény megadja a CHAR karakter ASCII kódját, míg a `chr SZAM` függvény a SZAM indexű ASCII karaktert adja vissza (éppúgy mint a `sprintf "%c",SZAM` függvény). Példák:

```
print ord("P"), "\n";   # -> 80
print chr(80),  "\n";   # -> P
print hex("ff"),"\n";   # -> 255
print oct("77"),"\n";   # -> 63
```

A `crypt SZOVEG, KULCS` függvény eredménye a SZOVEG string KULCS szerint titkosított formája. Ez a titkosítás nem megfordítható. Leginkább jelszavak titkosítására használt módszer. Például a következő program megpróbálja kitalálni az `/etc/passwd` fájlban szereplő első kulcsszót:

```
open PASSWD '/etc/passwd'; ($user,$passwd) = <PASSWD>; $guess = <>;
if(crypt("$guess,$passwd) eq $passwd){print "Your guess is correct!\n"}
```

11.3. Perl típusú formázott kiírás: `format, write`

A `format` utasítás szintaxisa meglehetősen furcsa, de viszonylag jól áttekinthető. Különösebb magyarázat helyett íme egy rövid példa:

```
format STDOUT_TOP =
   item    VAT   price   notes
--------- ----- ------- ---------------
.
```

```
format STDOUT =
@|||||||| @>>>% $@##.## ^<<<<<<<<<<<<<
$name,    $vat, $price, $notes
~~                          ^<<<<<<<<<<<
                            $notes
.
$name="cat"; $price=119.99; $notes="not available"; $vat=25;
write STDOUT;
$name="small dog"; $price= 99.9 ; $notes="in stock but someone has booked it";
write STDOUT;
```

A `format` utasítás egy kép-sorral kezdődik, melyben a @ kezdetű részek helyettesítik az értékeket. A $<, |, >$ karakterek a balra, középre, jobbra igazítást, a ##.## pedig jobbra igazított numerikus mezőket jelent. A képsort követik azok a változók, melyeket ki akarunk írni. A ^ karaktert követő mező hossza rugalmas. A hullámmal ~ kezdődő sor csak akkor íródik ki, ha nem üres. A dupla hullámmál kezdődő sor addig ismétli magát, amíg van mit kiírni. A `format` utasítást a pont zárja.

A kiírás a `write FAJLKEZELO` utasítással történik. Az `STDOUT_TOP` formátum az első `write` utasításnál jelenik meg. A fenti program eredménye:

```
   item    VAT   price   notes
--------- ----- ------- ---------------
   cat      25% $119.99 not available
small dog   25% $ 99.90 in stock but
                            someone has
                            booked it
```

Ha nem az STDOUT fájlkezelőre akarunk írni, akkor értelemszerűen annak a fájlkezelőnek a nevét adjuk meg a `format` és az `write` utasításokban. Az STDOUT az alapértelmezett fájlkezelő, így el lehet hagyni, azaz `format =` és `write;` írható a fenti `format STDOUT =` és `write STDOUT` helyett, de az `STDOUT_TOP` nem hagyható el.

12. Bináris adatok kezelése

12.1. Konverzió bináris formátumba: `pack, unpack, vec`

A `pack FORMA, LISTA` függvény egy bináris stringet ad vissza, mely a LISTA értékeit tartalmazza a FORMA által meghatározott formában. A FORMA alakja a következő:

```
xN xN ...
```

ahol az `x` karakter a típust, az `N` egész pedig a darabszámot jelöli (kivéve a string típusnál, ahol a string hosszát). A szóközöknek nincs jelentősége. Ha `N` hiányzik, akkor 1 darab `x` típusú lista elemet (illetve 1 karakter hosszú stringet) alakít át. Ha `N` helyén egy csillag áll, akkor az összes további lista elemet berakja a stringbe. Az `x` típus fontosabb lehetséges értékei:

x	jelentés
a	ASCII string \0 -val kibővítve
A	ASCII string szóközzel kibővítve
b	bit string bal oldalon a legkisebb helyértékkel
B	bit string jobb oldalon a legkisebb helyértékkel
c	előjeles karakter érték
C	előjel nélküli karakter érték
d	8 byte-s valós (double precision) szám
f	4 byte-s valós (float) szám
l	előjeles 4 byte-s (long) egész
L	előjel nélküli 4 byte-s (long) egész
n	2 byte-s egész network (big-endian) sorrendben
N	4 byte-s egész network (big-endian) sorrendben
s	előjeles 2 byte-s (short) egész
S	előjel nélküli 2 byte-s (short)egész
v	2 byte-s egész VAX (little-endian) sorrendben
V	4 byte-s egész VAX (little-endian) sorrendben

Az `unpack FORMA, STRING` függvény egy tömböt ad vissza, mely a STRING stringben FORMA formátumban binárisan tárolt elemeket tartalmazza. Példák:

```
print FILE pack "l*",@longs;           # store @longs in binary file
@longs = unpack "l*",<FILE>;           # read @longs from binary file
print pack("c4",80,101,114,108)."\n"; # -> "Perl"
$hex = unpack("H*","Perl");            # -> "5065726c"
@chars = unpack("c*","Perl");          # -> (80,101,114,108)
```

A `vec STRING, INDEX, BITSZAM` függvény visszaadja a STRINBEN tárolt BITSZAM bites egészek közül az INDEX sorszámút (balról nézve). Ha a `vec` egy értékadás bal oldalán áll, akkor a STRING megfelelő bitjeit lehet beállítani. A BITSZAM értéke csak 1, 2, 4, 8, 16 vagy 32 lehet. A bit vektor rendkívül kompakt tárolást tesz lehetővé, például 8 logikai változó tárolható egy bájtban (BITSZAM=1). A `pack` és `unpack` függvények `b*` formátuma a bit vektort 1-sekből és 0-kból álló stringre tudja konvertálni, illetve fordítva.

12.2. Műveletek bináris számok között: | & ^ ~

A | (vagy), & (és), és a ^ (kizáró vagy) műveletek két string baj1tjai, vagy két egész bitjei között hatnak. A ~ (negálás) operátor pedig egy string bájtjait negálja, vagy egy egész számot (ami 32 vagy 64 bites is lehet a géptől függően). A » és a « operátorok a bal oldalon álló egészben a biteket a jobb oldalon álló számú bittel shiftelik jobbra illetve balra.

12.3. Feladatok

Írjon egy programot, ami eldönti, hogy a gép amin fut, little-endian vagy big-endian formában írja a bináris számokat.

Írjon programot, ami 4 bájtos big-endian számokból (lehet egész és valós is) 4 bájtos little-endian-t csinál, és fordítva.

Írjon egy programot, ami egy fájlban tárolt bináris duplaprecíziós valósakat szimpla precizitásúakra cseréli ki.

