Abonyi-Tóth Andor • Holler János • Rozgonyi-Borus Ferenc

# Képzeld el!

## algoritmusok, játékok

Imagine

az általános iskolától
az emelt szintű érettségiig

ABAX

# Imagine

## Algoritmusok és játékok

ű

Abonyi-Tóth Andor . Holler János . Rozgonyi-Borus Ferenc

# Imagine

## Algoritmusok
## és
## játékok

haladók számára

az általános iskolától
az emelt szintű érettségiig

ABAX
2007

**Szerző:**
***Rozgonyi-Borus Ferenc***
*szakvezető tanár*

**Lektor:**
***Abonyi-Tóth Andor***
*egyetemi tanársegéd*
***Holler János***
*vezető tanár*

A könyv segédanyagai a
http://www.abax.hu/imagine
címen érhetők el

Minden jog fenntartva, beleértve a sokszorosítás, a mű bővített, illetve rövidített változata kiadásának jogát is. A kiadó írásbeli hozzájárulása nélkül sem a teljes mű, sem annak része semmiféle formában nem sokszorosítható.

*© ABAX BT - SZEGED, 2007*

# Rövid tartalomjegyzék


Bevezető ..... ?
Programkészítés alapjai ..... 9
Változó létrehozása, értékadás ..... 9
Vezérlési szerkezetek ..... 17
Matematikai algoritmusok ..... 31
Játékok készítése ..... 35
Funkcionális LOGO alapjai ..... 61
Programozási tételek ..... 75
Elemi programozási tételek ..... 75
Programozási tételek alkalmazása ..... 79
Halmazműveletek ..... 85
Alapvető rendezések ..... 89
Nevezetes feladatok ..... 93
Rekurzió alkalmazása ..... 93
Visszalépéses algoritmus ..... 98
Mohó algoritmus ..... 101
Programozási hibák ..... 103
Emelt szintű érettségi Imagine környezetben ..... 109
2005. május 19. Lottó ..... 109
2005. október 27. Vigenére tábla ..... 114
2006. február 28. Telefonszámla ..... 118
2006. május 17. Fehérje ..... 122
2006. november 3. Zenei adók ..... 127
Felhasznált és ajánlott irodalom ..... 133
Források a weben ..... 133
Ajánlott irodalom LOGO-hoz ..... 134
Tárgymutató ..... 135
Tártai omj egyzék ..... 139

## *Kedves Olvasó!*

Az Imagine programozási környezetet bemutató sorozatunk haladóknak szóló kötetét tartja a kezében.

Több ország informatika oktatása történetében is példa nélküli, hogy a közoktatásban résztvevők számára egy olyan programozási környezetet bocsátanak ingyen(!) rendelkezésre, amely a legkorszerűbb programozástechnikai elemeket is alkalmazza.

Az Imagine 2006 márciusától a Sulinet honlapjáról, a http://logo.sulinet.hu címről, ingyenesen letölthető az Oktatási Minisztérium döntésének jóvoltából a közoktatásban résztvevő összes személy számára, legyen akár diák vagy tanár.

A LOGO eredetileg oktatási célra, főleg szövegfeldolgozás céljára kidolgozott nyelvként született meg. Ehhez adta Seymour Papert azt a pedagógiai célt, amely révén a nyelv alkalmassá vált kisgyermekek programozás oktatására is. Megszületett Irving, a padlóteknőc, ezzel a teknőcgrafikai utasítások készlete, majd később a képernyőteknőc grafikai lehetőségei.

A korai mikrogépekre készült LOGO környezetek, megvalósítások inkább a grafikai bővítmények kezelésére szolgáltak. Az alapot adó LISP eredeti, funkcionális részei erősen háttérbe szorultak, így az oktatásuk is.

Az Imagine-t 2001-ben jelentette meg Peter Tomcsányi, Ivan Kalas, Lubomir Salanci és Andrej Blaho. Elődje, a Windows 3.1 alá készített, saját korában ugyancsak zseniálisnak számító Comenius LOGO volt. A közel két évtizedes fejlesztőmunkának köszönhetően a Windows XP-re készített, és az időközben igencsak megváltozott, kiteljesedett multimédia környezetre is felkészítették az Imagine rendszert.

Az Imagine első magyar nyelvű változata 2005-re jelent meg az ELTE TEAM Labornak köszönhetően, Abonyi-Tóth Andor, Turcsányi-Szabó Márta és Windisch József munkájának eredményeként.

Az Imagine a ComLOGO-hoz képest nem csak egyszerűen operációs rendszert és multimédiás környezetet váltott. Magába integrálta a paperti elvek szem előtt tartása mellett a programozási nyelvek elmúlt 30 év alatti fejlesztéseit, és eleget tett a felhasználók azon igényének, hogy ne csak értelmezővel, hanem

fordítóval is rendelkezzen, valamint egyszerű lehetőség nyíljon a weben való szinte azonnali megjelentetésre is.

A fejlesztés nem állt le, és remélhetőleg nem is fog leállni. A program mindenkori legújabb demóverziója és sok más hasznos információ megtalálható az Imagine kiadójának, a Logotron cég honlapjának címén: http://www.logo.com.

Külön köszönetet is kell mondanunk. Ez a három kötetes tankönyvsorozat és a munkafüzetek nem jöttek volna létre, ha Dr. Zsakó László docens, az ELTE Média- és Oktatásinformatikai tanszékének oktatója nem küzdene évtizedek óta elismerten a programozás oktatásának hazai műveléséért, és nem fejtené ki versenyszervező munkáját ismert és névtelen segítőinek tömegével.

## A szóhasználatról

A könyvben a *parancs* megnevezést általánosan használjuk. Minden esetben ezt írjuk, ha nem akarjuk konkrétan megnevezni, hogy milyen funkciójú az adott elem vagy az elkészített programmodul.

Ha az *eljárás* egy konkrét értéket ad vissza, azaz szerepel benne például az **eredmény** utasítás, akkor *függvény*-ről beszélünk. Ugyancsak függvénynek tekintjük a matematikai értelemben vett függvényeket is, mint például az egész rész kiszámítását. *Művelet* alatt a matematikában megszokott fogalmat értjük, de művelet lesz az infix írást használók mellett az eltérő írású művelet is, mint például a **:a mod :b** helyett az Imagine-ben használatos **maradék :a :b** is.

# Programkészítés alapjai

A ComLOGO-ból jól ismert, és szinte mindenki által is használt **ismétlés** és **ha** utasítás mellett az Imagine számos más vezérlési szerkezetet is tartalmaz.

## *Változó létrehozása, értékadás*

Az Imagine már tartalmazza a globális és lokális változó használatának lehetőségét.

### *Globális változók*

*A* **globálisváltozó** - vagy röviden **globvál** - paranccsal globális változó hozható létre, illetve a ComLOGO-ban már létező, de itt eltérő paraméter sorrendű **név** paranccsal is. Az így létrehozott változók a program minden részében elérhetőek lesznek, de természetesen ha azonos nevű lokális változó is van egy adott eljáráson belül, akkor az általános szabályt kell alkalmaznunk. Erre később az eljárásoknál is látunk példát.

```
? név 10 "maci
? globvál "ubul 8
? mutat :maci
10
? mutat :ubul
8
```

### *Lokális változók*

*A* lokális változó létrehozására szolgál a **lokálisváltozó** parancs - röviden **lokvál.** Ez csak magát a változót hozza létre, de értéket még nem ad neki.

```
? lokálisváltozó "ubul
? lokálisváltozó [erre arra]
? mutat :ubul
A(z) ubul változónak nincs értéke
 Használd a NÉV parancsot a változó értékének megadásához!
```

A változó neve előtt " idézőjel szerepel, a : kettőspont pedig az értékének használatakor. Ez azért is fontos, mert az ***"ubul*** név vagy cím szerinti változó meghívást jelent, ahol a változó értéke módosulhat, míg az ***:ubul*** érték szerinti meghívás, a változó értékét kiolvassuk, de nem módosítható.

Értéket adni a már létrehozott változónak a **név** paranccsal is lehet.

```
? név 8 "ubul
? mutat :ubul
8
```

Figyeljünk arra, hogy a **név** paranccsal az értékadás használata nem a megszokott sorrendű, azaz elsőként az érték, utána a változó szerepel.

Ha fordítva adnánk meg a sorrendet, akkor érdekes dolog alakul ki:

```
? név "ubul 8
? mutat :ubul

A(z) ubul változónak nincs értéke
 Használd a NÉV parancsot a változó értékének megadásához!
```

Létrehoztunk viszont egy új, globális változót 8 néven, amit ellenőrizhetünk is. Az alábbi második parancs mutatja a különbséget is.

```
? mutat :8
ubul
? mutat "8
8
```

A lokális változók használata esetén célszerűbb azonnal az értékadást is lehetővé tevő **lokálisérték** - röviden **lókért** - parancsot használni.

```
? lokálisérték "maci 4
? mutat :maci
4
```

A lokális csak az adott eljárásban, míg a globális változó mindenhol látható.

A parancsszóval és az objektumokon belül létrehozott változók között különbség van, erre az objektumok tárgyalása során térünk ki a 3. kötetben.

A létrehozott változókkal végezhető műveletek közül a **növel** igen hasznos. Ez alapértelmezetten 1-gyel való növelés, de megadhatunk növekményt is.

```
mutat :maci
10
? növel "maci
? mutat :maci
11
? növel "maci 2

Nem tudom mit csináljak a(z) 2-val
 Nem mondtad meg, mit csináljak az eredménnyel.
```

A hiba oka, hogy alapértelmezetten csak a változó a **növel** bemenete, ezért a hiba a **növel** feldolgozása után lép fel, maga az értéknövelés megtörténik!

```
mutat :maci
12
```

A növekmény megadásánál ki kell jelölni a hatókört.

```
? (növel "maci 2)
? mutat :maci
14
? (növel "maci -5)
? mutat :maci
```

Teljesen hasonlóan használható a **csökkent** parancs is.

## Adat be- és kivitel

### *Kiírás*

A változók írólapon való megjelenítésére már használtuk a **mutat** parancsot. Ha több elemből áll, amit ki akarunk íratni, akkor hatókört kell kijelölni a zárójelek használatával.

```
? (mutat "|maci értéke:| :maci "|, ubul értéke:| :ubul)
maci értéke: 12 , ubul értéke: 8
? mutat "Lajos mutat [a b c]
Laj os
 [a b c]
```

Vegyük észre, hogy a **mutat** parancs a kiírt elemek közé automatikusan szóközt helyez el, ami felhasználói szempontból néha felesleges. A kiírás után új sor elejére teszi a kurzort, valamint a listát listaként jeleníti meg [] jelek között.

A ComLOGO-ban is létezett már az általánosabban használható **kiír** - röviden **ki** - parancs. Ez is átirányítható megjelenítést tesz lehetővé, azaz nem csak az írólapon, hanem a megadott kiíró eszközön is tud adatot megjeleníteni, viszont a listát szögletes zárójel nélkül adja vissza, különben használata megegyezik a **mutat** paranccsal.

```
? (kiír "|maci értéke:| :maci "|, ubul értéke:| :ubul)
maci értéke: 12 , ubul értéke: 8
? kiír "Lajos kiír [a b c]
Lajos
a b c
```

A **kiírérték** parancs a kiírt bemeneteket nem választja el szóközökkel, és a kurzort sem viszi a következő sorba, azaz alkalmas formázottabb kiírásokra, valamint átirányítás esetén is használható.

```
? (kiírérték "|maci értéke:| :maci "|, ubul értéke:| :ubul)
maci értéke:12, ubul értéke:8
kiírérték "Lajos kiírérték [a b c]
Lajosa b c
```

> A speciális feladatokra szolgáló **kiírsor** és **kiírszövegdobozba** parancsokat most nem tárgyaljuk, ezeket a 3. kötetben ismertetjük.

Jó tudnunk, hogy az írólapra kerülő kiírásokat a **törölszöveg** – röviden **törölszöv** – paranccsal tüntethetjük el.

## Szöveg kiírása grafikus képernyőre

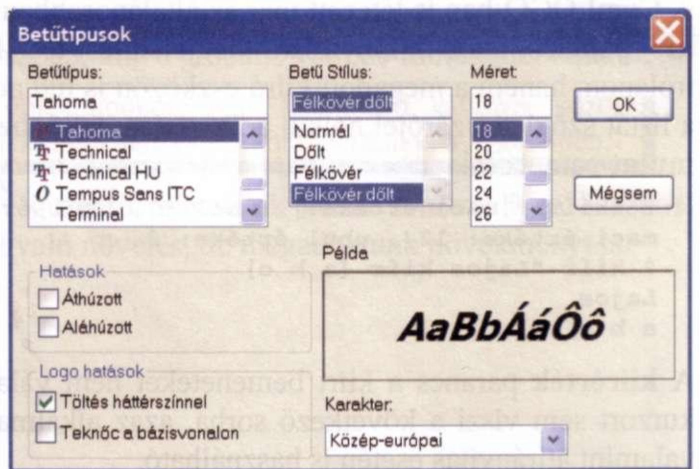
A szöveg kiírásához a rajzlapra két utasításunk is van. Mindkettő tetszőleges irányban teszi lehetővé a szöveg kiírását, ha a betűtípus vektoros leírású.

> Ha például a System betűtípust használjuk, ami az alapértelmezett, akkor is kiírható a szöveg, de nem forgatható el, nem méretezhető szabadon át, stb.

A **címke** parancs kiadása esetén minden aktív teknőc kiírja a bemeneti szöveget a helyén, az aktuális pozíciójától kezdve, aktuális betűtípusával. A szöveg kiírásához a toll aktuális színét használja. Ha a szöveg egy lista, azt a **címke** parancs szögletes zárójelek között írja ki. A szöveget a teknőc irányára merőlegesen, attól jobbra 90°-kal elfordítva írja ki. Éppen ezért, ha a szöveget vízszintesen akarjuk kiíratni, akkor a teknőcnek északra kell néznie.

A szövegben a ¶ – Alt+0182 – vagy \13 bekezdésvég karakter szimbólumok sorvégnek értelmeződnek, így a **címke** parancs több sorban is képes kiírni.

A **címke2** parancs alapvetően azonos működésű, de míg a **címke** parancs esetén a teknőc nem változtatja meg a pozícióját, addig a **címke2** esetén a teknőc a szöveg jobb oldali végére ugrik. Ennek megfelelően a további kiírás ugyanebben a sorban folytatódik, az előző szöveg után.




A betűtípus megadásához – minden aktív teknőchöz vagy csak egy szövegdobozhoz – a **betűtípus!** parancs használható.

Ehhez egy [[*betűtípusnév*] [*méret félkövérség stílus átlátszatlan alapVonal karakter*]] alakú listát kell használnunk, ahol:

- a *méret* pontokban van megadva, például 8, 10 vagy 13 vagy 28... stb.
- a *félkövérség* lehet 400 a szabályoshoz, 700 a félkövérhez.
- a *stílus*

| | |
|---|---|
| 0 ⇨ szokásos | 3 ⇨ aláhúzott és dőlt |
| 1 ⇨ dőlt | 4 ⇨ áthúzott |
| 2 ⇨ aláhúzott | 5 ⇨ áthúzott és dőlt. |

Ha az *átlátszatlan* értéke 1, akkor az egész négyszög, amelybe a **címke** parancsra a szöveg kerül, kitöltődik az aktuális háttérszínnel. Ha az értéke 0, a négyszög átlátszó marad, azaz a korábban odaírt szöveg is látszani fog.

Az *alapVonal* a szöveg helyét határozza meg. Ha 0, akkor a teknőc pozíciója annak a négyszögnek a bal felső sarka lesz, amelybe a szöveg kerül. Ha 1, akkor a teknőc pozíciója a szöveg alapvonalának kezdete. Ez megkönnyíti például a különböző betűtípus-méretekből álló szöveg egy sorba való írását.

A *karakter* egy egész szám 0-tól 255-ig. Ez azonosítja a karakterkészlet alváltozatát, azaz például a magyar ékezetes betűk használatához a 238-as kód kerül ide, mivel ez a közép-európai kódkészlet azonosítója.

Az ablakon beállított betűtípusnak megfelelő parancs hatására az oldalt látható formában jelenik meg a szöveg:

```
j 30
? betűtípus! [[Tahoma][18 700 1 1 0 238]]
? címke "|Az Imagine az igazi!|
```

Az Imagine az igazi!

Vegyük észre, hogy a teknőc nem mozdult el, és az előképen látott kalapok helyett szerencsére helyesen jelentek meg az ékezetek.

1. *Írható-e 9,5 pontos betűméret?*
2. *Miben mérik a félkövérséget?*
3. *Vizsgáljuk meg, ha a kiírandó szöveg egy lista, az miként jelenik meg!*

### *Bevitel*

Az **olvasjel** parancs - röviden **oj** - egyetlen karaktert olvas be a beviteli eszközről, ami nem jelenik meg a képernyőn. Amíg nem történik billentyű-leütés, addig várakozik. Az alábbi feladat mutatja be ennek működését!

1. *Adjuk ki az alábbi, első sorban látható parancsot, majd nyomjunk meg egy billentyűt! Ezután adjuk ki a második,* **mutat** *parancsot is!*

```
név olvasjel "betű
? mutat :betű
j
```

Az **olvasszó** parancs - röviden **osz** - visszatérési értéke egy szó lesz, amely speciális karaktert tartalmazhat, így például szóközt, szögletes zárójeleket is. A bevitel egy kettőspont kezdetű új sorban valósul meg. Az ENTER lenyomásával zárjuk le a bevitelt, a beolvasott szó megjelenik az írólapon is.

```
? név olvasszó "ezt
: ez egy szó most [csak több jelből áll]
? mutat :ezt
ez egy szó most [csak több jelből áll]
```

```
? mutat elemszám :ezt
38
? mutat első :ezt
e
```

Az **olvaslista** parancs – röviden **ol** – visszatérési értéke egy lista lesz, ami speciális karaktert tartalmazhat, így például szóközt is, szögletes zárójeleket is. A bevitel új sorban valósul meg, és egy kettőspont jelzi ezt. Az ENTER lenyomásával történik meg a bevitel lezárása, és a beolvasott lista megjelenik a képernyőn is.

```
? név olvaslista "ezt
: egy lista [pár elemmel]
? mutat :ezt
[egy lista [pár elemmel]]
? mutat elemszám :ezt
3
? mutat első :ezt
egy
```

A működés bemutatására használt **elemszám** és **első** parancsokat nemsokára részletesen is ismertetjük.

## Átirányítás

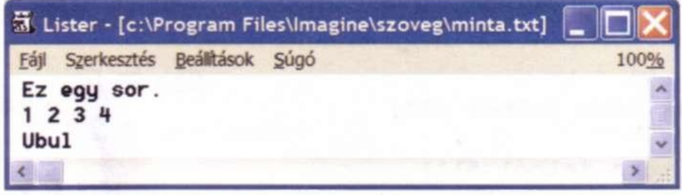
A kiviteli és beviteli eszköz nem csak a képernyő, illetve a standard beviteli eszköz – billentyűzet, egér – lehet, hanem átirányíthatjuk a műveleteket például fájlra vagy nyomtatóra is.

### Bevitel

Az **olvasóeszköz** paranccsal egy fájlból olvashatunk be. A fájl helye alapértelmezés szerint a SZOVEG mappa, de használhatjuk a **betöltőút** és az útvonal megadás lehetőségét is, illetve az **út** paranccsal azt le is kérdezhetjük.

Jó tudnunk, hogy ha kiterjesztés nélkül adtuk meg a fájl nevét, akkor azt az Imagine automatikusan TXT kiegészítéssel keresi.

A fájlból olvasáskor figyelnünk kell arra, hogy legyen még beolvasandó elem, azaz ellenőriznünk kell a tényleges olvasás előtt, hogy nem értük-e el már a fájl végét. Erre szolgál a **fájlvége?** parancs, ami IGAZ értéket ad vissza, ha már a fájl végén vagyunk, azaz kiolvastuk az utolsó sorát, szavát vagy karakterét, különben HAMIS lesz.



*2. Példaként olvassuk be az előző fájlt, amit hozzunk létre előtte MINTA.TXT néven az Imagine SZOVEG mappájában!*

A fájl létrehozása után nézzük az alapértelmezett helyről történő beolvasást a különböző olvasó parancsokkal!

```
? olvasóeszköz "minta
? mutat olvasszó
Ez egy sor.
? mutat olvaslista
[1 2 3 4]
? amíg [nem fájlvége?] [mutat olvasjel]
U
b
u
l
? olvasóeszköz []
```

Az itt használt, de még nem tárgyalt parancsok ismertetése is megtalálható e kötetben.

Az olvasó eszköz lezárására szolgál az **olvasóeszköz** [] parancs, amivel egyben vissza is irányítjuk az olvasást a standard beolvasó eszközökre.

Speciális lehetőség, hogy az olvasó eszköz a Vágólap is lehet, amire természetesen akár más alkalmazásokból is kerülhet tartalom.

E lehetőségre példaként nézzük meg a következő műveletsort!

```
? (kiír "Ezt "tegyük "vágólapra "|a CTRL+C paranccsal!|)
Ezt tegyük vágólapra a CTRL+C paranccsal!
```

A fenti sort jelöljük ki, és hajtsuk is végre a másolás műveletet a CTRL+C használatával, majd adjuk ki a következő parancsot:

```
? olvasóeszköz "vágólap
? mutat olvasszó
Ezt tegyük vágólapra a CTRL+C paranccsal!
```

Ne felejtsük el, hogy az átirányítást most is le kell zárni az **olvasóeszköz []** paranccsal, vagy új olvasó eszközt kell kijelölni.

Ha átirányítás történt, akkor az **olvasjel** a soron következő, egyetlen karaktert dolgozza fel, míg az **olvasszó** és az **olvaslista** a soron következő sort olvassa be, de az első sorként, a második listaként.

### Kivitel

A fájlba írást hasonlóan végezhetjük a **kiíróeszköz** paranccsal. Elsőként meg kell adnunk a kivitelre szolgáló fájl helyét és nevét. Ha nem adjuk meg a kiterjesztést, akkor az automatikusan TXT lesz.

Alapértelmezés szerint a fájl helye a SZÖVEG mappa lesz.

A fájlba folyamatosan történik az írás mindaddig, amíg más helyre nem irányítjuk a kiírást, vagy le nem zárjuk a folyamatot a standard kiíró eszközre irányító **kiíróeszköz** [] paranccsal.

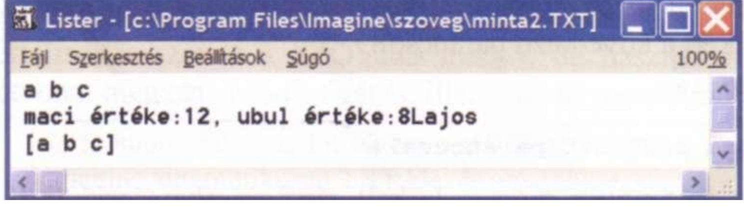
Figyeljünk arra, hogy az előző müveletek mindegyike lezárja a korábbi kimeneti fájlt. A már korábban létező fájlba történő írás, vagy egy újabb ráirányítással kiválasztott fájl is azonosan viselkedik: alapértelmezés szerint a fájl tartalma törlődik, tehát nem egészíti a korábbi tartalmat, hanem azt üres fájlként kezeli, és az fájl elejétől kezdi el a kiírást.

Példaként hozzuk létre a MINTA2 fájlt, amelybe írjunk három sort.

```
? kiíróeszköz "minta2
? kiír [a b c]
? (kiírérték "|maci értéke:| :maci "|, ubul értéke:| :ubul)
? mutat "Lajos mutat [a b c]
? kiíróeszköz []
```

A kapott végeredmény egy MINTA2.TXT fájl a SZÖVEG mappában.

Ugyan nem túl szép, de jól mutatja, hogy minden írás müvelet a megszokott módon működik.




A Vágólapra is átirányítható a kimenet, ami annak tartalmaként jelenik meg. Ezután természetesen az beolvasható más alkalmazások számára is, így elemek átemelhetőek programok között.

## *Vezérlési szerkezetek*

Az Imagine LOGO több és következesebben megadott programvezérlési lehetőséggel rendelkezik, mint azt más LOGO változatban megszokhattuk.

### Szekvencia

A parancsok végrehajtásának alapvető sorrendje a balról jobbra haladás, illetve a fentről lefelé való végrehajtás. A ComLOGO-hoz képest újdonság, hogy egy parancs több sorba is írható és ehhez nem kell ~ jelet írni a sorok végére. Ez az újítás azt is jelenti, hogy a régi programokból ezeket a jeleket ki kell venni.

Az átírási probléma megoldására készítettük el a honlapunkon megtalálható COM2IMG programot.

Mielőtt ismertetnénk az egyes elágazási szerkezeteket, azelőtt röviden nézzük meg a feltételek megadásánál alkalmazható logikai műveleteket!

### *Logikai műveletek*

Tagadás a **nem** paranccsal végezhető, a konjunkcióra az **és,** diszjunkcióra a **vagy** művelet áll rendelkezésre. Ezeken kívül a **kizáróvagy** - röviden **xor** - is használható.

Nézzünk meg néhány egyszerű példát!

```
? mutat nem "hamis
igaz
? mutat (vagy "igaz "hamis)
igaz
? mutat (és "igaz "hamis)
hamis
? mutat (kizáróvagy "igaz "hamis)
igaz
? mutat (kizáróvagy nem "hamis "igaz)
hamis
? mutat (vagy nem "igaz "hamis)
hamis
? mutat (és nem "hamis nem "hamis)
igaz
```

Vegyük észre, hogy az IGAZ logikai értéket az **"igaz** szókonstans, a HAMIS logikai értéket a **"hamis** szókonstanssal tudjuk megadni.

A műveletek természetesen egymásba is ágyazhatóak, és több operandussal is elvégezhetőek, például:

```
? mutat (és "igaz (vagy "igaz nem (és "hamis "igaz)))
igaz
? mutat (kizáróvagy "igaz "igaz "igaz)
igaz
```

4. *Készítsünk eljárást, amely elkészíti az egyes logikai müveletek igazságtábláját!*
5. *Készítsük el a **nemes** és a **nemvagy** műveletnek megfelelő eljárást!*
6. *Helyettesítsük az **és** és a **nem** müvelettel a **vagy** és a **Mzárávagy** müveleteket!*

## Szelekció

A másképpen *elágazási* vagy *kiválasztási* szerkezeteknek nevezett vezérlések egyértelműbbé váltak az Imagine-ben, a már jól ismert **ha** szerkezet egy- és kétágú változata külön kulcsszót kapott, de természetesen megmaradt az **elágazás** szerkezet is.

### Egyágú kiválasztás

A **ha** szerkezet első bemenete egy logikai értéket visszaadó kifejezés, a *feltétel,* amely ha IGAZ lesz, akkor a második bemenet utasításait végrehajtja, ha a *feltétel* HAMIS, akkor nincs hatása a műveletnek.

```
ha feltétel [utasításlista]
```

Nézzünk két egyszerű példát:

```
? ha 5>4 [mutat "helyes]
helyes
? ha 5=4 [mutat "helyes]
```

Utóbbi esetben nincs kiírás! Figyeljünk arra, hogy a ComLOGO-val ellentétben itt nem írható különben ág.

### Kétágú kiválasztás

*A* **hakülönben** - röviden **hak** - szerkezet első bemenete egy logikai értéket visszaadó kifejezés, a *feltétel,* ami ha IGAZ, akkor a második bemenetének megadott utasításlistát hajtja végre, ha HAMIS, akkor a harmadik bemeneteként megadott utasításlistát. Fontos, hogy mindkét ágnak szerepelnie kell!

```
hakülönben feltétel
   [utasításlista igaz esetén]
   [utasításlista hamis esetén]
```

Nézzünk két egyszerű példát:

```
? mutat hak 5>4 ["helyes]["helytelen]
helyes
? mutat hak 5=4 ["helyes]["helytelen]
helytelen
```

Lehetőségünk van arra is, hogy az elágazási feltételt külön értékeljük ki, majd adjuk meg, hogy mi történjen ha igaz, vagy ha hamis. Ezt a megoldást teszi

lehetővé a **teszt, halgaz, haHamis** parancshármas - röviden **hai** és **hah.** E szerkezet egy eljáráson belül használható, és nyilvánvalóan a **teszt** parancsnak meg kell előznie a **halgaz** és **haHamis** parancsokat, amelyekből több is lehet az eljáráson belül. Ez utóbbi gyakorlatilag azt jelenti, hogy az elágazási feltételt adó kifejezés többszöri kiértékelését lehet elkerülni, illetve mivel nem kötelező mindkét ág, így nem határozzuk meg előre, hogy egy- vagy kétágú kiválasztást akarunk-e megvalósítani.

Példaként nézzük meg az egyszerű értékvizsgálatot bemutató ***jellemez*** eljárást.

```
eljárás jellemez :szám
 teszt :szám >= 0
 halgaz [mutat "JA szám pozitívj]
 haHamis [mutat "|A szám negatív|]
 halgaz [mutat "|vagy nulla|]
vége
```

Hívjuk meg az eljárást:

```
? jellemez 5
A szám pozitív
vagy nulla
? jellemez -4
A szám negatív
```

Amint látható, az igaz ág részekre bontható lett.

### *Többágú kiválasztás*

*A* többágú kiválasztás az **elágazás** paranccsal valósítható meg. Itt a feltételként megadott *kifejezésnek* egy szóval kell visszatérnie, amelynek a lehetséges értékeit soroljuk fel az egyes ágak feltételeként. Fontos, hogy az értékek nem kerülnek kiértékelésre, azaz csak szókonstansok használhatóak esetként. Az egyes ágak egymás után kerülnek vizsgálatra, és azt az ágat hajtja csak végre, amelynél először teljesül az egyezés. Ha nem adunk meg az utolsó ág előtt feltételt, akkor az hajtódik végre, mint különben ág. Ha ilyen nincs, és egyik ág esetei sem adtak egyezést, akkor nincs hatása a szerkezetnek.

```
elágazás kifejezés [
 eseti [utasításlista]
 eset2 [utasitaslista]

 eseta [utasításlista]
  [utasításlista egyébként(elhagyható) ]
]
```

Például, ha egy beolvasott jelről szeretnénk eldönteni, hogy ékezetes vagy ékezet nélküli magánhangzó, vagy esetleg egyéb karakter, akkor ezt a következő módon tehetjük meg egy eljárásban megvalósítva.

```
eljárás magánhangzó
 mutat "|Nyomj le egy billentyűt!|
 lókért "jel olvasjel
  (kiírérték "|A(z) | :jel "| |)
 mutat elágazás :jel [
         a e i o u         ["|ékezet nélküli.)]
         á é í ó ö ő ú ü ű  ["ékezetes.]
                            ["|egyéb jel.|]

vége
```

Hívjuk meg rendre a ***magánhangzó*** eljárásunkat az u, ö és a p megadásával:

```
? magánhangzó
Nyomj le egy billentyűt!
A(z) u ékezet nélküli.
? magánhangzó
Nyomj le egy billentyűt!
A(z) ö ékezetes.
? magánhangzó
Nyomj le egy billentyűt!
A(z) p egyéb jel.
```

7. *Alakítsuk át úgy a **magánhangzó** eljárást, hogy az a mássalhangzó esetén is nyelvtanilag helyes kiírást adjon!*

## Iteráció

Az iterációnak, azaz az ismétlésnek számos, a korai LOGO megvalósításokban nem is létező változata került be az Imagine-be. Ez a változatosság könnyebbé, de egyben nehezebbé is teszi az Imagine-ben programozó feladatát: alaposan ismernie kell ugyanis a különböző szerkezeteket, különben nem a leghatékonyabb vezérlést fogja alkalmazni.

Elsőként a mindenki által jól ismert, **ismétlés** parancs új arcát mutatjuk meg.

### *Számlálásos ciklus, ciklusváltozó nélkül*

```
ismétlés ismétlésszám [utasításlista]
```

Az előre meghatározott számú ismétlés elvégzésére az **ismétlés** parancsot használhatjuk - röviden **ism.** A címből sokan talán rá sem ismernek, mivel az nem is teljesen pontos. A ciklus ugyanis rendelkezik ciklusszámlálóval, de ezt nem nevesíti, és a **hányadik** paranccsal lekérdezhető az értéke.

A működése jobb megértéséhez érdemes megfigyelni a következő utasításokat.

```
? ismétlés 1 [mutat "alma]
alma
? ismétlés 3 [mutat "alma]
alma
alma
alma
```

```
? ismétlés 2.5 [mutat "alma]
alma
alma
alma
? ismétlés 2.49999 [mutat "alma]
alma
alma
```

A fenti eredmény azért meglepő, mert a ComLOGO-ban az ismétlés nem végzett kerekítést, azaz a 2.5, de a 2.9999 esetén is csak két kiírás lett volna.

A számlálásos ciklusban alkalmazzuk ezután a **hányadik** eljárást is, ami tehát a ciklusszámláló szerepét tölti be, így értéke 1-gyel indul, ha egyáltalán történik végrehajtás legalább egyszer.

```
? ismétlés 2.4 [kiír hányadik]
1
2
? ismétlés 2.5 [kiír hányadik]
1
2
3
```

De lássuk, hogy is lehet ezt kombinálni!

```
? ismétlés 2 [(mutat hányadik "alma)]
1 alma
2 alma
? ismétlés -1 [(mutat hányadik "alma)]
? ismétlés 0 [(mutat hányadik "alma)]
? ismétlés 0.5 [(mutat hányadik "alma)]
1 alma
```

Figyeljünk fel arra, hogy a -1 és 0 ismétléseknél nincs kiírás, míg a 0.5 esetében már egyszer végrehajtja a futtató rendszer, és a **hányadik** is felveszi az 1 értéket.

Az **ismétlés** parancsok egymásba is ágyazhatóak, de ekkor a **hányadik** parancs működése nehezebben nyomon követhető!

```
? ismétlés 3 [(kiírérték hányadik "|  |) ismétlés 2 [kiír
hányadik]]
1 1
2
2 1
2
3 1
2
```

Mint a példákból is látszik, ebben az esetben a **hányadik** mind a külső, mind a belső ciklus végrehajtásszámát megadja, azaz úgy viselkedik, mintha mind a kettőnek külön-külön ciklusszámlálója lenne.

### Elöltesztelő ciklus

```
amíg [feltétel] [utasításlista]
```

Az **amíg** a klasszikus elöltesztelő ciklus megjelenése a LOGO-ban. Elsőként kiértékeli a *feltétel*-t. Ha értéke IGAZ, akkor a parancs lefuttatja az *utasításlista* utasításait mindaddig, amíg a feltétel IGAZ marad.

Ha a *feltétel* HAMIS, akkor a ciklusmag nem hajtódik végre, azaz ha már az első vizsgálatkor is HAMIS lett volna, akkor az *utasításlista* egyszer sem kerül végrehajtásra.

Az *utasításlistá*-n belül itt is használható a **hányadik** művelet, mely a már lefutott (beleértve az épp zajlót is) ismétlések számát adja meg.

8. *Elöl- vagy hátultesztelő ciklus az* **ismétlés**?

### Feltétel nélküli, végtelen ismétlés

```
végtelenszer utasításlista
```

Röviden **vszer**. Folytonosan ismételve futtatja az *utasításlista* utasításait egy végtelen ciklusban. A parancs hatása gyakorlatilag azonos a

```
amíg ["igaz] [utasításlista]
```

kiadásával. Az *utasításlista* utasításait „örökké" futtatja – hacsak meg nem szakítjuk azt a folyamatot, amely ezt a **végtelenszer** parancsot elindította.

> Bár a **végtelenszer** működése nagyon hasonlít hozzá, de nem szabad összekeverni ezt az egyszerű vezérlőparancsot az **örökké** paranccsal, amely egy új, független folyamatot indít el.

Egy érdekes programrészlettel mutatjuk be a parancs működését, amit csak az F12 gombot lenyomva vagy az eszköztár valamelyik megállító ikonjára kattintva állíthatunk le.

```
? törölképernyő
? végtelenszer [jobbra tetszőleges előre 5
ha absz poz > 60 [hátra 5] várj 1]
```

### Számlálós ciklus, ciklusváltozóval

Szinte hallani lehet, *tisztelt Olvasó*, hogy fellélegzett, ha elsőként ismerkedik a LOGO világával, és BASIC vagy Pascal nyelven nőtt fel. Végre! Itt a jó öreg, megszokott ***for***! A parancs őse már a ComLOGO-ban is létezett, csak ritkán használták. A kulcsszó **ciklus**, vagy röviden **cikl**.

```
ciklus szó [kezdet vég] [utasításlista]
```

vagy

```
ciklus szó [kezdet vég lépés] [utasításlista]
```

A vezérlési szerkezet intelligens módon viselkedik, ami emiatt szokatlan i egyben. Ha a *kezdet* <= *vég*, akkor eggyel növeli, különben, ha a *kezdet* > *vég* akkor eggyel csökkenti a *szó* ciklusváltozót.

A *szó* változó kezdeti értéke *kezdet* lesz, és lefut az *utasításlista*. Ezután a *sz* változó értéke 1-gyel vagy *lépés*-sel módosul, az előbbi viszonytól függően Ezt a folyamatot addig ismétli, amíg a *szó* változó nem lépi át a *vég* értékét.

Az *utasításlistá*-ban, a **hányadik**-kal lehet az eddig végrehajtott ismétlése pillanatnyi számára hivatkozni.

> Sajnos az **ismétlés** és a **ciklus** vezérlési szerkezetek nem teljesen úgy viselkednek, ahogy elvárnánk! Éppen ezért tanulságul szolgáljon a következő néhány talán extrémnek is nevezhető parancs, amelyek azonban előfordulhatnak programban is.

Kezdetben minden jól indul:

```
? ciklus "i [1 3][(kiír szó hányadik "|. futásnál i értéke) :i)]
1. futásnál i értéke 1
2. futásnál i értéke 2
3. futásnál i értéke 3
? ciklus "i [2 4][(kiír szó hányadik " j . futásnál i értéke| :i)]
1. futásnál i értéke 2
2. futásnál i értéke 3
3. futásnál i értéke 4
? ciklus "i [6 4][(kiír szó hányadik "|. futásnál i értéke| :i)]
1. futásnál i értéke 6
2. futásnál i értéke 5
3. futásnál i értéke 4
```

Állítsuk munkába a *lépés* paramétert is!

```
? ciklus "i [1 6 2][(kiír szó hányadik "|. futásnál i értéke) :i)]
1. futásnál i értéke 1
2. futásnál i értéke 3
3. futásnál i értéke 5

? ciklus "i [6 1 2][(kiír szó hányadik "|. futásnál i értéke| :i)]
```

Nem nyomdahiba, az eredmény egy üres, *tisztelt Olvasói*

A program várt csökkenő irányában a harmadik paraméterére valóban nincs kiírás! Vagy mégis?

```
? ciklus "i [6 1 -2][(kiír szó hányadik "|. futásnál i értéke) :i)]
1. futásnál i értéke 6
2. futásnál i értéke 4
3. futásnál i értéke 2
```

Igen! Nekünk kell megadni a *lépés* értékét negatívnak! A *lépés* minden esetben valódi növekmény, azaz hozzáadódik a *szó* értékéhez!