Írjon egy programot, ami egy decimális egész számot visszaír kettes számrendszerben.

Tároljon 8 logikai változót egyetlen bájtban, majd olvassa ki az értéküket.

13. Fájlok és könyvtárak kezelése

13.1. Fájlok olvasása: `getc, read, seek, tell, eof`

A `getc FILEHANDLE` függvény kiolvassa a következö bájtot a FILEHANDLE fájlkezelőről. illetve Ha a FILEHANDLE hiányzik, akkor az STDIN-ről. A billentyűzet pufferelését ez nem szünteti meg!

A `read FILEHANDLE, STR, HOSSZ, STRPOZ` függvény a FILEHANDLE fajlkezelőről HOSSZ darab bájtot próbál beolvasni az STR stringbe. Az STRPOZ index, ha szerepel, azt adja meg, hogy az STR stringbe hova írunk! Az STR string hossza szükség szerint változik. A függvény a ténylegesen beolvasott bájtok számát adja vissza, hiba esetén pedig definiálatlan értéket. Egy egész fájl így olvasható be egy stringbe:

```
open IN, "valami.gz";
read IN, $content, -s "valami.gz";
```

A fájl méretét a `-s` operátor adja meg. A `read` függvény akkor is nagyon hasznos, ha nagyon nagy méretű bináris fájlnak egyszerre csak egy részét akarjuk csak kiolvasni. Például:

```
while (read FROM, $buffer, 16384){print TO $buffer;}
```

A `seek FILEHANDLE, POZICIO, KEZDOPONT` függvény a FILEHANDLE kezelőhöz tartozó fájlmutatót a POZICIO bájthoz állítja a KEZDOPONT pozícióhoz képest. Ha a KEZDOPONT értéke 0, akkor a fájl elejéhez, ha 1 akkor a pillanatnyi mutatóhoz képest, ha 2 akkor a fájl végéhez képest. A függvény eredménye 1 ha sikerült a pozicionálás, és 0 ha nem. A `seek` a fájl vége mögé mutató pozíciót is elfogad, de persze ilyenkor az `eof` függvény hamis értékűvé válik.

A `tell FILEHANDLE` függvény megadja a FILEHANDLE kezelőhöz tartozó fájlmutató pillanatnyi értékét bájtban. A `tell` mindig a fájl elejéhez képesti pozíciót adja meg.

Az `eof FILEHANDLE` függvény igaz értéket (1) ad, ha a FILEHANDLE kezelőhöz tartozó mutató a fájl végére (vagy amögé) mutat. Az `eof` mindenféle paraméter nélkül az utoljára olvasott fájl végét jelzi. Az `eof()` viszont a parancssorban felsorolt fájlok közül az utolsóét!

13.2. Könyvtárak olvasása: `glob, opendir, closedir, readdir`

A `glob MINTA` a jelenlegi könyvtárban a MINTA stringre illeszkedő fájlok listáját adja vissza egy tömbbe. Például a Perl programok listája:

```
@perlcodes = glob('*.pl');
```

Linux alatt a `split /\n/, `ls -d MINTA`` hasonló eredményt ad, mint a `glob MINTA`.

Az `opendir DIRHANDLE, DIRNAME` megnyitja a DIRNAME nevű könyvtárat a DIRHANDLE könyvtárkezelővel.

A `closedir DIRHANDLE` bezárja a könyvtár olvasását.

A `readdir DIRHANDLE` függvény skalár kontextusban a könyvtár következő elemét (a könyvtárban szereplő fájlt vagy alkönyvtár nevét) adja vissza, ha nincs ilyen, akkor definiálatlan értéket. Lista kontextusban az egész (még visszalevő) könyvtárat megkapjuk egy tömbben. Például ha szükségünk van a könyvtár összes elemére, kivéve a `.` és `..` elemeket, akkor

```
opendir DIR, ".";
@entries = grep !/^\.\.?$/, readdir DIR;
closedir DIR;
```

13.3. Feladat

Írja ki megfelelően formázva egy könyvtár összes elemét abc sorrendben, majd az alkönyvtárak összes elemét, és így tovább. A kimenet formája legyen hasonló az `ls -R` utasítás eredményéhez.

13.4. Fájlok kezelése: `rename, unlink, link, symlink, readlink, stat, lstat, chmod, chown, utime`

A `rename REGINEV, UJNEV` függvény átnevezi a REGINEV nevű fájlt/könyvtárat az UJNEV stringre. A függvény eredménye 1 ha az átnevezés sikerült, 0 ha nem. Unix alatt megfelel a `` `qx mv -f REGINEV UJNEV` `` utasításnak.

Az `unlink FILELIST` függvény kitörli a FILELIST tömbben szereplő fájlokat. A függvény a sikeresen törölt fájlok számát adja vissza.

A `link REGINEV, UJNEV` függvény egy UJNEV nevű hard linket csinál a REGINEV nevű fájljoz. A függvény siker esetén 1-t, egyébként 0-t ad vissza. Unix alatt megfelel a `` `ln REGINEV UJNEV` `` utasításnak.

A `symlink FAJLNEV, LINKNEV` függvény egy LINKNEV nevű szimbolikus linket csinal a FAJLNEV nevű fájlhoz. A függvény siker esetén 1-t, egyébként 0-t ad vissza. Unix alatt megfelel a `` `ln -s REGINEV UJNEV` `` utasításnak.

A `readlink LINKNEV` függvény megadja annak a fájlnak a nevét, amire a LINKNEV nevű szimbolikus link mutat.

A `stat FILENEV` vagy `stat FILEKEZELO` függvény egy 13 elemű tömböt ad vissza, ami a fájl szinte minden létező tulajdonságát megadja. Az `lstat LINKNEV` ugyanezt csinálja szimbolikus linkekre (és nem a fájlra, amire a link mutat). Használat:

```
($device,$nodenumber,$mode,$number_of_links,$user_id,$group_id,
 $device_id,$size,$access_time,$modify_time,$change_time,
 $block_size, $number_of_blocks) = stat $filename;
```

A hozzáférési idők itt másodpercben adottak 1970 január 1 (UTC) óta, ellentétben a `-A, -C, -M` operátorokkal, amik az időt napokban adják meg (lásd az 5.5

részt a 24. oldalon).

A `chmod MOD, LISTA` beállitja a MOD hozzáférési módot a LISTA tömbben szereplő fájlokra. A MOD egy oktális szám, amit vagy 0 kezdetü konstansként, vagy az oct() függvénnyel adhatunk meg. A függvény eredménye a sikeresen megváltoztatott hozzáférési módú fájlok száma. Például:

```
chmod 0700, "my.dat"; chmod oct("777"), "our.dat";
```

A függvény hatása ugyanaz mint a `` `chmod MOD FILÉ FILE2 ...` `` utasításé, de ez utóbbi nagyon sok fájlra nem müködik, míg a `chmod` függvény igen.