Nézzük, van-e még más meglepetés is! Lássuk, mi van a törtekkel!

```
? ciklus "i [1 2 0.5][(kiír szó hányadik "|. futásnál i értéke|
:i)]
1. futásnál i értéke 1
2. futásnál i értéke 1.5
3. futásnál i értéke 2
? ciklus "i [1 2 0.6][(kiír szó hányadik "|. futásnál i értéke|
:i)]
1. futásnál i értéke 1
2. futásnál i értéke 1.6
```

Láthatóan elfogadta a tört lépés értéket is, de itt is a már **ismétlés**-nél tapasztalt kerekítés működik. És mily remek! A negatív és tört határok sem ejtik kétségbe a **ciklus**-t!

```
? ciklus "i [-1 1.5 0.7][(kiír szó hányadik "|. futásnál i
értéke| :i)]
1. futásnál i értéke -1
2. futásnál i értéke -0.3
3. futásnál i értéke 0.4
4. futásnál i értéke 1.1
```

És a Súgóban beígért 0 értékű *lépés*-sel mi is történik?

```
? ciklus "i [1 1 0][(kiír szó hányadik "|. futásnál i értéke|
:i)]
ciklus nem szereti [1 1 0] -t bemenetként.
 A bemenő adatok listájának felépítése nem megfelelő ciklus
számára!
? ciklus "i [1 2 0][(kiír szó hányadik "|. futásnál i értéke|
:i)]
ciklus nem szereti [1 2 0] -t bemenetként.
 A bemenő adatok listájának felépítése nem megfelelő ciklus
számára!
```

A Súgó ígérete szerint egyszer kerülne csak végrehajtásra a ciklusmag, ami láthatóan egyik esetben sem igaz, pontosabban persze részben igaz, mert a program se fut…

Külön felhívjuk a figyelmet még egyszer a következő két futási eredményre, ami jól megvilágítja a **hányadik** parancs és a *szó* változó funkcióját.

```
? ciklus "i [3 1][kiír hányadik]
1
2
3
? ciklus "i [3 1][kiír :i]
3
2
1
```

Sajnos vannak még további buktatók is. Például nézzük az alábbi kis feladatot, ahol az 1 ... *n* páratlan számokat szeretnénk egymás alá kiírni!

```
? globvál "n 5
? ciklus "i [1 :n 2][kiír :i]

ciklus nem szereti :n -t bemenetként.
  A(z) ciklus művelet számot vár bemenő adatként!
```

A probléma orvosolható: az ***:n*** változó értékét adjuk csak át, ami a **mondat** parancssal megoldható, vagy **!** jellel kényszerítjük a fordítót a szögletes zárójelben lévő kifejezés kiértékelésére.

```
? ciklus "i (mondat 1 :n 2)[kiír :i]
? ciklus "i ![1 :n 2][kiír :i]
1
3
5
```

A **ciklus** vezérlési szerkezet nagyon jól használható, de furcsaságait mindenképpen előzetesen fel kell derítenünk, mert különben értékes órákat veszíthetünk el programfejlesztés közben az őrület határára sodródva.

Nézzünk egy sokak által már biztos ismert feladatot a **ciklus** alkalmazására!

3. *Készítsünk programot gyufa néven, amely megmondja, hogy N darab gyufaszálból (3<=N<=1000) hány különböző háromszöget lehet összerakni!*

A feladatot fogalmazzuk át! Hány olyan különböző háromszög van, amelynek oldalai egész hosszúságúak, és az oldalak összege éppen N?

A három oldal összegének nyilvánvalóan N-nek kell lennie: i+j+k=N, ebből származóan a k=N-i-j lesz. Ki kell még zárnunk az ismétlődő megoldásokat, azaz a 3,4,5 és 4,3,5 megoldások nem különbözőek. A háromszögek oldalait ezért nagyság szerint rendezzük, azaz i<=j<=k álljon fenn.

Szintén a rendezettség miatt a három háromszög egyenlőtlenségből elegendő az i+j>k feltételt ellenőrizni, mivel a többi biztosan fennáll.

```
eljárás gyufa :n
 lokért "db 0
  ciklus "i (mondat 1 :n)
   [ciklus "j (mondat :i :n)
    [lokért "k :n-:i-:j
     ha (és :j<=:k :i+:j>:k ) [növel "db]
    ]
   ]
  eredmény :db
vége
```

Vegyük észre, hogy változókat közvetlen módon nem használhatjuk a ciklus alsó és felső határának, de például a **mondat** utasítás segítségével most is áthidalhattuk ezt a problémát.

```
? mutat gyufa 9
3
? mutat gyufa 7
2
? mutat gyufa 12
3
? mutat gyufa 13
5
? mutat gyufa 100
208
? mutat gyufa 1000
20833
```

### *Halmazon végrehajtott ismétlés*

```
ciklusegyenként szó bármi [utasításlista]
```

A parancsszó rövidítése **ciklegy.** A második - *bármi* - bemenet első eleme (egy karakter, ha szó; egy elem, ha lista; egy képkocka, ha képsor stb.) lesz a *szó* változó értéke, és lefut az utasítások listája - az *utasításlista.* Ezután a *szó* értéke *bármi* második eleme lesz, s újra lefut az *utasításlista.* Ez így megy tovább, végigléptetve *bármi* minden elemén.

Az *utasításlista* pontosan a *bármi* elemeinek számával megegyező alkalommal fut le. Ennek megfelelően, ha *bármi* üres, az utasításlista egyszer sem fut le. Az *utasításlistá-n* belül használható a **hányadik** ciklusszámlálóként, amellyel az éppen futó ismétlés sorszámára lehet hivatkozni, de használhatjuk a *szó* változót is egyszerű változóként.

```
? ciklusegyenként "i [13 5 7][(kiír hányadik "|: | :i)]
1 : 1
2 : 3
3 : 5
4 : 7
? ciklusegyenként "i [a d][mutat :i]
a
d
? ciklusegyenként "i [Anna Ottó Balázs][mutat :i]
Anna
Ottó
Balázs
```

A **ciklusegyenként** használatára nézzük meg a következő versenyfeladat megoldását!

## Mókusok (NTTV 2001/02. 5-8. évfolyam 2. forduló)

Egy patakban N követ raktak le átjárónak. A köveken M mókus ugrál át a túlsó partra úgy, hogy mindegyik csak bizonyos kövekre - például a parttól számítva minden második, minden negyedik kőre - ugrik.

4. *Készítsünk programot, amely beolvassa a kövek számát (1<=N<= 20), a mókusok számát (1<=M<=20), és azt, hogy az egyes mókusok minden hányadik kőre ugranak (l<=K(i)<=N, ahol 1<~ i <=M), majd megadja, hogy mely kövekre nem lép egyetlen mókus sem (üres legyen a válasz, ha nincs ilyen kő), illetve hogy melyekre lép a legtöbb!*

```
BEMENET          KIMENET
N=15             Egy mókus sem lép ezekre: 1, 5, 7, 11, 13
M=3
K=(2,3,4)        A legtöbb mókus lép ezekre: 12
```

A feladatot két részre bontjuk, és már szinte adja magát a megoldás. Elsőként nézzük meg, hogy a mókusok melyik kövekre ugranak, majd ezekből válasszuk ki a keresett köveket!

A megoldásban kihasználjuk, hogy a **ciklusegyenként** utasítás halmazon értelmezett ismétlést ad.

```
eljárás ugrál :n :ide
 lókért "nyom []
 lókért "i 0
 amíg [ti < :n]
   [lókért "rá 0 növel "i
    ciklusegyenként "j :ide
      [ha maradék :i (elem hányadik :ide) = 0 [növel "rá]]
       lókért "nyom utolsónak :rá :nyom]
 ki :nyom
 értékel :nyom
vége
```

A külső ciklus indexét ***i*** jelöli. Használhatnánk az **ismétlés :n** utasítást és a **hányadik** függvényt itt is, de akkor követhetetlen lenne a belső ciklus esetében a másik ciklusindex értéke.

A kapott ***nyom*** lista feldolgozása technikailag külön eljárásba került, de egybe is írhattuk volna az előzővel. A szétbontást az indokolja, hogy az itt alkalmazott lépések már eredményként kezelik az elkészített ***nyom*** listát.

```
eljárás értékel :nyom
 kiír [Egy mókus sem lép ezekre:]
 ciklusegyenként "j :nyom
   [ha elem hányadik :nyom = 0
     [( kiírérték hányadik ", )]]
 kiír []
 lókért "max első :nyom
```

```
 ciklusegyenként "j :nyom
   [ha elem hányadik :nyom > :max
     [lókért "max elem hányadik :nyom]]
 lókért "sok []
 ciklusegyenként "j :nyom
   [ha elem hányadik :nyom = :max
     [lókért "sok utolsónak hányadik :sok]]
 (kiír "|A legtöbb mókus lép ezekre,| :max "alkalommal:)
 kiír :sok
vége
```

A maximum megkeresése és az adott értékű elemek kigyűjtése szintén klasszikus programozási feladat, ami LOGO-ban is könnyen megoldható.

A futási eredmény a következő lehet például:

```
? ugrál 20 [2 3 4 5]
0 1 1 2 1 2 0 2 1 2 0 3 0 1 2 2 0 2 0 3
Egy mókus sem lép ezekre:
1,7,11,13,17,19,
A legtöbb mókus lép ezekre, 3 alkalommal:
12 20
```

9. *Alakítsuk át úgy az **értékel** eljárást, hogy felhasználjuk a maximum és adott tulajdonságú elemek kigyűjtése programozási tételeket!*

### *Hátul tesztelő ciklus*

Bármennyire is szokatlannak tűnik, de a klasszikus *repeat until feltétel* felépítésű hátul tesztelő ciklus nem található meg az Imagine-ben. De nem kell kétségbe esnünk emiatt, hisz ez megvalósítható az amíg segítségével is, csak a ciklusmagot célszerű eljárásként előállítani, majd a kilépési feltételt is célszerű tagadva megadni. Ennek megfelelően tehát a:

```
repeat ciklusmag until feltétel
```

szerkezet Imagine-ben így lesz megvalósítható:

```
ciklusmag
amíg [nem feltétel]
  [ciklusmag]
```

## Program futásának felfüggesztése

### *Várakozás adott ideig*

```
várj idő'
```

Hatására az Imagine szünetelteti azt a folyamatot, amelyben a **várj**-ot használjuk, annyi ezredmásodpercig, amelyet megadtunk *idő-ként.* Tehát a **várj 1000** pontosan egy másodpercre függeszti fel az éppen futó folyamatot.

Valójában ez a várakozási idő bizonyos esetekben függhet a processzortól, a memória és egyéb erőforrások kihasználtságától, a futtatás körülményeitől.

> Tapasztalataink szerint ha rövid időtartamokat adunk meg, és ezeket sokszor kívánjuk kivárni, akkor az óra az előkészítések miatt pontatlannak tűnik, és másként viselkedik webes alkalmazás esetén, másként Imagine alatt, és másként önállóan futó alkalmazásban. Talán ez utóbbiban a legkiszámíthatóbb, ezért érdemes mindenképpen itt is megvizsgálni, nem lett-e, lesz-e túl kevés a meghatározott várakozási idő az elkészült alkalmazásban.

### *Várakozás egy feltétel teljesüléséig*

```
váramig [feltétel]
```

Ismételten kiértékeli a bemeneti *feltételt-i,* amíg értéke az IGAZ nem lesz. Az utasítás érdekessége, hogy nincs ciklusmagja, azaz egyszerű várakozás az adott feltétel teljesüléséig.

A **várj** paranccsal a program futását tudjuk felfüggeszteni, de a tényleges futás idejét is tudjuk mérni. Jó tudni, hogy ha parancsmódban dolgozunk, akkor az első futtatás mindig lassabb, mint a második, mivel a tényleges fordítás ideje is az első alkalommal hozzáadódik, míg a második - módosítás nélküli futtatás - során már az Imagine intelligensen felismeri, hogy a végrehajtandó kódot nem kell újra előállítani.

### *Időmérés stopperrel*

Az idő mérésére szolgál a **stopper** és **nullázstopper** parancspár. A **stopper** használható paraméter nélkül is, ekkor az Imagine elindítása óta eltelt időt mutatja ezredmásodpercben megadva.

```
? mutat stopper
2322289
```

*10. Hogyan mérhetnénk meg egy program kifejlesztésére szánt időt?*

A számítógép belső idejét, az aktuális időt az **időformátum** és **idő** parancs adja vissza, előbbi szóként, utóbbi listaként.

```
? mutat időformátum
10:31:7 de.
? mutat idő
[10 31 7 867]
```

Az időformátum megjelenítése azonos az operációs rendszerben beállítottál.

Használhatjuk a **(stopper "óranév)** formát is, amikor is egy névvel ellátott óra idejét kérdezzük le. A **nullázstopper** és az új óra létrehozása értelemszerűen 0-ra állítja a stopper idejét, de ez nem befolyásolja a tényleges időt megjelenítő órát.

A stopperek használatára nézzük meg a következő utasítássort, amelynél a gépelésünk sebessége dönti el, hogy mit kapunk vissza értékként.

```
? mutat (stopper "egy)
0
? mutat (stopper "kettő)
0
? mutat (stopper "egy)
18857
? (nullázstopper "egy)
? mutat (stopper "egy)
8671
? mutat (stopper "kettő)
71313
```

Mint látható, a több stopper egymástól függetlenül is működtethető.

## Eljárás

A programozási nyelvek ugyan igen nagy számú parancsot tartalmaznak, de hatékonyságukat és rugalmasságukat az adja, hogy újabb utasítássorozatokat láthatunk el önálló névvel, és hozzájuk bemeneti változókból álló listát rendelhetünk. Erre szolgál az **eljárás** parancs, röviden **elj.**

Az eljárás megadását a **vége** parancs zárja le.

```
eljárás  eljárás_név bemenő_paraméterek
  u tasítások_sorozata
•vége
```

## Függvény

Függvényről beszélünk majd, ha az eljárás visszatérő értékkel fejezi be működését. Ehhez az eljárásban az **eredmény** - röviden **er** - parancsnak szerepelnie kell.

Az |x| függvény megvalósítása például ez lehet:

```
eljárás abszolútérték :x
 hak :x > 0 [eredmény :x]  [eredmény -:x]
vége
```

Például a meghívása így történet:

```
? mutat abszolútérték -2
2
```

## Rekurzió

Rekurzív eljáráson az önmagát meghívó eljárást értjük. A rekurziót igen gyakran használjuk mind a rajzolás, mind a funkcionális LOGO területén.

> A rekurzió gyakori alkalmazásának fő oka, hogy az automataelvű és a funkcionális nyelvek természetes megoldása az eljárások állapotmódosulás vagy paraméter változtatás után önmagukat hívják meg. Az Imagine LOGO mindkettő nyelvtípus sajátosságait tartalmazza.

# Matematikai algoritmusok

A továbbiakban nézzük meg néhány középiskolában felmerülő matematikai feladat megoldását LOGO környezetben!

Az itt ismertetett feladatok megoldásának részletes matematikai indoklása megtalálható a középiskolai tankönyvekben, illetve e könyv honlapján is.

*5. Végezzük el a Newton-féle gyökvonást!*

A módszer alapvetően az intervallumfelezéses közelítésen alapszik. ISAAC NEWTON (1642-1727) egy korábban már az indiai matematikusok által is ismert, alkalmazott módszert tett általánossá. Az eljárás alapgondolata az, hogy a keresett gyököt közrefogjuk egy nála kisebb és egy nála nagyobb értékkel, majd az új közelítő értéknek ezen intervallum felezőpontját választjuk. Az eljárást addig folytatjuk, amíg két egymást követő közelítő érték eltérése kisebb lesz a megkívánt pontosságnál.

```
eljárás newton :a :p :x0 :x1
 ha absz(:x0 - :x1)<:p [eredmény :x1]
 eredmény newton :a :p :x1 (:x1+:a/:x1)/2
vége
```

Próbáljuk ki a függvényt! A meghívása ***newton*** *alap pontosság* 1 *alap* formában történhet:

```
? mutat newton 4 0.1 1 4
2.00061
? mutat newton 5 0.01 1 5
2.23607
```

*6. Számoljuk ki egy szám hatványát Legendre módszerével!*

AZ ADRIEN MARIE LEGENDRE (1752 – 1833) francia matematikus nevéhez fűződő, a hatványozást hatékonyabbá tévő módszer alapgondolata az, hogy ha a kitevő páros, akkor az alapot emeljük négyzetre, ha pedig páratlan, akkor egy tényező leválasztásával érjük el a kitevő újbóli párossá tételét. A fenti két lépést addig ismételjük, amíg a kitevő 0 nem lesz.

```
eljárás legendre :a :k :s
 ha :k=0 [eredmény :s]
 hak maradék :k 2 = 0
   [eredmény legendre :a*:a :k/2 :s]
   [eredmény legendre :a :k-1 :s*:a]
vége
```

Próbáljuk ki a függvényt! A meghívása ***legendre*** *alap kitevő* 1 formában történhet:

```
? mutat legendre 3 5 1
243
```

*7. Határozzuk meg két szám legkisebb közös osztóját euklideszi algoritmust alkalmazva!*

EUKLIDÉSZ időszámításunk előtt 300 körül írt Elemek című művében már közkeletű eljárásként beszél a módszerről.

Maga a prímtényezős felbontáson alapuló módszernél jóval hatékonyabb eljárás azon a felismerésen alapszik, hogy a két szám legnagyobb közös osztója a két szám egymással való osztása maradékának is osztója kell, hogy legyen.

```
eljárás euklidesz :a :b
 ha :b=0 [eredmény :a]
 eredmény euklidesz :b maradék :a :b
vége
```

Próbáljuk ki a függvényt! A meghívása ***euklidesz*** *szám1 szám2* alakú:

```
? mutat euklidesz 42 24
6
? mutat euklidesz 5 8
1
? mutat euklidesz -84 36
12
```

*8. Készítsünk prímszitát Eratoszthenész módszerével!*

ERATOSZTHENÉSZ (i. e 276-194) között élt görög csillagász és matematikus nem túl hatékony, de igen egyszerű módszere megadja egy adott értékig a prímszámok halmazát.

A feladat megoldását két részre bontjuk. Elsőként előállítjuk a ***feltölt*** eljárással az egész számokat tartalmazó listát.

```
eljárás feltölt :db
 ha :db=0 [eredmény []]
 eredmény utolsónak :db feltölt :db-1
vége
```

Erre a célra használhatjuk a **generál :db [:% + 1] 1** parancsot is, ami hatásában ugyanezt eredményezi, és lényegesen gyorsabb is, viszont csak maximum ***tízezer*** elemű lista előállítására használható.

Ezután kiejtjük azokat a számokat a listából, amelyek nem lehetnek prímek. Elsőként az 1-et húzzuk ki, majd ami elsőként megmaradt, azt eredményként kiírjuk, mivel az prím, ez tehát elsőként a 2. Ennek többszöröseit kiejtjük.

Ezután ami elsőként megmaradt, az megint biztos prím, kiírjuk, és a többszöröseit kiejtjük. Ezt csináljuk mindaddig, amíg az első még bennmaradt szám kisebb, mint az elemszám gyöke.

Első lépésként oldjuk meg a többszörösök kiejtését!

```
eljárás szitál :l :o
 ha üres? :l [eredmény []]
 hak maradék első :l :o = 0
   [eredmény szitál en :l :o]
   [eredmény elsőnek első :l szitál en :l :o]
vége
```

Az ***o*** jelöli a szitáló prímet, amivel osztunk. Például az első 20 számból a párosak törlése is elvégezhető vele.

```
? mutat szitál feltölt 20 2
 [1 3 5 7 9 11 13 15 17 19]
```

A *szita* eljárásunk végzi el az egyes részfeladatok összekapcsolását. A ***prím*** nevű listában gyűjtjük össze a prímeket.

```
eljárás szita :n
 globvál "l feltölt :n
 globvál "prím []
 ; 1-t töröljük
 globvál "l en :l
 ; az első prím, kiírjuk majd kiejtjük a többszöröseit
 amíg [első :l <= gyök :n]
   [globvál "prím utolsónak első :l :prím
    globvál "l szitál :l első :l]
 ; ami maradt, az még mind prím
 eredmény (mondat :prím :l)
vége
```

Az eljárásban újdonság a gyök parancs, amely egy nemnegatív szám gyökét adja vissza.

Figyeljük meg, hogy a szitáló prímeket és a gyök elérése után még megmaradt számokat egyszerűen a **mondat** paranccsal fűzzük össze egyetlen listává.

A programmal megkereshetjük például 500-ig a prímeket

```
? mutat szita 500
 [2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79
83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163
167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251
257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349
353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443
449 457 461 463 467 479 487 491 499]
```

*11. Alakítsuk át úgy a szita nevű eljárást, hogy ne alkalmazzunk globális változókat!*

## *Kiíratási gyakorlatok*

Az alábbi néhány példa a vezérlési szerkezetek és a kiíratás gyakorlati alkalmazására és bemutatására szolgálnak példaként.

9. *írassuk ki a számokat táblázatos formában 1-től 5-ig, majd a 2, 3 és 4-szeres értéküket is!*

```
? ciklus "i [1 4] [ciklus "j [1 5] [(kiírérték :i*:j "| |)]
kiír[]]
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
```

10. *Szorozzuk össze soronként kiíratva az 1-9 páratlan számokat a 2-6 páros számokkal!*

```
? ciklegy "i [13 5 7 9]
   [ciklegy "j [2 4 6][(kiírérték :i*:j "| |)] kiír[]]
2 4 6
6 12 18
10 20 30
14 28 42
18 36 54
```

A fenti minták példájára készítsük el karakteres és szöveges felületre is a következő feladatokat!

12. *írassuk ki az összeadó és a szorzótáblát fejlécekkel együtt! A tábla 1-től 10-ig mutassa be az eredményeket!*

13. *írassuk ki a 2 és a 3 többszöröseit 100-ig egy-egy sorban!*

14. *Készítsünk formázott kiíratást lehetővé tevő eljárást! Az eljárás egyik paramétere a kiírandó szó legyen, a másik egy számérték, amely azt adja meg, hogy jobbra igazítva hány helyet foglaljon el!*

# Játékok készítése

Az Imagine talán egyik legnagyobb erőssége, hogy nagyon egyszerűen lehet benne játékokat fejleszteni. Erre mutat néhány példát ez a fejezet.

A fejezet olvasásának megkezdése előtt a korábban ComLOGO-t használók számára mindenképpen javasolt megnézni a Sulinet Informatika rovatában szereplő, az áttérést megkönnyíteni kívánó, királylány kiszabadításáról szóló játékot a http://informatika.sulinet.hu/inform/lovag/lovag.zip lapon!

Ezen kívül mindenképpen tanácsos az itt látható http://imagine.elte.hu/ lapon is járni, hogy lássuk a korábban ComLOGO-ban használt kulcsszavak Imagine megfelelőjét!

Az itt bemutatott játékokat letölthetjük ugyan a könyv honlapjáról, ennek ellenére érdemes olvasás közben el is készíteni, mivel a bennük alkalmazott megoldások és lépések rutinszerűek, és sajnos csak gyakorlással válnak készséggé. Az első néhány ismertetett játékban alkalmazott fogásokat, megoldásokat részletesen adjuk meg. A későbbiek esetében ezt már nem mindig tesszük meg, részben helytakarékossági megfontolásokból.

***Minden játékunk esetén érdemes a továbbfejlesztésen gondolkodni, amit nagy örömmel fogadunk is honlapunkon!***

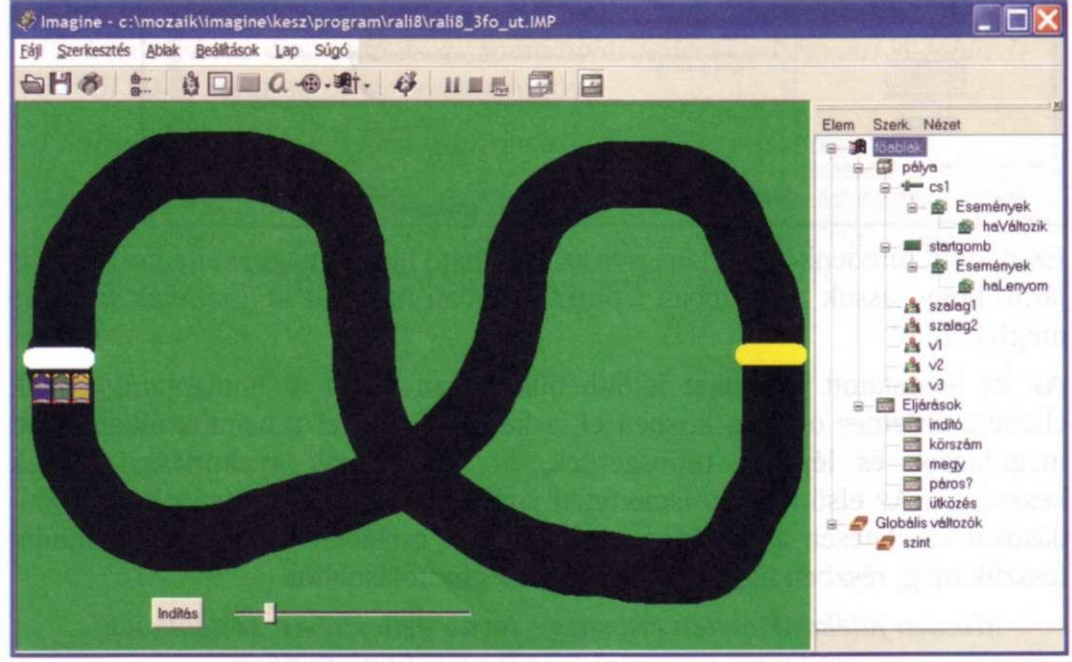
## Rali!

Az Imagine-nel egyszerre több folyamat is végrehajtható. Ezt használjuk ki az autóversenyes játékunkban.

*11. Készítsünk három személyes játékot! Mindhárom játékos egy nyolcas alakú pályán halad versenyautójával. A cél három kör megtétele. Akinek elsőként sikerül, az nyer! De ne csak rójuk a köröket! Igyekezzünk a többieket megakadályozni a célba jutásban!*

A játékban több apró elemet is megvalósítunk a feladat kitűzésének megfelelően:

- Ha valaki nekimegy a másiknak, akkor visszaléptetjük véletlenszerűen, véletlenszerű irányba.
- Ha egy kocsi kimegy a fűre, akkor lefékeződik, azaz lassabban tud csak mozogni.
- Az nyer, aki elsőként végigmegy a nyolcas alakú pályán egymás után háromszor! A pályán két csík – egy fehér és egy sárga – átlépésével ellenőrizzük, hogy sikerült-e a kör megtétele. Az irány nem fontos, csak a két csík felváltva való érintése.
- A kész játékot önállóan futóvá tesszük, mentjük EXE fájlként és webprojektként is!



## Hogyan készítjük el?

A játék egyes elemeit célszerű a megadott sorrendben elkészíteni. Ez a későbbi takarások miatt is hasznos.

### *A pálya elemei*

A pálya méretét állítjuk be elsőként a Lap menüben a Változtat... pontra kattintva. A név legyen *pálya* az Alapok fülön, valamint a pálya háttérszínét állítsuk be zöldre. A Megjelenés fülön a pálya méretét módosítsuk 796×499-re, ha origó, a középpont helye pedig nem (398;249), akkor azt is.

Az alapértelmezett lap mérete azért ekkora, mert így egy 800×600 képpontos képernyőn is teljes egészében látható lesz a pálya az önálló futtatáskor, illetve a weben a böngésző ablakában is látszik a teljes pálya a kiírásokkal együtt.

Az út elkészítéséhez ezután a Festő eszköztár mutatását kérjük az ikonjára kattintva. Ezen elsőként válasszuk ki a vonalvastagság beállítását. Itt kézzel írjuk be a 60 pont vastagságot. A vonal színe maradjon fekete.

A ceruza ikonra kattintva készíthetjük el a pályát. Ha javítani szeretnénk, akkor a Szerkesztés/Rajz visszavonása menü, illetve a Radír ikon is rendelkezésünkre áll.

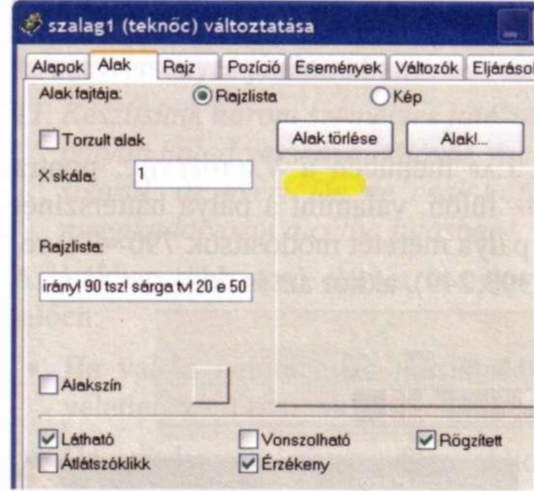
Az elkészített pályát mentsük el képként is! Erre azért is szükség lesz, mert a győztes nevét kiírjuk a képernyőre, és ez a háttérre kerül, amit egy újabb játék megkezdésekor törölni kell. Ezt itt az újbóli betöltéssel oldjuk meg.

15. *Az újbóli betöltés helyett a háttér színével egyező színnel írjuk felül a szöveget!*

A mentést megtehetjük a Festő eszköztár Háttér ikonjára kattintva vagy a Lap menüben a Rajzlapmentés... pontot választva. Emellett még választhatjuk a lap helyi menüjéből is a Rajzlapmentés... parancsot.

Alapértelmezés szerint a KEPSOR mappába szeretne menteni, de a mentési helyet most ne felejtsük el úgy módosítani, hogy a projektünk mappájába írja ki a kész képet!

### Teknőcök




A pályán egy fehér és egy sárga szalagot is elhelyezünk. A teknőc létrehozása után az ALAPOK fülön elnevezzük *szalag1*-nek, felemeljük a tollát és az ALAK fülön **Rajzlistaként** megadjuk az alakját, ami

```
irány! 90 tsz! sárga
tv! 20 e 50
```

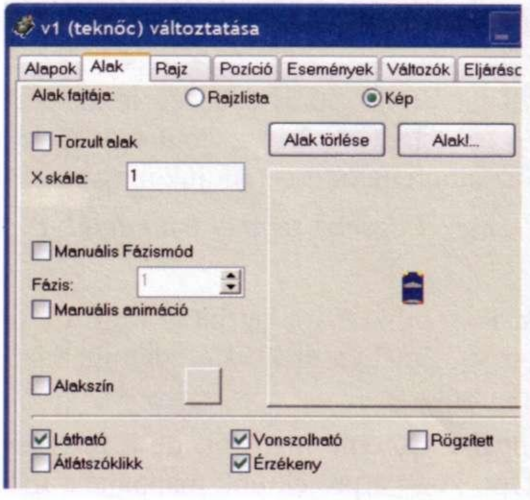
utasítássor lesz. A parancsok begépelésének és bármilyen módosításának eredménye azonnal megjelenik a jobb oldali, alak előnézet területen.

A *szalag1* teknőcünket ne felejtsük el **Érzékennyé** tenni, mert különben nem tudjuk az autókkal való ütközést figyelni.

A kész teknőcöt mozgassuk a pálya jobb szélére.

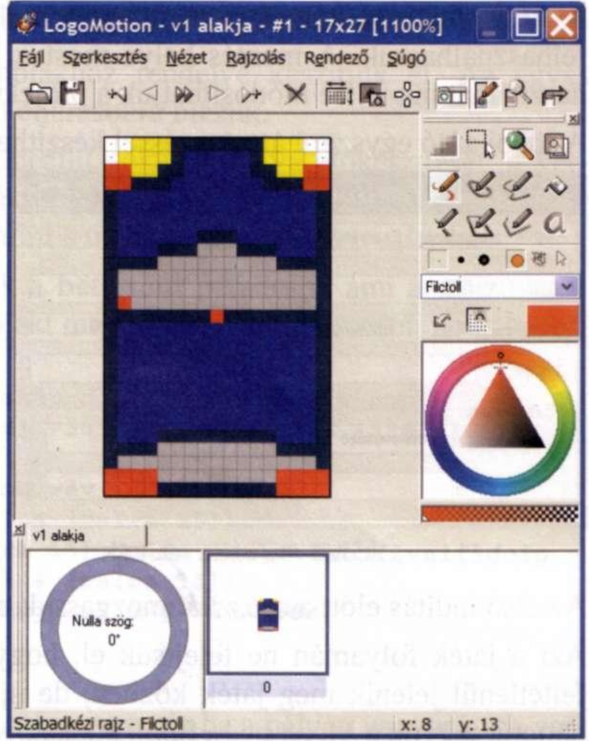
A *szalag2* teknőcöt hasonlóan készítjük el, de esetében a tollszínt fehérnek választjuk.

A versenyautókat v1, v2, v3 néven hozzuk létre. Ezek esetében is felemeljük a tollat, az ALAK fülön **Vonszolható** és **Érzékeny** állapotúvá tesszük, és a POZÍCIÓ fülön **Visszapattan** tulajdonsággal ruházzuk fel. Ez utóbbi azért lényeges, mert így a versenyző nem tud kilépni a területből, de nem is ragad le a pálya szélére érve.




v1 (teknőc) változtatása
Alapok | Alak | Rajz | Pozíció | Események | Változók | Eljárásc
Pozíció
Pozíció: X: -374 Y: -5
Apoz: X: 24 Y: 254
Irány!... 0
Alaphelyzet
Pozíció: X: -376 Y: -12
Irány!... 0 Aktuális!
Tartomány
Körkörös
Téglalap:
Ablak
Bekerített
Visszapattan

A versenyautók azonos alakúak, de más színűek. A kék kocsi elkészítését kísérjük nyomon!

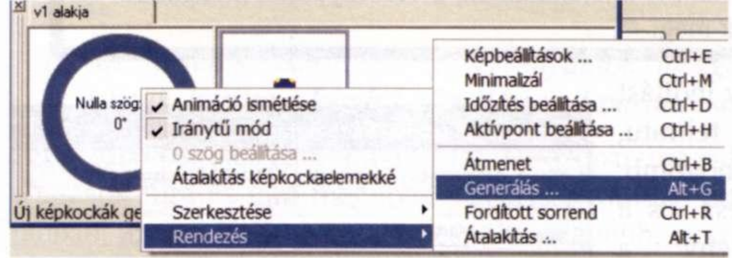



A teknőc előnézeti képére kattintva indítsuk el a LogoMotiont! Elsőként használjuk az eszköztár Képbeállítások ikonját! Az autó mérete legyen 17×27 képpont, a többi jellemzőjét ne módosítsuk.

Az első képfázis teljesen töltse ki a területet. Ha mégsem, akkor a Kép minimalizálása ikonnal hagyjuk el a felesleges részeket, majd az Aktívpont ikonnal jelöljük meg a (8;13) pontot, azaz nagyjából a kocsi közepét.

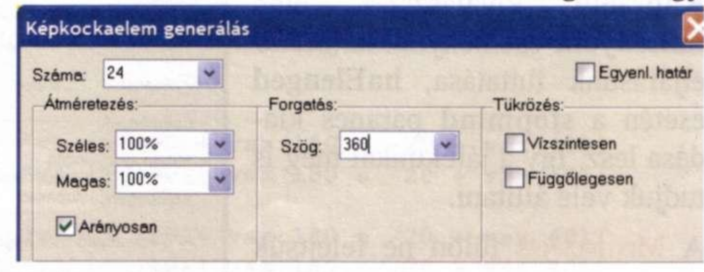
A szerkesztés közben a munkaterületen egérgörgetéssel tudunk nagyítást, kicsinyítést elérni.

Ha elkészítettük a 0 szöghöz tartozó képkockát, akkor itt az ideje a többi irányhoz tartozó elkészítésének is. Ehhez a bal oldalon Képkocka rendezése nézeten jobb gombbal kattintva megjelenő menüben a Rendezés almenüjében a Generálás… pontot válasszuk ki.




Mint a képen is látható, ez a menüpont azonnal is megnyitható lett volna az ALT-G gyorsbillentyűvel.

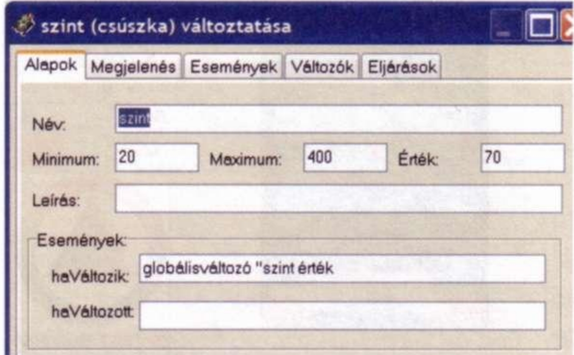
15 fokonként akarjuk a kocsikat forgatni. Így 24 képkockát kell generálnunk a teljes 360 fokos körbeforduláshoz.




Vegyük észre, hogy így összesen 25 képkocka lesz, mivel az utolsó generált elem iránya éppen megegyezik az eredetivel, így azt töröljük ki. Megtehettük volna azt is, hogy csak 23 képkockát generálunk, de akkor csak 345 fokos elfordulás kell.

A kész versenyautót el is menthetjük fájlba, hogy később más játékban is felhasználhassuk. A mentés helye most a KEPSOR mappa lenne, amit most használhatunk is, de módosíthatjuk a projekt mappájára.

A többi autó egyszerű átszínezéssel készíthető el, a kékkel azonos módon.




A pályára még felkerülhet egy *szint* nevű csúszka.

Az ALAPOK fülön adjuk meg a 20 **Minimum**, 400 **Maximum** és 70 kezdő **Érték** nagyságát, valamint a **haVáltozik** sorában az ugyancsak *"szint* nevű globális változó beállítására vonatkozó parancsot a **haVáltozik** sorába:

```
globálisváltozó "szint érték
```

Az első indítás előtt a csúszkát mozgassuk meg, hogy a változó kapjon értéket.

Azt a játék folyamán ne felejtsük el, hogy a *"szint* változó módosítása nem feltétlenül jelenik meg játék közben, de újraindítás után biztosan. A csúszka hátterét tegyük a MEGJELENÉS fülön átlátszóvá, és a hosszát 240-re állítsuk be.

> Sajnos az Imagine sem mentes kisebb hibáktól. Ez egyik ilyen hiba a nyomógombok és a **gombnyomás**? parancs együttes használatánál léphet fel. Ha van nyomógombunk az alkalmazásban, akkor a **gombnyomás?** nem működik megfelelően, csak **gombmenü!** használható együtt vele hiba nélkül. Erre majd figyelünk is a játékunkban, a versenyautók mozgatását ez utóbbi paranccsal oldjuk meg.

A pálya utolsó eleme egy indítást lehetővé tevő Start feliratú, *startgomb* nevű nyomógomb. Ennek felirata Indítás lesz, és a hozzá rendelt esemény a **Kapcsoló** kiválasztása után **haLenyom** eseménynél az ***indító*** eljárásunk futtatása, **haElenged** esetén a **stopmind** parancs kiadása lesz. Így a játékunkat meg is tudjuk vele állítani.

A MEGJELENÉS fülön ne felejtsük el a gomb szélességét legalább 50 képpontnyira megnövelni.

## *Az eljárások*

*A* program összes eljárását a főablakhoz rendelve készítjük el. Az egyes részfeladatok megvalósításait külön eljárásokra bízzuk.

Az ***indító*** eljárásunk végzi az alapbeállításokat: a pálya előkészítését, a versenyautók elhelyezését, az indítási hang és a késleltetés megvalósítását, a gombvezérlések megadását, valamint a mozgások indítását.

A háttérkép betöltésénél használjuk a **betöltőút** parancsot, ami azt biztosítja, hogy nem a KÉPSOR, hanem a projekt mappájában keresi az adott nevű képet.

```
eljárás indító
 törölháttérkép betöltháttérkép betöltőút "8palya.bmp
 vl'xypoz! -374 -10 v2'xypoz! -355 -10 v3'xypoz! -336 -10
 kér [vl v2 v3] [irány! 0]
 lejátszikfáj 1 betöltőút "dob.wav
 vl'gombmenü! [s [jobbra 15] a [balra 15]]
 v2'gombmenü! [jobbra [jobbra 15] balra [balra 15]]
 v3'gombmenü! [6 [jobbra 15] 4 [balra 15]]
 minden :szint [vl'megy v2'megy v3'megy ütközés]
 körszám 0 0 0
vége
```

A kocsik mozgását a ***megy*** eljárás határozza meg: ha a pályán vagyunk - a versenyautó alatti **pontszín** fekete - , akkor 5, ha más színű, akkor 2 képpontot haladunk előre. Ezzel oldjuk meg azt az elvárást, hogy a pályán gyorsabban haladjanak a versenyautók.

```
eljárás megy
 hak pontszín = "fekete [előre 5][előre 2]
vége
```

*16. Mi történik akkor, ha a versenyautók mozgását nem külön-külön kérjük, hanem a **kér fvl v2 v3][megy]** paranccsal?*

*A* **minden** parancs hatására a *"szint* változóban megadott időközönként ezek az eljárások újra és újra meghívódnak. Ha a változó értéke kisebb, akkor gyakrabban, azaz a játék gyorsabb lesz, ha nagyobb, akkor ritkábban. Az idő itt is ezredmásodpercben értendő.

Az elindított folyamatok utolsó eleme az ***ütközés*** eljárásunk.

```
eljárás ütközés
 figyelj "vl
  ha takar? "v2 [kér "v2 [j -90 + vsz 180 e -20 + vsz 40]]
  ha takar? "v3 [kér "v3 [j -90 + vsz 180 e -20 + vsz 40]]
 figyelj "v2
  ha takar? "vl [kér "vl [j -90 + vsz 180 e -20 + vsz 40]]
  ha takar? "v3 [kér "v3 [j -90 + vsz 180 e -20 + vsz 40]]
 figyelj "v3
  ha takar? "vl [kér "vl [j -90 + vsz 180 e -20 + vsz 40]]
```

```
  ha takar? "v2 [kér "v2 [j -90 + vsz 180 e -20 + vsz 40]]
vége
```

Az eljárásban figyeljük meg a **figyelj** és a **kér** parancsok hatókörének használatát! A **figyelj** folyamatos aktívvá tevést jelent, míg a **kér** csak időlegesen, a megadott műveletsor elvégzéséig aktivizálja a teknőcöt.

*17. Emeljük ki az **ütközés** eljárásból a közös elemeket egy új eljárásba! Alakítsuk át úgy az ütközési eseményt, hogy az autók ne ennyire véletlenszerűen mozduljanak el!*

Az egyik legösszetettebb eljárásnak a ***körszám*** eljárásunk látszik, ami a megtett köröket figyeli úgy, hogy minden egyes szalagon való áthaladáskor megnézi, hogy előtte melyiken ment át, és ha a másikon, akkor eggyel megnöveli a versenyző szalagátlépéseinek darabszámát.

Figyeljük meg azt a megoldást, ahogy a szalagokon való felváltva történő áthaladást ellenőrizzük! Például a *v1* versenyző *db1* változóját csak akkor növeljük a sárga *szalag1* teknőcön való áthaladáskor, ha a *db1* páros, azaz utoljára a fehér *szalag2-n* ment át.

```
eljárás körszám :db1 :db2 :db3
 ha :db1 = 6 [xypoz! -300 0 címke "|kék versenyző nyert!|
              irány! 0 e 20 stopmind]
 ha :db2 = 6 [xypoz! -300 0 címke "|zöld versenyző nyert!|
              irány! 0 e 20 stopmind]
 ha :db3 = 6 [xypoz! -300 0 címke "|sárga versenyző nyert!|
              irány! 0 e 20 stopmind]
 kér "v1 [ha (és takar? "szalag1 páros? :db1)
             [körszám :db1+1 :db2 :db3]]
 kér "v1 [ha (és takar? "szalag2 nem páros? :db1)
             [körszám :db1+1 :db2 :db3]]
 kér "v2 [ha (és takar? "szalag1 páros? :db2)
             [körszám :db1 :db2+1 :db3]]
 kér "v2 [ha (és takar? "szalag2 nem páros? :db2)
             [körszám :db1 :db2+1 :db3]]
 kér "v3 [ha (és takar? "szalag1 páros? :db3)
             [körszám :db1 :db2 :db3+1]]
 kér "v3 [ha (és takar? "szalag2 nem páros? :db3)
             [körszám :db1 :db2 :db3+1]]
 körszám :db1 :db2 :db3
vége
```

A ***körszám*** eljárásban felhasználtuk a ***páros?*** nevű, általunk készített függvényt, ami egyszerűen a körök párosságát figyeli.

```
eljárás páros? :n
 eredmény (maradék :n 2) =0
vége
```

A ***körszám*** eljárás egyszerűbbé is tehető, ha kihasználunk pár specialitást.

*18. Tegyük egyszerűbbé a **körszám** eljárást! Használjuk ki a **kér** helyett a **figyelj** használatának lehetőségét, egyszerűsítsük az eredménykiírást!*

A körszám eljárás három paraméterrel, ***körszám 0 0 0*** paranccsal hívódik meg. Az eljárás feleslegesen terheli a memóriát, hiszen rekurzívan folyamatosan hívja önmagát. Ez a hívás ugyan jobb- vagy másképp végrekurzív, azaz nincs utána már visszatérés a hívó eljáráshoz, de akkor sem túl szép.

*19. Alakítsuk át úgy a **körszám** eljárást, hogy nem hívja meg magát feleslegesen, illetve úgy, hogy globális változókat használjunk a megtett körök számának figyelésére!*

*20. Alakítsd át úgy a programot, hogy a szalagok helyett a pályára rajzolt fehér és sárga területen való áthaladást figyeljünk!*

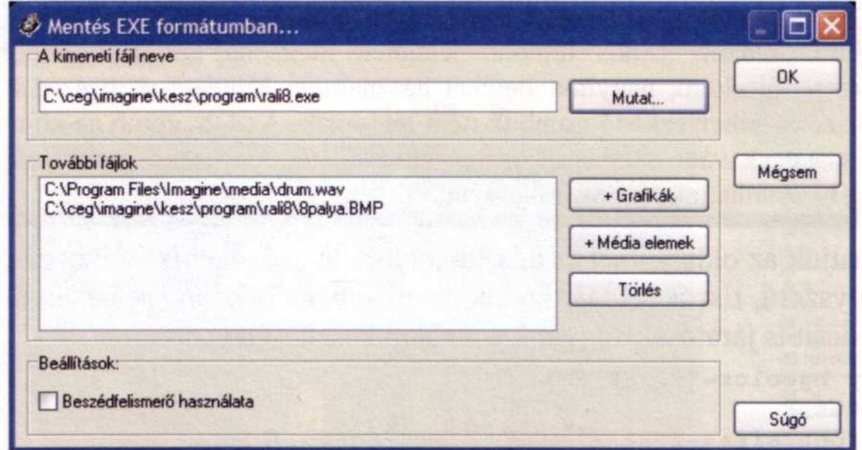
*21. Egészítsük ki a programot úgy, hogy egy negyedik, egér bal és jobb gombjával irányított versenyző is színre léphessen!*

*22. A megtett a játék állásáról adjunk folyamatos tájékoztatást!*

## Önálló program készítése

Az elkészült programot elmenthetjük önállóan futó, EXE kiterjesztésű alkalmazásként, de módunk van webprojekt formájában is menteni, amit a leggyakrabban használt böngészőkkel tudunk megtekinteni.

A FÁJL menü MENTÉS EXE FÁJLKÉNT... pontját választva nyithatjuk meg az alábbi párbeszédablakot.

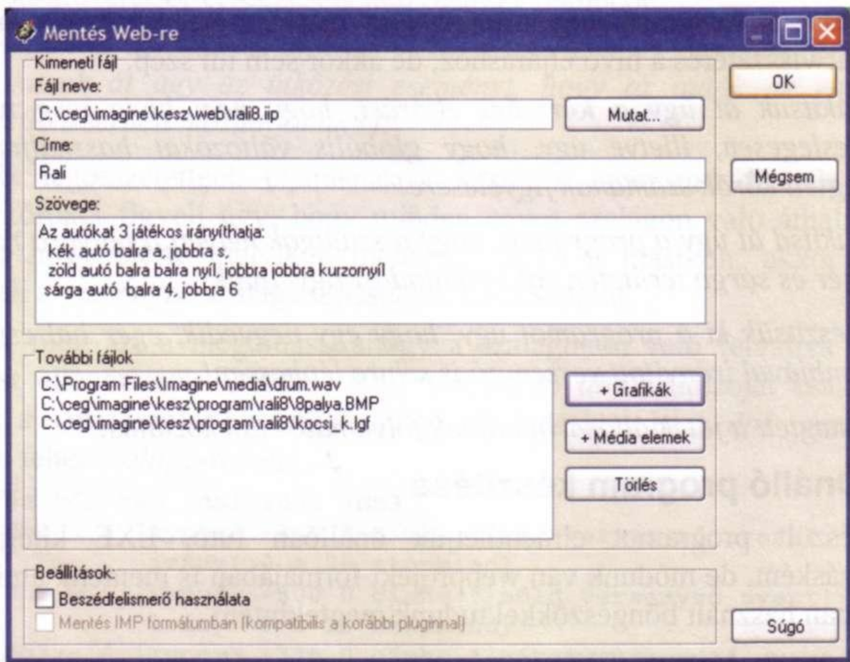



A MUTAT... gombra kattintva nyílik meg az a párbeszédablak, ahol az alkalmazás helyét adhatjuk meg.

A **További fájlok** ablakrészben nem minden esetben jelenik meg az összes programunkban felhasznált grafikai és média elem. Ha ez a lista hiányos, akkor a teknőcruhákat, háttérképeket a +GRAFIKÁK gombra kattintva vehetjük

fel, a hangot, videót pedig a +Média elemek gombra kattintva. Ha használni akarunk hangparancsot is, akkor ne felejtsük el a **Beszédfelismerő használata** kapcsolót bejelölni.

A Fájl menü Webprojekt készítése... pontja hasonló felépítésű.



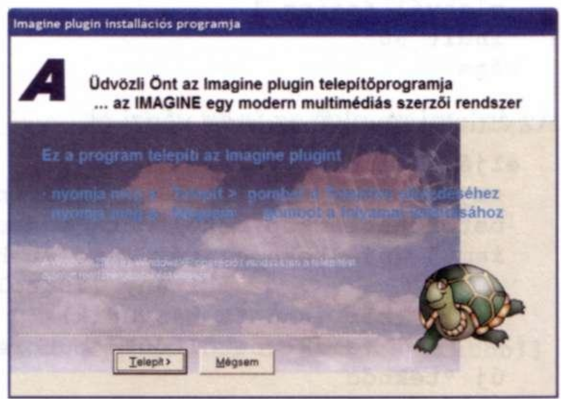

A hely és a már korábban látott elemek megadásán túl lehetőségünk van a létrehozott IIP kiterjesztésű állományt elindító HTML oldalra is írnunk.

A fenti ablakok sajnos hibásan jelennek meg, ha az alapértelmezett betűmérettől eltérő, nagyított betűket használunk. Mindkét párbeszédablak jobb szélén elhelyezkedő gombok nem látszanak. Az OK gomb az ablakok szélességének változtatásával viszont előcsalható. A Mégsem gomb helyett pedig használhatjuk az ESC billentyűt.

Megadhatjuk az oldal címét és az oldalra kerülő szöveget is. Mivel ezek a beírások egyszerű, tiszta HTML kódok, ezért utólag is könnyen belenyúlhatunk, ha egy kicsit is járatosak vagyunk a weblap készítésében.

```
<body bgcolor="#FFFFFF">
<center>
<b><font size=+2>alkalmazás címe</font></b><p>
<font size=-1>ismertető szövege</font><p>
<embed src="prognév" align="middle" border="0"
width="szélesség" height="magasság"
...
```

Ne felejtsük el, a böngészők alapértelmezetten nem alkalmasak az Imagine webprojektek futtatására, ezért az első alkalommal mindenképpen telepítenünk kell az ActiveX-vezérlő futtatásának engedélyezése mellett a kért plugint a http://imagine.elte.hu/plugin lapról.




A 2,21 MB méretű fájl időnként frissülhet, ezért célszerű az oldalra ellátogatni később is.

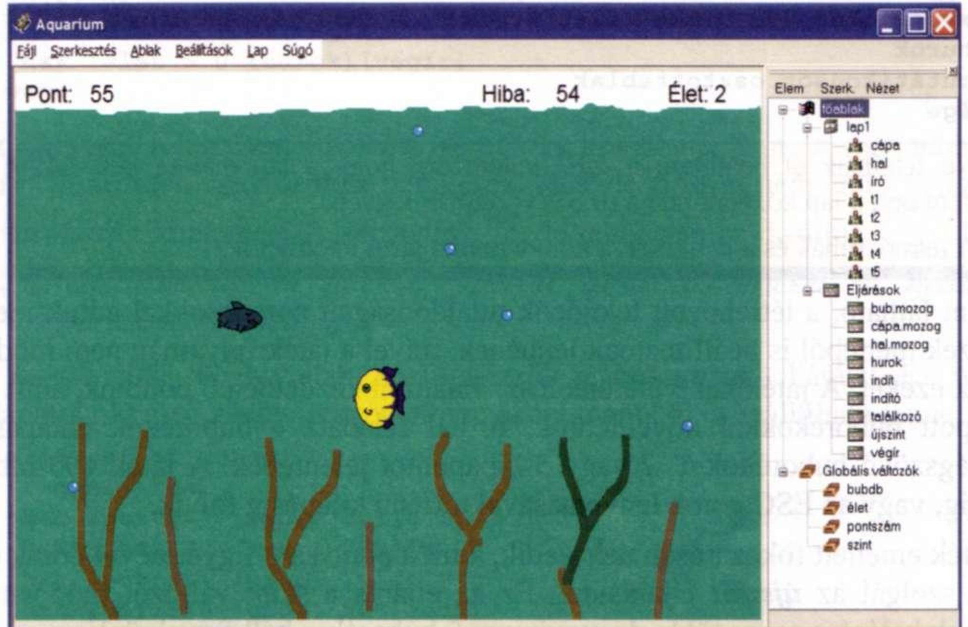
A webprojekt indítása és kezelése csak akkor lehetséges, ha abba először belekattintunk.

Célszerű éppen ezért az indító eljárást kattintás vagy gombnyomás figyeléssel ellátni.

## Cápa az akváriumban

E játékunk az eddigiekben látott programozási stílustól teljesen eltérően készült el. Azt is mondhatnánk, hogy a klasszikus LOGO-beli programfejlesztést mutatja be. Éppen ezért érdemes a leírásban feladott átalakításokat megvalósítani az Imagine tényleges lehetőségeinek jobb megismerése végett.

12. *Készítsünk játékot, amelyben egy akváriumban úszó halat kell irányítanunk úgy, hogy elkerüljük a minket üldöző cápát, és minél több vízből fellebegő buborékot kelljen összegyűjtenünk.*



A programot ***indító*** eljárás a kezdő paraméter átadása miatt az ***indít*** nevűt hívja meg, így lesz értéke a kezdetben a ***szint*** változónak.

```
eljárás indító
 globvál "szint 1
 indít 50
vége
```

Az érdemi munkát az ***indít*** végzi el.

```
eljárás indít :várakozás
 rejtikonsor rajzlapablak bezárintéző
 betölthättérkép betöltőút "medence
 ism 5 [új "teknőc [név (szó "t hányadik) poz [10 10]
                    irány 270 toll tf látható igaz]]
 ism 5 [kér (szó "t hányadik)
         [alak! betöltőút "buborék xypoz! 50 + vsz 700 5]]
 új "teknőc
     [név cápa poz [400 20] irány 90 toll tf látható igaz]
 figyelj "cápa alak! betöltőút "capa
 új "teknőc
     [név hal poz [400 400] irány 90 toll tf látható igaz]
 kér "hal [ alak! betöltőút "hal
      gombmenü! [balra [irány! 270] jobbra [irány! 90]
                 fel [irány! 0]         le [irány! 180]
                 esc [végír]]]
 új "teknőc [név író irány 0 toll tf látható hamis]
 író'betűtípus! [[Arial][14 400 0 10 238]]
 globvál "pontszám 0 globvál "bubdb 0 globvál "élet 3
 figyelj "író
 xypoz! 10 530 címke "Pont: xypoz! 700 530 címke "Élet:
 xypoz! 750 530 címke :élet xypoz! 500 530 címke "Hiba:
 hurok
 mutatikonsor osztottablak
vége
```

Ne felejtsük el beállítani a *lap1* jellemzőit, hogy a lap mérete 800×540 képpont, a lap középpontja a (0;539) pontban legyen.

A teknőcruhák és a háttérkép a könyv honlapjáról letölthetők.

Amint látható, a létrehozott teknőcök tulajdonságait parancsokkal adtuk meg, de ezek menüből is beállíthatóak lennének, mivel a játék folyamán nem módosítjuk ezeket. A játékban 3 életünk lesz, valamint kezdetben 0 pontunk, amit az elkapott buborékokkal növelhetünk. A hal feladata a buborékok elkapása. A megszökő buborékok 1, 2 vagy 3 hibapontot jelentenek. A játék 100 hibapontig, vagy az ESC gomb lenyomásával történő kilépésig folyik.

A játék emellett fokozatosan nehezedik, amit a pontszám figyelésével érünk el. Erre szolgál az *új szint* eljárásunk. Ez az eljárás a szint változót módosítja, aminek hatására egyre több, de maximum 5 buborékra kell figyelnünk.

```
eljárás újszlnt
 elágazás :pontszám
   [5 [globvál "szint 2]
    15 [globvál "szint 3]
    30 [globvál "szint 4]
    50 [globvál "szint 5]]
vége
```

Az eljárás hatását a ***bub.mozog*** eljárásban láthatjuk, a buborékok számát adja.

```
eljárás bub.mozog
 ism :szint
   [figyelj (szó "t hányadik)
     irány! 45 - vsz 90
      e 2 + vsz 3
      ha ypoz > 535
         [növel "bubdb figyelj "író xypoz! 580 530 címke :bubdb]]
vége
```

Érdemes megfigyelni a buborék életszerű mozgását megvalósító programrészt. Itt oldjuk meg az elszökő buborékok figyelését is.

Az eljárások meghívását is klasszikus módon végezzük el. A ***hurok*** nevű eljárásunk hívja a mozgató és a vizsgálatokat végző eljárásokat.

```
eljárás hurok
 hal.mozog
 cápa.mozog
 bub.mozog
 találkozó
 várj :várakozás
 újszint
 ha :bubdb > 100 [végír]
 hak :élet > 0 [hurok][végír]
vége
```

Az önmagát rekurzívan hívó ***hurok*** eljárás kiváltható lenne a már többször használt **minden** vagy örökké parancsokkal is, de most ez nem volt a célunk.

A ***várakozás*** paramétert itt használjuk fel, de sehol sem módosítjuk.

*23. Nehezítsük úgy a játékot, hogy a várakozás értékét az újszint eljárásban módosítjuk!*

A ***hurok*** által meghívottak közül nézzük a ***hatmozog*** eljárást! Ez olyan feladatot old meg, amely teknőcjellemzőként menüből is beállítható lenne.

```
eljárás hal.mozog
 figyelj "hal
 hak xpoz<20  [xpoz! 20] [e 2]
 hak xpoz>780 [xpoz! 780][e 2]
 hak ypoz<20  [ypoz! 20][e 2]
 hak ypoz>520 [ypoz! 520][e 2]
vége
```

*24. Helyettesítsük a területmegadást a **hal** jellemzőinek menüből történő beállításával, vagy a* **tartomány** *parancs használatával!*

A cápa mozgása jóval egyszerűbb, mivel annak csak követnie kell a halunkat.

```
eljárás cápa.mozog
 figyelj "cápa
 irány! irányszög kér "hal [poz]
 e 1
vége
```

A találkozások vizsgálatát és az elkapott buborékok helyett új létrehozását végzi a ***találkozó*** eljárás. Az eljárás emellett kiírást is végez.

```
eljárás találkozó
 ism :szint
   [figyelj "hal
    ha takar? (szó "t hányadik)
       [növel "pontszám
        figyelj "író xypoz! 80 530 (címke :pontszám)
        figyelj (szó "t hányadik) xypoz! 50 + vsz 700 5
        hangsor [S0 1122 T120 L4 02 G 16B]]]
 figyelj "cápa
  ha takar? "hal [xypoz! vsz 700 vsz 400 ( növel "élet -1 )
 figyelj "író xypoz! 750 530 (címke :élet) hanghullám "jaj]
vége
```

*25. Amikor a hal elkap egy buborékot, akkor megszólal egy hangsor. Ez megállítja a játék futását. Oldjuk meg, hogy a játék továbbfusson!*

Már csak a ***végír*** eljárásunk maradt ki, amely a játék befejezésekor kap szerepet.

```
eljárás végír
 figyelj "író
 betűtípus! [[Arial][80 700 10 0 238]]
 xypoz! 150 300
 tsz! 12
 címke "VÉGE!
 stopmind
vége
```

Ez a programrészlet is fejleszthető!

*26. Oldjuk meg, hogy a VÉGE felirat fokozatosan megnőve jelenjen meg a képernyő közepén! Figyeljünk arra, hogy a* **címke** *parancs az író teknőc pozíciójában kezdi a kiírást, tehát azt is mozgatnunk kell.*

Utolsó feladatunk a meghívási módokra hívja fel a figyelmet.

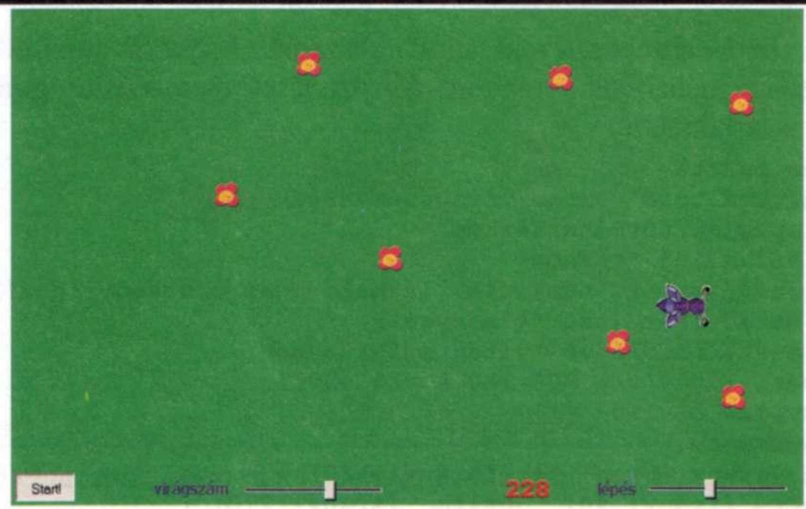
*27. A* **figyelj** *és* **kér** *parancsokkal szólítjuk meg szinte mindenhol a teknőcöket. írjuk át a programot úgy, hogy csak ott használjuk ezeket, ahol ténylegesen nem elkerülhető!*

## *Szedjünk virágot!*

A következő játékunk a beállítások gyakorlását szolgálja. Éppen ezért e játék esetén sem írjuk le a kiválasztandó elemek pontos helyét, de a beállítandó paramétereket pontosan megadjuk.

13. *Készítsünk virágszedő méhecskés játékot! A méhecskét a kurzornyilakkal lehessen irányítani, a virágok száma 1 és 20 között állítható legyen, a rendelkezésre álló idő 100 és 1000 lépés közötti időtartam legyen!*

> A programban használt JAJ és HURRA hangokat, valamint a VIRAG.LGF fájlt a könyv honlapjáról le lehet tölteni, vagy más elemmel helyettesíthető.




A *lap1* nevű játéklap méretét állítsuk be 799×499 képpontra, a kitöltésének színét zöldre, a lap középpontját a (398;249) pontra.

Hozzuk létre a Start feliratú *start* gombot. A gomb lenyomásra indítsa el az ***indító*** eljárást! A szélességét ne felejtsük el legalább 60 képpontra állítani!

Helyezzünk el egy csúszkát a *lap1*-re, amit nevezzünk el *virágszám*-nak. A csúszka 1 és 20 között adjon értéket, a kezdőértéke 10 legyen! Ha a csúszkát mozgatjuk, akkor hajtsa végre a **globálisváltozó "darab érték** parancsot. Próbáljuk is ki, hogy a ***darab*** változó létrejöjjön! A MEGJELENÉS fülön tegyük **Átlátszóvá** is! Írjuk ki mellé egy szövegdobozba a „virágszám" szöveget, 10 pontos, kékszínű Tahoma betűkkel.

Hasonlóan készítsük el a *„lépés"* kiírást tartalmazó szövegdobozt, és mellé a ***lépés*** változót beállító *lépés* nevű csúszkát is! A csúszka a változó értékét 100 és 1000 között állítsa, 400 kezdőértékkel! A csúszka változtatásakor végrehajtandó parancs **globálisváltozó "lépés érték** legyen! Az átlátszóság beállítását ne felejtsük el! Próbáljuk is ki, hogy a ***lépés*** változó létrejöjjön!

Az alapértemezett *tl* teknőcöt nevezzünk el zró-nak! A teknőcöt helyezzük el a (100;-217) pontba, nézzen észak felé, ne legyen látható, de a tollát tartsa lenn. A **R**AJZ fülön állítsunk be piros tintaszínt és 14 pontos Arial Black betűt!

Hozzunk létre egy *méh* nevű teknőcöt is! A teknőc vegye fel a BOGI2 alakot a KÉPSOR mappából, emeljük fel a tollát, és tegyük láthatóvá. Az ALAK fülön tegyük érzékennyé! Az Események fülön a **haÜtközik** esetén adjuk ki a találkozást feldolgozó **kér takart [elrejt] globálisváltozó "darab :darab-l** parancsot! A Pozíció fülön, a **Tartomány** részen a mozgást megadó téglalap bal felső sarka legyen a (-390 ; 245) pont, a szélessége 780, magassága 470, a stílusa visszapattanó!

Mindezen előkészületek után a *főablakon* hozzuk létre az eljárásokat!

Az *indító* eljárás létrehozza a kívánt darabszámú virágot, és teszi el a kezdőpozíciójába a méhecskét, valamint a vezérlését megvalósító **gombmenü!** parancsot is tartalmazza.

```
eljárás indító
 globálisváltozó "darab virágszám'érték
 rajzlapablak törölháttérkép
 ism :darab [új "teknőc
       [név (szó "v hányadik) látható igaz érzékeny igaz]]
 ism :darab [kér (szó "v hányadik)
   [tf alak! betöltőút "virág mutatteknőc
   xypoz! 350 - vsz 700 240 - vsz 420]]
 méh"xypoz! 0 -150
 méh'gombmenü!
   [fel [irány! 0]    le  [irány! 180]
    jobbra [irány! 90]   balra  [irány! 270]
    esc [gombmenü! [] stopmind]]
 játék :lépés
vége
```

A hasonlóan egyszerű ***játék*** eljárás a méh mozgását figyeli, és elvégzi a szükséges eseményeket.

```
eljárás játék :idő
 méh'e 5 várj 30
 ha :idő<0 [hanghullám "jaj.wav várj 2000
           ism 20
              [ kér (szó "v hányadik) [elrejt]] stopmind]
 ha :darab = 0 [hanghullám "hurra.wav stopmind]
 kér "író [címke (szó :idő "|   |)]
 játék :idő - 1
vége
```

Vegyük észre, hogy a kiírásra csak akkor kerül sor, ha az azt megelőző feltételek egyike sem teljesült, azaz nem fogyott el az idő és még van virág is.

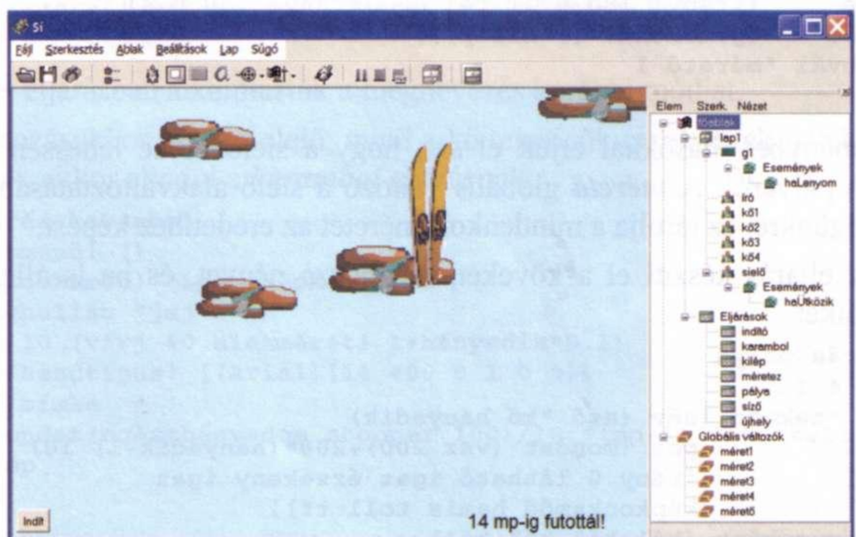
*28. Készítsük el a leírt elemeket programból!*

## *Síeljünk!*

Következő játékunkat úgy készítjük el, hogy az összes objektum létrehozását és jellemzőinek beállítását a programban paranccsal végezzük el. Ez alól egyetlen kivétel lesz, a játékot indító gomb elhelyezése a képernyőn.

A játékban felhasznált KO és SI képeket a könyv honlapjáról tölthetjük le.

*14. Készítsünk egy hegyoldalon lecsúszó síelést utánzó játékot! A játékban mérjük az eltelt időt! A kövek mérete változzon, ahogy közelednek!*




Az ***indító*** eljárásunk végzi el a képernyő beállításait, majd meghívja a ***pálya*** és a ***síző*** eljárásokat, amelyek a mozgó objektumokat hozzák létre. A programban még használjuk az *író* nevű teknőcöt is a kért időkijelzésre. A játék egy **hangsor** paranccsal megadott dallam után indul, és persze a stoppert is nulláznunk kell.

```
eljárás indító
 törölképernyő teljesképernyő rejtikonsor bezárintéző
 kér "lap1 [méret! [796 540] kiindulópont! [0 539]
            háttérszín! "ibolya11]
 törölobjektum mindenteknőc
 pálya
 síző
 új "teknőc [név író poz [550 30] irány 0 látható hamis toll tl]
 hangsor [S0 I60 T120 L4 O2 16C 16E 16C 16E 16C
          16E 16C 16E 16C 16E 16C 16E 16C 16E 16F 16G 2B]
 várj 500
 nullázstopper
 minden 10 [
  kér "sielő [ e 1 méretez -0.1]
  kéregyenként [kő1 kő2 kő3 kő4] [e 2 méretez 0.2 újhely]]
vége
```

A síző létrehozásáért a ***síző*** eljárás felelős.

```
eljárás síző
 új "teknőc [név sielő poz [400 450] irány 180
             látható igaz érzékeny igaz toll tf
             képkockamód hamis haütközik karambol]
 kér "sielő [
  alak! betöltőút "si
  tartomány! [[50 500] [790 10]] tartománystílus! "visszapattan
  gombmenü! [
   jobbra [irány! 90 e 3]  balra [irány! 270 e 3]
   le     [irány! 180 e 1] fel   [irány! 180 h 4]
   esc    [gombmenü! [] kilép]]]
 globvál "mérető 1
vége
```

A tartománybeállításokkal érjük el azt, hogy a sielővel ne lehessen kilépni a látható pályáról. A ***mérető*** globális változó a sielő alakváltoztatásában lesz a segítségünkre, ez tárolja a mindenkori méretét az eredetihez képest.

A ***pálya*** eljárás készíti el a köveket, méghozzá négyet, és be is állítja a fő jellemzőiket.

```
eljárás pálya
 ism 4 [
  új "teknőc [név (szó "kő hányadik)
              poz (mondat (vsz 200)+200*(hányadik-1) 10)
              irány 0 látható igaz érzékeny igaz
              képkockamód hamis toll tf]]
 kéregyenként [kői kő2 kő3 kő4]
  [alak! betöltőút "ko
  globvál (szó "méret utolsó kiaktív) 0.5 + 0.1*vsz 10
  alakméret! (értéke (szó "méret utolsó kiaktív))
  tartomány! [[-120 500] [799 0]] tartománystílus! "nézőablak
  láthatótartomány! [[0 800] [799 0]]]
vége
```

A kövek nagyjából egyenletesen, de mégis véletlenszerűen jelennek meg a **poz** tulajdonság megadásánál alkalmazott kifejezésnek köszönhetően.

Érdemes figyelnünk a ***kő*** teknőcök általunk látható és mozgásuk számára rendelkezésre álló tartományainak beállítására. Erre azért van szükség, hogy a pálya tetején kicsúszó kövek fokozatosan tűnjenek el, ne azonnal, amikor a tollpontjuk eléri a felső szélt, ugyanis a KO kép tollpontja a tetején található.

Külön felhívjuk a figyelmet a méretjellemzőket tároló ***méreti, méret2,*** stb változók létrehozási módjára. Ezt a nevük utolsó betűjéből és a ***"méret*** szóból alakítjuk ki a **(szó "méret utolsó kiaktív)** paranccsal.

Figyeljük meg azt is, hogy ezt a szót az **értéke** paranccsal tudjuk felhasználni egy változó tartalmának lekéréséhez.

A kövek és a síelő méretmódosítását a ***méretez*** nevű eljárással valósítjuk meg.

```
eljárás méretez :nő
 alakméret! (értéke (szó "méret utolsó kiaktív))+:nő*(ypoz+500)/500
vége
```

A képből kisikló kövek új helyét az ***újhely*** eljárásunk állítja elő.

```
eljárás újhely
 ha ypoz >700
  [xypoz! ((utolsó kiaktív) - 1)*200 + vsz 200 -120 + vsz 120
   globvál (szó "méret utolsó kiaktív) 0.2 + 0.1*vsz 20
   alakméret! értéke (szó "méret utolsó kiaktív)]
vége
```

Mindkét eljárásban alkalmaztuk a megnevezés korábbi módját.

A létrehozásukkor mind a síelő, mind a kő teknőcök érzékenyek lettek, így ha ütköznek, akkor elindul a ***karambol*** eljárásunk.

```
eljárás karambol
 gombmenü! []
 képkockamód! "igaz képkocka! 1
 hanghullám "jaj
 ism 10 [várj 40 alakméret! 1+hányadik*0.1]
 író'betűtípus! [[Arial][14 400 0 10 0]]
 író1címke
   (mondat(egészhányados stopper 100)/10 "|mp-ig futottál!|)
 kilép
vége
```

Az eljárás egyrészt megnöveli a síelőt, majd elvégzi a futási idő kiírását, és végül meghívja a ***kilép*** eljárást, ami a környezet visszaállítását végzi el.

```
eljárás kilép
 várj 2000
 maxméret
 mutatikonsor
 stopmind
vége
```

Ne felejtsük el végezetül, hogy program újraindíthatósága érdekében vegyünk fel egy új gombot az ÚJ GOMB ikonra kattintva, Az **ALAPOK** fülön a **Felirat** legyen Indító, a **haLenyom** esemény legyen az *indító* eljárás meghívása. Célszerű még a **MEGJELENÉS** fülön a **MéretrSzéles** értékét 60-ra állítani.

*29. Egészítsük ki programunkat az indító gomb létrehozásával is!*

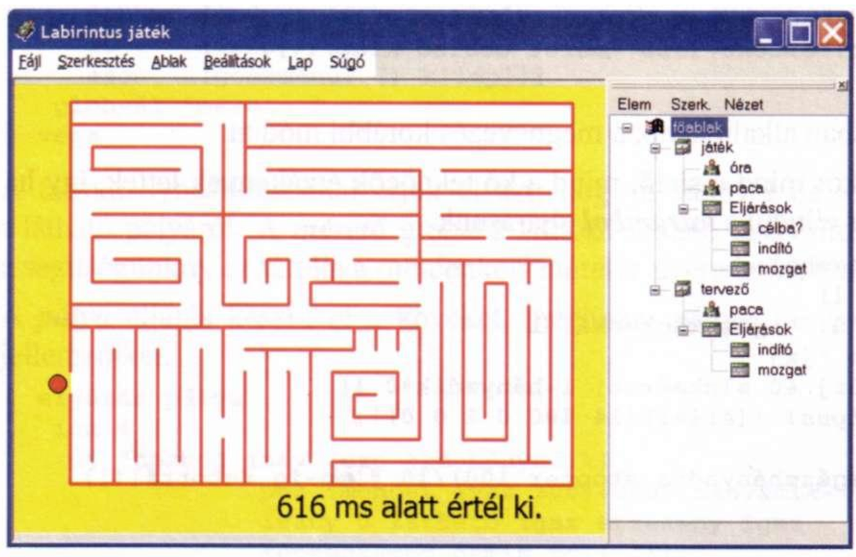
*30. A programot fejlesszük úgy tovább, hogy bizonyos időközönként jelenjen meg egy tárgy, például egy ajándékcsomag, amit ha felveszünk, plusz pontokat tudunk gyűjteni!*

*31. Tegyük lehetővé, hogy a játék ne érjen véget az első ütközéssel! Például időközönként kapjunk új életet, vagy eleve több legyen!*

## Menjünk végig a labirintuson!

*15. Készítsünk labirintust! A kész labirintusban haladjunk végig, miközben mérjük az eltelt időt!*

*A* játékot két lapon valósítjuk meg. Az egyik lapon - ezt *tervezőnek* nevezzük - a labirintust lehet készíteni, a másik lapon - a *játék* nevűn -, az elkészített labirintusban lehet mozogni.