A `chown UID, GID, LISTA` a UID számú felhasználót, és a GID számú csoportot rendeli hozzá tulajdonosként a LISTA tömbben szereplő fájlokhoz. A függvény eredménye a sikeresen megváltoztatott tulajdonosú fájlok száma. A függvény hatása ugyanaz mint a `` `chown UID:GID FILÉ FILE2 ...` `` utasításé.

Az `utime ACCESSTIME, MODTIME, FILELIST` függvény beállítja a hozzáférési és a módosítási időt az ACCESSTIME illetve a MODTIME idejére (másodpercben) a FILELIST tömbben szereplő összes fájlra. A függvény eredménye a sikeres időmódosítások száma. A Unix `` `touch FILÉ FILE2 ...` `` parancsnak a `utime time, time, FILELIST` felel meg, bár ez a `touch` paranccsal ellentétben nem hozza létre a fájlt, ha korábban nem létezett.

13.5. Könyvtárak kezelése: `chdir, mkdir, rmdir`

A `chdir DIRNEV` függvény a DIRNEV könyvtárra állítja be a munkakönyvtár értékét. Ha a DIRNEV hiányzik, a felhasználó home könyvtárába megy. A függvény eredménye siker esetén 1, egyébként 0. Unix alatt megfelel a `` `cd DIRNEV` `` utasításnak.

A `mkdir DIRNEV, MOD` egy DIRNAME nevű könyvtárat készít, melynek hozzáférési módját a MOD oktális szám adja meg. A függvény eredménye siker esetén 1, egyébként 0. Hatása megegyezik a `` `mkdir DIRNEV` `` utasítással.

Az `rmdir DIRNEV` letörli a DIRNEV nevű könyvtárat, ha üres(!). Ha a DIRNEV hiányzik, a `$_` könyvtárat törli. A függvény eredménye siker esetén 1, egyébként 0. Hatása megegyezik a `` `rmdir DIRNEV` `` utasítással.

13.6. Feladat

Írjon programot, ami lemásol egy könyvtár fát. Csak a könyvtárakat készítse el, a fájlokat nem kell lemásolni. Ügyeljen arra, hogy a könyvtárak tulajdonosa,

csoportja, hozzáférési módja megegyezzen az eredetivel.

14. Idővel kapcsolatos függvények: `time, gmtime, localtime, times, sleep`

A `time` függvény az 1970. január 1. (UTC) óta eltelt időt adja meg másodpercekben, így kompatibilis a `stat` és `utime` függvényekkel (lásd a 13.4 részt az 55. oldalon).

A `gmtime IDO` függvény az IDO által másodpercekben megadott időt konvertálja a greenichi középidőre (GMT illetve UTC). Ha az IDO paraméter hiányzik, akkor a `time` függvény értékét, azaz a pillanatny időt alakítja át. Skalár kontextusban egy stringet ad vissza, ami az időt tartalmazza a

```
Wed Nov 20 18:40:46 2002
```

formátumban. Tömb kontextusban pedig egy 9 egész számból álló tömböt kapunk:

```
($sec, $min, $hour, $day_of_month, $month, $year_since_1900,
 $day_of_week, $day_of_year, $is_daylight_saving_time) = gmtime;
```

Fontos megemlíteni, hogy a `$month` a 0..11, a `$day_of_week` pedig a 0..6 tartományba esik, így alkalmasak egy megfelelő hónapnevekből illetve napnevekből álló tömb indexelésére, valamint azt is, hogy a hét első napja az angolszász szokásoknak megfelelően vasárnap, és nem a hétfő. Az utolsó tömb elem nyári időszámítás esetén 1, egyébként 0.

A `localtime IDO` függvény ugyanúgy működik, mint a `gmtime` függvény, azzal a különbséggel, hogy ez a helyi (pontosabban a számítógépen beállított) időzónának megfelelő időt írja ki.

A `times` függvény egy négy elemű valós számokból álló tömböt ad vissza, mely a felhasználói és a rendszer időket tartalmazzák az éppen futó eljárásra, illetve az ezek által elindított "child" processzekre. A `times` függvény alkalmas egy Perl program futásidejének pontos mérésére:

```
$start=(times)[0];
for $i (1..1000000){$n++};
$end  =(times)[0];
printf "This took %.2f seconds",$end-$start;
```

A `sleep MASODPERCEK` függvény a MASODPERCEK változóban megadott másodpercet várakozik, majd továbbfut. Ha a paraméter hiányzik, akkor csak egy ALARM szignál hatására ébred fel a program. Például:

```
print "You have 10 seconds to read this\n";
sleep 10;
print "\n" x 25;
```

14.1. Feladatok

Írja ki a pillanatnyi időt és dátumot a helyi idő szerint magyarul! Például: "Most 2002. november 20. szerda 21 óra 51 perc 10 másodperc van."

Mérje meg, hogy meddig tart a szinusz függvény kiszámítása!

15. Referenciák

A Perl nyelv általában lineáris (1 dimenziós) adatstruktúrákat használ. Ilyenek a rendezett tömbök és az asszociatív tömbök is. Ha ennél bonyolultabb adatszerkezetre van szükségünk, akkor azt a referenciák segítségével lehet a legjobban megoldani.

A Perl nyelvben a referencia ugyanúgy működik, mint a UNIX operációs nyelvben a kemény link. Amíg egy adatra mutat legalább egy referencia, addig az adat létezik és elérhető. Az adat lehet egy változóhoz tartozó adat, de lehet egy név nélküli anonim adat is. A referencia maga egy skalár mennyiség ami mutathat tetszőlegesen bonyolult adatra. Az adatot magát csak a referencia explicit feloldásával (dereferencing) lehet elérni illetve megváltoztatni.

15.1. Referencia létrehozása változókra: \

A visszafele dőlő tört vonal segítségével lehet egy változóra vonatkozó referenciát létrehozni. Például:

```
$a="this is a string";
$refscalar=\$a;                              # Reference to $a
@a=("this","is","an","array");
$refarray=\@a;                               # Reference to @a
%a=("this" => "is", "a" => "hash");
$refhash = \%a;                              # Reference to %a
```

```
sub a {print "this is a subroutine: ",@_,"\n"};
$refsub  = \&a;                              # Reference to &a
```

15.2. Referenciák létrehozása név nélküli adatokra: \, [ ], { }, sub { }

Név nélküli skalárra mutató referenciát a visszafele dőlő törtvonallal lehet létrehozni:

```
$refnumber=\3.1415926;
$refscalar=\"this is a string";
```

Név nélküli rendezett tömbre mutató referencia létrehozása a szögletes zárojellel történik:

```
$refarray = ["this","is","an","array"];
```

Név nélküli asszociatív tömbre mutató referencia létrehozása a kapcsos zárojellel történik:

```
$refhash = {"this" => "is", "a" => "hash"];
```

Név nélküli eljárásra mutató referenciát a `sub { }` utasítás hoz létre:

```
$refsub = sub {print "this is a subroutine:",@a,"\n"};
```

Az eljárás nem kerül végrehajtásra, csupán a referencia kap értéket. Ellentétben a tömb és hash referenciákkal, a fenti utasítás többszöri kiadása (például egy ciklusban) nem fog létrehozni több eljárást.