### Tervező lap

Az ***indító*** eljárásunk nem a főablakhoz, hanem a laphoz tartozik. Az összes műveletet itt végezzük most el. Hogy futás közben ne zavarjon az Imagine környezet, az egyes kezelő elemeket parancs segítségével bezárjuk, majd meg is nyitjuk a munka végeztével.

```
eljárás Indító
 tervező'méret! [600 400]
 törölképernyő rejtikonsor rajzlapablak
 törölobjektum mindenteknőc
 új "teknőc
    [név paca poz [-100 -100] irány 0
    toll tl tsz 12 tv 3
    látható igaz alak (szó betöltőút "paca) képkockamód igaz]
 paca'mozgat
 mutatikonsor osztottablak
vége
```

Az eljárásban azt figyeljük meg, hogy a *paca* nevű teknőc szinte minden lehetséges tulajdonságát a létrehozásakor, eljárásban adjuk meg, nem párbeszédablakban. Érdemes megfigyelni a teknőc osztályba kerülő új objektum tulajdonságait, hogy milyen párok alkotják!

A létrehozott teknőcöt mozgatjuk a képernyőn. A mozgatáshoz a leütött billentyű kódját figyeljük. A szóközzel szabályozzuk a toll helyzetét.

```
eljárás mozgat
 elágazás olvaskőd
   [-72 [irány! 0 e 20]
    -80 [irány! 180 e 20]
    -75 [irány! 270 e 20]
    -77 [irány! 90 e 20]
    27 [stopmind]
    32 [hak toll = "tollatfel [képkocka! 1 tl][képkocka! 2 tf]]]
 mozgat
vége
```

A vezérlés megadásához a ComLOGO-ban is használható ASCII kódokból kialakított kódokat alkalmaztuk. Az ESC kódja a 27, a szóközé a 32, a kurzornyilak kódjai pedig rendre: a fel -72, jobbra -77, le -80, balra -75.

*32. Alakítsuk át úgy a **mozgat** eljárást, hogy a kódok helyett az Imagine-ben használatos megnevezések szerepeljenek!*

## A játék lap

A tervező lapon elkészített és elmentett pályánkon azonnal indulhatna a játék, de előtte az elkészített labirintuson végezzünk el egy kis módosítást: a külső részt fessük be sárgára a rajzeszközök segítségével!

Az *indító* eljárásunk a *játék* lapon az alapbeállítások elvégzését, és a mozgatás indítását oldja meg.

Elsőként a lap jellemzőit adjuk meg, a **méret!** paranccsal a tényleges méretét képpontban, a **kiindulópont!** parancs pedig a bal felső sarokhoz képest adja meg a lap középpontját.

A megoldás idejét is szeretnénk mérni, ezért szükségünk lesz egy stopperre, amelyet a tényleges játék megkezdése előtt - **paca'mozgat** - indítunk el.

```
eljárás indító
 játék'méret! [600 400] játék'kiindulópont! [299 199]
 rejtikonsor rajzlapablak
 betöltháttérkép betöltőút "labi2.bmp
 törölobjektum mindenteknőc
 új "teknőc
   [név paca poz [-229 168] irány 0 toll tf képkockamód igaz
    látható igaz alak (sző betöltőút "paca) vonszolható igaz]
 új "teknőc
   [név óra poz [-50 -165] irány 0 toll tl látható hamis]
 óra'betűtípus! [[Tahoma][16 400 0 0 0 238]]
 nullázstopper
 paca'mozgat
 mutatikonsor osztottablak
vége
```

A létrehozott teknőcök jellemzőit itt is érdemes megfigyelnünk. Az *óra* teknőc az időpont kiírását segíti majd, de magát a teknőcöt nem fogjuk látni a képernyőn.

A *paca* nevű teknőcöt - amely a játék lapon él és így nem azonos a tervező lapon megadott teknőccel - kell végig vinnünk a labirintuson. Mozgatása már Imagine környezetben a **gombmenü!** paranccsal szebben megoldható.

```
eljárás mozgat
 paca'gombmenü!
     [fel [irány! 0]
      le [irány! 180]
      jobbra [irány! 90]
      balra [irány! 270]
      esc [mutatikonsor osztottablak stopmind]]
 paca'előre 10
 ha paca'pontszín = "vörös [hátra 10]
 célba?
 várj 100
 mozgat
vége
```

Az irányítás már ismert, az újdonság a ***célba?*** függvény. Ez logikai értéket ad vissza.

```
eljárás célba?
 ha paca'pontszín = "sárga [kér "óra [címke (mondat maradék
stopper 1000 "|ms alatt értél ki.|)] átdob "felsőszintre]
vége
```

Az eljárás érdekessége, hogy ha a feltétel teljesül, akkor nem térünk vissza a meghívó eljáráshoz ***(mozgat),*** hanem a fő meghívó eljáráshoz az **átdob** alkalmazásával.

*33. Tegyük programunkat századmásodperc pontossá!*

Az elkészített pálya fontos eleme a start és a cél koordinátája, valamint maga a pálya. Ezeket most színekkel jelöltük, de megadhatjuk koordinátákkal is.

*34. Adjuk meg a célba? függvényt úgy, hogy a célterület koordinátáját ismerve határozzuk meg az eredményt!*

Ezeket az elemeket külön listában tároljuk el. A szerkezete ***[fstartx startyj fcélx cély]*** pályanév] legyen.

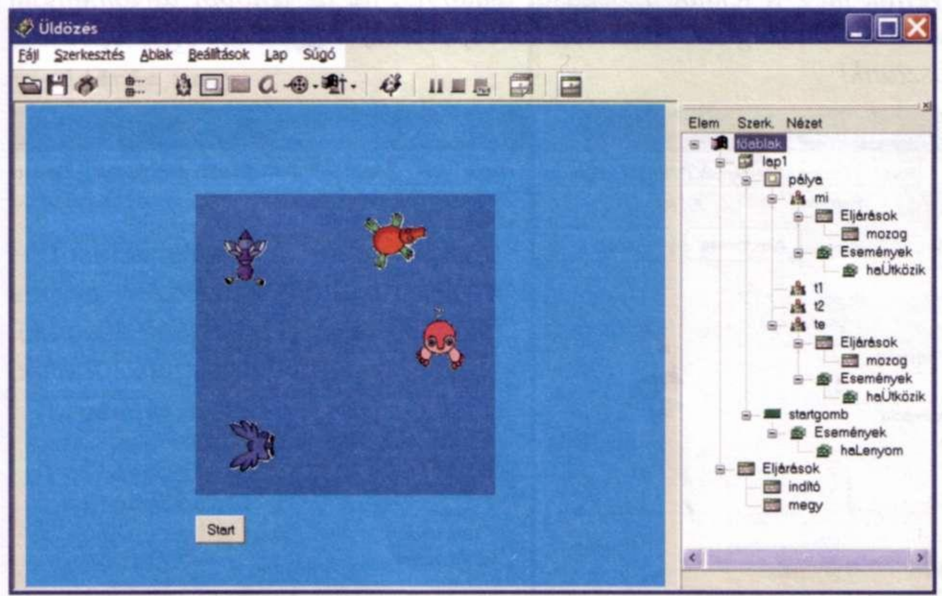
*35. Készítsük el a labirintus módosított változatát, ahol a tervező és szerkesztő nézet között válthatunk, valamint a fenti adatszerkezetben mentjük tervező nézetben a pályát, és a játék nézet a fenti adatszerkezetből veszi a pályát!*

*36. Készítsünk eredménylistát a feladat megoldásához!*

## Fogócska

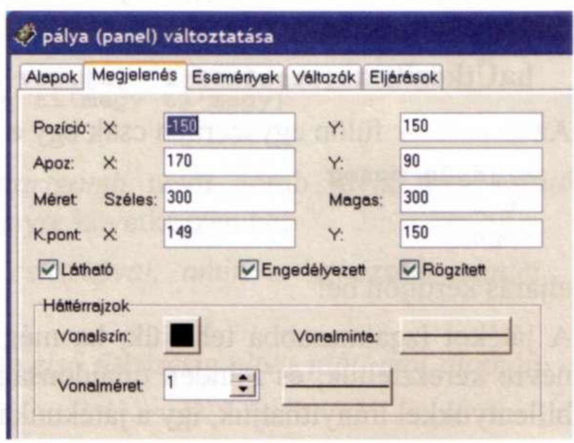
A panel használatának bemutatására egy egyszerű kis játékot készítünk el, ami jelenleg kettő, de többszemélyes is lehet.

Minden panel meghatároz egy területet a rajta élő objektumok számára: ez az a hely, ahol mozoghatnak, rajzolhatnak, vagy ahova nyomtathatnak.




*16. Adjunk meg egy 300×300 képpontos területet, ahol az általunk irányított figura ütközését kell elkerülnünk a területen mozgó többi figurával. Ha az ütközés bekövetkezett, akkor ezt hanggal jelezzük! A játék a Start gomb megnyomásával induljon el újra!*

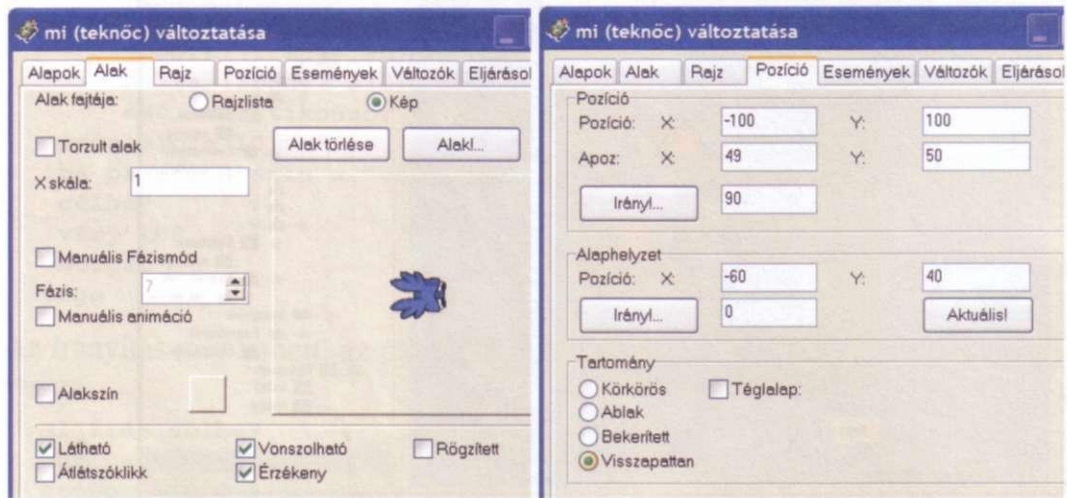
Elsőként a lap jellemzőit állítsuk be, ahol a pálya nevű panelt létrehozzuk! A *lap1* nagysága legyen 640×480 képpont, és az origó a (320;240) pontban legyen a lap közepén. A háttér színe legyen ibolya!




Az ÚJ PANEL ikonra kattintva hozzunk létre a *lap1*-enegy panelt, nevezzük el *pályá*-nak, a hátterét állítsuk be világoskéknek, majd MEGJELENÉS fülön állítsuk be a méretét 300×300-asra, a lapbeli pozícióját pedig (-150;150)-re.

A *mi* nevű teknőcöt létrehozásakor a panelre helyezzük, így nem a teljes lap, hanem csak a panel területe lesz a mozgástere. A panel határára érve a teknőc ne tudjon kilépni. Azt is szeretnénk, ha nem ragadna le, amit úgy tudunk elérni, hogy a Pozíció fülön a **Tartomány** részben **Visszapattan** tulajdonságot választjuk ki!

37. *Figyeljük meg a teknőc mozgását akkor is, ha az a többi tartomány beállítással mozoghat, azaz Körkörös, Ablak, vagy Bekerített opciót választunk!*




Ne felejtsük el az Alapok fülön **Tollatle** opciót kiiktatni, mert különben a teknőc mozgás közben nyomot hagy a panelen.

A *mi* teknőchöz egy eseményt is rendelünk: ha ütközik valamelyik másik teknőccel, akkor a JAJ hangot adja, és leállnak a teknőcök. Ezért az Események fülön beállítjuk, hogy:

```
haÜtközik lejátszikfájl "jaj.wav stopmind
```

Az Eljárások fülön egyszerűen csak egy **e 2** mozgást adunk meg:

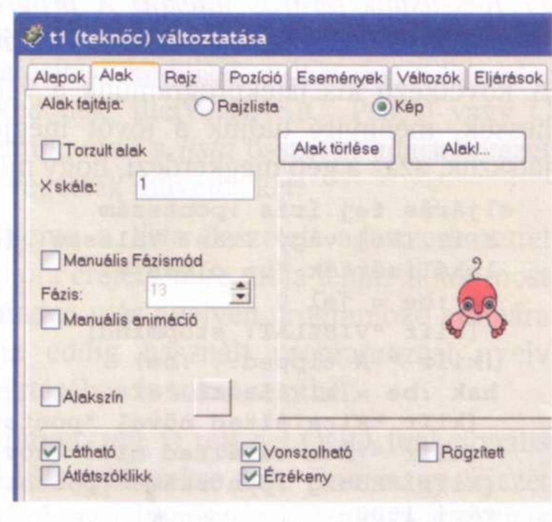
```
eljárás mozog
 e 2
vége
```

eljárás kerüljön be!

A játékot izgalmasabbá tehetjük, ha még egy teknőcöt irányíthatunk. Ezt *te* névre kereszteltük, és minden tulajdonsága azonos a *mi* teknőccel, csak más billentyűkkel irányíthatjuk, így a játékunkat kétszemélyessé tettük.

Az üldöző teknőcöket *t1* és *t2* néven szerepeltetjük majd, mindkettőt a panelen hozzuk létre.

Az ALAPOK fülön ne felejtsük el a tollakat felemelni. A többi beállításuk az ALAK fülön látható.




A *t1* és *t2* teknőcöket mozgató eljárásokat a főablakban hozzuk létre. Ebben az eljárásban a játékosok haladnak előre, de időnként véletlenszerűen irányt váltanak.

```
eljárás megy
 e 2
 ha vsz 40 = 0
  [irány! tetszőleges]
vége
```

A főablak eleme az ***indító*** eljárás, ami a teknőcök starthelyzetét, a teknőcök irányítását, és a mozgató eljárások futtatását indítja el.

```
eljárás indító
 mi'xypoz! -100 -100 te'xypoz! 100 100
 t1'xypoz! -100 100 t2'xypoz! 100 -100
 mi'gombmenü! [
  fel [irány! 0]
  le [irány! 180]
  jobbra [irány! 90]
  balra [irány! 270]
  esc [stopmind]]
 te'gombmenü! [
  w [irány! 0]
  y [irány! 180]
  s [irány! 90]
  a [irány! 270]
  esc [stopmind]]
 minden 80 [mi'mozog te'mozog t1'megy t2'megy]
vége
```

Végezetül a *lap1*-en létrehozott *startgomb* nevű gomb hívja meg ***indító*** eljárásunkat! Ez a **haLenyom** eseményre következzen be!

38. *A játékot egészítsük ki egy csúszkával, amin a nehézségi szintet szabályozhatjuk!*

39. *A játék indítása után mérjük az első ütközésig eltelt időt, ami alapján készítsünk egy eredménylistát!*

40. *Nehezítsük úgy a játékot, hogy minden perc elteltével az üldözők mozgása egy kicsit felgyorsul!*

## Fej-írás játék

*17. Készítsünk fej-írás játékot! A program számolja a találatokat! Az ESC billentyű lenyomásáig lehessen játszani!*

A következő kis játékprogramunk a véletlent hívja segítségül, hogy eldönthessük, mennyire tudjuk a jövőt megjósolni. A klasszikus fej-írás játékot játsszuk, azaz a gép megkérdezi, hogy mi a tippünk, és utána sorsol.

```
eljárás fej.írás :pontszám
 kiír [Fej vagy Irás? Válassz! [Esc kilép] |(F/I/ESC)|]
 lokálisérték "be olvasjel
 ha :be = jel 0
  [kiír "VISZLÁT! stopmind]
 (kiír "|A tipped:| :be)
 hak :be = kiválaszt "FI
  [kiír "Eltaláltad növel "pontszám]
  [kiír "|Nem találtad el!| (növel "pontszám -1)]
 (kiír "Eddig :pontszám "|pontot szereztél!|)
 várj 1000
 fej.írás :pontszám
vége
```

A **kiválaszt** parancs egy szó vagy egy lista tetszőleges karakterét vagy elemét adja vissza. A hatása azonos az **elem 1 + véletlenszám elemszám *bármi bármi*** parancsával.

Az eljárást most is egy ***indító*** nevű másik eljárással hívjuk meg.

```
eljárás indító
 írólapablak törölszöveg
 fej.írás 0
vége
```

A program kiírásában figyeljük meg, hogyan jelentek meg az üzenetek!

```
Fej vagy Irás? Válassz! [Esc kilép] (F/I/ESC)
A tipped: F
Nem találtad el!
Eddig -1 pontot szereztél!
Fej vagy Irás? Válassz! [Esc kilép] (F/I/ESC)
A tipped: F
Eltaláltad
Eddig 0 pontot szereztél!
Fej vagy Irás? Válassz! [Esc kilép] (F/I/ESC)
VISZLÁT!
```

*41. Készítsünk kő-papír-olló játékot! A program előre kérdezze meg, hogy hány menet lesz, és az alapján írja ki a végeredményt!*

*42. A fenti programot alakítsd át úgy, hogy figyelje, mit adunk meg tippnek, és az alapján súlyozottan választ!*

# Funkcionális LOGO

Kötetünk ezen fejezetében az Imagine tulajdonképpen csak programfejlesztési környezet, a LOGO nyelv csak megvalósítási, kódolási eszköz. Ezt a fejezetet minden olyan programozással ismerkedőnek, illetve BASIC, Pascal vagy C nyelven programozónak is bátran ajánljuk, akik a *lista* összetett adatszerkezet helyett eddig csak a *tömb* szerkezettel végeztek müveleteket.

*Kedves Olvasó!* Ha a LOGO-val, illetve a lista összetett adatszerkezettel szemben támasztott előítéleteit néhány óra erejéig félre tudja tenni, akkor most egy olyan világot tárunk fel, amely minden más nyelven programozó számára is izgalmas kaland helyszíne, és az eddig használt programozási nyelv működésének és korlátainak jobb megértésében is sokat segíthet.

Néhány egyszerű példán követjük nyomon, mit is rejt a LOGO funkcionális része. Ez a terület, ami leginkább a LISP gyökerekre épít. Ezen nagy fejezet külön érdekessége, hogy ami itt szerepel, az nagyrészt mindenfajta átalakítás nélkül a ComLOGO-ban, de jó néhány más LOGO változatban is teljesen azonosan valósítható meg, és más programozási nyelvekben sem kell másképpen, nem lehet hatékonyabban megvalósítani az adott algoritmust.

## Adattípusok

Minden programozási nyelv alapvető építőkövei az alkalmazható adattípusok és a rajtuk értelmezett műveletek.

A LOGO nyelv elemi adata a karakter. A karakterekből előállhat például *szó,* ami értelmezhető *egész* vagy *valós számként,* illetve építhető belőle *mondat* vagy *lista.*

### Egyszerű adatobjektumok, primitívek

#### Szó

*A szó* karakterek sorozata. A parancsszavaktól és egyéb elemektől megkülönböztetendő a szó írása " idézőjellel történik, ha önállóan áll. így például *szó* lesz a ***"körte,*** de a ***"2.alma*** is. Néhány speciális karakter nem szerepelhet a szóban egyszerűen beírva, így a \ és / jel, vagy a műveleti jelek sem írhatóak be. Éppen ezek miatt az Imagine-ben megalkották az *általános szóírási módot.* Ekkor a szó karaktereit | | jelek közé írjuk, és így például már szóköz is lehet a szó része, például a ***"\7/B tanulói*** | egy szónak számít. Célszerű ezt a szóírásmódot választanunk mindannyiszor, amikor betűkön kívül más jeleket is írnunk kell, illetve ha a szóközt akarjuk használni.

*43. Azonos jelsornak számít-e a "matrac és a "\matrac\ karaktersorozat?*

*44. Vizsgáljuk meg, hogy a következő jelsorozatok szavak-e? Használjuk erre a* **szó?** *parancsot!*

*a) maci* *b) "6* *c) \7 törpe*

*d) "\Holnap kirándulni megyünk a hegyekbe!*

*45. Hogyan adható meg az üres szó?*

A **szó** parancs is az Imagine LOGO-ban. Figyeljük meg, hogyan lesz a megadott elemekből egyetlen, felesleges szóköz nélküli kiírás!

```
? ciklusegyenként "i [alma szilva banán]
  [(kiír szó hányadik ". "szó elemszám :i "betűből "áll)]
1. szó 4 betűből áll
2. sző 6 betűből áll
3. szó 5 betűből áll
```

A **szó?** parancs viszont annak a megállapítására szolgál, hogy a kérdéses jelsorozat szó-e.

```
? mutat szó? "szöveg
igaz
? mutat szó? 34
igaz
? mutat szó? [maci 2]
hamis
```

Mint látható a 34 is szó!

A karakterekből alkotott nem üres szavak speciális sorozata a *szám.* A szám alapvetően számjegyek sorozata, például a 123, de szerepelhet a . (tizedespont) és a - (előjel) karakter is a tizedesek elválasztására és az előjel jelölésére: -34.6 is szám lesz. Ezen kívül lehet betű is a szám része, mert például szám a 3.5e-4 jelsor: normál alakban megadott. Fontos, hogy az Imagine számalakjában is tizedespont van, és nem tizedesvessző!

*46. Állapítsuk meg, hogy a következő jelsorozatok számok-e?*

*a)+6* *b)-6.6E6* *c) 6E+6* *d) 3.4e5.6* *e) 6,78*

*A* **szám?** parancs a bemenetként megadott szóról állapítja meg, hogy értelmezhető-e a számként.

```
? mutat szám? 34
igaz
? mutat szám? "szöveg
hamis
? mutat szám? "34,7
hamis
```

Vegyük észre, hogy az utóbbi esetben a tizedesvessző okozza a hibát! A törtrész elválasztására a többi ország pontot használ, a vessző használatának következményét sajnos el kell fogadnunk, és nagyon figyelnünk kell rá.

## Műveletek szavakkal

A *szó* tehát az általános adatobjektum, amelynek egy részhalmazát alkotják csak a *számok.* E sajátosságra tekintettel kell lennünk, ha müveleteket végzünk a számokkal. Elsőre például meglepő lehet, de a fentiekből következik, hogy a

```
? mutat 7 + első "|3. sor|
10
```

eredmény logikus, mivel a szó első karaktere számként is értelmezhető.

Az alkalmazott **első** parancs a szó első karakterét adja vissza. Értelemszerűen az **utolsó** parancs az utolsó karaktert jeleníti meg.

```
? mutat első "kéreg
k
? mutat első "|

? mutat utolsó "kéreg
g
```

Mint látható az **első** parancs számára nem okoz hibát, ha az üres szóra alkalmazzuk, az üres karaktert adja vissza.

A nem üres szóra alkalmazható még az **elsőnélküli, utolsónélküli** - röviden **en** és **un** - parancs is. Ezek a szó első vagy utolsó karakterét elhagyják.

```
? mutat elsőnélküli "falak
alak
? mutat utolsónélküli "falak
fala
```

Az **első** paranccsal kombinálva hozzáférhetünk a szó második karakteréhez is.

```
? mutat első elsőnélküli "répa
é
? mutat utolsó utolsónélküli "répa
P
```

Az **elemnélküli** és az **elemsorszámnélküli** parancsok már a szó belsejében is ki tudják fejteni hatásukat. Mindkettő két bemenetű.

Az **elemsorszámnélküli** - röviden **elemsn** - esetében az első bemenet egy szám, a második egy szó. A művelet eredménye egy olyan szó lesz, amely az eredeti szóból a szám helyén lévő karaktert nem tartalmazza.

```
? mutat elemsorszámnélküli 4 "körte
köre
```

Az **elemnélküli** első bemenete egy karakter, a második egy szó. A művelet eredménye egy olyan szó lesz, amelyből hiányzik az összes megadott karakter.

```
? mutat elemnélküli "e "elemes
lms
```

Ha a szóban nem fordul elő a megadott karakter, akkor természetesen a szó nem változik meg!

*18. Egy általános szóírási móddal megadott szóból szedjük ki az esetlegesen előforduló szóközöket!*

```
? ki elemnélküli "|   "|bármi is lehet itt
bármiislehetitt
```

Igen hasznos lehet a két szó bemenetes **elemtől** parancs. A parancs balról megkeresi az első bemenet első előfordulását a második bemenetben, és a második bemenetnek azzal a részével tér vissza, amely attól az előfordulástól kezdődik. Ha az nem található meg benne, akkor az üres szóval tér vissza.

```
? mutat elemtől "me "mamut

? mutat elemtől "ma "mamut
mamut
? mutat elemtől "mu "mamut
mut
```

Az ugyancsak két bemenetű **elsőnek** és **utolsónak** parancsokkal az első bemenetként megadott szót illeszthetjük a második bemenetként megadott szó elé vagy után.

```
? mutat elsőnek "kör "lap
körlap
? mutat utolsónak "kör "lap
lapkör
```

A **szó** paranccsal hasonló eredményt érhetünk el, és ezt több bemenettel is használhatjuk. Ekkor viszont ki kell jelölni a parancs hatókörét.

```
? mutat (szó "kör "lap "szél)
körlapszél
```

Az **üres?** paranccsal megállapíthatjuk, hogy egy szó tartalmaz-e karaktert vagy sem. Akkor kapunk igaz értéket, ha a szó üres, különben hamis lesz a visszatérési értéke.

```
? ki üres? "|egy két|
hamis
? ki üres? "|
igaz
```

Az **elemszám** parancs a szó karaktereinek számát adja meg. Az üres szó nyilván 0 értékű lesz.

```
? ki elemszám "|egy két|
7
? ki elemszám "
0
```

*47. Helyettesítsük az **üres?** parancsot az **elemszám** segítségével!*

A **szó** parancs mintájára lehetőségünk van külön álló szavakból mondatot készíteni. Erre szolgál a **mondat** parancs.

```
? mutat (mondat "Jó "|LOGO-ban| "programozni!)
[Jó LOGO-ban programozni!]
```

Felmerülhet a kérdés: egy mondatot hogyan lehet szavakra bontani? Erre a kérdésre a listaműveleteknél adunk választ, de annyit már most elárulhatunk, hogy az **első** parancs például a mondat első szavát adja vissza, és a többi itt már ismertetett parancs hasonlóan alkalmazható mondatra is, listára is.

A szó belsejében is kutakodhatunk. Az **eleme?** és **eleme** parancsok a tartalmazást vizsgálják. Az **eleme?** a létezést, míg az **eleme** a tartalmazást, pontosabban annak helyét vizsgálja.

```
? mutat eleme? "e "telep
igaz
? mutat eleme "e "telep
2
```

Az első bemenet lehet szó is:

```
? mutat eleme? "le "telep
igaz
? mutat eleme "le "telep
3
```

A visszatérési érték hamis, ha nem fordul elő a karakter a szóban, illetve 0.

```
? mutat eleme? "a "telep
hamis
? mutat eleme "a "telep
0
```

Az első bemenet ugyancsak lehet szó is:

```
? mutat eleme? "eb "telep
hamis
? mutat eleme "eb "telep
0
```

## Feladatok szókezelésre

A szó adaton értelmezett legfontosabb parancsok áttekintése után nézzünk meg néhány klasszikus szókezelő feladatot!

*19. Egy szó összes magánhangzóját cseréljük ki az e betűre!*

A feladat megoldását bontsuk két eljárásra. Az első eljárásban adjuk meg a magánhangzókat. Ez a konstans függvényes megadás tipikus fogás arra, hogy egy halmazban való előfordulást tudjunk vizsgálni.

```
eljárás magánhangzó
 eredmény [a á e é i í o ó ö ő u ú ü ű]
vége
```

Vizsgáljuk meg az eljárást:

```
? mutat eleme? "a magánhangzó
Igaz
? mutat eleme? "b magánhangzó
hamis
```

A tényleges ***cserél*** eljárásban végighaladunk majd a szó betűin, mindaddig, amíg üres szót nem kapunk. Ez lesz egyben az eredmény. Ha nem üres a szó, akkor viszont az első betűjéről el kell döntenünk, hogy eleme-e a magánhangzók halmazának, vagy sem. Ha igaz, akkor az eredménybe az e betűt írjuk helyette, ha hamis, akkor az eredeti betű kerül be, és az első betű elhagyásával kapott szót vizsgáljuk tovább.

```
eljárás cserél :szó
 ha üres? :szó [eredmény :szó]
 hak eleme? első :szó magánhangzó
   [eredmény elsőnek "e cserél elsőnélküli :szó]
   [eredmény elsőnek első :szó cserél elsőnélküli :szó]
vége
```

Vizsgáljuk meg az elkészített eljárást:

```
? mutat cserél "kitárul
keterel
? mutat cserél "stb
stb
```

A csere megvalósítására létezik a **csere** parancs is a LOGO nyelvben.

Ha a **csere** első bemenet egy *szám,* akkor a **csere** a második bemenetével úgy tér vissza, hogy kicseréli annak szám-adik elemét a harmadik bemenetre. Ha a *szám* kisebb, mint 1, akkor a harmadik bemenetet a második első elemeként szúrja be. Ha a *szám* nagyobb, mint a második bemenet elemeinek száma, akkor a harmadik bemenetet utolsó elemként szúrja be a másodikba.

```
? mutat csere 2 "bénán "a
banán
? mutat csere 8 "banán "héj
banánhéj
? mutat csere 0 "banán "zöld
zöldbanán
```

Ha a *szám* nem egész, akkor a csere eljárás kerekít - ellentétben a Súgóban leírtakkal -, és az így kapott helyen kerül sor a fenti műveletre.

```
? mutat csere 2.8 "bénán "a
béaán
? mutat csere 2.4 "bénán "a
banán
```

Ha az első bemenet egy egyszerű *[számi szám2] szegmens,* azaz egy tartomány, akkor a művelet a második bemenetével úgy tér vissza, hogy

kicseréli a szegmens által meghatározott összefüggő karaktereit a harmadik bementre. Ez persze túl is nyúlhat az eredeti bemeneten, ekkor azt felülírja.

```
? mutat csere [2 2] "bunda "or
borda
? mutat csere [3 4] "banán "jonett
bajonett
```

Figyeljünk arra, hogy ha az első bemenet egy szegmens, akkor a harmadik bementnek mindig ugyanolyan típusúnak kell lennie, mint a másodiknak.

*20. Készítsünk eljárást, amely megállapítja egy megadott szóról, hogy mély, magas vagy vegyes hangrendü-e!*

Elsőként adjuk meg külön függvényekben a magas és mély magánhangzókat!

```
eljárás magas
 eredmény [e é i í ö ő ü ű]
vége

eljárás mély
 eredmény [a á o ó u ú]
vége
```

A bemeneti szóban megkeressük, hogy szerepel-e magas illetve mély magánhangzó a ***vaiumagas?*** és a ***varuméfy?*** eljárások segítségével.

```
eljárás van.magas? :sző
 ha üres? :szó [eredmény "hamis]
 ha eleme? első :szó magas[eredmény "igaz]
 eredmény van.magas? elsőnélküli :szó
vége

eljárás van.mély? :szó
 ha üres? :szó [eredmény "hamis]
 ha eleme? első :szó mély [eredmény "igaz]
 eredmény van.mély? elsőnélküli :szó
vége
```

Ezután a szó betűit egyenként vizsgáljuk meg a ***hangrend*** eljárásban. Ha van magas rendű, akkor megvizsgáljuk, hogy van-e mély is. Ha igen, akkor *vegyes,* ha nem, akkor tiszta *magas* hangrendű a bemeneti szó. Ha nem volt magas, de van benne mély magánhangzó, csak tisztán *mély* lehet, viszont ez a feltétel sem teljesül, akkor a bemeneti szóban *nem volt magánhangzó.*

```
eljárás hangrend :szó
 ha van.magas? :szó
   [hak van.mély? :szó
     [eredmény "vegyes][eredmény "magas]]
 hak van.mély? :szó
   [eredmény "mély][eredmény "|nincs benne magánhangzó|]
vége
```

Kész eljárásunkat vizsgáljuk meg a lehetséges tesztesetekre!

```
? mutat van.magas? "baba
hamis
? mutat van.magas? "bébi
igaz
? mutat van.magas? "stb
hamis
? mutat van.magas? "ciklon
igaz
? mutat van.mély? "ciklon
igaz

? mutat hangrend "baba
mély
? mutat hangrend "bébi
magas
? mutat hangrend "ciklon
vegyes
? mutat hangrend "stb
nincs benne magánhangzó
```

Bemeneti szavunkat magánhangzói hangrendje szerint kódolhatjuk is. A szó betűit egyenként megvizsgáljuk: Ha magas a magánhangzó, akkor ' jelet írunk, ha mély, akkor , jelet. Ha nem magánhangzó, akkor semmit.

```
eljárás rend.kód :szó
 ha üres? :szó [eredmény :szó]
 ha eleme? első :szó mély
   [eredmény elsőnek "|,| rend.kód elsőnélküli :sző]
 ha eleme? első :sző magas
   [eredmény elsőnek "|'| rend.kód elsőnélküli :szó]
 eredmény rend.kőd elsőnélküli :szó
vége
```

Az eljárás eredménye egy kódolt szó lesz, amely az eredeti szóban található magánhangzók számával azonos hosszú. Ezt a kódolt kimenetet vizsgáljuk meg, hogy azonos jeleket tartalmaz-e, vagy vegyesen fordulnak elő a magánhangzók, esetleg nincs is benne magánhangzó.

```
? mutat rend.kód "lapos

? mutat rend.kőd "éléskamra

? mutat rend.kód "|kihagytál-e|
' , , '
```

*48. Készítsünk eljárást, amely a bemeneteként megadott szót megfordítja!*

*49. írassuk ki egy mondatot betűnként megfordítva!*

*50. Egy mondat palindrom, ha a benne szereplő betűk a szóközöktől eltekintve azonos sorrendben helyezkednek el mindkét irányból. Készítsünk programot, amely eldönti egy mondatról, hogy palindrom-e!*

## Lista

A *lista* a LOGO, és így az Imagine egyik legfontosabb összetett adatstruktúrája. A lista szavakból, képsorokból és egyéb listákból is állhat. Az [] az *üres lista* jele, amely olyan lista, amelynek nincs eleme.

A *mondat* szintén fontos, szavakból előálló szerkezet. A szavakat alapvetően szóközökkel választjuk el, ezért a mondat esetében is ezt használhatjuk elválasztóként. A mondat határait a [ ] jelek jelölik ki. Üres mondatot nem adhatunk meg [] írásával, mivel ezt egy másik fontos elemnek, az üres listának tartjuk fenn. Azt is mondhatjuk, hogy a mondat egy speciális, csak szavakból felépülő lista.

A **szó** parancshoz hasonlóan a **mondat** parancsként is megjelenik az Imagine LOGO-jában. A **mondat** parancs a mögötte felsorolt elemekből készít listát, és mint már korábban többször láttuk, egyik alkalmazása a változók értékének konstanssá módosítása.

A *tömb* adatszerkezet a Pascalban vagy a BASIC-ben megszokott formában a LOGO-ban és az Imagine-ben nem létezik. A lista műveletei között viszont szerepelnek olyanok, amelyekkel a listát tömbszerűen kezelhetjük.

A tömb szerkezet látszólagos hiánya miatt sokan a LOGO nyelvet alacsonyabb rendűnek tekintik, és úgy gondolják, hogy az erre az adatszerkezetre kifejlesztett algoritmusok nem ültethetőek át. Azt persze nem teszik hozzá, hogy a lista adatszerkezetre kidolgozott speciális algoritmusokat viszont tömb szerkezetre körülményes igen sokszor átültetni.

A tömb adatszerkezet LOGO-beli specifikációjának hiányaként ok lehetne, hogy a logó, illetve az alapot adó LISP kifejlesztése idején szalagos tárakat alkalmaztak, így nem is volt mód szabadon elérni az adatelemeket, csak a lineáris feldolgozás volt megvalósítható egyszerűen, ezért hatékonyan. A BASIC és a Pascal születésekor már a lemezes tárak is kezdtek elterjedni háttértárolóként, és így szabadabbá vált az elemek elérése. Ez viszont nem teljesen helyes érvelés: a LOGO eredendően szövegfeldolgozási feladatokra kifejlesztett nyelv, ahol a lista a természetes adatszerkezet, mivel a szöveg karaktereit alapvetően sorban dolgozzuk fel.

Néhányan a *bekezdés* fogalmát is használják, ami alatt a csak szavakból felépülő mondatok listáját értik. Például a

```
[[1 egy] [2 kettő] [3 három]]
```

egy bekezdésnek tekinthető, ami némileg egyszerűbb megnevezése a mondatokból felépülő listának.

### Műveletek listákkal

A lista adatszerkezet alapvetőbb műveleteivel foglalkozunk a következőkben. Az első művelet a lista hozzárendelése egy változóhoz, ami a

```
lokálisérték :l [ e1 e2 e3 ... en-1 en]
```

paranccsal tehető meg például.

Az **első, elsőnek, elsőnélküli** a lista első, ei eleméhez fér hozzá, míg az **utolsó, utolsónak, utolsónélküli** az utolsó elemet, e„-t tudja kezelni. Az üres lista létrehozása történhet egyszerűen az értékadás utasítással is: a **lokért :l []** parancs egy :l nevű üres listát hoz létre, amely parancs egyben alkalmas például egy lista kiürítésére is.