15.3. Referenciák használata: ${ }, @{ }, %{ }, &{ }

A referenciák használhatók mindenhol, ahol egy alfanumerikus változó vagy eljárás név szerepelne. Például

```
print "$$refscalar\n";          # -> this is a string
print join ",",@$refarray;      # -> this,is,an,array
print join ",",keys %$refhash;  # -> this,a
```

```
&$subref("param");              # -> this is a subroutine: param
print $$refarray[3];            # -> array
print $$refhash{"a"};           # -> hash
```

Ha a referencia nem egy egyszerű változó, hanem például egy tömb vagy hash eleme, akkor a referenciát egy blokkba, azaz kapcsos zárójelek közé tesszük:

```
@refs = (\"pi", \3.14);
print "${$refs[0]} = ${$refs[1]}\n";  # pi = 3.14
@arrays = ([1,2,3], [4,5]);
print join(",",@{$arrays[1]}),"\n";   # 4,5
```

15.4. Referenciák használata tömb és hash elemekre: ->

Tömbre vagy hashre mutató referenciák elemeire a nyíl -> operátor segítségével is lehet hivatkozni. Például:

```
print $refarray->[3],"\n";  # same as $$refarray[3]  or ${$refarray}[3]
print $refhash->{"a"},"\n"; # same as $$refhash{"a"} or ${$refhash}{"a"}
```

Itt jegyezzük meg, hogy a konstans hash kulcsoknál az idézőjel elhagyható, ha a string nem tartalmaz szóközt. Ez bizonyos esetekben javítja az olvashatóságot.

További egyszerűsítést ad a jelölésben, hogy a szögletes illetve kapcsos zárójelek között álló nyíl operátor elhagyható! Így például

```
@multi = ([11,12,13],    # array of array referencies
          [21,22,23]);
print $multi[1][2];      # same as $multi[1]->[2] or ${$multi[1]}[2]
```

Ha maga a több dimenziós tömb is egy referencia, akkor

```
$multiref = [ [11,12,13],   # reference to an array of array referencies
              [21,22,23] ];
print $multiref->[1][2];    # same as $$multiref[1][2] but clearer
```

15.5. Referenciák implicit létrehozása

Ha egy referenciát értékadásban használunk, akkor a referencia és a hozzátartozó adat automatikusan létrejön. Ez az egyik legszokásosabb módja annak, hogy bonyolult adatszerkezeteket hozzunk létre:

```
$emberek->{Gabor}->{szemek}->{jobb}->{szin} = "barna";
```

Ez létrehoz egy `$emberek` referenciát, ami egy hashre mutat, amiben a "Gabor" kulcshoz egy hash referencia tartozik, amiben a "jobb" kulcshoz egy újabb hash referencia tartozik, amiben a "szin" kulcshoz a "barna" értéket rendeltük hozzá.

15.6. Szimbolikus referenciák

Hasonlítanak a UNIX szimbolikus linkekre. A szimbolikus referencia egyszerűen a változó nevét tartalmazza. Ha egy értéket referenciaként használunk, noha az nem referencia, akkor változó névként kerül kiértékelésre. Például:

```
$name      = "index";
$$name     = 1;       # $index    = 1;
@$name     = (1,2,3); # @index    = (1,2,3);
print $name->[0];     # print $index[0];
```

A szimbolikus referenciák használata meglehetősen veszélyes. Amikor egy igazi referenciát akarunk használni, ami program hiba folytán nem egy referencia, akkor a string értékének megfelelő változót használjuk, és nem kapunk hiba üzenetet. Ez elkerülhető a

```
use strict 'refs';
```

utasítással, ami a bezáró blokk végéig tart. Ezt minden olyan programba érdemes beírni, ami használ referenciákat. Ha valahol valóban szimbolikus referenciára van szükség, akkor a

```
no strict 'refs';
```

utasítással kapcsolhatjuk ki az ellenőrzést a bezáró blokk végéig.

15.7. Referencia típusának lekérdezése: `ref`

A `ref REFERENCIA` függvény visszaadja a referencia típusát. Az eredmény egy string, ami lehet

- '', üres string ha nem referencia
- 'SCALAR', skalárra mutat

- 'ARRAY', tömbre mutat
- 'HASH', asszociatív tömbre mutat
- 'CODE', eljárásra mutat
- 'REF', azaz a referencia egy referenicára mutat

Ha egy referenciát string kontextusban írunk ki, akkor az tartalmazza a típust és egy címet hexadecimális formában. Például:

```
$a=12;
$aref=\$a;
print ref $aref;  # -> SCALAR
print $aref;      # -> SCALAR(0x810734c)
```

15.8. Listák listája

Egy kétdimenziós tömb létrehozása:

```
@tomb2D = (["kicsi","kozepes","nagy"],
           [1,2,3],
           ["ket","elem"]);
```

Mint látható, a tömbben lévő referenciák mutathatnak különböző méretű tömbökre. Ha eleve egy referenciát akarunk, ami 2 dimenziós tömbre mutat, akkor szögletes zárójelet kell használni

```
$ref2D = [ [1,2,3], [4,5,6], [7,8]];
```

Új elemeket nagyon egyszerű hozzáadni:

```
$tomb2D[2][3] = 4;
$ref2D->[2][3] = 6.5;
```

Ha egy fájlból akarunk beolvasni egy 2 dimenziós tömböt:

```
while(<>){
    push @tomb, [split];
    @line = split;
    push @$tombref, [@line];
}
```

A @ operátor tömböt csinál a tömb referenciából, míg a [ ] operátor referenciát csinált a tömbből. Fontos megjegyezni, hogy a

```
push @$tombref, \@line
```

nem lett volna jó, mert akkor minden sor ugyanazt a referenciát tartalmazná, míg a `@line` tömb értéke újra és újra felülíródik.

Ha egy sorhoz akarunk hozzáadni új elemeket

```
push @{ $tomb2D[1] }, "meg", "ketto";
push @{ $ref2D->[1] }, "tovabbi", "harom", "elem";
```

Itt szükség van a `@{..}` operátorra, hogy az 1-es indexű tömbreferenciából igazi tömböt csináljunk, ugyanis a `push` függvény csak tömbökre értelmezett.

A többdimenziós tömb kiírásakor a referenciákat fel kell oldani. Például:

```
foreach $ref (@tomb2D){
   print join(',',@$ref),"\n"
}
for $i (0..$#{$ref2D}){
   print "$i:",join(' ',@{$ref2D->[$i]}),"\n";
}
```

15.9. Komplex adatszerkezetek

Készítsünk egy listát az első emberek adataival! Az emberek tömbjének minden egyes eleme egy referencia egy névtelen asszociatív tömbre, ami egyelőre csak a nevet tartalmazza:

```
foreach $nev ("Adam","Eva","Kain","Abel"){
    push @emberek,{nev => $nev};
}
```

A házastárs legyen egy újabb elem az asszociatív tömbben, ami egy mutató egy másik emberre:

```
$emberek[0]{hazastars} = $emberek[1];
$emberek[1]{hazastars} = $emberek[0];
```

Ezek után Ádám feleségének nevét a

```
print "Adam felesege: $emberek[0]{hazastars}{nev}\n";
```

utasítással írhatjuk ki. A gyerekeket egy tömb referenciába írjuk, melynek elemei szintén referenciák:

```
$emberek[0]{gyerekek} = [$emberek[2], $emberek[3]];
$emberek[1]{gyerekek} = [$emberek[2], $emberek[3]];
```

Éva első gyerekének neve

```
print "Eva elso gyermeke $emberek[1]{gyerekek}[0]{nev}\n";
```

Készítsünk egy asszociatív tömböt, ami név alapján megtalálja a megfelelő adatokat:

```
foreach $ember (@emberek){
  $emberek{$ember->{nev}} = $ember;
}
```

Most már név szerint is hivatkozhatunk az emberekre (feltéve, hogy nincs köztük azonos nevű). Adjuk meg az emberek nemét, ami egy új elem lesz az asszociatív tömbben:

```
$emberek{Adam}{nem}="ferfi"; $emberek{Eva}{nem}="no";
$emberek{Kain}{nem}="ferfi"; $emberek{Abel}{nem}="ferfi";
```

Vegyük észre, hogy ezek az új adatok a @emberek tömbön át is elérhetőek:

```
print "Az elso ember elso gyerekenek neme $emberek[0]{gyerekek}[0]{nem}\n"
```

Írjunk egy eljárást, ami a szülőt hozzárendeli a gyerekhez:

```
sub szulok{
    my $ember = shift;
    my $szulo = $ember->{nem} eq "ferfi" ? "apa" : "anya";
    foreach $gyerek (@{$ember->{gyerekek}}){
        $gyerek->{$szulo} = $ember;
    }
```

Ezzel az eljárással az ember-gyerek kapcsolatból kiszámíthatjuk az ember-szulo kapcsolatot:

```
foreach $ember (@emberek){&szulok($ember)};
print "Abel apja  $emberek{Abel}{apa}{nev}\n";
print "Kain anyja $emberek{Kain}{anya}{nev}\n";
```

15.10. Feladatok

Irjon egy függvényt, ami 1-t ad vissza, ha az argumentumban referenciaként átadott két numerikus tömb elemenként egyezik!

Olvasson be egy 2 dimenziós numerikus tömböt egy fájlból, majd írja ki a transzponáltját, azaz az eredeti oszlopokból sorok, a sorokból oszlopok lesznek!

Irjon egy eljárást, ami az ismert gyerek-szülő kapcsolatokból kiszámítja a testvér, féltestvér kapcsolatokat. A testvéreket egy tömbre mutató referenciában tároljuk el!

16. Külső és belső programok, modulok

16.1. Külső programok végrehajtása: `, qx, system, exec

A `PARANCS`, illetve a qx(PARANCS) utasítások végrehajtják az operációs rendszerben (UNIX-ban az sh shellben) a PARANCS stringben található utasítást. A PARANCS tartalmazhat változókat, ezeket a Perl ugyanúgy helyettesíti, mintha dupla idézöjelben állna a PARANCS. Ha hiba történt, akkor a $? speciális változóban egy 0-tól különböző szám lesz. Például a "ls -l" perl-n keresztül:

```
#!/usr/bin/perl
while($file = shift){
    $ls = qx(ls -l $file);
    if($?){
        print "$file could not be listed, error code = $?\n";
    }else{
        print $ls;
    }
}
```

Ha nem létező fájlnevet adunk meg, akkor kapunk egy hibaüzenetet a shell-ből és a programból is. A shell hibaüzenetét a

```
$ls = qx(ls -l $file 2>&1)
```

utasítással lehet átirányítani az STDOUT-ra, amit a `$ls` változó kap meg.

A `system PARANCSLISTA` függvény végrehajtja a PARANCSLISTA által megadott utasítást, majd a végrehajtás után visszatér a Perl programba és folytatja azt. A függvény eredménye a hiba kód, ami sikeres végrehajtás esetén 0 (azaz nincs hiba). Ha a PARANCSLISTA egyetlen elemből áll, akkor azt a /bin/sh shell hajtja végre, ha több eleme van, akkor viszont a shell megkerülésével (lásd man execvp) hajtódik végre az első elemben szereplő parancs a további elemekben szereplő argumentumokkal. Ilyenkor nincs shell behelyettesítés a $, *, vagy ? karakterekre. Hiba esetén a $! tartalmazza a hibaüzenetet. Például:

```
$error = system "ls -l *.pl";       # lists perl codes on STDOUT
print "error = $error, msg = $!\n"; # error = 0, msg =
$error = system "ls", "-l", "*.pl"; # prints on STDERR:
                                    #   ls: *.pl: No such file or directory
print "error = $error, msg = $!\n"; # error = 256, msg =
$error = system "ls-l*.pl";         # prints on STDERR:
                                    #   sh: ls-l*.pl: command not found
print "error = $error, msg = $!\n"; # error = 32512, msg =
$error = system "ls-l", "*.pl";     # prints nothing
print "error = $error, msg = $!\n"; # error = -1,
                                    #   msg = No such file or directory
```

Az `exec PARANCSLISTA` pontosan ugyanazt csinálja, mint a `system` függvény, csakhogy normális végrehajtás esetén nem tér vissza a program végrehajtás. Ha valamiért a PARANCSLISTA nem végrehajtható, akkor viszont egy hibakódot ad vissza.

16.2. Perl programok végrehajtása: `eval, require`

Az `eval STRING` illetve `eval BLOCK` függvények végrehajtják a STRING illetve BLOCK által megadott programot. Ha paraméter nélkül hívjuk meg, akkor a $_ hajtódik végre. A függvény eredménye az utolsó utasítás értéke, illetve egy esetleges `return` utasítással visszaadott érték, hasonlóan mint a szubrutinoknál. Hiba esetén egy üres stringet kapunk vissza, és a $@ változóban kapunk egy hibaüzenetet.