A lista elemszáma, üressége is vizsgálható. Erre szolgál az **elemszám** és az **üres?** parancs. A többi listaművelet közül kilóg az **elemnélküli,** amely a paraméterként megkapott érték összes előfordulását kiveszi a listából. Ez azért is kilóg a sorból, mert a listának általában az első vagy az utolsó elemét tudjuk kezelni, a belső elemek külön mutatókkal érhetők el. Az elemnélküli eljárás tulajdonképpen egy egyszerűsítés, amely a többi művelettel helyettesíthető is.

Nézzünk meg néhány példát ezen parancsok működésére!

```
? mutat elemszám [2 14 5]
4
? mutat elemszám [2 [1 4 5]]
2
? mutat eleme 4 [2 14 5]
3
? mutat eleme? 6 [2 14 5]
hamis
? mutat elemnélküli 2 [12 [12 3]]
 [1 [1 2 3]]
? mutat első [12 [12 3]]
1
? mutat utolsó [12 [12 3]]
 [1 2 3]
? kiír utolsó [12 [12 3]]
1 2 3
```

Ez utóbbi két parancs ismételten példa két kiírás közti különbségre.

### *Feladatok listákkal*

*21. Egy beolvasott mondat szavai helyett adjuk meg rendre azok hosszát!*

A feladat értelmezhető úgy, hogy ha a bemeneti mondat üres, akkor egy üres listát kell megadnunk, különben az első eleme helyett annak hosszát kell ími a végeredmény listába, majd ezután a maradék mondatra kell alkalmazni az eljárást.

```
eljárás szóhossz.lista :m
 ha üres? :m [eredmény []]
 eredmény elsőnek elemszám első :m szőhossz.lista en :m
vége
```

Nézzük meg, hogyan is működik az eljárásunk!

```
? mutat szóhossz.lista [Itt az idő!]
[3 2 4]
? mutat szóhossz.lista (szó "Itt "az "idő!)
[ 1 1 1 1 1 1 1 1 1 ]
```

Mint látható, a ! jel is karakter, a második esetben lista helyett szó a bemenet, amit nem listaként, hanem karakterenként vizsgál meg.

51. *Javítsuk ki a* ***szóhossz.lista*** *eljárást úgy, hogy szó bemenet esetén a szó hosszát adja vissza!*

*A* lista elemeinek sorrendjét is tudjuk változtatni, például ha a lista első elemét akarjuk a lista végére helyezni, akkor tulajdonképpen egy jobbra léptetést hajtatunk végre.

```
eljárás lép.jobbra :s
 eredmény elsőnek utolsó :s utolsónélküli :s
vége
```

Nézzünk meg egy futási példát:

```
? mutat lép.jobbra "lakó
ólak
```

52. *Készítsünk* ***lép.balra*** *nevű függvényt, amely a lista utolsó elemet teszi át a lista első helyére!*

53. *Készítsünk eljárást, amely az* **utolsó** *parancsot helyettesíti az* **első, elsőnélküli, elsőnek** *parancsok felhasználásával!*

Vegyük észre, hogy a listaműveletek kevés kivétellel azonosak a szóra alkalmazható műveletekkel.

54. *Készítsünk* ***egyelemű?*** *nevü függvényt, amely akkor ad igaz eredményt, ha a paramétereként megadott lista vagy szöveg egy elemű!*

55. *Készítsünk* ***szerepel*** *nevű függvényt, amely megállapítja, hogy a paramétereként megadott listában hányszor szerepel egy érték!*

56. *Készítsünk* ***elemkihagy*** *nevü függvényt, amely az* **elemnélküli** *eljárásnak megfelelően működik, de azt nem használja!*

57. *Készítsünk* ***egykihagy*** *nevű függvényt, amely az első paramétereként megkapott érték első előfordulását hagyja ki a második paraméterként megkapott listából. A feladat megoldásánál használjuk fel az elemi algoritmusokat!*

## Listakezelés haladó szinten

A LOGO nyelv nagy hiányosságának tartják sokan, hogy nem rendelkezik tömb adattípussal. Ez a szó klasszikus értelmében igaz is, de ha speciális felépítésű listákat készítünk, akkor a vektor, mátrix és tetszőleges dimenziójú tömb is előállítható és kezelhető az Imagine-ben.

### *Vektor létrehozása és kezelése*

Egy lista tetszőleges elemét tudjuk megjeleníteni a **mélyelem** paranccsal. Ennek használatakor jó tudni, hogy vektorként kezeli a lista elemeit 1-től indexelve. Ha a vektor határain túllépünk, akkor az üres listát adja vissza.

```
? mutat mélyelem 3  [a b c d]
c

? mutat mélyelem 6  [a b c d]
[]

? mutat mélyelem 0  [a b c d]
[]
```

A vektorba írni is tudunk. Erre szolgál a **mélycserél** parancs.

```
? mutat mélycserél 4  [a b c d e f]  "d2
[a b c d2 e f]
```

Ha az indexhatárokon kívülre mutat az első hely paraméter, akkor a vektor nem változik meg.

```
? mutat mélycserél 8  [a b c d e f]  "d2
[a b c d e f]
```

Vegyük észre a különbséget a **csere** és a **mélycserél** parancs között!

```
? mutat csere 8  [a b c d e f]  "d2
[a b c d e f d2]
```

### *Kétdimenziós tömb létrehozása és kezelése*

Az azonos hosszúságú listákból felépülő listákkal kétdimenziós tömböt tudunk megadni. A jobb oldali 4×3-as mátrix például megadható

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

```
[[1 2 3 4][5 6 7 8][9 10 11 12]]
```

lista formájában.

A **mélyelem** és **mélycserél** parancsok első paramétere lista is lehet. Ebben az esetben az első lista első eleme a főlista elemét azonosítja, az első lista második eleme már az allistán belüli elemet.

```
? mutat mélyelem [2 3]  [[1 2 3 4][5 6 7 8][9 10 11 12]]
7
```

Hasonlóan használható a **mély cserél** eljárás is. Ha például a mátrix 2. sorában lévő 3. elemet akarjuk cserélni, akkor a

```
? mutat mélycserél [2 3] [[1 2 3 4] [5 6 7 8] [9 10 11 12]] "d
[[1 2 3 4] [5 6 d 8] [9 10 11 12]]
```

Itt is fontos megjegyeznünk, hogy a változást tárolni is kell, a második bemenetként megadott lista ténylegesen nem módosul.

*22. Készítsünk egy **tömb.létrehoz** nevü függvényt, amely három paraméterrel rendelkezik: az első a tömböt alkotó sorok, a második az oszlopok száma. A tömb elemeit alkotó elem legyen a harmadik paraméter!*

A függvény megvalósításához használjuk fel a **lokálisérték** parancsot. Elsőként a sorokat, majd az oszlopokat töltjük fel.

```
eljárás tömb.létrehoz :i :j :e
 lokálisérték "1 []
 ism :j [lokalisérték "1 elsőnek :e :1]
 lokálisérték "m []
 ism :i [lokálisérték "m elsőnek :1 :m]
 eredmény :m
vége
```

A függvény használatát mutatja a következő példánk:

```
? mutat tömb.létrehoz 2 5 "#
[[#####] [#####]]
```

Ne felejtsük el, hogy a létrehozott tömb alapvetően kétdimenziós, így az alábbi eredményeket kapjuk, ha speciális sor- és oszlopszámot adunk meg.

```
? mutat tömb.létrehoz 1 5 "#
[[#####]]
? mutat tömb.létrehoz 1 1 "#
[[#]]
? mutat tömb.létrehoz 1 0 "#
[[]]
? mutat tömb.létrehoz 0 0 "#
[]
? mutat tömb.létrehoz 0 5 "#
[]
```

A feltöltő elem persze lista is lehet.

*58. Alakítsuk át az eljárást úgy, hogy ha a sor vagy oszlop paraméterek közül az egyik 1, akkor csak egyszerű listát hozzon létre!*

Vegyük észre, hogy ha lenne egy ***vektor.létrehoz*** nevü, elemszámot és elemet tartalmazó függvényünk, akkor azzal a ***tömb.létrehoz*** helyettesíthető lenne!

*59. Készítsük el a **vektor.létrehoz** nevü, elemszám és egy elem paraméterű függvényt!*

*23. Hozzuk létre az alábbi tömbnek megfelelő listát! Azaz egy olyan tömböt, amelynek az i. sorában és a j. oszlopában az i+j összeg áll! A feladat megoldásához az* ***ismétlés*** *parancsot használjuk!*

$$\begin{pmatrix} 2 & 3 & 4 & \cdots & 1+n \\ 3 & 4 & 5 & \cdots & 2+n \\ \vdots & \vdots & \vdots & i+j & \vdots \\ m+1 & m+2 & m+3 & \cdots & m+n \end{pmatrix}$$

A feladat nehézsége abban rejlik, hogy a **hányadik** parancs nem kötődik szorosan egy ciklushoz, hanem általában jelenti a ciklus változóját. Éppen ezért kénytelenek vagyunk a külső ciklus indexét eltárolni egy ***k*** változóba, és ezután már a belső ciklusban szabadon használhatjuk a **hányadik** parancsot.

```
eljárás példa :i :j
 globvál "m tömb.létrehoz :i :j 0
 ism :i
  [lokért "k hányadik
   ism :j
    [globvál "m mélycserél(mondat :k hányadik) :m hányadik+:k]]
 mutat :m
vége
```

Figyeljük meg, hogy a **mélycserél** parancs nem szereti a változót, de a **mondat** paranccsal tulajdonképpen konstanslistává alakítjuk a helyet meghatározó első paramétereit.

```
? példa 3 4
[[2 3 4 5] [3 4 5 6] [4 5 6 7]]
```

A vektor és tömb adatszerkezet műveleteit az Imagine lehetőségeit megmutató sorozatunk harmadik részében a rendezések és a gráfok kapcsán tovább is tárgyaljuk.

Az Imagine egyik sajátossága pont itt mutatkozik meg: a korábbi LOGO verziókban már megismert és megszokott automata és funkcionális nyelvi elemek elérhetősége mellett a procedurális nyelvek egyes elemeit is tartalmazza, emiatt talán az egyik legérdekesebb programozási nyelvi kísérlet.

*60. Készítsünk eljárást, amely megadja egy bemeneti vektorra, hogy milyen elemszámú vektorok az elemei! Az elkészített eljárást tároljuk el* ***vektor.részelemszám*** *néven!*

*61. Egy baráti társaság tagjai részben telefonszámot cseréltek. Ábrázoljuk tömb segítségével, hogy ki kinek adta meg a számát, majd döntsük el, hogy egy kiválasztott személy ismeri-e egy másik személy számát! Az elkészített függvényt* ***tudja?*** *néven mentsük el! A tömbben 0 jelentse, hogy nem tudja, 1 ha tudja a telefonszámot!*

# Programozási tételek

A programkészítés során a részekre bontott feladat megoldásának nagy része sok azonos részproblémát megoldó, egyszerűen megfogalmazható kis algoritmusra vezethető vissza. Ezeket nevezzük *elemi algoritmusnak,* vagy másképpen *programozási tételnek.*

E fejezet kulcsszava is ez: itt az ***algoritmizálás*** alapjait mutatjuk be! Éppen ezért ajánljuk mindenkinek, hogy a könyv honlapján keresse meg a más változatokra való átalakítás lehetőségét!

## *Elemi programozási tételek*

A feladatcsoportban a keresett tulajdonságot a "> 10" feltétel helyettesíti az egyszerűség kedvéért. Ezt a feltételt ki is emelhetnénk paraméterként, de akkor nehezebben lenne átlátható a működésük.

A másik egyszerűsítés, hogy többnyire nem az első adott tulajdonságú elem helyét keressük meg, hanem magát az első elemet. Ez itt indokoltabb, mivel a listán belül sokkal kevésbé fontos a hely, nem úgy, mint a tömb szerkezetben.

### Összegzés

A bemeneti sorozat – vagy lista – alapján egyetlen értéket kell meghatároznunk. Lényeges, hogy a kimenet meghatározásában a bemenet összes eleme szerepet játszik.

Tipikus probléma alapján ezt a csoportot összegzésnek nevezzük majd.

*24. Készítsük el egy lista elemeinek összegét megadó eljárást!*

```
eljárás összegzés :l
 ha üres? :l [eredmény 0]
 eredmény ( első :l ) + (összegzés en :l )
vége
```

Hasonló feladat lehet, azaz összegzésre visszavezethető a következő probléma:

*62. Határozzuk meg egy tantárgyból kapott osztályzatok átlagát!*

*63. Adjuk meg az első n természetes szám szorzatát!*

*64. Egyesítsük egyetlen szóvá egy sorozatba (listába) tárolt egymást követő karaktereket!*

### Eldöntés

Ebbe a csoportba azon típusú problémákat soroljuk be, amikor a bemenet minden eleme alapján legalább egy elemére meglévő tulajdonság meglétét vagy éppen egységes hiányát célunk megállapítani.

A konkrét megvalósításban azt várjuk el, hogy az eljárás visszatérési értéke IGAZ legyen, ha a keresett tulajdonság teljesül, és HAMIS, ha nem.

*25. Készítsünk eljárást, amely megválaszolja, hogy egy listában van–e 10-nél nagyobb értékü elem!*

```
eljárás eldöntés :l
 ha üres? :l [eredmény "hamis]
 hak első :l > 10
  [eredmény "igaz]
  [eredmény eldöntés elsőnélküli :l]
vége
```

*65. Döntsük el, egy diák év végi jegyei alapján, hogy kitűnő lett-e vagy sem!*

*66. Állapítsuk meg egy szóról, hogy magas hangrendű-e vagy sem!*

*67. Döntsük el egy számról, hogy összetett-e, vagy sem!*

## Kiválasztás

A kiválasztási probléma esetén azt már tudjuk és elvárjuk, hogy a keresett tulajdonság teljesüljön. Emellett szeretnénk tudni azt is, hogy melyik az első eleme a bemenetnek, amelyre teljesül a tulajdonság.

> A kiválasztási probléma megválaszolásának feltétele, hogy az eldöntési vizsgálat IGAZ értéket adjon vissza, azaz feltesszük, hogy van keresett tulajdonságú elem a bemenetben.

Eredménynek két dolgot várhatunk: magát az első megfelelő értéket, vagy a bemenetben elfoglalt helyének sorszámát. Ennek megfelelően kétfajta megvalósítása is lehetséges.

*26. Készítsünk eljárást, amely megadja egy listában az első 10-nél nagyobb elemének a helyét!*

```
eljárás kiválasztás.hely :l
 hak első :l > 10
  [eredmény 1]
  [eredmény 1 + kiválasztás.hely en :l]
vége
```

A hely mellett az értéket is visszaadhatjuk:

*27. Készítsünk eljárást, amely megadja egy listában az első 10-nél nagyobb elemét!*

```
eljárás kiválasztás.érték :l
 hak első :l > 10
  [eredmény első :l]
  [eredmény kiválasztás.érték en :l]
vége
```

68. *Adjuk meg egy összetett szám legnagyobb, vele nem egyenlő - valódi - osztóját!*

69. *Adjuk meg egy összetett szám legkisebb valódi - nem 1-gyel egyenlő - osztóját!*

70. *Adjuk meg egy legalább két szótagos szóban az első magánhangzó helyét!*

## Lineáris keresés

Az ilyen típusú problémák esetén nem tudjuk biztosan, hogy létezik-e egyáltalán a bemenetben az adott tulajdonságú elem, de ha igen, akkor az első ilyen helyét vagy értékét várjuk visszatérő értékként. Ha az adott tulajdonságú elem nem létezik, akkor a hely keresése esetén a visszatérési érték 0 legyen!

A konkrét megvalósításban egy ***mit*** nevű érték előfordulási helyét keressük a bemenetben!

28. *Készítsünk eljárást, amely egy adott érték első előfordulási helyét határozza meg egy listában!*

```
eljárás lin.keresés :mit :l
 ha üres? :l [eredmény 0]
 hak első :l = :mit
   [eredmény 1]
   [eredmény 1 + lin.keresés :mit en :l]
vége
```

71. *A fenti megvalósítás, ha nem eleme a keresett **:mit** a listának, akkor az elemszámot adja vissza! Javítsuk ki az eljárást!*

72. *Adjuk meg egy sorozat első olyan elemét és helyét, amely kisebb, mint az előtte álló!*

73. *Egy hónapon keresztül mérjük a reggeli hőmérsékleteket. A mérések alapján adjuk meg az első negatív értékű nap sorszámát!*

74. *Adjuk meg egy egész szám 1-től és önmagától különböző egyik osztóját!*

## Megszámlálás

A probléma csoportban azt határozzuk meg, hogy a bemenet elemei közül hány rendelkezik a keresett tulajdonsággal.

```
eljárás megszámlálás :l
 ha üres? :l [eredmény 0]
 hak első :l > 10
     [eredmény 1 + megszámlálás en :l]
     [eredmény megszámlálás en :l]
vége
```

75. *Egy osztály tanulóinak átlaga alapján adjuk meg a jeles rendűek számát!*

76. *Adjuk meg egy szó mássalhangzóinak számát!*

77. *Határozzuk meg egy mondat szavaira kiszámított magánhangzók és mássalhangzók arányát!*

## Minimum megkeresése

A bemenet értékein, ha értelmezett egy rendezés, akkor problémaként felmerülhet, hogy melyik a legkisebb elem, azaz vagy hol helyezkedik el a bemeneten belül, vagy konkrétan mi az értéke. A bemenetről feltesszük, hogy legalább egy elemű.

```
eljárás minimum :l
 ha üres? en :l [eredmény első :l]
 hak első :l > utolsó :l
    [eredmény minimum en :l]
    [eredmény minimum un :l]
vége
```

78. *Készítsünk eljárást, amely a minimum első előfordulási helyét adja meg!*

Az eljárás minimális átalakításával megkaphatjuk a maximum meghatározását is!

### *Maximum keresése*

```
eljárás maximum :l
 ha üres? en :l [eredmény első :l]
 hak első :l < utolsó :l
    [eredmény maximum en :l]
    [eredmény maximum un :l]
vége
```

Vegyük észre, hogy az eljárások alkalmasak ebben a formában is egyetlen szó legnagyobb karakterkódú elemének meghatározására is.

79. *Adjuk meg, hogy egy beteg napi hőmérséklet mérései alapján melyik nap volt a legmagasabb a testhője!*

80. *Keressük meg egy osztály névsora alapján a legutolsó tanulót!*

81. *Adjuk meg egy szám legkisebb és legnagyobb számjegyének összegét!*

82. *Keressük meg egy verseny első három legjobb eredményt elért indulóját! Tegyük fel, hogy holtverseny nem lehet és legalább hárman indultak a versenyen!*

### Kiválogatás

A probléma csoportba azokat soroljuk, amelyekben bemenet elemeiből egy új adatszerkezetbe kigyújtjuk az összes adott tulajdonságú elemet. E feladat esetén is lehetséges, hogy nem magukat az értékeket, hanem csak a bemeneti sorozatban elfoglalt helyüket kell kigyűjteni.

```
eljárás kiválogatás :l
 ha üres? :l [eredmény []]
 hak első :l > 10
   [eredmény elsőnek első :l kiválogatás elsőnélküli :l]
   [eredmény kiválogatás elsőnélküli :l]
vége
```

Itt most a bemeneti lista 10-nél nagyobb elemeit gyűjtöttük ki.

*83. Készítsük el azt az eljárást, amely nem az adott tulajdonságú elemeket, hanem azok helyét gyűjti ki egy listába!*

*84. Gyűjtsük ki egy osztály jeles tanulóinak nevét!*

*85. Gyűjtsük ki egy iskola osztályainak létszámadatai alapján azokat, amelyekbe több fiú jár mint lány! A feladat megoldásához használjuk a tulajdonság szerkezetet!*

*86. Adjuk meg egy szám összetett osztóit!*

## Programozási tételek alkalmazása

*A* funkcionális LOGO használatának igen fontos területe, hogy felismerjük, milyen programozási tételek bújnak meg a megoldandó problémában.

### Medián meghatározása

*29. Határozzuk meg egy lista mediánját!*

Egy sorozat mediánján a nagyság szerint rendezett elemei közül a középső helyen állót értjük. Ha a sorozat elemszáma páros, akkor vehetjük a két középső közül bármelyiket vagy éppen a két középső elem átlagát. Mi ez utóbbit értjük most alatta.

A megoldás ötlete hasonlít a minimum megkereséséhez. Ha a sorozat egy elemű, ez az eleme maga a médián. Ha a sorozat kételemű, akkor a két elem átlaga, különben pedig elhagyjuk a legkisebb és a legnagyobb elemet, azaz kettesével csökkentjük a bemenet hosszát.

```
eljárás medián :l
 ha üres? en :l [eredmény első :l]
 ha esz :l = 2 [eredmény ((első :l) + utolsó :l)/2]
 eredmény medián en un :l
vége
```

Figyeljünk arra, hogy a bemeneti sorozatnak rendezettnek kell lennie, éppen ezért vagy rendezetten adjuk meg, vagy használjuk a **rendez** parancsot.

```
? mutat medián rendez [1 6 2 3 4]
3
? mutat medián rendez [16 2 3 4 5]
3.5
```

87. *Készítsük el úgy a **medián** meghatározását, hogy nem rendezzük a bemeneti sorozatot! Vegyük észre, hogy a rendezett sorozat legkisebb és legnagyobb elemét hagyjuk el, ami megoldható a **minimum** és a **maximum** elhagyásával is, ha kettőnél több elemű! Figyeljünk arra, hogy csak egy-egy ilyen értékű elemet szabad elhagynunk!*

88. *Vizsgáljuk meg, hogy nagy elemszám esetén a rendezés vagy a minimum és maximum megkeresése hatékonyabb-e!*

## Prímtényezőkre bontás

Egy természetes számot prímtényezőire bonthatjuk, ha a szám összetett. Az 1 nem összetett szám, prímtényezős felbontása nincs. A többi pozitív egész szám esetében a lehetséges legkisebb tényező a 2.

*30. Készítsük el egy pozitív egész szám prímtényezős felbontását!*

Az algoritmusunk alapgondolata az, hogy a számot elsőként megpróbáljuk 2-vel osztani. Ha osztót találunk, akkor addig írjuk ki a tényezők közé és osztjuk vele a számot, amíg lehet. Ha már nem osztó, akkor az osztót 1-gyel megnöveljük, és ez lesz az új osztó.

Az eljárást addig végezzük, amíg 1 nem lesz a szám.

```
eljárás tényező :n :o :l
 ha :n=1 [eredmény :l]
 hak (maradék :n :o) = 0
   [eredmény tényező (egészhányados :n :o) :o (utolsónak :o :l)]
   [eredmény tényező :n :o+1 :l]
vége
```

Az eljárás meghívása a **tényező :n 2** [] formában történhet.

```
? mutat tényező 28 2 []
[2 2 7]
? mutat tényező 12 []
[]
```

Vegyük észre, hogy feleslegesen próbálkozunk a 2 és 3 után a 4 értékkel, mivel az nem prím, és mint a 2 többszöröse, már biztos nem is szerepelhet tényezőként. Míg az 5-tel biztosan kell vizsgálatot végeznünk, a 6-tal nem, mivel a 2 és a 3 többszöröse. Vegyük észre, hogy az eljárás hatékonyabbá tehető, ha a 2 és 3 után már csak a $6k\pm1$ alakú osztókkal kísérletezünk!

*89. Módosítsuk az előbbi eljárást úgy, hogy csak a 6k±1 alakú számokkal próbálkozik a 2 és a 3 után!*

*90. Alakítsuk át úgy az eljárást, hogy a prímek ismeretében csak a prímszámokkal végzi el az osztáspróbát! A prímszámokat egy listában kapja meg az eljárás!*

## Írjuk ki fordítva!

*31. Készítsünk függvényt, amely a paramétereként megadott szó vagy lista fordítottját adja meg!*

```
eljárás fordít :l
 ha üres? :l [eredmény :l]
 eredmény utolsónak első :l fordít en :l
vége
```

A megvalósításunk nem lett általános:

```
? mutat fordít "kanna
annak
? mutat fordít [kanna]
[kanna]
```

## Nőjön mind!

*32. Egy lista minden elemét növeljük meg egy előre megadott értékkel! Az elkészített eljárásunk ezeket a megnövelt értékeket adja vissza eredményül!*

Az eredményként listát kell kapnunk, ezért a **mondat** eljárást alkalmazzuk az elemből új lista képzésére.

```
eljárás növelő :d :l
 ha üres? en :l [eredmény (mondat :d + első :l)]
  eredmény (mondat :d + (első :l ) növelő :d en :l)
vége
```

A kapott megoldás jól működik, de használható egy szó esetén is? Sajnos nem!

*91. Alakítsuk át úgy az eljárást, hogy a paraméternek kapott szó esetén annak karakterenkénti kódját növelje meg a megadott értékkel! A kész eljárást szó**.tol** néven mentsük el!*

## Vonjuk le mindből!

*33. Készítsünk függvényt, amely egy sorozat minden eleméből kivonja annak legkisebb elemét, és az így kapott sorozatot adja vissza eredményül!*

*A* feladat megoldásához nyilvánvalóan felhasználhatjuk a minimum meghatározására szolgáló, korábban már elkészített függvényünket.

A feladatot két lépésben oldjuk meg: elsőként meghatározzuk a minimumot, és ennek ismeretében hívjuk meg a kivonást végző eljárást!

Az eredményként listát kell kapnunk, ezért a **mondat** eljárást alkalmazzuk ismét az elemből a lista képzésére.

```
eljárás kivon0 :l
 ha üres? en :l [eredmény (mondat (első :l) - minimum :l)]
  eredmény (mondat (első :l) - minimum :l kivon0 en :l)
vége
```

Kész is vagyunk! Próbáljuk ki!

```
? mutat kivon0 [4 3 2 1]
[3 2 1 0]
```

Az eredmény pont az, amit vártunk! De azért az ördög nem alszik, nézzünk meg egy másik esetet is!

```
? mutat kivon0 [1 2 3 4]
[0 0 0 0]
```

Na bumm! Nem ennek kellett volna kijönnie! De a hiba most sem a gépben van! Gondolkodásunkban követtük el azt a kis hibát, ami miatt ezt az eredményt kaptuk!

A ***kivon0*** függvény újra és újra meghatározza a még fel nem dolgozott elemek minimumát. Emiatt nem az egész bemeneti lista, hanem a még fel nem dolgozott részlista minimumát vonjuk ki a soron következő elemből!

Az igazi megoldás is kézenfekvő: külön, előre meg kell határozni a teljes lista minimumát, majd ezt kell kivonni minden elemből! A feladat megvalósítása két eljárásban foglal helyet: az elsőben meghatározzuk az egész lista minimumát, és ennek ismeretében hívjuk meg a tényleges kivonást végző második eljárást!

```
eljárás kivon :l
 eredmény kivon_s minimum :l :l
vége

eljárás kivon_s :min :l
 ha üres? en :l [eredmény (mondat (első :l) - :min)]
  eredmény (mondat (első :l) - :min kivon_s :min en :l)
vége
```

92. *Vizsgáljuk meg, hogy mely zárójelek feleslegesek az előbb elkészített **kivon_s** eljárásban!*

93. *Alakítsuk úgy át az eljárást, hogy alkalmas legyen egy szó betűiből való kivonásra is! A kivonás alatt a karakterek kódja szerinti kivonást értsünk!*

94. *Az eljárások ismeretében készítsük el a Caesar-féle betűeltolásos titkosítás eljárását! A módszer lényege az, hogy a karakterek helyett egy megadott értékkel módosított karaktereket írunk ki!*

*95. Egy cég havi bevételei és kiadásai alapján állapítsuk meg, hogy melyik hónapban volt a legnagyobb a havi nyereségük! A bemenet [hónapnév bevétel kiadás] hármasokból épül fel!*

## Számzár ( NTTV 2001/02. 5-8. évfolyam 2. forduló)

Adjuk össze egy legfeljebb 100 jegyű természetes szám számjegyeit! Ha a kapott szám nem egyjegyű, akkor ennek újra adjuk össze a számjegyeit, s az eljárást folytassuk, amíg egyjegyű számot nem kapunk. Egyes kiinduló számok esetén a végeredmény 1 lesz. Nevezzük ezeket a kiinduló számokat *„egyes "* számoknak.

```
91: (91-->10-->1)
1998: (1998-->27-->9)
1999: (1999-->28-->10-->1)
```

*34. Készítsünk programot, amely beolvas egy legfeljebb 100 jegyű természetes számot, majd kiírja, hogy a szám „ egyes " szám-e!*

| BEMENET | KIMENET |
|---|---|
| 91 | EGYES |
| 1998 | NEM EGYES |
| 1999 | EGYES |

A feladat klasszikus megoldása az összegzés elemi algoritmusa, kissé megfűszerezve a maradék és hányados képzés felhasználásával a számjegyek leválasztásához. Mindez legtöbbször viszont sajnos csak 12 számjegyig működik.

Ez után külön gondként jelentkezik az a probléma, hogy 12 jegynél nagyobb egész típus nemigen fordul elő, így stringként kell a számot beolvasni, majd típuskonvertálással a karakterekből számot képezni.

A megoldás LOGO-ban persze azonnal adódik: nem kell számértékkel dolgozni, dolgozzunk a szóval, ami sokkal természetesebb is, és a számjegyek leválasztása egyszerűbb is lesz. A szó hossza pedig szinte bármekkora lehet!

```
eljárás jegyösszeg :szám
 ha :szám < 10 [eredmény :szám]
 eredmény (első :szám) + jegyösszeg en :szám
vége
```

A LOGO szövegfeldolgozási hatékonyságát bemutatandó írjunk be 300 darab egyes számjegyből álló számot, ami nyilvánvalóan nem lehet már egész, és a szokásos stringhosszon is átlép. Ennek ellenére a bevitt értékkel gond nélkül megbirkózik az eljárás, és ki is írja a várt 300-as eredményt.

```
? mutat jegyösszeg
111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111
300
```

A ténylegesen kért szöveg kiírása most már egyszerű: azt kell csak ellenőriznünk, egyjegyű lett-e már az összeg. Ha nem, akkor egy újabb összegzést kell kémünk.

```
eljárás egyes :szám
 ha jegyösszeg :szám > 9 [eredmény egyes jegyösszeg :szám]
 hak jegyösszeg :szám = 1 [eredmény "EGYES][eredmény "|NEM
EGYES|]
vége
```

Nézzünk meg egy-két futási eredményt!

```
? mutat egyes 18
NEM EGYES
? mutat egyes 19
EGYES
? mutat egyes 1999
EGYES
```

A feladat meggyőzően mutatja, hogy a LOGO különösen alkalmas szövegfeldolgozási problémák megoldására, de semmivel sem kevésbé alkalmas a nyelv vektorokra épülő feladatainak megoldására.

## Halmazműveletek

A következő algoritmusok halmazokkal végzett műveletek megvalósítását mutatják be. Feltesszük, hogy a halmazok olyan listák, amelyek nem tartalmaznak azonos elemeket. A lista ezen tulajdonságát ellenőrizhetjük.

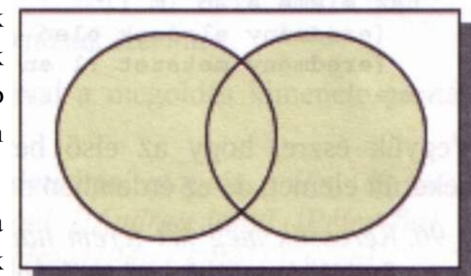
*35. Készítsünk függvényt, amely egy listáról megállapítja, hogy lehet-e halmaz!*

A megoldás első gondolata az, hogy ha a lista üres, vagy egy elemű, akkor igaz, hogy halmaz. Ha legalább két eleme van, akkor nézzük meg, hogy az első eleme eleme-e a lista első nélküli részének! Ha igaz, akkor nem lehet halmaz, ha hamis, akkor hagyjuk el az első elemet, és nézzük meg, hogy a lista többi részére fennáll-e a tulajdonság.

```
eljárás halmaz? :l
 ha elemszám :l < 2 [eredmény "Igaz]
 hak eleme? első :l en :l
   [eredmény "hamis]
   [eredmény halmaz? en :l]
vége
```

### Unió

Két halmaz egyesítésén azon elemek halmazát értjük, amelyek legalább az egyik bemeneti halmaz elemei voltak. Az unió elkészítésénél ezért a következőképpen járunk el.



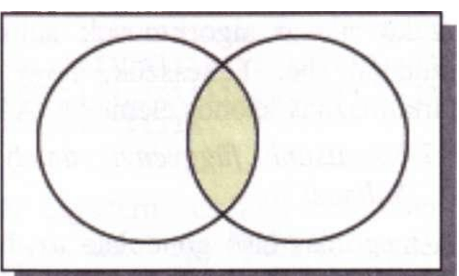
Az egyik bemeneti lista lesz a kimenet, ha a másik lista üres. Ha nem üres a második lista, akkor megnézzük az első elemét, hogy eleme-e az első bemenetnek. Ha igen, akkor nem kell az unióhoz csatolnunk, ha nem, akkor az unió új, például utolsó eleme lesz. Mindkét esetben ezután a feldolgozott elemet elhagyjuk, és az eljárást meghívjuk az új bemenetekkel.

így az első lista gyűjti az unió elemeit, a második lista pedig egyre fogy.

*36. Készítsünk függvényt, amely két halmaz unióját adja meg!*

```
eljárás unió :l :m
 ha üres? :m [eredmény :l]
 hak eleme? első :m :l
    [eredmény unió :l en :m]
    [eredmény unió utolsónak első :m :l en :m]
vége
```

## Metszet



Ha két halmaz metszetét szeretnénk meghatározni, akkor azokat az elemeket kell összegyűjtenünk, azaz kiválogatnunk, amelyek mindkét halmazban, esetünkben tehát mindkét bemeneti listában benne vannak.

Ez tulajdonképpen egy kiválogatás. A metszet kezdetben üres lista lesz, és is marad, ha valamelyik bemenet üres. Például, ha a második bemeneti lista üres, készen is vagyunk, a kimenet az üres lista lesz.

Ha nem üres a második bemenet, akkor az első elemét megnézzük, hogy szerepel-e az első listában. Ha igen, akkor ez az elem benne lesz a metszetben, hozzá kell majd fűzni az e nélküli maradék elemek metszetéhez. Ha nem eleme, akkor egyszerűen elhagyjuk, és nélküle hívjuk meg a többi elem metszetét meghatározó eljárásunkat.

*37. Készítsünk függvényt, amely két halmaz metszetét adja meg!*

```
eljárás metszet :l :m
 ha üres? :m [eredmény []]
 hak eleme első :m :l
    [eredmény elsőnek első :m metszet :l en :m]
    [eredmény metszet :l en :m]
vége
```

Vegyük észre, hogy az első bemenetből elhagyhatnánk a már metszetbe bekerült elemet, de ez érdemben nem csökkentené a vizsgálatok idejét.

*96. Keressük meg két terem napi órarendjének ismeretében, hogy mikor foglalt legalább az egyik közülük! Az órarendlista az órák sorszámát tartalmazza!*

*97. Adjuk meg azokat a diákokat, akik több csapatban is játszanak a három csapat névsorlistája alapján!*

## Szétválogatás

A kiválogatási tétel bemeneti listájából azokat gyűjti csak ki, amelyek a keresett tulajdonsággal rendelkeznek. Szétválogatásról akkor beszélünk, ha a bemeneti listát két - vagy több - kimeneti listára bontjuk széjjel, a megadott tulajdonság alapján.

Legegyszerűbb esetben egy bemenet alapján két kimenetünk lesz. E speciális eset megoldása elegendő is, hiszen ha több részre kell bontanunk a bemenetet, akkor első lépésként leválasztjuk az első tulajdonság szerinti kimenetet, majd a többi elemből a második és így tovább részlistákat.

Az **eredmény** parancs csak egyetlen listát adhat vissza, ezért egy kis trükkhöz folyamodunk: a bemenetet egy olyan listába fogjuk szétválogatni, amely két részlistából áll, az első része a keresett tulajdonsággal rendelkező, a második része az ezzel nem rendelkező elemeket tartalmazza.

*38. Készítsünk függvényt, amely a bemeneti lista elemeit szétválogatja aszerint, hogy melyik elem nagyobb vagy nem kisebb 10-nél!*

```
eljárás szétválogat :l :kl :k2
 ha üres? :l [eredmény utolsónak :k2 elsőnek :kl []]
 hak első :l > 10
    [eredmény szétválogat en :l utolsónak első :l :kl :k2]
    [eredmény szétválogat en :l :kl utolsónak első :l :k2]
vége
```

A megoldás érdekessége a két üres lista összefűzésének módja egy kételemű listává. Ha ugyanezt mondat paranccsal adtuk volna meg, akkor egyetlen listát kaptunk volna!

```
? mutat szétválogat [] [] []
 [[] []]
? mutat szétválogat [1 2 3 23 1 34 3] [] []
 [[23 34] [12 3 13]]
```

Az alapeset megoldása alapján készítsük el a következő feladatok megoldását!

*98. Válogassuk szét egy sorozat elemeit paritásuk szerint!*

Például a **[1 2 3 5 7 8 9 0]** bemeneti listával a megoldás kimenete paritás szerint **[[13 5 7 9][2 8 0]]** lesz.

*99. Adjuk meg egy osztály név és nem adatpárjai alapján a lány és fiú tanulóinak névsorát! A bemenet például [[Andrea lány] [Péter fiú] [Ottó fiú]] alakú, akkor a kimenet [[Andrea][Péter Ottó]] legyen!*

A megvalósítás alapján elkészíthetjük a több részlistára bontó eljárásokat is.

*100.Bontsuk szét a tanulókat elért osztályzatuk alapján csoportokra! Az osztályzatokat név- és jegypárokat tartalmazó listában adjuk meg!*

*101. Válogassuk szét egy lista számait aszerint, hogy melyek oszthatóak a 2 különböző hatványaival! Az első részlista a kettővel nem osztható számokat tartalmazza, a második a csak 2-vel, a harmadik a 4-gyel és így tovább! Figyeljünk arra, hogy az utolsó részlista nem lehet üres!*

*102.Három munkás egy heti munkanapjait tartalmazó listák alapján határozzuk meg, hogy mely napokon nem dolgozott közülük senki, csak egyikük, csak ketten, vagy mindannyian azon a héten!*

## Fésűs egyesítés

A szétválogatás mellett gyakran szükséges listákat egyesítenünk. Két vagy több lista egyetlen listába fűzése egyszerűen megvalósítható a **mondat** utasítással.

```
? mutat (mondat [1 3][2 3 4][2 5])
 [1 3 2 3 4 2 5]
```

Vegyük észre, hogy ekkor a kapott lista elemeinek ismétlődésére és rendezettségére sem fordítunk figyelmet. Ha az egyesített listában nem lehet elemismétlődés, akkor unióképzésről beszélünk majd, amit halmazműveletként oldottunk meg.