Az `eval` függvény segítségével kihasználhatjuk, hogy a Perl egy interpreter,

hiszen a programot az adatokból állíthatjuk elő. Íme egy egyszerű számológép ami még a hibákat is kezeli:

```
perl -pe '$_ = eval()."\n"; $_ = $@ if $@'
2+3
5
sin(3)
0.141120008059867
asin(3)
Undefined subroutine &main::asin called at (eval 3) line 1, <> line 3.
$x = 3+4
7
2*$x
14
```

A `require FILENAME` függvény végrehajtja a FILENAME stringben található Perl programot, amennyiben az még nem volt beolvasva. Lényegében úgy viselkedik, mint egy `include` utasítás a C vagy Fortran nyelvekben. A fájlt a `@INC` (include directories) speciális változóban tárolt könyvtárakban keresi. Ez alapértelmezésben is rengeteg könyvtárat tartalmaz, ami továbbbővíthető expliciten, illetve a `use libs KONYVTARLISTA` direktívával. A már beolvasott fájlok a `%INC` speciális változóba kerülnek be. A `require` eredménye a fájlban végrehajtott utolsó utasítás értéke, aminek igaznak kell lennie, különben a `require` megáll egy hibaüzenettel. Ezért a beolvasott fájlok utolsó sorába egy `1;` utasítást szokás írni.

A `require` használata lehetővé teszi, hogy egy nagyobb programot több fájlban tároljunk. Például a főprogram lehet a következő:

```
# Main program
use libs "lib";
require "reader.pl";
require "sorter.pl";
require "writer.pl";

my @values = &reader;
&sorter(@values);
&writer(@values);
```

és a `reader.pl` fájl tartalmazhatja a következőt:

```
# reader
sub reader{
  open IN "data.dat";
```

```
  @values = <IN>;
}
1;
```

16.3. Csomagok: package

Mindezidáig a globális változók egyetlen névtérhez (name space) tartoztak. Csak a `my` deklaráció állt rendelkezésünkre ahhoz, hogy egy változó használáti terét (scope) leszűkítsük egy blokkra. A csomagok (package) mintegy előnevet rendelnek hozzá a változókhoz. A szokásos változók a `main` csomag részei. A csomag változóira a csomag nevét követő dupla kettősponttal majd a változó nevével lehet hivatkozni:

```
$a = 1;
print "$a == $main::a \n";
```

A `package` utasítás segítségével egy másik csomagra válthatunk:

```
package CSOMAG;
$a=2;
print "main::a = $main::a, a = $a = $CSOMAG::a\n"; # main::a = 1, a = 2 = 2
```

A csomagnak akkor van vége, ha a blokknak vége van, vagy ha egy másik csomagot deklarálunk. Ez (ellentétben a `my` deklarációval), nem jelenti azt, hogy a csomagban lévő változók elvesztek, csupáncsak már nem lehet rájuk a csomag nevének megadása nélkül hivatkozni. Például

```
$a=1;
LOCAL:{
    package LOCAL;
    $a=3;
    print "a = $a = $LOCAL::a\n";
}
print "a = $a, LOCAL::a = $LOCAL::a\n";
```

A csomagok egymásba is ágyazhatóak. Ehhez expliciten meg kell adni a csomag nevében, hogy hova tartozik:

```
$a = 1;
package KULSO;
$a = 2;
```

```
package KULSO::BELSO;
$ a = 3;
print "BELSO::a=$BELSO::a\n"; # undefined !!!
package main;
print "a = $a, KULSO=$KULSO::a, BELSO=$KULSO::BELSO::a\n"; # 1, 2, 3
```

Megjegyzés: csak a normál változókra vonatkozik a csomag deklaráció, a speciális változókra, mint $_, $?, ..., @INC, %INC, STDOUT, STRDERR stb. nem, ezek mindig a `main` csomaghoz tartoznak. A `my` változók pedig semmilyen csomagban sincsenek benne, azok a blokkon belül léteznek csupán.

16.4. Modulok: `use, no`

A modul egy csomagot tartalmazó fájl, aminek a neve megegyezik a csomag nevével kiegészítve a `.pm` végződéssel. Például a Cwd modul a Cwd.pm fájlban nagyjából a következőképpen néz ki:

```
package Cwd;
require Exporter;
@ISA = qw(Exporter);                          # inherit methods from Exporter
@EXPORT = qw(cwd getcwd fastcwd fastgetcwd);
@EXPORT_OK = qw(chdir abs_path fast_abs_path realpath fast_realpath);
....
sub getcwd {
...
}
...
1;
```

Az első sor a csomagot deklarálja. Az Exporter modul beolvasása után a benne deklarált eljárásokat elérhetővé tesszük az `@ISA` tömb megadásával. Az Exporter modul `import` eljárása (amit a `use Cwd` hív meg), lehetővé teszi, hogy a Cwd csomagban deklarált eljárások egy részét exportáljuk, azaz ezek a `Cwd::` nélkül is elérhetőek legyenek. A `@EXPORT` tömbben felsorolt eljárások és változók közvetlenül elérhetők lesznek, ha a Cwd modult a

```
use Cwd;
```

utasítással beolvassuk. A `use` utasítas esetében a modul beolvasása, ellentétben a require utasítással, a fordításkor történik meg.

A `use MODUL LISTA` utasítás tehát beolvassa a MODUL nevű modult, és importálja a LISTA-ban felsorolt változókát és eljárásokat. A LISTA csak olyan

elemeket tartalmazhat, amit az `@EPORT` és `@EXPORT_OK` tömbök valamelyike tartalmaz, például

```
use Cwd qw(getcwd chdir abs_path)
```

Ha a lista hiányzik, akkor a `@EXPORT` tömb elemeit importáljuk. Ha nem akarunk semmit sem importálni, akkor a

```
use Cwd ();
```

utasítást használhatjuk. Ilyenkor az eljárások a `Cwd::getcwd` formában érhetőek csak el.

Ha a csomag nevében :: van, azaz egy csomag része, akkor a megfelelő alkönyvtárban keresendő. Például a `Data::Dumper` csomag a `Data/Dumper.pm` fájlban található.

A `no MODUL` közvetlenül el nem érhetővé teszi a MODUL modulból importált változókat és eljárásokat.

16.5. Fordító direktívák: `use`, `no`

Az előző részekben már többször említettük egyes a fordítóknak adható direktívákat (a Perl ezeket `pragma`-nak nevezi). A direktívát a `use DIREKTIVA PARAMETEREK` utasítással kapcsoljuk be, és a `no DIREKTIVA PARAMETEREK` utasítással kapcsolhatjuk ki. A PARAMETEREK jelentése a DIREKTIVA-tól függ. Néhány fontosabb direktíva.

Az `integer` direktíva hatására a bezáró blokk végéig a egész szám aritmetikát használ. Ez gyorsabb, mint az alapértelmezésben használt valós. Például:

```
use integer;
print 5/3;      # = 1
no integer;
print 5/3;      # = 1.66666666666667
```

A `strict PARAMETER` direktíva bizonyos deklarációk betartását teszi kötelezővé a bezáró blokk végéig. Így elkerülhető, hogy a gépelési hibák nehezen észrevehető és javítható program hibákká váljanak. A PARAMETER lehetséges értékei:

- 'vars', azaz a változókat kell deklarálni

- 'subs', az eljárások meghívásánál ki kell írni az & jelet
- 'refs', azaz a szimbolikus referenciák nem használhatóak

Ha nincs paraméter, akkor mindhárom megkötés érvényes. Hosszabb programok írásánál a `use strict` használata feltétlenül ajánlott. Ahol szükséges kivételt tenni, ott ez többféleképpen is megtehető.

A vars VALTOZOLISTA direktiva lehetővé teszi, hogy felsorolunk bizonyos változókat, amiket nem kell deklarálni a `strict 'vars'` ellenére sem. Például egy program kaphat változókat a parancs sorról a `-s` kapcsoló segítségével. Ha ezeket a változókat expliciten felsoroljuk, akkor számos potenciális programhibát el lehet kerülni:

```
#!/usr/bin/perl -s
use strict;
use vars qw($h $q $min);
print "Switches -h=$h -q=$q -min=$min";
my $Help = $h;
my $Quiet = $q;
...
```

A `subs ELJARASNEVLISTA` direktívával egyes eljárások deklarálhatók.

A `libs KONYVTARLISTA` a `@INC` változót manipulálja, azaz a felhasznált modulok kereséséhez adhatóak meg új könyvtárak (vagy törölhetök a `no libs KONYVTAR` utasítással).

16.6. Feladatok

Írjon egy programot, ami kitalálja, hogy milyen műveleti jeleket kell írni 3 megadott szám közé, hogy a negyedik számot kapjuk eredményül.

Tanulja meg a Data::Dumper modul használatát, és írjon ki egy bonyolult adatszerkezetet egy fájlba, majd olvassa be. Tipp: írja be, hogy `locate Data/Dumper.pm`.

17. Ami kimaradt

Az előadáson idő hiányában, valamint az érthetőség kedvéért sok mindenről nem volt szó, vagy csak részlegesen került sorra. Íme egy lista néhány kimaradt témáról:

- Általános típus (type glob, *valtozo), mivel a használata szinte mindig elkerülhető: `man perldata`
- Biztonságos programozás: `man perlsec`
- Egyes speciális változók: `man perlvar`
- Egyes függvények: `man perlfunc`
- Gyakran használt modulok (pl. CGI, HTML): `http://www.perl.com/CPAN`
- Kapcsolat más nyelvekkel: `man perlembed`
- Minta illesztés egyes részei: `man perlre`
- Műveletek fölülírása: `overload.pm`
- Objektum orientált programozás: `man perlboot, perltoot ...`
- Perl belső dokumentáció: `man perlpod`
- Perl debugger: `man perldebug`
- Prototype a szubrutinok számára: `man perlsub`
- és még sok minden más: `man perl`

A Perl nyelv folyamatosan fejlődik. Új függvények jelennek meg, a régi függvények új felhasználása, és főként új modulok sokasága. Ezekről a fejleményekről a Perl honlapokon lehet tájékozódni.

18. Függvények listája

Ebben a fejezetben téma szerint, azon belül pedig abc rendben találhatóak a Perl függvények és annak a fejezetnek a száma, ahol a függvény leírása megtalálható, illetve egy rövid magyarázat. A `man perlfunc` segítségével bőséges információ kapható valamennyi függvényről.

18.1. Matematikai függvények

`abs`	abszolút érték
`atan2`	x,y koordinátához tartozó polárszög
`cos`	koszinusz (az argumentum radiánban)
`exp`	exponenciális
`int`	egész rész a 0 felé kerekítve
`log`	természetes logaritmus
`rand`	véletlen szám 0 és EXPR (vagy 1) között
`sin`	szinusz (az argumentum radiánban)
`sqrt`	négyzetgyök
`srand`	a `rand` kezdőértékét teszi véletlenszerűvé

18.2. String manipuláló függvények

`m//`	lásd a 8.2 részt a 34. oldalon
`s///`	lásd a 8.12 részt a 39. oldalon
`chomp`	lásd az 5.4 részt a 22. oldalon
`chop`	lásd az 5.4 részt a 22. oldalon
`chr`	lásd a 11.2 részt az 50. oldalon
`crypt`	lásd a 11.2 részt az 50. oldalon
`format`	lásd a 11.3 részt az 50. oldalon
`formline`	a format belső eljárása
`grep`	lásd a 9.2 részt a 43. oldalon
`hex`	lásd a 11.2 részt az 50. oldalon
`index`	lásd a 8.1 részt a 33. oldalon
`join`	lásd a 9.1 részt a 42. oldalon
`lc`	lásd a 8.15 részt a 40. oldalon
`lcfirst`	lásd a 8.15 részt a 40. oldalon
`length`	lásd a 8.17 részt a 42. oldalon
`oct`	lásd a 11.2 részt az 50. oldalon
`ord`	lásd a 11.2 részt az 50. oldalon
`pos`	lásd a 8.7 részt a 37. oldalon
`q//`	lásd a 8.16 részt a 41. oldalon
`qq//`	lásd a 8.16 részt a 41. oldalon
`qx//`	lásd a 8.16 és a 16.1 részeket a 41. és a 65. oldalakon
`qw//`	lásd a 8.16 részt a 41. oldalon
`quotemeta`	lásd a 8.15 részt a 40. oldalon
`reverse`	lásd a 9.3 részt a 44. oldalon
`rindex`	lásd a 8.1 részt a 33. oldalon
`split`	lásd a 9.1 részt a 42. oldalon
`sprintf`	lásd a 11.1 részt a 49. oldalon
`study`	optimalizálja a minta illesztést
`substr`	lásd a 8.17 részt a 42. oldalon
`tr///`	lásd a 8.14 részt a 40. oldalon
`uc`	lásd a 8.15 részt a 40. oldalon
`ucfirst`	lásd a 8.15 részt a 40. oldalon
`vec`	lásd a 12.1 részt az 51. oldalon
`y///`	ugyanaz mint a tr/// függvény
`write`	lásd a 11.3 részt az 50. oldalon