Érdekesebb számunkra az az eset, amikor a bemeneti listák rendezettek, és a rendezettséget meg kell tartanunk. Ekkor beszélünk fésűs egyesítésről.

*39. Két, már rendezett listát egyesítsünk egyetlen listába a rendezés megtartása mellett!*

A feladat megoldásának alapgondolata az, hogy ha az egyik bemeneti lista üres, akkor a feladattal elkészültünk, a másik bemenet és a már egyesített rész összefűzése a végeredmény. Ha nem üres egyik lista sem, akkor a kimeneti listába elhelyezzük a két listából a sorban következő elemet, ami a két első elem közül a kisebb lesz.

```
eljárás fésűs :l :m :k
 ha üres? :l [eredmény (mondat :k :m)]
 ha üres? :m [eredmény (mondat :k :l)]
 hak első :l > első :m
   [eredmény fésűs :l en :m utolsónak első :m :k]
   [eredmény fésűs en :l :m utolsónak első :l :k]
vége
```

Az eljárás három paraméteres, az utolsó a kimenetet jelenti, ami kezdetben üres lista, így a meghívása

```
fésűs :l :m []
```

alakú lesz.

*103. Lányok és fiúk külön névsora alapján adjuk meg az osztálynévsort!*

*104. A különböző csapatok névsora alapján adjuk meg a csapatjátékosok névsorát! Figyeljünk arra, hogy egy diák több csapatban is játszhat!*

## Mégis, hány elemű?

A listák elemei újabb listák is lehetnek.

*40. Állapítsuk meg egy ilyen összetett listáról, hogy ténylegesen hány elemi adatot tartalmaz!*

Az üres lista elemszámát a továbbiakban is 0-nak tekintjük, ahogy azt az **elemszám** parancs is megadja.

```
eljárás elem.szám :l
 ha üres? :l [eredmény 0]
 ha lista? első :l
    [eredmény (elem.szám első :l) + elem.szám elsőnélküli :l]
 eredmény 1 + elem.szám elsőnélküli :l
vége
```

Az eljárás eredményét vizsgáljuk meg néhány kimenetre!

```
? mutat elem.szám [[2 3 5] [] [4]]
4
? mutat elem. szám [[2 3 5 a] [] [4 t] ]
6
? mutat elem. szám [ [] [] [] ]
0
? mutat elem.szám [ [ 2 3 5 1 ] [3] [4 t [z 6]]]
9
? mutat elem.szám [[[a b]]]
2
```

Hogy érzékeljük, nem az **elemszám** függvényről van itt szó, adjuk ki az **elemszám** parancsot is a két legutóbbi listára!

```
? mutat elemszám [[[a b]]]
1
? mutat elemszám [ [ 2 3 5 1 ] [3] [4 t [z 6]]]
3
```

Az eredmény alapján látható, hogy jelentős eltérés van a két eljárás között.

## *Alapvető rendezések*

*A* listák kezelése során gyakori probléma, hogy a lista elemeit egy megadott szempont - a lista elemein értelmezett rendezési reláció - szerint rendezni kell. Ezt a feladatot a legtöbb programozási nyelvben saját magunknak kell megoldani. Az Imagine-ben viszont létezik a **rendez** parancs!

```
? mutat rendez [3 2 15 6]
 [2 3 6 15]
? mutat rendez "maci
Nem tudom hogy csináljam a(z) rendez-t
```

Mint látható, nem alkalmazhatjuk a **rendez** parancsot szóra, de képsorra sem. Ezen kis hátrányt ellensúlyozza, hogy a rendezést viszont alkalmazhatjuk akkor is, ha az elemek maguk is listák!

```
? mutat rendez [[1 c][3 a][2 b]]
 [[1 c] [2 b] [3 a]]
? mutat rendez [[c 1] [a 3] [b 2]]
 [[a 3] [b 2] [c 1]]
```

Mint látható, ebben az esetben az allisták első eleme szerint rendez.

```
? mutat rendez [[1 c][1 a][1 b]]
 [[1 a] [1 b] [1 c]]
? mutat rendez [[a 3] [b 1] [a 2]]
 [[a 2] [a 3] [b 1]]
```

A fenti kép példa pedig azt mutatja be, hogy ha az első elem szerint nem egyértelmű a sorrend, akkor a második elem alapján végzi a rendezést.

*105.Készítsünk eljárást, amely csökkenő sorba rendez!*

*106.Készítsünk eljárást, amely a kételemű listákból álló listát a második eleme szerint rendezi!*

*A* hagyományos, tömbszerkezetre építő rendezési módszerek közül a buborék, a minimum és a beszúró rendezés elvét célszerű megismerniük. Ezen elvek átültetése lista szerkezetre is lehetséges.

### Buborékrendezés

A buborékrendezés esetén a szomszédos elemeket hasonlítjuk össze.

Ha például növekvő sorba kell rendezni, akkor a bal oldali elem nem lehet nagyobb a jobb oldalinál. Ha az első nagyobb, akkor a két elem sorrendjét meg kell cserélnünk.

*41. Rendezzünk egy listát a buborék elv alkalmazásával!*

Ez azt jelenti számunkra, hogy egy végignézés folyamán a bemeneti sorozat első és második elemét kell vizsgálnunk. Ha az első nagyobb a másodiknál, akkor csere kell, különben nem. Ezt úgy valósítjuk meg, hogy a kettő közül a kisebbet eltesszük a végeredménybe, és a többi elemre tovább keressük, hogy melyik az első kettő közül a kisebb, amíg nem lesz a bemenet egy elemű.

```
eljárás menet :l
 ha üres? en :l [eredmény :l]
 hak első :l > első en :l
   [eredmény elsőnek (első en :l) menet elsőnek (első :l) (en en
il)]
   [eredmény elsőnek (első :l) menet en :l]
vége
```

A ***menet*** eljárás végén kapott lista olyan, hogy az utolsó eleme biztosan mindegyik elemnél nem kisebb. Ezt válasszuk le, és a továbbiakban utolsónak fűzzük hozzá a többi elem rendezett listájához.

```
eljárás rend.bub :l
 ha üres? en :l [eredmény :l]
 eredmény utolsónak (utolsó menet :l) (rend.bub un menet :l)
vége
```

A kialakult eljárásban nem szép és felesleges is, hogy a menet eljárást kétszer kell meghívni ugyanazzal a paraméterrel.

### *Javított buborékrendezés*

A ***menet*** eljárás során előfordul az, hogy már nincs tényleges sorrendbeli hiba, azaz az igaz ágat nem hívjuk meg. Ha nincs már csere, akkor egy logikai változóval ezt tárolhatjuk. Ha ez igaz marad, akkor nem volt csere, különben ismét szükséges.

*107. Módosítsuk úgy a menet és **rendbub** függvényt, hogy az első figyelje, történt-e csere a menetfolyamán, és ha már nem, akkor a **rena\bub** ne hívja meg többször a függvényt!*

### Minimum elvű rendezés

Ha a lista első eleme nem nagyobb, mint a legkisebb elem a lista többi részéből, akkor legyen a rendezett lista első eleme az első elem, különben a lista eddigi első elemét helyezzük el a lista fennmaradó részének a végére. Az így kapott átrendezett listával dolgozzunk tovább.

*42. Rendezzünk egy listát a minimum elv alkalmazásával!*

Az egyszerűnek tűnő minimum kiírása az első helyre azért nem működik, mert a minimum helyét a listában nem tudjuk. Persze ha tudnánk, egyszerűen ki lehetne emelni a minimum értékű elemet a listából, és áthelyezni az első helyre. Ennek a műveletigénye viszont nagyobb lenne, mint egyszerűen a nem legkisebb elemek újrafeldolgozása.

```
eljárás rend.min :l
 ha üres? en :l [eredmény :l]
 hak első :l > minimum en :l
    [eredmény rend.min utolsónak első :l en :l]
    [eredmény elsőnek első :l rend.min en :l]
vége
```

*108. Alakítsuk át az eljárást úgy, hogy a bemeneti listát csökkenő sorrendbe rendezze! A feladat megoldásához alkalmazzuk a **maximum** függvényt!*

*109.Alakítsuk át az eljárást úgy, hogy számolja meg, hány minimum függvény meghívás és hány darab önmeghívás történik meg! Az eljárást futtassuk le 100 elemű listákon, amelyek alapvetően rendezett, fordítottan rendezett, illetve véletlenszerűen generált 100 db számot tartalmaznak. Mit tapasztalunk? Mennyi a legjobb, legrosszabb és az átlagos meghívásszám?*

## Beszúró rendezés

A beszúró rendezés alapgondolata, hogy az elemek egy része már rendezett. A már rendezett elemek sorozatába kell egy újabb elemet beszúrnunk.

Ezt a feladatot két eljárással oldhatjuk meg kényelmesen.

*43. Rendezzünk egy listát a beszúró elv alkalmazásával!*

Az első eljárás egy rendezett sorozatba illeszt be egyetlen új elemet. Ha a lista üres, akkor az elem maga lesz az eredmény egyetlen eleme. Ha nagyobb a beszúrandó elem mint a már rendezett rész legutolsó eleme, akkor a végére illesztjük egyszerűen, különben az utolsó elemet eltesszük, és megvizsgáljuk, hogy az előtte lévő részben hová kell az új elemet elhelyezni.

```
eljárás beszúr :e :l
 ha üres? :l [eredmény utolsónak :e :l]
 hak utolsó :l < :e
   [eredmény utolsónak :e :l]
   [eredmény utolsónak utolsó :l beszúr :e un :l]
vége
```

Vegyük észre, hogy ez a ***beszúr*** eljárás önmagában is jól használható, mivel egy már rendezett listába tudunk a segítségével egy újabb elemet, a rendezést megtartva elhelyezni.

A fő eljárás egy teljesen rendezetlen sorozatból kiindulva végzi el a beillesztéseket. Az alapgondolat itt az, hogy az egy elemű illetve az üres lista rendezettnek tekinthető. Máskülönben az utolsó elemet leválasztjuk, és beszúrjuk a többi elemből álló listába.

```
eljárás rend.beszúr :l
 ha üres? un :l [eredmény :l]
 eredmény beszúr utolsó :l rend.beszúr un :l
vége
```

# Nevezetes feladatok

Ebben a fejezetben híresebb problémákat oldunk meg, amelyek azért különösen fontosak a számunkra, mert valamilyen megoldási módszert mutatnak be.

## Rekurzió alkalmazása

Könyvünkben már igen sokszor alkalmaztunk rekurziót, jóllehet a LOGO nyelv erőssége ez, de a következő két példánkban azt mutatjuk be, hogy néha jó, de lehet, hogy inkább célszerű kerülni.

### Hanoi torony

A történet szerint az egyik Brahma templom vezető szerzetese látomást élt át: az örökkévalóság elnyeréséhez a kolostor udvarán álló ezüst rúdra elhelyezett 64 darab porfír korongot át kell helyeznie az arany rúdra, de úgy, hogy a korongokat csak egyesével mozgathatja, nagyobbra csak a kisebbet tehet, és a feladat megoldásához segítségül felhasználhatja a réz rudat is.

A történet szerint az idős szerzetes elgondolkodott: *„Az ő feladata lényegében csak a legnagyobb, 64. korong áthelyezéséről gondoskodni a szabályok szerint az arany rúdra. A többi 63 korongot legidősebb tanítványa is átrakhatja helyette elsőként a réz rúdra, majd miután ő áthelyezte saját kezűleg a legnagyobb korongot az arany rúdra, a tanítvány átteszi rá a többi korongot is."*

A legidősebb tanítvány hasonlóan úgy gondolta, hogy az ő feladata sem igazán mind a 63 korong mozgatása, hanem a 63. korongon lévő 62 korong áthelyezését az arany rúdra rábízhatja a második legidősebb tanítványra. Miután ő elvégezte a feladatát, már át tudja tenni a réz rúdra a 63. korongot. A 2. tanítvány rápakolja a 62 korongot az arany rúdról, majd a mester átteszi az ezüst rúdról a 64. korongot. Ezután a 2. tanítvány lép újra a rudakhoz: a réz rúdról a 62 felső korongot az ezüst rúdra teszi, az így kiszabadított 63. korongot az első számú tanítvány átteszi az arany rúdra, majd a 2. tanítvány befejezi a feladatot: az ezüst rúdon lévő 62 korongot áthelyezi az arany rúdra.

A második legidősebb tanítvány is elgondolkodott, hogy mi is a feladata. És szólt a 3. legidősebb tanítványnak...

Nem folytatjuk, visszaélve az olvasó türelmével. De tudva azt, hogy egy porfír korong legalább 100 kg lehetett volna ilyen méretben, nem valószínű, hogy egy másodpercnél rövidebb idő alatt át lehetett volna helyezni. Ha mégis, akkor vegyük észre, hogy a mester 1, az első tanítvány 2, a második 4, a harmadik 8 és így tovább alkalommal helyezett volna át korongot, azaz a 63.

tanítvány $2^{63}$-on áthelyezést végzett volna. Ez összesen $2^{64}$-1 alkalom, ami legkevesebb ennyi másodpercet vett volna legalább igénybe, ami évekre átszámolva és kerekítve 584,5 milliárd év!

E szám láttán biztos állíthatjuk, hogy ha a feladat teljesíthető, az örökkévalóság a jutalma!

A feladathoz kapcsolódó történet lehet hogy igaz, lehet hogy nem. Az viszont tény, hogy François Édouard Anatole Lucas (1842-1891) francia matematikus 1883-ban saját ötleteként publikálta a feladatot, annak bemutatására, hogy létezik olyan probléma, amely nem oldható meg csak exponenciális időigénnyel.

Pontosítsuk a feladat szövegét!

44. *Három rúd közül az egyiken n db korong helyezkedik el, csökkenő nagyságban elhelyezve. A korongokat át akarjuk helyezni egy másik rúdra. A feladat megoldása folyamán az alábbi szabályokat kell betartanunk:*

- Egyszerre csak egy korong helyezhető át.
- Nagyobb korongra tehető csak kisebb korong.
- A korongok elhelyezésére csak a rudak használhatók.

A rudakat célszerű elneveznünk: legyen az A rúd a kiindulási hely neve, és C legyen a célrúd. Ha az A rúdról n-1 korongot áthelyezünk a B rúdra, akkor ezután már a legnagyobb korongot A-ról C-re áthelyezhetjük, és nincs más feladatunk, mint a B-n lévő n-1 korongot C-re átrakjuk. A megoldást ezzel visszavezettük eggyel kevesebb korong áthelyezésének problémájára.

```
eljárás hanoi :n :r :e :a
 hak :n = 1 [(kiír "tégy :r "oszlopról :e "oszlopra. )]
 [hanoi :n - 1 :r :a :e
  (kiír "tégy :r "oszlopról :e "oszlopra. )
  hanoi :n - 1 :a :e :r]
vége
```

Nézzük a program futásának eredményét 3 korong esetén!

```
? hanoi 3 "réz "ezüst "arany
tégy réz oszlopról ezüst oszlopra.
tégy réz oszlopról arany oszlopra.
tégy ezüst oszlopról arany oszlopra.
tégy réz oszlopról ezüst oszlopra.
tégy arany oszlopról réz oszlopra.
tégy arany oszlopról ezüst oszlopra.
tégy réz oszlopról ezüst oszlopra.
```

110. *Készítsünk programot, amely nem csak szövegesen, hanem grafikusan is bemutatja a korongok áthelyezését!*

## Váltsunk!

A korábban már megismert algoritmusok közül rekurzióra épül a számrendszerek közötti átváltást végző eljárás is.

*45. Készítsünk eljárást, amely egy megadott számot tetszőleges számrendszerbe vált át tízesből!*

```
eljárás váltó :y :x
 ha :y > :x - 1 [váltó egészhányados :y :x :x]
 kiírérték maradék :y :x
vége
```

Az eljárás közvetlenül a képernyőre ír, így a tényleges átváltási eredményt nem tárolja el:

```
? váltó 27 3
1000? váltó 20 2
10100
```

Az eljárás másik kényelmetlensége, hogy a **kiírérték** parancs nem emel sort, emiatt a következő kiírás az előző utolsó sorában folytatódik.

*111.Alakítsuk át a **váltó** eljárást úgy, hogy a kiírás végén sort emeljen!*

*112.Alakítsuk át a **váltó** eljárást úgy, hogy az átváltott értéket kimenetként adja meg!*

*113.Alakítsuk át a **váltó** eljárást úgy, hogy az alkalmas legyen tizenhatos számrendszerbeli számok kijelzésére is! Ebben a számrendszerben a 10 értéket A, a 11-et B, a 12-t C, a 13-at D, a 14-et E és a 15-öt F jelöli!*

## Permutáció

Gyakran előforduló feladat, hogy egy lista elemeinek egy eredetitől különböző sorrendjét kell előállítanunk. Egy ilyen lehetséges sorrendet az elemek permutációjának nevezzük.

A permutáció megvalósítására használhatjuk az **összekever** parancsot.

```
? mutat összekever [12 3 4 5 6]
 [6 3 1 5 4 2]
? mutat összekever [12 3 4 5 6]
 [5 1 3 6 4 2]
```

Mint látható, két egymást követő meghívás más eredményt ad.

*114. Készítsünk eljárást, amely egy bemenetként adott lista összes lehetséges permutációját előállítja!*

*115. Mit tapasztalunk, ha a lista elemszáma 10, 20, 30?*

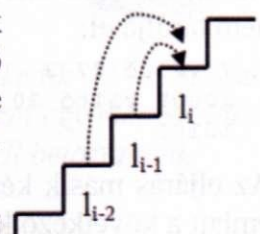
## *Jó a rekurzió, de mindig?*

Vannak olyan problémák, amelyek megoldása során a rekurzív definíció egy az egyben való átvétele ugyan kézenfekvő lenne, de helyette inkább némileg átgondoltabb rekurziót tartalmazó eljárást kell készítenünk.

A Fibonacci sorozat talán az egyik legismertebb rekurzív számsorozat. A feladat számtalan átfogalmazásban ismert. Nézzünk példaként egyet!

*46. Sári elhatározza, hogy az iskola kapujához vezető 10 lépcsőn minden nap másképp halad fel. Hány nap tud különböző módon felmenni, ha egy vagy két lépcsőfokot tud csak lépni?*

Mint az ábrán is látható, az i-edik lépcsőfokra az i–1–edik és a i–2–edik fokról lehet feljutni, azaz sorozatra jellemző $l_i=l_{i-1}+l_{i-2}$ képlettel kell itt is dolgoznunk, és ha nem lenne lépcső vagy ha csak egy lépcső lenne, akkor is csak 1-1 felmenetel lenne lehetséges.




Feladatunk tehát meghatározni, hogy a mennyi Fibonacci sorozat n-edik eleme.

A sorozat az ismert rekurzív definícióval adott: $f_1$ és $f_2$ legyen 1, és ezeknél nagyobb n-re a sorozat n-ik eleme az előző két elem összegeként kapható meg, azaz $f_i=f_{i-1}+f_{i-2}$.

A Fibonacci sorozat definíciója szinte sugallja, hogy készítsük el a fenti meghatározás mintájára a rekurzív függvényt!

```
eljárás fibonacci :n
 ha :n<=2 [eredmény 1]
 eredmény (fibonacci :n-1) + (fibonacci :n-2)
vége
```

Az eljárással semmi gondunk sincs mindaddig, amíg ki nem próbáljuk például a

```
ki fibonacci 30
```

meghívást. A türelmetlenebbek megijedhetnek a közel 10 másodperces futástól, de aki kivárja, az megkapja a kívánt értéket, a 832040 számot.

Még rosszabbnak látszik a ***ki fibonacci 40*** parancs után a helyzet, mivel ekkor a futás már szinte kivárhatatlanul hosszú, a mi gépünkön 1 óra után még nem adott eredményt!

Miért lesz ilyen lassú a 40. elem meghatározása?

*116.A kérdés megválaszolása érdekében alakítsuk át egy kicsit a programot! Számoljuk meg, hány alkalommal kerül meghívásra a **fibonacci** függvény!*

Az tapasztalhatjuk, hogy az elemszám eggyel való növelése megduplázza a meghívások számát! Így már nem meglepő, hogy a 30. és a 40. elem kiszámítása között ezerszer több idő telik el, és szinte reménytelen lesz kivárni az 50. elem meghatározását!

*117.Készítsünk időmérőt a **fibonacci** programhoz! A keretprogram mérje meg és mutassa meg az 1-től 30-ig meghatározott elemek kiszámolási idejét! Mit tapasztalunk az értékek vizsgálatát elvégezve?*

*118.Készítsük el a **fibonacci** program azon változatát, amely kiírja, hogy melyik értékkel hányszor hívjuk meg az eljárást!*

Jóval hatékonyabb módszer a felesleges elem újraszámolások elkerülése. Kézi számolás esetén is leírjuk a már kiszámolt elemeket, és így határozzuk meg a következő elemet, és az n. meghatározása során tulajdonképpen minden előzőt is kiszámolunk, de csak egyszer. Az egyszeri számolás érdekében tegyük el egy listába a már ismert elemeket, amelyek *n*-nél nem nagyobb a sorszáma, és amikor a lista *n* elemű lesz, akkor eljutottunk a kért elemhez, azaz a kért elem a lista utolsó eleme lesz!

```
eljárás fibonacci :n :l
 ha :n=1 [eredmény 1]
 ha elemszám :l = :n [eredmény utolsó :l]
 eredmény fibonacci :n utolsónak (utolsó :l)+(utolsó un :l) :l
vége
```

A módosított Fibonacci függvény meghívása például a

```
 ki fibonacci 8 [1 1]
```

paranccsal történhet meg. És mint várható volt, az előbb tapasztalt közel három óra helyett a 40. elemet szinte azonnal kiszámolja: 102334155, ami láthatóan nem olyan nagy érték, hogy emiatt lassult volna le ennyire a program.

*119.Valóban szükség van a módosított Fibonacci függvény első utasítására? Elhagyható-e ez akkor is, ha általánosítjuk a sorozatot?*

*120.Készítsünk **másik.irány** nevű eljárást, amely kiírva az eredeti Fibonacci sorozat szabályát, alkalmazva a 1 1 előtti elemeket is! Az eljárás paraméterként kapja meg a negatív irányban kiírandó elemek számát! Például 7 esetén a kiírandó sorozat [-8 5 -3 2 -1 1 0] lesz!*

*121.Határozzuk meg n! értékét, ha n egész szám! Az n! – kiolvasva n faktoriális – azt a számot jelenti, amit úgy kapunk, hogy 1-től n-ig összeszorozzuk az egész számokat egymással. A 0! és az 1! definíció szerint 1-et jelent.*

*122.Adjuk meg a Lucas-számok közül az n-ediket, ha tudjuk, $L_0=2$, $L_1=1$ és $L_{n+2}=L_n+L_{n+1}$!*

# *Visszalépéses algoritmus*

## 8 királynő probléma

Állítólag Carl Friedrich Gauss (1777. 4. 30. – 1855. 2. 23.) vetette fel 1850-ben, hogy el lehet-e helyezni a sakktáblán úgy 8 királynőt – hivatalos szóhasználattal vezért –, hogy azok ne üssék egymást.

A feladat megoldása nagy kitartást igényel, és alapvetően próbálgatással lehet megoldani. A számítógép pedig különösen alkalmas a rendszeres formában történő próbálgatásra.

*47. Helyezzünk el 8 vezért a sakktáblán úgy, hogy ne üssék egymást!*

A feladat megoldásaként most megelégszünk egyetlen jó elrendezés megkeresésével, de a következő megállapítások igazak az összes lehetséges elrendezés meghatározása esetében is.

A következő megállapítások nyilvánvalók:

- Minden sorban csak egy vezér lehet.
- Minden oszlopban csak egy vezér lehet.
- Minden ↖ irányú átlóban csak egy vezér lehet.
- Minden ↙ irányú átlóban csak egy vezér lehet.

Kevésbé nyilvánvaló, de a fenti állítások következményeként adódik, hogy a jó elrendezés egy olyan 8 elemű lista, amelyben az i-edik elem az i-edik oszlopban lévő vezér sorát adja meg:

- Minden szám 1-től 8-ig csak egyszer szerepelhet.
- A ↙ átlók esetén a sor és oszlopkoordináták különbsége állandó.
- A ↖ átlók esetén a sor és oszlopkoordináták összege állandó.

Az utóbbi tények alapján az új vezér a már korábban elhelyezettekhez akkor tehető hozzá, ha:

- még nem szerepelt a már felhelyezettek listájában;
- a már felhelyezettek listáján végighaladva ↙ irányba nem találhatunk vezért, azaz a már fenn lévő elemek listájának utolsó eleme nem lehet 1-gyel kisebb, az utolsó előtti 2-vel, és így tovább;

- a már felhelyezettek listáján végighaladva K irányba nem találhatunk vezért, azaz a már fenn lévő elemek listájának utolsó eleme nem lehet 1-gyel nagyobb, az utolsó előtti 2-vel, és így tovább.

Ez utóbbi átalakítás alapján már elkészíthetjük azt a függvényt, amely a £ irányú átlókat ellenőrzi:

```
eljárás le? :l sitt
 ha (vagy üres? :l :itt=0) [eredmény "IGAZ]
 hak utolsó :l = :itt-1
   [eredmény "HAMIS]
   [eredmény le? un :l :itt-1]
vége
```

Igaz eredményt akkor kapunk, ha kimentünk a tábláról - *:ití=0* -, vagy ha a már feltettek listáján végighaladtunk.

Hasonlóan a K irányba haladás vizsgálata is megadható:

```
eljárás fel? :l sitt
 ha (vagy üres? :l :itt=9) [eredmény "IGAZ]
 hak utolsó :l = :itt+1
   [eredmény "HAMIS] [eredmény fel? un :l :itt+1]
vége
```

A sorfoglaltság ellenőrzéséhez elegendő azt megvizsgálni, hogy az új sor eleme-e már a felhelyezettek listájának. Ha nem, akkor megvizsgáljuk, ha igen, akkor továbbléptetjük. Ez utóbbira persze csak akkor kerülhet sor, ha a tábla felső sorát még nem értük el. Ha ez áll fenn, akkor elakadtunk, le kell venni az új vezért, és az előző oszlopban lévővel kell próbálkoznunk, azaz ezt kell - ha lehetséges - egy sorral feljebb helyezni, és megvizsgálni.

```
eljárás próbál :l :új
 ha esz :l= 8 [eredmény :l]
 ha :új=9 [eredmény próbál un :l 1 + utolsó :l]
 hak (és (:új<9) eleme? :új :l)
   [eredmény próbál :l :új+1]
   [hak (és fel? :l :új le? :l :új)
    [eredmény próbál utolsónak :új :l 1]
    [hak :új=8
     [hak utolsó :l <8
      [eredmény próbál un :l 1+ utolsó :l]
      [eredmény próbál un :l 1]]
     [eredmény próbál :l súj+1]
    ]
   ]
vége
```

Figyeljük meg a fenti eljárásban, hogy az új vezér csak akkor kerül be a már elhelyezett vezérek listájába, ha megfelel a feltételeknek, és visszalépünk

akkor – az utolsó elemet töröljük a vezérlistából –, ha már nem tudjuk továbbléptetni az új sorban a vezért.

A ***próbál*** eljárás meghívása a **mutat** ***próbál [] 1*** módon történhet, de megadhatunk tetszőleges kezdőlistát is, amely azért legyen helyes. A mintaként megadott elhelyezés a ***próbál [2 4 1] 1*** meghívással alakult ki.

*123.A képen nem az első lehetséges elrendezés látszik, hanem egy azzal fedésbe nem hozható. Add meg az első lehetséges elrendezést!*

*124.Adjuk meg az összes lehetséges elrendezést!*

*48. Készítsük el a 8 királynő feladat grafikus megjelenítését!*

A feladat kiegészítéseként adjuk hozzá már meglévő eljárásaink mellé a sakktáblát megjelenítő, a vezérek megjelenítését és mozgatását végző eljárásokat.

Az ***indító*** eljárásban egy ***"tábla*** nevű, 400×400 képpontos, sakktábla alakú teknőcruha lenyomatát kértük. Az 1-től 8-ig nevű vezéreket megjelenítő teknőcöket létrehoztuk már a memóriában.

```
eljárás indító
 törölképernyő
 kér "tábla [xypoz! 50 0 lenyomat]
 ism 8 [kér hányadik [tf xypoz! -175+hányadik*50 -225
        alak! betöltőüt "svezer]]
 ki próbál [] 1
vége
```

Az SVEZER.LGF a sötét vezér alakja. Ezt a sakktáblával – TABLA.JPG – együtt letölthetjük a könyv honlapjáról.

A már táblán lévő vezéreket a ***rak***, míg a fel nem használt vezéreket a ***többi*** eljárás mozgatja, utóbbi veszi le a tábláról a 8. sorba érkezett és nem elhelyezhető vezéreket.

```
eljárás rak :l
 ha nem üres? :l [kér esz :l [ypoz! -225+50* utolsó :l rak un
:l]]
vége

eljárás többi :l
 ism 8-esz :l [kér 9-hányadik [ypoz! -225]]
vége
```

A ***rak*** eljárást a ***próbál*** első utasításaként célszerű meghívni:

```
eljárás próbál :l :új
 rak :l többi :l várj 10
...
```

A **várj 10** parancs a láthatóság miatt lényeges. Az eljárás többi része változatlan maradt.

## Mohó algoritmus

A sokszor célravezető probléma megoldási stratégia bemutatására a címletezés feladatot oldjuk meg. E probléma pont olyan, hogy a rendelkezésre álló címletektől függően kell eldöntenünk, hogy alkalmazható-e a módszer.

### Amikor lehetünk mohók

*49. A feladat a következő: a lehető legkevesebb címlettel fizessünk ki egy összeget! Az elkészített eljárás kérje be a fizetendő összeg nagyságát, valamint a rendelkezésre álló címleteket!*

A feladat megoldására a mohó algoritmus megfelelő, ha a valós életben használatos címleteket adjuk meg, azaz a 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000 Ft-os áll rendelkezésre.

> Vegyük észre, hogy ezek a címletek olyanok, hogy egyik sem nagyobb a rákövetkező címlet felénél, valamint a legkisebb címlet 1 Ft.

```
eljárás fizet :mit :címlet
 ha :mit = 0 [eredmény "vége]
 hak :mit >= első :címlet
 [(ki (egészhányados :mit első :címlet) "db első :címlet "kell)
   eredmény fizet (maradék :mit (első :címlet)) en :címlet]
 [eredmény fizet :mit en :címlet]
vége
```

Meghívása a címletlistával például így történhet:

```
? ki fizet 159 [1000 500 200 100 50 20 10 5 2 1]
1 db 100 kell
1 db 50 kell
1 db 5 kell
2 db 2 kell
vége
```

### *Fizessünk hármas pénzzel!*

Az előző algoritmus kényelmetlensége, hogy a címleteket nekünk kell megadnunk. Tegyük fel, hogy a pénzérmék speciális sorozatot alkotnak, 1, 3, 9, 27, 81, stb nagyságúak, azaz a három hatványai.

Ekkor a címletek előállítása is megoldható programmal. Ezt oldjuk meg elsőként, és hívjuk meg rá az előbbi eljárást!

A feladat tehát a következő módon oldható meg:

```
eljárás hármas :mit :címlet
 ha :mit >= 3* első :címlet [eredmény hármas :mit (elsőnek
3*első :címlet :címlet)]
 eredmény fizet :mit :címlet
vége
```

A meghívása most egyszerűbb lesz:

```
hármas :mit [1]
```

### Amikor nem lehetünk mohók

A feladat viszont nem oldható meg általános esetben is a mohó algoritmussal.

Ha például a 10 Ft a kifizetendő összeg, és [8 5 1] Ft-os címletek állnak rendelkezésünkre a korábbi eljárás hibás lesz, ugyanis a ***fizet*** algoritmus ebben az esetben a 8 Ft-os érmét választaná ki elsőként, és 2 db 1 Ft-os érmét venne mellé, jóllehet a 2 db 5 Ft-os érme választása lenne a tényleges megoldás.

> A címletezést általános címletekkel dinamikus programozással kell megoldanunk. Az általános megvalósítást a harmadik kötetben adjuk meg.

## *Vegyes gyakorló feladatok*

*125. Készíts függvényt, amely összeszorozza egy lista minden elemét!*

*126. Készíts függvényt, amely két rendezett lista elemeit hasonlítja össze, ha azonos a két lista, akkor IGAZ, ha eltérést talál, akkor HAMIS értéket ad vissza!*

*127. Készíts függvényt, amely kiírja a lista középső elemét! Ha a lista páros elemszámú, akkor a két középső átlagát adja meg!*

*128. Készíts függvényt, amely három beírt számról eldönti, hogy az lehet-e egy háromszög három oldala - ekkor IGAZ - vagy sem - ekkor HAMIS értékkel tér vissza.*

*129. Készíts függvényt, amely egy lista legnagyobb és a legkisebb eleme közötti különbséget adja meg!*

*130. Készíts függvényt, amely megállapítja egy listáról, hogy hány helyen van benne sorrendbeli hiba!*

*131. Készíts eljárást, amely egy nagyjából rendezett lista esetében megadja azt, hogy milyen hosszú rendezett részekből áll!*

# Programozási hibák

Ebben a fejezetben a leggyakrabban elkövetett programkészítési hibákat mutatjuk be, egyszerű példákon keresztül.

A LOGO-ban, de más programozási nyelvekben is a hibák nagy része abból adódik, hogy a programkészítő nincs tisztában az egyes változótípusok tárolási módjuktól függő értéktartományával, magával a lehetséges értékkészlettel, illetve azzal, hogy milyen típusú elemi adatokkal is dolgozik az elkészített eljárás, hogyan is zajlik a típus meghatározása.

Egy másik, már látott hibakör a műveletek sorrendjének helytelen megadásával kapcsolatos, illetve a felesleges vezérlési szerkezetek használatával.

Ez utóbbi bemutatására szolgál első példánk!

## *Igaz vagy hamis?*

Tegyük fel, hogy azt akarjuk eldönteni egy függvény segítségével, hogy három megadott szakasz ismeretében azok alkothatnak-e háromszöget.

Ehhez elegendő a háromszög egyenlőtlenség vizsgálata, ami három feltétel vizsgálatát írja elő.

A helyes eredményt adó, de csúnya megoldás például ez.

```
eljárás lehet? :a :b :c
 hak (és :a+:b>:c :b+:c>:a :c+:a>:b)
   [eredmény "IGAZ]
   [eredmény "HAMIS]
vége
```

Miért is csúnya? A kétágú kiválasztás akkor hajtódik végre, ha a feltétel IGAZ, a különben ága, a feltétel HAMIS, és ezt ki is írja. A hiba tehát az, hogy feleslegesen alkalmazzuk a kétágú kiválasztást, elegendő a feltételt visszaadni.

```
eljárás lehet? :a :b :c
 eredmény (és :a+:b>:c :b+:c>:a :c+:a>:b)
vége
```

Vegyük észre, hogy a feltétel három logikai kifejezés vizsgálatát jelenti, ezért kell kijelölni zárójelekkel az **és** művelet hatókörét.

*132. Készítsünk **számhármas?** nevű függvényt, amely kiírja, hogy három szám lehet-e pitagoraszi számhármas három tagja! A beírt számok csak egészek lehetnek, és akkor lesz igaz a feltétel, ha a legncagyobb négyzete éppen egyenlő a másik kettő négyzetének összegevei*

## Emeljük a tétet!

A hatványok kiszámítására megismertük már Legendre módszerét, de készíthetünk erre a célra rekurzív eljárást is. Ezen kis ***emel*** nevű programunk éppen ezt mutatja be.

```
eljárás emel :a :b
 ha :b=0 [eredmény 1]
 eredmény :a*emel :a :b-1
vége
```

Fontos, hogy az eljárásban az ***:a*** alap tetszőleges valós szám lehet, de a ***:b*** kitevő csak pozitív egész.

Nézzük meg, hogy mi lesz pár futási eredménye!

```
? ki emel 2 3
8
? ki emel 2 30
1073741824
? ki emel 2 31
2.14748E9
```

Ez utóbbi kifejezés talán már meglepő lehet, de semmi gond, a számok megjelenítése az Imagine-ben bizonyos érték felett már normál alakban történik. Az egészeket 32 biten tároljuk lebegőpontos formában, így a legnagyobb ábrázolható szám $\mathbf{2^{31}-1}$ lesz. De mennyi is ez? írassuk ki!

```
? ki emel 2 31 - 1
1073741824
```

Az eredmény nem helyes, ami onnan látszik, hogy páros szám!

A hiba oka az Imagine postfix feldolgozásának figyelmen kívül hagyásában keresendő. Azért kaptunk azonos értéket éppen a $\mathbf{2^{30}}$ kifejezéssel, mert azt számítattuk ki! Az értelmező ugyanis elsőként elvégzi a kivonást, mivel az az utolsó művelet a képletben, és mi csak az általunk megszokott írásmód miatt érezzük úgy, hogy a kivonásra majd csak később kerül sor, elsőként tehát nem a paraméterekkel történő függvénymeghívást végzi el.

A megoldás természetesen a zárójel alkalmazása.

```
? ki (emel 2 31) - 1
2.14748E9
```

Az eredmény látszólag ugyanaz, mint az előbbi $\mathbf{2^{31}}$ kiszámolásánál.

*50. Kényszerítsük rá a kiírást, hogy a pontos értéket jelenítse meg!*

Ehhez például alkalmazzuk az **egészhányados** - röviden **eh** - műveletet, amely meghatározása szerint csak egész értéket adhat vissza, így ha kiszámoljuk a $\mathbf{((2^{31}-1)\ div\ 2)*2+1}$ értéket, pontosan $\mathbf{2^{31}-1}$ lesz.

```
? ki (egészhányados (emel 2 31)-1 2)*2 + 1
2147483647
```

```
? ki (egészhányados (emel 2 31)-1 2)*2 + 2
2.14748E9
```

Azt vegyük észre, hogy 2-vel megnövelve már csak valós számként tudja kezelni a végeredményt!

A vizsgálatunkat még ne hagyjuk abba! Nézzük meg ezeket a kiírásokat!

```
? mutat emel 2 32
4.29497E9
? mutat emel 2 33
0
```

Ez utóbbi kissé meglepő lehetne, de miért is nem az?

Az ok egyszerű: az eljárásunk egész értékkel dolgozik, emiatt a $2^{32}$-n esetén ugyan még kiszámolható és tárolható az eredmény, de a $2^{33}$ értéknél már a 4 byte-nyi hely és a túlcsordulást jelző bit is 0 számjeggyel lesz feltöltve, emiatt azt írja ki, amit ott talál tartalomként, és ez a hibátlannak tűnő 0 érték lesz.

Az eljárás kis módosítással jól működővé tehető, hiszen elegendő csak egész aritmetikáról valósra rábírni a fordítót, amit több módon is megtehetünk.

Vagy az alapot adjuk meg valósként, azaz a

```
? mutat emel 2.0 33
8.58993E9
```

meghívással ugyan normál alakban, de helyesen kapjuk meg az eredményt, és ezt akár $10^{4932}$-ig is elvégezhetjük, de tovább nem.

```
? mutat emel 10.0 4932
1E4932
? mutat emel 10.0 4933

Hiba a(z) emel eljárás 3. sorában: Valós aritmetikai hiba
lépett fel * végrehajtásakor.
```

A másik lehetőség a *típuskényszerítés,* amit speciális meghívással érhetünk el.

```
? mutat 0.0+emel 2 32
4.29497E9
? mutat (emel 2 32)+0.0
4.29497E9
```

Ezen eredménynek a helyessége annak köszönhető, hogy az összegzés egyik tagja valós ábrázolású, ami kikényszeríti a másik tag esetében is, hogy valós aritmetikával végezzük el, és mint látható, ez a kényszerítés sorrendtől független.

Ennél sokkal meglepőbb lehet a kényszerítés alkalmazására a már korábban látott hatványokra a következő néhány eredmény!

```
? mutat 0+emel 2 33
0
? mutat 0+emel 2 32
0
? mutat 0+emel 2 31
-2147483648
? mutat 0+emel 2 30
1073741824
```

A második és a harmadik eredmény talán meglepő is lehetne, ha nem tudnánk, hogy most a 0 hozzáadása miatt 4 byte-os előjeles egész aritmetika használatára kényszerül az eljárás, és ekkor teljesen normális, hogy a fenti értékek adódnak.

*133.Alakítsuk át az* **emel** *függvényt úgy, hogy 0 és negatív egész kitevőre is meg tudja határozni a hatvány értékét!*

*134.Határozzuk meg a legkisebb egész számként megjelenő szám értékét is, és írassuk is ki!*

Itt jegyezzük meg, hogy arra is figyelnünk kell, hogy az 1 kivonásának beírásánál vagy a mínusz jel előtt és után is van szóköz, vagy sehol sem, mert az **a-1** azonos a **a - 1** kifejezéssel, de egyik sem azonos az **a -1** írással, mivel ez utóbbi hibás, a mínusz jelet az értelmező ugyanis előjelnek veszi!

## Most akkor hány kell?!

Egy program segítségével szeretnénk kiszámolni, hogy hány ötös lottószelvényt kell kitöltenünk ahhoz, hogy telitalálatunk legyen, azaz 90 számból mind az ötöt eltaláljuk. Ez a probléma megoldható egy egyszerű kis programmal, amely kiszámolja a 90 alatt az 5 értékét, azaz hogy 90 elem közül 5-öt hányféleképpen lehet kiválasztani úgy, ha a sorrend nem fontos. Ez a 90*89*88*87*86/1*2*3*4*5 kifejezés kiszámolását jelenti.

```
eljárás ötöslottó
 lokálisérték "db 1
 ism 5 [lokálisérték "db :db*(91-hányadik)]
 ism 5 [lokálisérték "db egészhányados :db hányadik]
 (mutat :db "szelvényt "kell "kitölteni.)
vége
```

A program szépen lefut, és eredményképpen a

```
8157873 szelvényt kell kitölteni.
```

üzenettel leáll, és látszólag minden hibátlanul ment.

Jól látható, hogy a két ismétlést egybe is írhatnánk, mivel a 90 alatt az 5 elemeit átcsoportosítva a kitöltendő szelvények darabszáma kiszámítható úgy is, hogy 90/1*89/2*88/3*87/4*86/5.

```
eljárás ötöslottó2
 lokért "db 1
 ism 5 [lokért "db egészhányados :db*(91-hányadik) hányadik]
 (mutat :db "szelvényt "kell "kitölteni.)
vége
```

Ekkor viszont meglepő lesz az eredmény! A program ismét hibajelzés nélkül, szépen lefut, de eredményképpen a

```
43949268 szelvényt kell kitölteni.
```

üzenet jelenik meg!

Mi az oka a jelenségnek? Az Imagine eljárásai között szerepelnek olyanok, amelyek csak egész számértékekkel értelmezettek. Ilyen művelet például az **egészhányados** és a **maradék** is. Az előjeles, 4 B-os egész változók értelmezési tartománya mint már láttuk a $-2^{31}$ és $2^{31}$-1 közé eső intervallum, ami a *-2147483648* és *2147483647* közötti értékek tartománya. Az első esetben a hibás eredmény oka az, hogy az első ciklusban a részeredmények meghaladják ezt a felső értéket, az előjeles egész ábrázolás sajátosságai miatt az eredmény átfordul, és az újabb szorzásnál éppen pozitív értéke lesz az eredménynek, amellyel tovább számol, és hihető végeredményt ad.

Az átfordulás nem okoz hibát a kiolvasás szempontjából, de igencsak lényeges lehet a végérték szempontjából!

Annak érdekében, hogy lássuk, az ábrázolás sajátosságában van a hiba, módosítsunk ismét az eredeti programunkon! Az első értékadásban írjunk **1.0** kezdőértéket, és cseréljük az **egészhányados** műveletet osztásra, amit a / jelöl!

```
eljárás ötöslottó3
 lokálisérték "db 1.0
 ism 5 [lokálisérték "db :db*(91-hányadik)]
 ism 5 [lokálisérték "db :db/hányadik]
 (mutat :db "szelvényt "kell "kitölteni.)
vége
```

A program „*4.39493E7 szelvényt kell kitölteni.*" üzenettel tér vissza, amely a javított, helyes változatnak felel meg, mivel láthatóan a 6 megjelenített értékes jegyben meg is egyezik vele.

*135.Készítsünk **általános_lottó** nevű eljárást, amely azt adja meg, hogy hány szelvényt kell kitölteni ahhoz, hogy biztosan legyen az :**ebből** szám közül kihúzott :**ennyi** számmal nyertes szelvényünk!*

Az Imagine több szempontot figyelembe véve jeleníti meg az egész értéket kiíratáskor. Ez az eljárásokban alkalmazott műveletektől és konstans és változó értékektől is függ.

## Valós számok?

A következő kis programocska meglepő eredményt ad, ha megfelelő lépésben végezzük el: 1/3 = 1/2! A program a valós számok kezelésének egy sajátosságára mutat rá, ami könnyen programbeli hibához vezethet.

*51. Számoljuk ki meghatározott alkalommal az 1/3 értéket úgy, hogy szorozzuk 4-gyel, majd 1-et kivonunk belőle!*

A ***meglepő*** nevű kis program az ***:x*** változóban tárolja az egyharmad értéket, majd azt ***:db*** alkalommal újra számolja ugyan, de várhatóan változatlanul is hagyja, mivel az egyharmad négyszerese négyharmad, és abból levonva egyet ugyancsak egyharmadot kapunk.

```
eljárás meglepő :db
 lokálisérték "x 1/3
 ism :db [lokálisérték "x :x*4-1]
 (mutat :db "végrehajtás "után :x)
vége
```

A programot futtassuk le több paraméterrel is! Azt tapasztaljuk, hogy 22-ig semmi meglepőt nem kapunk, minden alkalommal kiírja a 0.333333 értéket, azaz az adott hat tizedesjegyes kijelzés mellett semmi meglepő nem tapasztalható. Ha viszont 23-at adjuk meg paraméternek, annál meglepőbb eredmény születik! Az eredmény 0,333334 lesz, 32 esetén pedig pont 0,5!

Mi is történt?

Sajnos a ComLOGO-ban megszokott megjelenített jegyszámot szabályozó **`tizedes jegy!`** parancs, illetve annak megfelelője nem létezik! A valós értékek kijelzése rögzített 6 értékes jeggyel történik meg, emiatt nem látható az ok!

Az Imagine ugyanúgy, mint minden más programozási nyelv, csak egy előre jól meghatározott pontossággal tudja kezelni a számokat. A matematikai értelemben vett valós számok emiatt csak bizonyos pontossággal ábrázolhatóak a számítógépen a lebegőpontos ábrázolás segítségével. Az egyharmad ráadásul olyan érték, amely pontosan nem ábrázolható kettes számrendszerben, mivel a kettes törtalakja végtelen kettedes jegyű, viszont egy bizonyos pontossággal, jegyszámmal igen. A ***meglepő*** program a pontosságot, jegyszámot leplezi le a 32 esetén, ami eszerint 64 bites, mivel a program minden lépésénél az ábrázolás hibáját 4-szeresére növeli meg, azaz két bitet emészt fel az értékes jegyekből. Ez a 64 bites ábrázolás olyan precíz, hogy igen sok lépés után okoz csak észrevehető pontatlanságot, a tízes számrendszerbeli átíráskor a kb. 20 bites pontosság is még hat tizedesnyi pontosságot biztosít.

> A programozónak kell figyelnie, hogy a programjában ne okozzanak hibát a valós számok lebegőpontos ábrázolásának sajátosságai.

# Emelt szintű érettségi Imagine környezetben

A kétszintű érettségi bevezetése óta tart a vita, hogy lehet-e, alkalmas-e a LOGO nyelv arra, hogy a diákok érettségizzenek a használatával, azaz hogy az emelt szintű érettségi feladatait megoldjuk vele. Az Imagine megjelenése óta ez már nem lehet kérdés! Ideje a LOGO-t is felvenni az érettségin használható nyelvek közé!

Ezen állításunkat igazolandó nézzük meg az elmúlt évek érettségi feladatainak megoldását Imagine környezetben, azaz LOGO nyelven. Ez utóbbi a fontos, azért hangsúlyozzuk újfent, hogy az érettségi követelményrendszere nem egy konkrét programozási nyelv ismeretét kéri számon, hanem azt, hogy az érettségiző algoritmizálási képességei elérik-e a kívánt szintet, valamint meg tudja-e valósítani az algoritmusokat egy általa ismert programozási nyelven.

Az érettségi feladatok teljes szövegét gyakran felesleges terjengősségük miatt nem jelöltük meg külön kidolgozott feladatként. Az eredeti, teljes szövegek megtalálhatók az Oktatási és Kulturális Minisztérium honlapján az ÉRETTSÉGI pontja alatt: http://www.okm.gov.hu/main.php?folderID=266.

A feladatok teljes szövegéből nem emeltük külön ki a magyarázó elemeket, csak dőlten szedtük. Az eredeti szövegen csak az értelmezést segítő mértékben módosítottunk. A feladatok sorszámozása után szereplő zárójelbe tett számok az eredeti részfeladat sorszámát jelölik.

## *2005. május 19. Lottó*

*Magyarországon 1957 óta lehet ötös lottót játszani. A játék lényege a következő: a lottószelvényeken 90 szám közül 5 számot kell a fogadónak megjelölnie. Ha ezek közül 2 vagy annál több megegyezik a kisorsolt számokkal, akkor nyer. Az évek során egyre többen hódoltak ennek a szerencsejátéknak és a nyeremények is egyre nőttek.*

*Adottak a* LOTTOSZ.DAT *szöveges állományban a 2003. év 51 hetének ötös lottó számai. Az első sorában az első héten húzott számok vannak, szóközzel elválasztva, a második sorban a második hét lottószámai vannak stb.*

*Például:*

```
37 42 44 61 62
18 42 54 83 89
...
9 20 21 59 68
```

*A lottószámok minden sorban emelkedő számsorrendben szerepelnek. Az állományból kimaradtak az 52. hét lottószámai. Ezek a következők voltak: 89 24 3411 64.*

*Készítsen **lottó** nevű programot a következő feladatok megoldására!*

*52. (1.) Kérje be a felhasználótól az 52. hét megadott lottószámait!*

*A* számokat egyenként olvassuk be az olvasszó segítségével egy listába.

```
eljárás feladati :l
 ha elemszám :l = 5 [eredmény :l]
 ism 5 [mutat (sző "A | 1+elemszám :l "|. lottószám: |)
  eredmény feladati elsőnek olvasszó :l]
vége
```

*53. (2.) A program rendezze a bekért lottószámokat emelkedő sorrendbe! A rendezett számokat írja ki a képernyőre!*

*A* beolvasott számokat egy ***hét52*** nevű változóban eltároljuk, és a rendez paranccsal elvégezzük a növekvő sorba rendezést.

```
eljárás feladat2
 globvál "hét52 rendez feladati []
 mutat :hét52
vége
```

*54. (3.) Kérjen be a felhasználótól egy egész számot 1-51 között! A bekért adatot nem kell ellenőrizni!*

*A* bekért szám egy korábbi hét sorszáma, ami a következő feladat szövegéből derül csak ki. Ezért is fontos, hogy a teljes feladatszöveget végigolvassuk a megoldás előtt. A ***hét*** változó tárolja ezt az adatot.

```
eljárás feladat3
  (mutat "|Add meg, melyik hét számaira vagy kíváncsi (1-51) : |)
 globvál "hét olvasszó
vége
```

5 5 *. (4.) írja k i a képernyőre a bekért számnak megfelelő sorszámú hét lottószámait, a LOTTOSZ.DAT állományban lévő adatok alapján!*

Az adatok beolvasását egy külön eljárással oldjuk meg, amely egy ***számok*** nevű változóban is eltárolja, mert később is szükségünk lesz rá.

```
eljárás olvas.adat :fájl
 lókért "1 []
 olvasóeszköz :fájl
 amíg [nem fájlvége?]
    [lókért "1 utolsónak olvaslista :l]
 globvál "számok :l
 olvasóeszköz []
 eredmény :számok
vége
```

Az adatbeolvasó eljárás egyben a kiíratást is elvégzi.

```
eljárás feladat4
 mutat elem :hét olvas.adat "|lottosz.dat|
vége
```

56. *(5.) A* LOTTOSZ.DAT *állományból beolvasott adatok alapján döntse el, hogy volt-e olyan szám, amit egyszer sem húztak ki az 51 hét alatt! A döntés eredményét (Van/Nincs) írja ki a képernyőre!*

Az eldöntés elemi algoritmus adja meg a választ, amit majd a ***számok*** változóra hívunk meg.

```
eljárás feladat5 :l
 ha üres? :l [eredmény "Nincs]
 ha első :l = 0 [eredmény "Van]
 eredmény feladat5 en :l
vége
```

57. *(6.) A* LOTTOSZ. DAT *állományban lévő adatok alapján állapítsa meg, hogy hányszor volt páratlan szám a kihúzott lottószámok között! Az eredményt a képernyőre írja ki!*

Egy újabb elemi algoritmus: a megszámlálás adja meg a választ a ***számok*** változóra alkalmazva.

```
eljárás feladatő :l
 ha üres? :l [eredmény 0]
 hak maradék első :l 2 • 1
   [eredmény (első :l) + feladatö en :l]
   [eredmény feladatő en s.l]
vége
```

58. *(7.) Fűzze hozzá a* LOTTOSZ.DAT *állományból beolvasott lottószámok után a felhasználótól bekért, és rendezett 52. hét lottószámait, majd írja ki az összes lottószámot a lotto52.ki szöveges fájlba! A fájlban egy sorba egy hét lottószámai kerüljenek, szóközzel elválasztva egymástól!*

Az eredeti állományt megnyitjuk olvasásra, a célállományt írásra, amit beolvastunk, azt egyből ki is írjuk. A beolvasás lezárulása után még egy sort írunk a célfájlba.

```
eljárás feladat7
 olvasóeszköz betöltőút "|lottosz.dat|
 kiíróeszköz betöltőút "|lotto52.ki|
 amíg
   [nem fájlvége?]
    [kiír olvaslista]
 olvasóeszköz []
 kiír :hét52
 kiíróeszköz []
vége
```

*59. (8.) Határozza meg a LOTTO52.KI állomány adatai alapján, hogy az egyes számokat hányszor húzták ki 2003-ban. Az eredményt írja ki a képernyőre a következő formában: az első sor első eleme az a szám legyen ahányszor az egyest kihúzták! Az első sor második eleme az az érték legyen, ahányszor a kettes számot kihúzták stb.! (Annyit biztosan tudunk az értékekről, hogy mindegyikük egyjegyű.)*

*Példa egy lehetséges eredmény elrendezésére (6 sorban, soronként 15 érték).*

```
4 2 2 4 2 2 6 1 1 2 1 5 2 1 1
1 3 5 0 5 5 2 6 6 5 1 0 6 4 3
3 3 5 4 3 1 4 2 2 4 2 4 1 2 3
4 2 1 2 3 2 2 2 4 4 5 1 3 5 5
5 2 0 2 2 4 4 3 1 3 6 1 5 6 2
4 3 2 2 3 1 1 4 1 3 3 2 1 5 3
```

A feladat elvégzésére a már korábbi beolvasó eljárást használjuk fel:

```
eljárás feladat8
 eredmény mi.volt olvas.adat betöltőút "|lotto52.ki|
vége
```

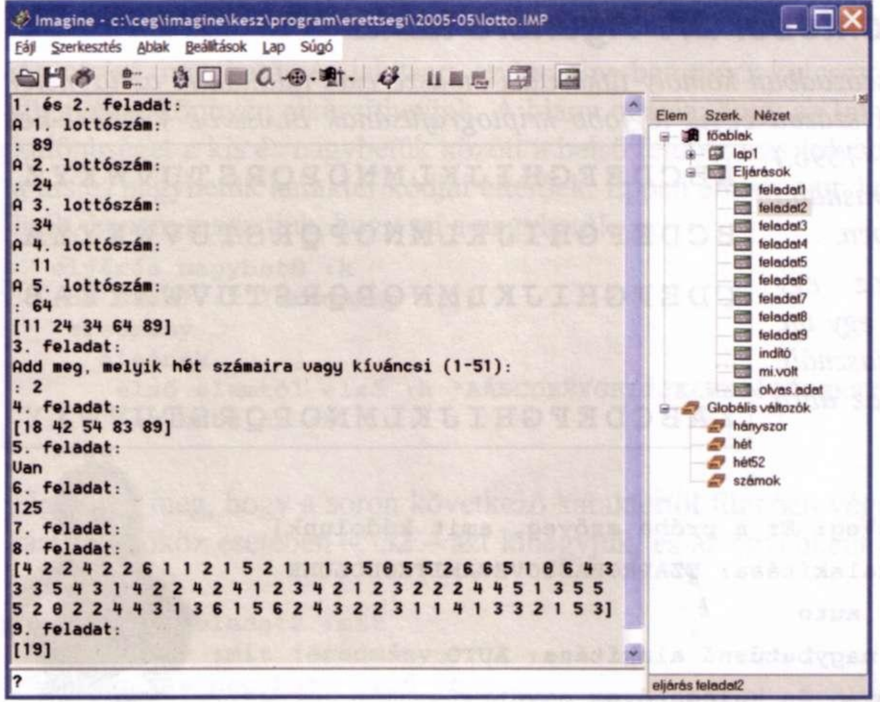
A tényleges számlálást a ***mivolt*** eljárásunk adja meg, ami el is tárolja az eredményt egy ***hányszor*** nevű változóba a következő feladat miatt.

```
eljárás mi.volt :számok
 lókért "s []
 ism 90 [lókért "s elsőnek 0 :s]
 ciklus "i (mondat 1 elemszám :számok)
   [ciklus "j [1 5]
    [lókért "sl mélyelem (mondat :i :j) :számok
     lókért "s mélycserél :sl :s 1 + elem :sl :s ]]
 globvál "hányszor :s
 eredmény :hányszor
vége
```

*60. (9.) Adja meg, hogy az 1-90 közötti prímszámokból melyiket nem húzták ki egyszer sem az elmúlt évben. A feladat megoldása során az itt megadott prímszámokat felhasználhatja vagy előállíthatja! (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89.)*

Egy újabb programozási tétellel találkozunk, ez az adott tulajdonságú elemek kiválogatása. A prímszámlistát felhasználjuk.

```
eljárás feladat9 :l :p
 ha üres? :p [eredmény []]
 hak elem első :p :l = 0
   [eredmény elsőnek első :p feladat9 :l en :p]
   [eredmény feladat9 :l en :p]
vége
```




Az egyes részfeladatokat összekötő eljárásunk ***indító*** néven szerepel. A megoldásban kényelmi okokból használtunk csak fel több helyen globális változókat, ezeket el is hagyhatnánk, de akkor ismételten kellene a korábbi eljárásokat meghívni.

```
eljárás indító
 mutat "|1. és 2. feladat:| feladat2
 mutat "|3. feladat:| feladat3
 mutat "|4. feladat:| feladat4
 mutat "|5. feladat:| mutat feladat5 mi.volt :számok
 mutat "|6. feladat:| mutat feladat6 :hányszor
 mutat "|7. feladat:| feladat7
 mutat "|8. feladat:| mutat feladat8
 mutat "|9. feladat:|
 mutat feladat9 mi.volt :számok [2 3 5 7 11 13 17 19 23 29 31
37 41 43 47 53 59 61 67 71 73 79 83 89]
vége
```

136. *Alakítsuk át az öt lottószám beolvasását végző **feladat1** eljárást úgy, hogy egyetlen szóközzel tagolt számötöst visz be!*

137. *Készítsük el a részfeladatok megoldását úgy, hogy nem használunk feleslegesen globális változókat!*

138. *Egészítsük ki az adatbeolvasásokat ellenőrzésekkel, azaz a lottószámok egészek legyenek 1 és 90 között, valamint csak az 1 és 51 hét közötti értéket fogadjuk el keresett hétnek!*

## 2005. október 27. Vigenère tábla

*Már a XVI. században komoly titkosítási módszereket találtak ki az üzenetek elrejtésére. A század egyik legjobb kriptográfusának Blaise de Vigenère-nek (1523. 4. 5 – 1596.) módszerét olvashatja a következőkben.*

*A kódoláshoz egy táblázatot és egy ún. kulcsszót használt. A táblázatot az ábra mutatja.*

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
BCDEFGHIJKLMNOPQRSTUVWXYZA
CDEFGHIJKLMNOPQRSTUVWXYZAB
...
ZABCDEFGHIJKLMNOPQRSTUVWXY
```

*Példa:*

```
Nyílt szöveg: Ez a próba szöveg, amit kódolunk!
Szöveg átalakítása: EZAPROBASZOVEGAMITKODOLUNK
Kulcsszó: auto
Kulcsszó nagybetűssé alakítása: AUTO
Nyílt szöveg és kulcsszöveg együtt:
E Z A P R O B A S Z O V E G A M I T K O D O L U N K
A U T O A U T O A U T O A U T O A U T O A U T O A U
Kódolt szöveg:
E T T D R I U O S T H J E A T A I N D C D I E I N E
```

*A tábla adatait a* VTABLA.DAT *fájlban találja a következő formában.*

*Készítsen programot* ***kodol*** *néven a következő feladatok végrehajtására!*

*61. (1.) Kérjen be a felhasználótól egy maximum 255 karakternyi, nem üres szöveget! A továbbiakban ez a nyílt szöveg.*

A beolvasást függvénnyel oldjuk meg.

```
eljárás feladat1
 (mutat "|Add meg a kódolandó szöveget:|)
 eredmény olvasszó
vége
```

*62. (2.) Alakítsa át a nyílt szöveget, hogy a későbbi kódolás feltételeinek megfeleljen! A kódolás feltételei:*

- *A magyar ékezetes karakterek helyett ékezetmenteseket kell használni. (Például á helyett a; ő helyett o stb.)*
- *A nyílt szövegben az átalakítás után csak az angol ábécé betűi szerepelhetnek.*
- *A nyílt szöveg az átalakítás után legyen csupa nagybetűs.*

Ez talán a feladatban a legbonyolultabb eljárásunk.

Az utolsó, nagybetűssé alakításra az Imagine-ben nincs kulcsszó, de egy ilyen függvényt könnyen elkészíthetünk. A hiány oka az, hogy az Imagine nem tesz különbséget a kis és nagybetűk között a belső feldolgozás folyamán, de persze a kis és nagybetűk karakter kódjai eltérőek. Éppen emiatt nem kódokat vizsgálunk, hanem megadjuk, hogy mi a nagybetű!

```
eljárás nagybetű :k
 ha üres? :k [eredmény "||]
 eredmény
   elsőnek
    első elemtől első :k "AÁBCDEÉFGHIÍJKLMNOÓÖŐPQRSTUÚÜŰVWXYZ
    nagybetű en :k
vége
```

Figyeljük meg, hogy a soron következő karaktertől függően végezzük el a kiírást. A szóköz esetében - **\32** - azt kihagyjuk, és az egyébként ág nem tartalmaz feltételt.

```
eljárás £eladat2 :mlt
 ha üres? :mit [eredmény •||]
 elágazás nagybetű első :mit
   [á     [eredmény elsőnek "a feladat2 en :mit]
    é     [eredmény elsőnek "e feladat2 en :mit]
    í     [eredmény elsőnek "i feladat2 en :mit]
    ó ö ő [eredmény elsőnek "o feladat2 en :mit]
    ú ü ű [eredmény elsőnek "u feladat2 en :mlt]
    \32   [eredmény feladat2 en :mit]
          [eredmény elsőnek első :mit £eladat2 en :mit]
   ]
vége
```

*63. (3.) írja ki a képernyőre az átalakított nyílt szöveget!*

A kiíratás mellett el is tároljuk az elkészített nyílt szöveget.

```
eljárás £eladat3 :mit
 globvál "nyílt nagybetű :mit
 kiír :nyílt
vége
```

*64. (4.) Kérjen be a felhasználótól egy maximum 5 karakteres, nem üres kulcsszót! A kulcsszó a kódolás feltételeinek megfelelő legyen! (Sem átalakítás, sem ellenőrzés nem kell!) Alakítsa át a kulcsszót csupa nagybetűssé!*

Beolvassuk a kulcsszót, és át is alakítjuk nagybetűssé.

```
eljárás feladat4
  (mutat "|Add meg kulcsszót (max 5 betű):|)
 eredmény nagybetű olvasszó
vége
```

*65. (5.) A kódolás első lépéseként fűzze össze a kulcsszót egymás után annyiszor, hogy az így kapott karaktersorozat (továbbiakban kulcsszöveg) hossza legyen egyenlő a kódolandó szöveg hosszával! Írja ki a képernyőre az így kapott kulcsszöveget!*

Addig írogatjuk egymás után a kulcsszót, amíg nem lesz megfelelő hosszú. Ha viszont hosszabb lenne, mint a kódolandó szöveg, akkor addig nyesegetjük a betűket a kapott láncról, amíg a hossza egyező nem lesz a kódolandó nyílt szöveggel.

```
eljárás feladat5 :kulcs :mit :ksz
 ha esz :ksz = esz :mit
   [globvál "kódszó :ksz eredmény :ksz]
 ha esz :ksz < esz :mit
   [eredmény feladat5 :kulcs :mit elsőnek :kulcs :ksz]
 eredmény feladat5 :kulcs :mit un :ksz
vége
```

*66. (6.) A kódolás második lépéseként a következőket hajtsa végre! Vegye az átalakított nyílt szöveg első karakterét, és keresse meg a* VTABLA.DAT *fájlból beolvasott táblázat első oszlopában! Ezután vegye a kulcsszöveg első karakterét, és keresse meg a táblázat első sorában! Az így kiválasztott sor és oszlop metszéspontjában lévő karakter lesz a kódolt szöveg első karaktere. Ezt ismételje a kódolandó szöveg többi karakterével is!*

> A megoldás során követjük a feladat előírását, de itt jegyezzük meg, hogy egyszerűbb lenne a kódtábla nélkül. Az eljárás ilyetén módja viszont mindenképpen szükséges akkor, ha a kódábécé sorok nem lennének ábécé rendben, hanem például összekeverték volna ezeket is.

A soron következő nyílt szöveg és kulcsszólánc betűjének oszlopát illetve sorát egy adott tulajdonságú elem helyét meghatározó elemi algoritmussal keressük meg, majd meghatározzuk a listában elfoglalt helyét.

```
eljárás hol? :l :mi
 ha első :l = :mi [eredmény 1]
 eredmény 1 + hol? en :l :mi
vége

eljárás feladat6 :nyílt :kódszó
 ha üres? :nyílt [eredmény "||]
 eredmény elsőnek
   elem (hol? :ábécé első :kódszó)
        (első elem hol? :ábécé első :nyílt :tábla)
   feladat6 en :nyílt en :kódszó
vége
```

*67. (7.) Írja ki a képernyőre és a KODOLT.DAT fájlba a kapott kódolt szöveget!*

A fájlba történő kiíratáshoz újra meghívjuk az előző eljárást.

```
eljárás feladat7
 kiíróeszköz betöltőút "|kodolt.dat|
 kiír feladat6 :nyílt :kódszó
 kiíróeszköz []
vége
```

Az eljárásokat összekötő elemet ismét ***indítónak*** nevezzük el.

Ennek két, eddig nem említett plusz eleme van. Az első elvégzi a kódtábla beolvasását, és eltárolja a ***tábla*** változóban, a másik létrehozza az ***ábécé*** változót, amely a hely meghatározásában lesz a segítségünkre.

```
eljárás indító
 globvál "tábla be.tábla "vtabla.dat
 mutat "|1-2-3. feladat:|
 globvál "nyílt feladat2 feladat1
 feladat3 :nyílt
 mutat "|4-5. feladat:|
 kiír feladat5 feladat4 :nyílt "||
 globvál "ábécé "abcdefghijklmnopqrstuvwxyz
 mutat "|6. feladat:|
 mutat feladat6 :nyílt :kódszó
 mutat "|7. feladat:|
 feladat7
 mutat "|Kiírás megtörtént.|
vége
```

Végül nézzünk néhány kapcsolódó gyakorló feladatot!

*139. Készítsük el a **be.tábla** eljárást!*

*140. Egészítsük ki a beolvasásokat ellenőrzésekkel!*

*141. Tegyük egyszerűbbé a kulcsszóláncot előállító eljárást!*

*142. Tegyük egyszerűbbé a kódolást végző eljárást! Az eljárás kapja meg a nyílt szöveget és a kulcsszót, és maga állítsa elő a kódolás alapját szolgáló táblát!*

## 2006. február 28. Telefonszámla

*Egy új szolgáltatás keretében ki lehet kérni a napi telefonbeszélgetéseink listáját. A listát egy fájlban küldik meg, amelyben a következő adatok szerepelnek: hívás kezdete, hívás vége, hívott telefonszám. A hívás kezdete és vége óra, perc, másodperc formában szerepel.*

*Például:*

| | | | | | | |
|---|---|---|---|---|---|---|
| 6 15 0 6 19 0 | Óra | perc | mperc | Óra | perc | mperc |
| 395682211 | Telefonszám | | | | | |
| 9 58 15 10 3 53 | Óra | perc | mperc | Óra | perc | mperc |
| 114571155 | Telefonszám | | | | | |

*A hívások listája időben rendezett módon tartalmazza az adatokat, és szigorúan csak egy napi adatot, azaz nincsenek olyan beszélgetések, amelyeket előző nap kezdtek vagy a következő napon fejeztek be. Továbbá az elmúlt időszak statisztikái alapján tudjuk, hogy a napi hívások száma nem haladja meg a kétszázat.*

*A telefonálás díjait a következő táblázat foglalja össze.*

| Hívásirány | Csúcsidőben $7^{00}$ - $18^{00}$ | Csúcsidőn kívül $0^{00}$ - $7^{00}$ és $18^{00}$ - $24^{00}$ |
|---|---|---|
| Vezetékes | 30 Ft/perc | 15 Ft/perc |
| Mobil társaság | 69,175 Ft/perc | 46,675 Ft/perc |

*További fontos információk:*

*A csúcsidő reggel 7:00:00-kor, a csúcsidőn kívüli időszak pedig 18:00:00-kor kezdődik. A díjazás számításakor az számít, hogy mikor kezdte az illető a beszélgetést. (Például: ha 17:55-kor kezdett egy beszélgetést, de azt 18:10-kor fejezte be, akkor is csúcsidőbeli díjakkal kell számlázni.)*

*Minden megkezdett perc egy egész percnek számít.*

*Minden telefonszám elején egy kétjegyű körzetszám, illetve mobil hívószám található. A mobil hívószámok: 39, 41, 71 kezdődnek, minden egyéb szám vezetékes hívószámnak felel meg.*

*A következő feladatokat oldja meg egy program segítségével! A programot mentse **szamla** néven!*

*68. (1.) Kérjen be a felhasználótól egy telefonszámot! Állapítsa meg a program segítségével, hogy a telefonszám mobil-e vagy sem! A megállapítást írja ki a képernyőre!*

Ismét megjegyezzük, hogy mindig fontos a feladat teljes szövegét elsőként alaposan végigolvasnunk, mivel azonos típusú részfeladatok előfordulhatnak, és ezeket célszerű külön eljárásokként megvalósítani.

Annak vizsgálata, hogy a szám mobil-e, előfordul később is, így ezt külön függvénnyel végezzük el.

```
eljárás mobil? :sz
  eredmény eleme (szó első :sz első en :sz) [39 41 71]
vége
```

A beolvasott számot a ***mobil?*** eljárással vizsgáljuk meg.

```
eljárás feladat1
 mutat "|Adj meg egy telefonszámot:|
 hak mobil? olvasszó
   [mutat "|A beírt szám mobilszám.|]
   [mutat "|A beírt szám vezetékes.|]
vége
```

*69. (2.) Kérjen be továbbá egy hívás kezdeti és hívás vége időpontot óra perc másodperc formában! A két időpont alapján határozza meg, hogy a számlázás szempontjából hány perces a beszélgetés! A kiszámított időtartamot írja ki a képernyőre!*

A két szám beolvasását listaként oldjuk meg, azaz azokat szóközzel elválasztva, egy sorban kérjük.

```
eljárás feladat2
 mutat "|Add a hívás kezdetét (óra perc mp):|
 lokért "kezd olvaslista
 mutat "|Add a hívás végét (óra perc mp):|
 lokért "vég olvaslista
 (mutat "|Számlázási idő| idő :kezd :vég "perc.)
vége
```

Az érdemi munkát az ***idő*** eljárás végzi, amely másodpercre váltja át a beolvasott időpontokat, és a különbségüket is képzi, majd visszaszámolja percre. Az egészre kerekítést az 59 hozzáadásával oldjuk meg.

```
eljárás idő :k :v
 lokért "k (első :k)*3600 + (első en :k)*60 + utolsó :k
 lokért "v (első :v)*3600 + (első en :v)*60 + utolsó :v
 eredmény egészhányados :v-:k+59 60
vége
```

*70. (3.) Állapítsa meg a HWASOK.TXT fájlban lévő hívások időpontja alapján, hogy hány számlázott percet telefonált a felhasználó hívásonként! A kiszámított számlázott perceket írja ki a* ***percek. txt*** *fájlba a következő formában: perc telefonszám*

A számok tényleges feldolgozása előtt egy ***hívások*** nevű változóba beolvassuk az adatokat.

```
eljárás beolvas :fájl
 olvasóeszköz betöltőút :fájl
 lőkért "1 []
 amíg [nem fájlvége?]
   [lőkért "1 utolsónak olvaslista :l]
 olvasőeszköz []
 globvál "hívások :l
vége
```

Az előbbi időszámító eljárást itt is felhasználjuk, ami után csak a fájlból való beolvasást és a fájlba való kiírást kell megoldanunk. A kissé összetettnek tűnő eljárás csak a beolvasott elemek átadását oldja meg.

```
eljárás feladat3 :l
 kiírőeszköz betöltőút "|percek.txt|
 ciklus "i (mondat 1 elemszám :l 2)
   [ (kiír idő
    (mondat első elem :i :l első en elem ti :l első en en elem :i :l)
    (mondat utolsó un un elem :i :l utolsó un elem :i :l
     utolsó elem :i :l)
     elem :i+l :l)
   ]
 kiíróeszköz []
vége
```

*71. (4.) Állapítsa meg a HÍVÁSOK, TXT fájl adatai alapján, hogy hány hívás volt csúcsidőben és csúcsidőn kívül! Az eredménytjelenítse meg a képernyőn!*

***A*** már beolvasott hívások listájával hívjuk meg az eljárást. Az eljárás magja tulajdonképpen az adott tulajdonságú elemek megszámlálására megvalósított elemi algoritmus átírása két számlálóra: az egyik a tulajdonsággal rendelkezőket, a másik a nem rendelkezőket számlálja.

```
eljárás feladat4 :l
 lókért "belül 0
 lókért "kívül 0
 ciklus "i (mondat 1 elemszám :l 2)
   [hak csúcsidő? első elem :i :l
     [növel "belül][növel "kívül]]
  (kiír "|Csúcsidőben | :belül "|db beszélgetés, kívül| :kívül "|db
         beszélgetés volt.|)
vége
```

Az érdemi munkát itt is egy külön függvény végzi, a ***csúcsidő?***. *A* függvény nem bonyolult, be is írhattuk volna az előző eljárásba, de így áttekinthetőbb a szerepe és könnyebben lehet módosítani is.

```
eljárás csúcsidő? :sz
  eredmény (és :sz>7 :sz<18)
vége
```

*72. (5.) A HÍVÁSOK, TXT fájlban lévő időpontok alapján határozza meg, hogy hány percet beszélt a felhasználó mobil számmal és hány percet vezetékessel! Az eredményt jelenítse meg a képernyőn!*

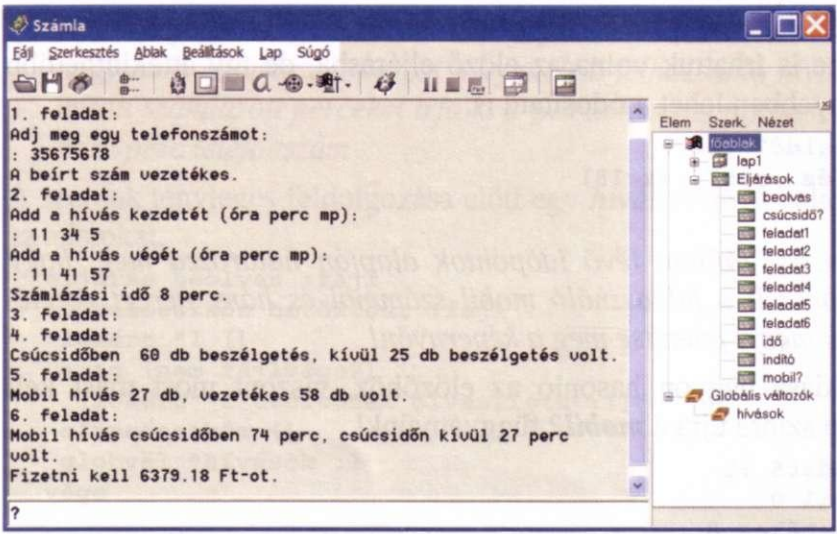
*A* feladat megoldása nagyon hasonló az előzőhöz, viszont most mást kell figyelnünk. Itt lép színre újra a ***mobil?*** függvényünk!