18.3. Kontrol utasítások

`die`	lásd a 7.2 részt a 30. oldalon
`do`	lásd a 6.5 részt a 28. oldalon
`dump`	egy bináris fájlt készít, amiből újra lehet indítani
`each`	lásd a 10.3 részt a 48. oldalon
`eval`	lásd a 16.2 részt a 66. oldalon
`exec`	lásd a 16.1 részt a 65. oldalon
`exit`	lásd a 7.2 részt a 30. oldalon
`fork`	a programot két példányban futtatja
`goto`	ugrás valahova (használata nem ajánlott)
`last`	lásd a 6.3 részt a 27. oldalon
`map`	lásd a 9.2 részt a 43. oldalon
`next`	lásd a 6.3 részt a 27. oldalon
`redo`	lásd a 6.3 részt a 27. oldalon
`return`	lásd a 7.3 részt a 31. oldalon
`system`	lásd a 16.1 részt a 65. oldalon
`wait`	vár egy child process végére
`waitpid`	egy konkrét child process végére vár

18.4. Fájl és könyvtár kezelö függvények

`binmode`	egy fájlkezelőt binárisként kezel (Unixban nem kell)
`chdir`	lásd a 13.5 részt az 56. oldalon
`chmod`	lásd a 13.4 részt az 55. oldalon
`chown`	lásd a 13.4 részt az 55. oldalon
`chroot`	megváltoztatja a szuper felhasználót
`close`	lásd az 5.4 részt a 22. oldalon
`closedir`	lásd a 13.2 részt az 54. oldalon
`eof`	lásd a 13.1 részt az 53. oldalon
`fcntl`	file control (UNIX)
`fileno`	file number (UNIX)
`flock`	file lock (UNIX)
`getc`	lásd a 13.1 részt az 53. oldalon
`glob`	lásd a 13.2 részt az 54. oldalon
`ioctl`	I/O control (UNIX)
`link`	lásd a 13.4 részt az 55. oldalon
`lstat`	lásd a 13.4 részt az 55. oldalon
`mkdir`	lásd a 13.5 részt az 56. oldalon
`open`	lásd az 5.4 részt a 22. oldalon
`opendir`	lásd a 13.2 részt az 54. oldalon
`pipe`	összeköt egy olvasó és egy író fájlkezelőt
`print`	lásd az 5.4 részt a 22. oldalon
`printf`	lásd a 11.1 részt a 49. oldalon
`read`	lásd a 13.1 részt az 53. oldalon
`readdir`	lásd a 13.2 részt az 54. oldalon
`readlink`	lásd a 13.4 részt az 55. oldalon
`rename`	lásd a 13.4 részt az 55. oldalon
`rewinddir`	a könyvtárkezelőt a kezdő pozicióra állítja
`rmdir`	lásd a 13.5 részt az 56. oldalon
`seek`	lásd a 13.1 részt az 53. oldalon
`seekdir`	a könyvtárkezelőt adott pozicióra állítja
`select`	az STDOUT helyett más fájl kezelőt választ írásra
`stat`	lásd a 13.4 részt az 55. oldalon
`symlink`	lásd a 13.4 részt az 55. oldalon
`tell`	lásd a 13.1 részt az 53. oldalon
`telldir`	a könyvtárkezelő pozicióját adja meg
`truncate`	lerövidít egy fájlt
`umask`	kiolvassa/beállítja a létrehozandó fájlok hozzáférését
`unlink`	lásd a 13.4 részt az 55. oldalon
`warn`	hibaüzenetet ír az STDERR fájlkezelőre
`utime`	lásd a 13.4 részt az 55. oldalon

18.5. Tömb kezelő függvények

`pack`	lásd a 12.1 részt az 51. oldalon
`pop`	lásd a 9.6 részt a 45. oldalon
`push`	lásd a 9.6 részt a 45. oldalon
`reverse`	lásd a 9.3 részt a 44. oldalon
`shift`	lásd a 9.6 részt a 45. oldalon
`sort`	lásd a 9.3 részt a 44. oldalon
`splice`	lásd a 9.6 részt a 45. oldalon
`unpack`	lásd a 12.1 részt az 51. oldalon
`unshift`	lásd a 9.6 részt a 45. oldalon

18.6. Hash és adatbázis kezelő függvények

`delete`	lásd a 10.2 részt a 47. oldalon
`each`	lásd a 10.3 részt a 48. oldalon
`exists`	lásd a 10.2 részt a 47. oldalon
`keys`	lásd a 10.1 részt a 47. oldalon
`values`	lásd a 10.1 részt a 47. oldalon
`dbmclose`	lásd a 10.3 részt a 48. oldalon
`dbmopen`	lásd a 10.3 részt a 48. oldalon

18.7. Változókat kezelő eljárások

`my`	lásd a 7.4 részt a 33. oldalon
`defined`	igazat ad vissza, ha a változó definiált
`local`	mint a `my`, de meghívott eljárásokra is kiterjed
`ref`	lásd a 15.7 részt a 61. oldalon
`reset`	?...? keresést újra kezdi, változókat töröl
`scalar`	lásd a 3.4 részt a 14. oldalon
`undef`	egy változót töröl (definiálatlanná tesz)

18.8. Csomagok, direktívák, objektum orientált

`bless`	egy referenciát hozzárendel egy osztályhoz
`import`	lásd a 16.4 részt a 69. oldalon
`new`	egy objektumot létrehozó eljárás
`no`	lásd a 16.4 és a 16.5 részeket a 69. és a 70. oldalakon
`package`	lásd a 16.3 részt a 68. oldalon
`require`	lásd a 16.2 részt a 66. oldalon
`sub`	lásd a 7.3 részt a 31. oldalon
`tie`	egy változót egy osztályhoz köt (dbmopen általánosítása)
`tied`	megadja az osztályt, amihez a változó kötődik
`untie`	megszünteti az osztályhoz rendelést (dbmclose általánosítása)
`use`	lásd a 16.4 és a 16.5 részeket a 69. és a 70. oldalakon

18.9. Idővel kapcsolatos függvények

`alarm`	SIGALARM jelet küld adott másodperccel később
`gmtime`	lásd a 14 részt az 57. oldalon
`localtime`	lásd a 14 részt az 57. oldalon
`sleep`	lásd a 14 részt az 57. oldalon
`time`	lásd a 14 részt az 57. oldalon
`times`	lásd a 14 részt az 57. oldalon

18.10. Egyéb függvények

Információ a meghívó eljárásról:

```
caller
wantarray
```

Hálózat kezelés:

```
accept
bind
connect
listen
recv
send
shutdown
socket
socketpair
```

Operációs rendszerrel kapcsolatos függvények:

```
get*
kill
msg*
sem*
set*
shm*
syscall
sysopen
sysread
syswrite
```