```
eljárás feladat5 :l
 lókért "mobil 0
 lókért "vezetékes 0
 ciklus "i (mondat 2 elemszám :l 2)
   [hak mobil? első elem :i :l
     [növel "mobil][növel "vezetékes]]
  (kiír "|Mobil hívás| :mobil "|db, vezetékes| :vezetékes "|
        db volt.|)
vége
```

*73. (6.) Összesítse a HTVASOK.TXT fájl adatai alapján, mennyit kell fizetnie a felhasználónak a csúcsdíjas hívásokért! Az eredményt a képernyőn jelenítse meg!*

Az eddigiek közül a legösszetettebb eljáráshoz érkeztünk el, de igazi újdonság nincs benne, a korábbiakat kell kicsit átalakítanunk és összevonnunk, újra használhatjuk a ***mobil?, csúcsidő?*** és ***idő*** eljárásainkat!

```
eljárás feladatő :l
 lókért "mobilb 0
 lókért "mobilk 0
  ciklus "i (mondat 1 elemszám :l 2)
   [ha mobil? első elem :i+l :l
     [lókért "idő
       idő (mondat első elem :i :l első en elem :i :l
             első en en elem :i :l)
            (mondat utolsó un un elem :i :l utolsó un elem :i :l
             utolsó elem :i :l)
      hak csúcsidő? első elem :i :l
        [(növel "mobilb :idő)][(növel "mobilk :idő)]]]
  (kiír
    "|Mobil hívás csúcsidőben| :mobil_b "|perc, csúcsidőn kívül|
    :mobil_k "|perc volt.|)
  (kiír
    "|Fizetni kell| :mobil_b*69.175+:mobil_k*46.675 "|Ft-ot.|)
vége
```




Végül nézzük most is az részfeladatokat összekötő, ***indító*** nevű eljárást!

```
eljárás indító
 mutat "|1. feladat:| feladat1
 mutat "|2. feladat:| feladat2
 beolvas "|hivasok.txt|
 mutat "|3. feladat:| feladat3 :hívások
 mutat "|4. feladat:| feladat4 :hívások
 mutat "|5. feladat:| feladat5 :hívások
 mutat "|6. feladat:| feladat6 :hívások
vége
```

> Vegyük észre, hogy ez az érettségi feladat, ha nem is tartalmazott nehezen megérthető elemeket, arra jó példa volt, hogy a megfelelően elkészített részeljárások milyen nagy mértékben egyszerűsíthetik az egész probléma megoldását.

*143.A 4. és 5. feladat egyszerűbben is megoldható! Érdemes-e külön számlálókat alkalmazni mindkét esetben?*

## 2006. május 17. Fehérje

*A fehérjék óriás molekulák, amelyeknek egy része az élő szervezetekben végbemenő folyamatokat katalizálják. Egy-egy fehérje aminosavak százaiból épül fel, melyek láncszerűen kapcsolódnak egymáshoz. A természetben a fehérjék fajtája több millió. Minden fehérje húszféle aminosav különböző mennyiségű és sorrendű összekapcsolódásával épül fel.*

*A következő táblázat tartalmazza az aminosavak legfontosabb adatait, a megnevezéseket és az őket alkotó atomok számát (az aminosavak mindegyike tartalmaz szenet, hidrogént, oxigént és nitrogént, néhányban kén is van):*

| Neve | Rövidítés | Betűjele | C | H | O | N | S |
|---|---|---|---|---|---|---|---|
| Glicin | Gly | G | 2 | 5 | 2 | 1 | 0 |
| Alanin | Ala | A | 3 | 7 | 2 | 1 | 0 |
| Arginin | Arg | R | 6 | 14 | 2 | 4 | 0 |
| Fenilalanin | Phe | F | 9 | 11 | 2 | 1 | 0 |
| Cisztein | Cys | C | 3 | 7 | 2 | 1 | 1 |
| Triptofán | Trp | W | 11 | 12 | 2 | 2 | 0 |
| Valin | Val | V | 5 | 11 | 2 | 1 | 0 |
| Leucin | Leu | L | 6 | 13 | 2 | 1 | 0 |
| Izoleucin | Ile | I | 6 | 13 | 2 | 1 | 0 |
| Metionin | Met | M | 5 | 11 | 2 | 1 | 1 |
| Prolin | Pro | P | 5 | 9 | 2 | 1 | 0 |
| Szerin | Ser | S | 3 | 7 | 3 | 1 | 0 |
| Treonin | Thr | T | 4 | 9 | 3 | 1 | 0 |
| Aszparagin | Asn | N | 4 | 8 | 3 | 2 | 0 |
| Glutamin | Gln | Q | 5 | 10 | 3 | 2 | 0 |
| Tirozin | Tyr | Y | 9 | 11 | 3 | 1 | 0 |
| Hisztidin | His | H | 6 | 9 | 2 | 3 | 0 |
| Lizin | Lys | K | 6 | 14 | 2 | 2 | 0 |
| Aszparaginsav | Asp | D | 4 | 7 | 4 | 1 | 0 |
| Glutaminsav | Glu | E | 5 | 9 | 4 | 1 | 0 |

*Készítsen programot **feherje** néven, ami megoldja a következő feladatokat! Ügyeljen arra, hogy a program forráskódját a megadott helyre mentse!*

*74. (1.) Töltse be az* AMINOSAV.TXT *fájlból az aminosavak adatait! A fájlban minden adat külön sorban található, a fájl az aminosavak nevét nem tartalmazza. Ha az adatbetöltés nem sikerül, vegye fel a fenti táblázat alapján állandóként az első öt adatsort, és azzal dolgozzon!*

```
eljárás feladat1 :mit
 lokért "l []
 olvasóeszköz :mit
 amíg [nem fájlvége?]
   [lokért "l utolsónak olvasszó :l]
 eredmény :l
 olvasóeszköz []
vége
```

A beolvasást az **olvasszó** végezte, így minden elem egyetlen listába került.

*75. (2.) Határozza meg az aminosavak relatív molekulatömegét, ha a szén atomtömege 12, a hidrogéné 1, az oxigéné 16, a nitrogéné 14 és a kén atomtömege 32! Például a Glicin esetén a relatív molekulatömeg 2-12 + 51 +2-16 + 1-14 + 0-32 = 75.*

Az előző feladatban beolvasott aminosav adatok az ***:m*** globális változóba kerültek, az :m tömb elemszáma 140. Ezt tudva a feladat megoldásánál 1-től 140-ig visszük a ciklusváltozót, de mivel minden aminosav 7 adattal szerepel, ezért hetesével lépkedünk. Az eljárásban a tömeg meghatározásához felhasználjuk az összegzés elemi algoritmusát megvalósító **összeg** eljárást is.

```
eljárás feladat2
 ciklus "i [1 140 7]
   [(kiír elem :i :m
     (összeg 12*elem :i+2 :m l*elem ii+3 :m
     16*elem :i+4 :m 14*elem :i+5 :m 32*elem :i+6 :m))]
vége
```

*A következő feladatok eredményeit írja képernyőre, illetve az* EREDMÉNY, TXT *fájlba! A kiírást a feladat sorszámának feltüntetésével kezdje (például: 4. feladat)!*

*76. (3.) Rendezze növekvő sorrendbe az aminosavakat a relatív molekulatömeg szerint! írja ki a képernyőre és az* EREDMENY.TXT *fájlba az aminosavak hárombetűs azonosítóját és a molekulatömeget! Az azonosítót és hozzátartozó molekulatömeget egy sorba, szóközzel elválasztva írja ki!*

A feladatot két részre bontva oldjuk meg a kiírások miatt. Itt végül is egy listát kell a második eleme szerint rendezni, amelyhez a rendez parancs közvetlenül nem használható fel, de ha az elemek sorrendjét a rendezéshez megcseréljük már igen! Ezért egy ***:l*** listába nem a ténylegesen kért [azonosító tömeg] párt adjuk meg, hanem annak a fordítottját, és ezt cseréljük meg a rendezés után.

```
eljárás feladat3
;elemek sorrendjének megfordítása
 lókért "1 []
 ciklus "i [1 140 7]
   [lókért "1 utolsónak
   (mondat (összeg 12*elem :i+2 :m l*elem :i+3 :m
    16*elem :i+4 :m 14*elem :i+5 :m 32*elem :i+6 :m)
    elem :i :m) :l]
;rendezés
 lókért "1 rendez :l
;kiírás
 ciklus "i [1 20]
     [(kiír utolsó elem :i :l első elem :i :l)]
vége
```

A fájlba írást még nem tettük meg, ezt később intézzük el.

77. *(4.) A BSA. TXT a BSA nevű fehérje aminosav sorrendjét tartalmazza - egybetűs jelöléssel. (A fehérjelánc legfeljebb 1000 aminosavat tartalmaz.) Határozza meg a fehérje összegképletét (azaz a C, H, O, N és S számát)! A meghatározásánál vegye figyelembe, hogy az aminosavak összekapcsolódása során minden kapcsolat létrejöttekor egy vízmolekula ($H_2O$) lép ki! Az összegképletet a képernyőre és az EREDMENY.TXT fájlba az alábbi formában írja ki:*

`Például: C 16321 H 34324 O 4234 N 8210 S 2231`

*(Amennyiben a BSA.TXT beolvasása sikertelen, helyette tárolja a G,A,R,F,C betűjeleket tízszer egymás után és a feladatokat erre a „ láncra " oldja meg!)*

A fájl beolvasását az feladati eljárással oldhatjuk meg elsőként. Az adatok a ***:bsa*** változóba kerülnek be, majd ezt dolgozzuk fel. A belső ciklus egy öt elemű tömbben végzi el az atomok összegzését úgy, hogy a korábban ott lévő értéket növeli meg a soron következő aminosav atomszámaival. A **H2O** kilépését a -2 és -1 levonással oldjuk meg, mivel minden aminosav tartalmaz hidrogént és oxigént.

```
eljárás feladat4 :bsa
 lókért "atom (mondat 0 -2*elemszám :bsa -1*elemszám :bsa 0 0)
 ism elemszám :bsa
  [lókért "itt elemtől első :bsa :m
   ciklus "i [1 5]
     [lókért "atom
       mélycserél :i :atom (elem :i :atom) + (elem :i en :itt )]
   lókért "bsa en :bsa]
 eredmény :atom
vége
```

*A fehérjék szekvencia szerkezetét hasításos eljárással határozzák meg. Egyes enzimek bizonyos aminosavak után kettéhasítják a fehérjemolekulát. Például a Kimotripszin enzim a Tirozin (Y), Fenilalanin (W) és a Triptofán (F) után hasít.*

78. *(5.) Határozza meg, és írja ki képernyőre a Kimotripszin enzimmel széthasított BSA lánc leghosszabb darabjának hosszát és az eredeti láncban elfoglalt helyét (első és utolsó aminosavának sorszámát)! A kiíráskor nevezze meg a kiírt adatot, például: „ kezdet helye:"!*

*A* válasz megadásához tudnunk kell, hogy hol kezdődik a leghosszabb darab, mi a tényleges hossza. Az elkészített eljárásunk ezért a megkeresi a lehetséges vágási helyet, majd ha hosszabb részt talál, akkor a kezdés helyét és a hosszát kicseréli.

```
eljárás feladat5 :bsa :maxh :kezd :hossz :lép
 ha üres? :bsa
   [eredmény (mondat "|kezdet helye| :kezd "hossza :maxh)]
 ha eleme? első :bsa [y w f]
  [hak :maxh < :hossz
    [eredmény feladat5 en :bsa :hossz 1+:lép-:hossz 1 :lép+1]
    [eredmény feladat5 en :bsa :maxh :kezd 0 :lép+1]
  ]
 eredmény feladat5 en :bsa :maxh :kezd :hossz+1 :lép+1
vége
```

*79. (6.) Egy másik enzim (a Factor XI) az Arginin (R) után hasít, de csak akkor, ha Alinin (A) vagy Valin (V) követi. Határozza meg, hogy a hasítás során keletkező első fehérjelánc részletben hány Cisztein (C) található! A választ teljes mondatba illesztve írja ki a képernyőre!*

Az érdemi munkát egy külön eljárásra bízzuk. Ez keresi meg az első vágás helyét, és adja vissza az első részletet.

```
eljárás bsa.eleje :feh :ki
 ha üres? :feh [eredmény :ki]
 hak (és első :feh = "r eleme? első en :feh [a v])
   [eredmény utolsónak első :feh :ki]
   [eredmény bsa.eleje en :feh utolsónak első :feh :ki]
vége
```

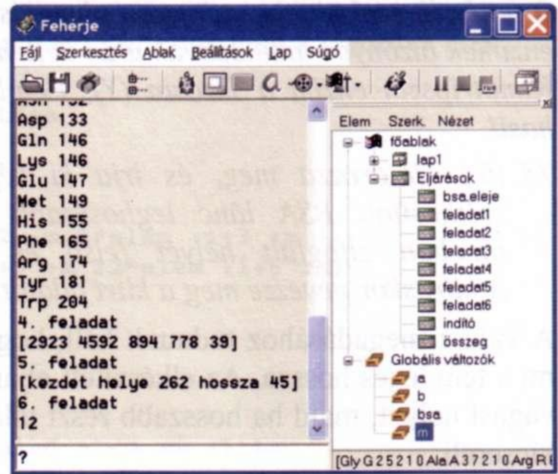
A tényleges számolást egy újabb elemi algoritmus végzi, a megszámlálás.

```
eljárás feladat6 :mit
 ha üres? :mit [eredmény 0]
 hak első :mit = "c
   [eredmény 1 + feladat6 en :mit]
   [eredmény feladat6 en :mit]
vége
```

Ne felejtsük el, hogy a fájlba írásokat még nem végeztük el!

Ezt viszont hála a **kiir** parancs átirányíthatóságának nem is kell külön eljárásokban megoldanunk, hanem egyszerűen elvégezzük a fájlba irányítást, és meghívjuk a megfelelő eljárásainkat!

Az ***indító*** eljárásunk ennek megfelelően a következő lesz.



```
eljárás indító
 globvál "m feladati "|aminosav.txt|
 kiír "|2. feladat| feladat2
 kiír "|3. feladat|  feladat3
 globvál "bsa feladati "|bsa.txt|
 kiír "|4. feladat| mutat feladat4 :bsa
 kiír "|5. feladatj mutat feladat5 :bsa 1 1 1 1
 kiír "|6. feladat| mutat feladatö bsa.eleje :bsa []
;faljba írás
 kiíróeszköz betöltőút "|eredmény.txt|
 kiír "|2. feladat) feladat2
 kiír "|3. feladat| feladat3
 kiír ")4. feladat) kiír feladat4 :bsa
 kiíróeszköz []
vége
```

*144.0ldjuk meg a 2. feladatot úgy, hogy nem tekintjük ismertnek az aminosav adatait tartalmazó* ***:m*** *lista elemszámát!*

## 2006. november 3. Zenei adók

*A rádióhallgatás ma már egyre inkább zene vagy hírek hallgatására korlátozódik. Ez a feladat három, folyamatosan zenét sugárzó adóról szól, azok egyetlen napi műsorát feldolgozva.*

*A reklám elkerülése érdekében az adókat nevük helyett egyetlen számmal azonosítottuk.*

*A MŰSOR, TXT állomány első sorában az olvasható, hogy hány zeneszám (z<=1000) szólt aznap a rádiókban, majd ezt z darab sor követi. Minden sor négy, egymástól egyetlen szóközzel elválasztott adatot tartalmaz: a rádió sorszámát, amit a szám hossza követ két egész szám (perc és másodperc) formában, majd a játszott szám azonosítója szerepel, ami a szám előadójából és címéből áll. A rádió sorszáma az 1, 2, 3 számok egyike. Az adás minden adón 0 óra 0 perckor kezdődik. Egyik szám sem hosszabb 30 percnél, tehát a perc értéke legfeljebb 30, a másodperc pedig legfeljebb 59 lehet. A szám azonosítója legfeljebb 50 karakter hosszú, benne legfeljebb egy kettőspont szerepel, ami az előadó és a cím között található. A számok az elhangzás sorrendjében szerepelnek az állományban, tehát a később kezdődő szám későbbi sorban található. Minden zeneszám legfeljebb egyszer szerepel.*

```
Például:
677
15 3 Deep Purple:Bad Attitűdé
2 3 36 Eric Clapton:Terraplane Blues
3 2 46 Eric Clapton:Crazy Country Hop
3 3 25 Omega:Ablakok
```

*Készítsen programot **zene** néven, amely az alábbi kérdésekre válaszol! Ügyeljen arra, hogy a program forráskódját a megadott helyre mentse!*

*A képernyőre írást igénylő részfeladatok eredményének megjelenítése előtt írja a képernyőre a feladat sorszámát (például: 3. feladat:). Ha a billentyűzetről olvas be adatot, jelenítse meg a képernyőn, hogy milyen értéket vár.*

*Az adatszerkezet készítése során vegye figyelembe az Ön által használt programozási környezetben az adatok tárfoglalási igényét!*

*80. (1.) Olvassa be a **musor.txt** állományban talált adatokat, s annak felhasználásával oldja meg a következő feladatokat! Ha az állományt nem tudja beolvasni, akkor a forrás első 10 sorának adatait jegyezze be a programba, s úgy oldja meg a következő feladatokat!*

Az első feladat egy már korábban már többször megvalósított fájlból történő beolvasás, amit most ***beolvas*** néven készítsünk el. Ezt itt az Olvasóra bízzuk. Az eljárás az **olvaslista** segítségével soronként végezze a munkáját. A *z* helyett a ***db*** változó tárolja a számok mennyiségét.

```
eljárás feladati
 globvál "műsor beolvas "|műsor.txt|
.globvál "db elemszám :műsor
vége
```

*81. (2.) írja a képernyőre, hogy melyik csatornán hány számot lehetett meghallgatni!*

A későbbiekben is szükségünk lesz az első adón lejátszott számok mennyiségére, emiatt változókat hozunk létre, és a műsor lista részlistáinak első elemének értékétől függően a neki megfelelő számlálót megnöveljük.

```
eljárás feladat2 :l
 globvál "adói 0
 lókért "adó2 0
 lókért "adó3 0
 ism :db
   [elágazás első elem hányadik :l
    [1 [növel "adói]
     2 [növel "adó2]
     3 [növel "adó3]
    ]
   ]
  (mutat "|Az első adón| :adól "|db, a második adón|
         :adó2 "|db, a harmadik adón| :adó3 "|db számot
játszottak.|)
vége
```

*82. (3.) Adja meg, mennyi idő telt el az első Eric Clapton szám kezdete és az utolsó Eric Clapton szám vége között az 1. adón! Az eredményt óra:perc-.másodpercformában írja a képernyőre!*

A válasz megadásához elsőként az első adó műsorát kiírjuk egy új fájlba, ami egyrészt tartalmazza az adón lejátszott számok számát, majd soronként két adat szerepel: másodpercben megadva, hogy melyik szám mikor ért véget, és persze maga a szám címe. Ezek az előkészületek a későbbi feladatok miatt is fontosak lesznek.

```
eljárás első.adó :l
 kiíróeszköz betöltőút "jelso.txt
 kiír :adől
 lókért "idő 0
 ism :db
  [ha első elem hányadik :l = 1
   [lókért "idő :idő + (60*első en elem hányadik :l)
                     + első en en elem hányadik :l
    (kiír :idő en en en elem hányadik :l)]]
 kiírőeszköz []
vége
```

A feladat megoldásában feltesszük, hogy Eric Clapton mellett más nem szerepel Eric néven, és csak ezt keressük.

```
eljárás feladat3 :l
 lókért "első 0 lókért "utolsó 0
 amíg [nem (első en elem hányadik :l = "|Eric|)]
  [lókért "első első elem hányadik+1 :l]
 ism :db
  [ha (első en elem hányadik :l) = "|Eric|
    [lókért "utolsó első elem hányadik :l]]
 lókért "eltelt :utolsó-:első
 (mutat "Az eltelt idő: egészhányados :eltelt 3600 "|:|
   egészhányados maradék :eltelt 3600 60 "|:|
      maradék :eltelt 60)
vége
```

*83. (4.) Amikor az „Omega.Legenda" című száma elkezdődött, Eszter rögtön csatornát váltott. írja a képernyőre, hogy a szám melyik adón volt hallható, és azt, hogy a másik két adón milyen számok szóltak ekkor. Mivel a számok a kezdés időpontja szerint növekvő sorrendben vannak, így a másik két adón már elkezdődött a számok lejátszása. Feltételezheti, hogy a másik két adón volt még adás.*

***A*** feladat szerint egyrészt megkeressük, hol szólalt meg a keresett szám, ez lesz a ***hol?*** eljárás, majd a műsorban megnézzük időben visszafelé haladva, hogy a másik két adón mikor szerepelt hozzá képest utoljára szám.

```
eljárás hol? :l :cím
 ha üres? :l [eredmény 0]
 hak (első en en en első :l) = :cím
  [eredmény 1]
  [eredmény 1 + (hol? en :l :cím)]
vége
```

```
eljárás feladat4 :l :hely
  (mutat "A keresett szám aj első elem :hely :műsor
         "|. adón szólt.|)
  lókért "adó elemnélküli első elem :hely :műsor [1 2 3]
  (mutat "|A többi adón ekkor:|)
  ciklusegyenként "j :adó
   [lókért "itt :hely
    amíg [:j <> első elem sitt :műsor]
     [lókért "itt :itt-1]
    (mutat "|Az| első elem :itt :műsor "|. adón|
            en en en elem :itt :műsor "|szólt.|)]
vége
```

84. *(5.) Az egyik rádióműsorban sms-ben, telefonon, de akár képeslapon is kérhető szám. Ám a sokszor csak odafirkált kéréseket olykor nehéz kibetűzni. Előfordul, hogy csak ennyi olvasható: „gaoaf", tehát ezek a betűk biztosan szerepelnek, mégpedig pontosan ebben a sorrendben. Annyi biztos, hogy először a szerző neve szerepel, majd utána a szám címe. Olvassa be a billentyűzetről a felismert karaktereket, majd írja a KERES.TXT állományba azokat a számokat, amelyek ennek a feltételnek megfelelnek. Az állomány első sorába a beolvasott karaktersorozat, majd utána soronként egy zeneszám azonosítója kerüljön! A feladat megoldása során ne különböztesse meg a kis- és a nagybetűket!*

A válasz megadása újabb előkészületeket igényel. Az első részeljárásunk számok címéből eltávolítja a szóközöket, így egyetlen szóvá téve azokat. Erre azért van szükségünk, mert a ***műsor*** változónk nem egy elemként tartalmazza a műsorszám címét, és emiatt körülményes kezelni. A # hozzáfűzése a címhez azért kell, hogy a keresésnél ne kapjunk üres elemet.

```
eljárás szóköz.ki :l :no
 ha üres? :l [eredmény []]
 eredmény elsőnek (mondat (sző irt első :l "#) :no) szóköz.ki
en :l :no+l
vége
```

A szóközök tényleges kiirtását végző eljárásunkat listára is fel kell készítenünk, mert az eredeti fájl tartalmaz [ ] jelek közé írt részt is, ettől meg kell szabadulnunk.

```
eljárás irt :l
 ha üres? :l [eredmény "||]
 hak lista? első :l
   [eredmény elsőnek irt első :l irt en :l]
   [eredmény elsőnek első :l irt en :l]
vége
```

Magát a szám címét az adó és időpont adatoktól megtisztítva kell szóköztelenítenünk, ezt oldja meg a ***számcím*** eljárás.

```
eljárás számcím :l
 ha üres? :l [eredmény []]
 eredmény elsőnek en en en első :l számcím en :l
vége
```

A tényleges választ megadó eljárásunkban a töredék, *f* elemeit egyesével vizsgáljuk, és ha nem eleme az adó, akkor figyelmen kívül hagyjuk.

```
eljárás feladat5 :f :l
 ciklus "i (mondat 1 elemszám :l)
   [ciklusegyenként "j :f
     [hak eleme? :j mélyelem (mondat :i 1) :l
       [lókért "l mélycserél (mondat :i 1) :l
                 en elemtől :j mélyelem (mondat :i 1) :l]
       [lókért "l mélycserél (mondat :i 1) :l "||]
   ]]
 ciklus "i (mondat 1 elemszám :l)
   [ha nem üres? mélyelem (mondat :i 1) :l
    [mutat elem :i :műsor]]
vége
```

*85. (6.) Az 1. adón változik a műsor szerkezete: minden számot egy rövid, egy-perces bevezető előz majd meg, és műsorkezdéstől minden egész órakor 3 perces híreket mondanak. Természetesen minden szám egy részletben hangzik el továbbra is, közvetlenül a bevezető perc után. így ha egy szám nem fejeződne be a hírekig, el sem kezdik, az üres időt a műsorvezető tölti ki. írja a képernyőre óra:perc:másodperc formában, hogy mikor lenne vége az adásnak az új műsorszerkezetben!*

Ezután a kapott listából könnyen ki tudjuk nyerni a kért információt.

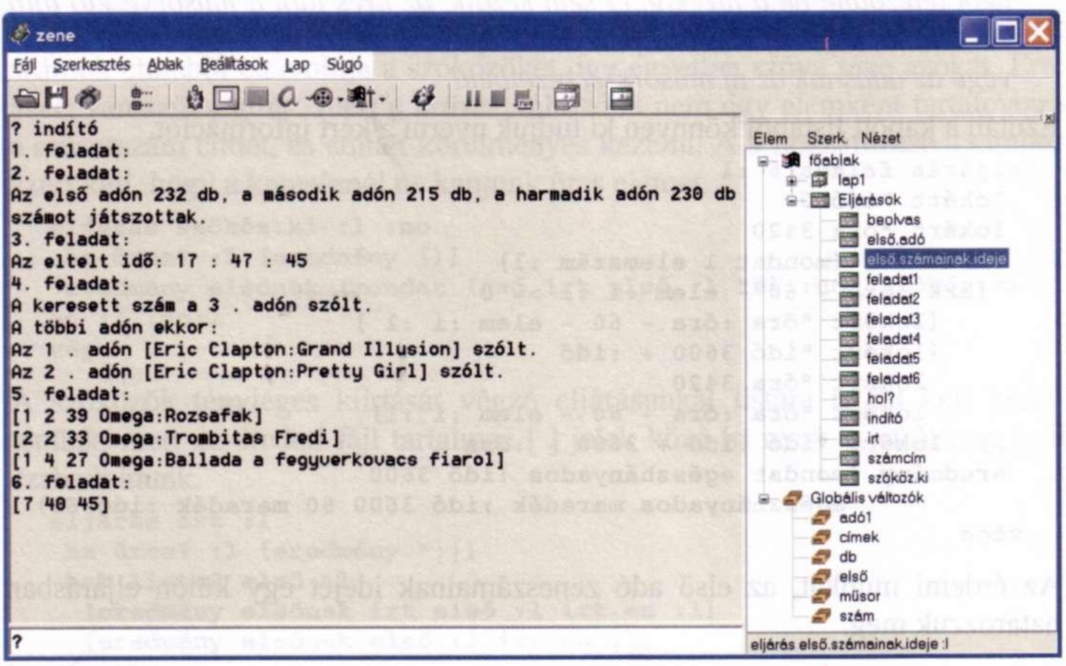
```
eljárás feladatö :l
 lókért "idő 0
 lókért "óra 3420
 ciklus "i (mondat 1 elemszám :l)
   [hak :óra - 60 - elem :i :l >= 0
      [lókért "óra :óra - 60 - elem :i :l ]
      [lókért "idő 3600 + :idő
       lókért "óra 3420
       lókért "óra :óra - 60 - elem :i :l]
   ]  lókért "idő :idő + 3600 - :óra
 eredmény (mondat egészhányados :idő 3600
           egészhányados maradék :idő 3600 60 maradék :idő 60)
vége
```

Az érdemi munkát, az első adó zeneszámainak idejét egy külön eljárásban határozzuk meg.

```
eljárás első.számainak.ideje :l
 lokért "s []
 ism :db
  [ha első elem hányadik :l = 1
   [ lokért "s
    elsőnek (60*első en elem hányadik :l) +
               első en en elem hányadik :l :s]]
 eredmény :s
vége
```

A program indítója az alábbi lesz.

```
eljárás indító
 mutat "|1. feladat:| feladat1
 mutat "|2. feladat:| feladat2 :műsor
 mutat "|3. feladat:|
 első.adó :műsor
 globvál "első beolvas "|elso.txt|
 feladat3 :első
 mutat "|4. feladat:|
 feladat4 :műsor hol? :műsor "|Omega:Legenda|
 mutat "|5. feladat:|
 feladat5 "gaoaf szóköz.ki számcím :műsor 1
 mutat "|6. feladat:|
 mutat feladat6 első.számainak.ideje :műsor
vége
```



# Felhasznált és ajánlott irodalom

Könyvünk megírásában az Imagine Súgója igen nagy segítségünkre volt. Ezt minden korábbi Comenius LOGO felhasználó is alátámaszthatja, aki megpróbált akár egyszer is eligazodni annak Súgójában.

> Mindezek ellenére azt tudnunk kell, hogy a jelenlegi Súgó sem hibátlan, mivel az időközben bekerült fejlesztések, módosítások és javítások hatását nem sikerült mindenhol rögzíteni.

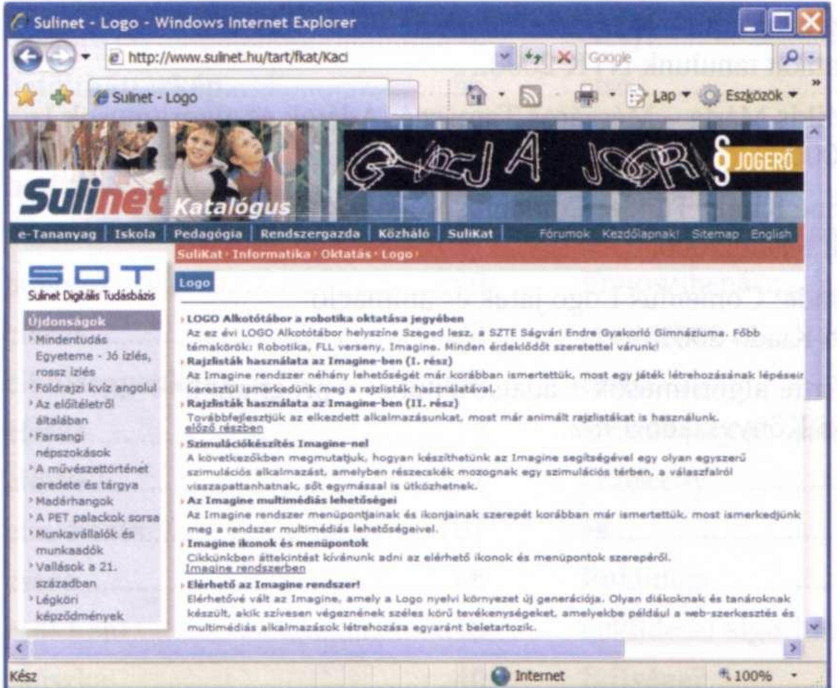
Alapmű gyanánt mindezek ellenére két forrást javaslunk:

- Az Imagine telepítése után olvasható magyar fordításának súgói és a munkafüzete mindenképpen fontos és meghatározó szerepű ismerettár lesz a munkánk során.
- Seymour Papert
  Észrengés, a gyermeki gondolkodás titkos útjai
  SZÁMALK 1988

## *Források a weben*

Az SDT tartalmazza a Digitális írásbeliség fejezetében a magyar és angol parancsszavak leírását és bemutatását is.

E hely kezdőoldala a http://www.sulinet.hu/tart/fkat/Kaci című.



A Sulinet Informatika rovatának szerkesztője, ABONYI-TÓTH ANDOR, aki az Imagine program egyik magyarra fordítója, több cikket is megjelentetett a program képességeiről, ezeket az oldalakat mindenképpen célszerű felkeresni. Kiindulási pontként a rovat LOGO-s katalógusát célszerű felkeresni.

Másik forrásként a már említett http://imagine.elte.hu/ oldalt javasoljuk.

## Ajánlott irodalom LOGO-hoz

Bár az Imagine sok tekintetben túlszárnyalja a ComLOGO lehetőségeit, a jelenleg még szerény magyar nyelvű Imagine kiadványok mellett a következő korábbi műveket is sok haszonnal forgattuk, és forgathatja az Olvasó is.


- Dietrich Senftleben - Turcsányiné Szabó Márta:
  A Logo programozási nyelv MK 1986.
- Seymour Papért: Észrengés
  SZÁMALK 1988.
- Farkas Károly: Logo példatár
  Pest Megyei Pedagógiai Intézet 1990.
- Turcsányiné Szabó Márta - Zsakó László: Comenius Logo gyakorlatok
  Kossuth Kiadó 1997.
- Mészáros Tamásné: Logo-világ
  NTK 1997.
- Kőrösné Mikis Márta - Mészáros Tamásné:
  Informatikát tanulunk NTK 2000.
- Kőrösné Mikis Márta - Mészáros Tamásné: Adatok és algoritmusok I.
  NTK 2000.
- Szentpéteriné Király Tünde: Comenius Logo teknőcgrafika
  Kossuth Kiadó 2000.
- Dancsó Tünde: Comenius Logo játék és animáció
  Kossuth Kiadó 2001.
- Niklaus Wirth: algoritmusok + adatstruktúrák = programok
  Műszaki Könyvkiadó 1982.

# Tárgymutató

A tárgymutatóba helyeztük el a könyvben ténylegesen alkalmazott Imagine parancsokat, ezeket félkövéren szedtük. A parancsok részletesebb leírását a könyv honlapján találhatjuk meg, ahol kereshetőek, illetve ha létezett, akkor az eredeti ComLOGO alapszót is megadjuk.


8 királynő probléma 98
adatbevitel 13
adatkivitel 11
adattípus 61
**aktívpont** 39
**alak!** 50
algoritmizálás 75
általános szóírás 61
**amíg** 22
**átdob** 56
átirányítás 14
backtrack 98
beszúró elv 92
**betölthátterkép** 55
**betöltőút** 14, 41
**betűtípus!** 12
**bezárintéző** 46
buborék elv 90
**ciklus** 22
**ciklusegyenként** 26
**címke** 12
**címke2** 12
címletezés 101
**csere** 66
**csökkent** 10
csúszka 40
de Vigenére 114
**egészhányados** 53, 80, 107
**elágazás** 19, 47
eldöntés 75
**eleme** 65
**eleme?** 65
elemi algoritmus 75
**elemnélküli** 63, 70
**elemsorszámnélküli** 63
**elemszám** 64, 70
**elemtől** 64
**eljárás** 30
**elrejt** 50
**első** 63, 70
**elsőnek** 64, 70
**elsőnélküli** 63, 70
emelt szintű érettségi 109
Eratoszthenész 32
**eredmény** 30, 65
**érték** 40, 50
értékes jegyek száma 108
**érzékeny** 38
**és** 17
Euklidész 32
euklideszi algoritmus 32
**fájlvége?** 14

**felsőszintre** 56
fésűs egyesítés 88
Fibonacci sorozat 96
**figyelj** 42
függvény 30
Gauss 98
generálás 39
globális változók 9
**globálisváltozó** 9
**gombmenü!** 56
**gombnyomás?** 40
**gyök** 33
gyökvonás 31
**ha** 18
**haHamis** 19
**haIgaz** 19
**hakülönben** 18
halmazműveletek 85
**hamis** 17
**hanghullám** 48
**hangsor** 51
Hanoi-torony 93
**hányadik** 21, 48
**háttérszín!** 51
hátul tesztelő ciklus 28
hatványozás 31
időmérés 29
**igaz** 17
IIP kiterjesztés 44
Imagine plugin 45
**irány!** 55
**irányszög** 48
**ismétlés** 20
iteráció 20
**képkocka!** 53, 55
**képkockamód** 52
**képkockamód!** 53
**kér** 42
**kéregyenként** 51
**kiaktív** 52
**kiindulópont!** 51, 55
**kiír** 11
**kiírérték** 11, 95
**kiíróeszköz** 16
**kiválaszt** 60
kiválasztás 76
kiválogatás 79
**kizáróvagy** 17
**láthatótartomány!** 52
lebegőpontos ábrázolás 108
Legendre 31
legnagyobb egész szám 104
legnagyobb valós szám 105
**lejátszikfájl** 41
lineáris keresés 77
lista 69
listaműveletek 70
logikai műveletek 17
LogoMotion 39
Logotron 8
lokális változók 9
**lokálisérték** 10
**lokálisváltozó** 9
Lucas 94

**!** 25
**maradék** 42, 107
matematikai algoritmusok 31
mátrix 72
maximum keresése 78
**maxméret** 53
medián 79
megszámlálás 77
**mélycserél** 72
**mélyelem** 72
**méret!** 51, 54, 55
metszet 86
**minden** 41
**mindenteknőc** 54
minimum elv 91
minimum megkeresése 78
mohó algoritmus 101
**mondat** 25, 33, 52, 69, 81
**mutat** 11
**mutatikonsor** 53
**mutatikonsor** 54
**mutatteknőc** 50
műveletek szavakkal 63
**nem** 17
**név** 9
Newton 31
nézőablak 52
**növel** 10, 48
**nullázstopper** 29
**olvasjel** 13
**olvaskód** 55
**olvaslista** 14
**olvasóeszköz** 14
**olvasszó** 13
**osztottablak** 54
önálló EXE 43
**összeg** 124
összegzés 75
**összekever** 95
ötös lottó 109
panel 57
parancs 8
permutáció 95
**pontszín** 41
postfix 104
prímszita 32
prímtényezőkre bontás 80
programozási hibák 103
programozási tétel 75
**rajzlapablak** 54
rajzlista 38
rejtikonsor 54
rekurzió 30
**rendez** 80, 89
rendezés 89
**stopmind** 55
**stopper** 29
stringhossz 83
**szám?** 62
számrendszerek közötti átváltás 95
szegmens 66
szekvencia 17
szelekció 18
szétválogatás 86

**szó** 48, 61, 62
**szó?** 62
**takar?** 41
**takart** 50
**tartomány!** 52
**tartománystílus!** 52
**teszt** 19
**tetszőleges** 59
típuskényszerítés 105
tömb 72
**törölháttérkép** 50
**törölképernyő** 54
**törölobjektum** 54
**törölszöveg** 11
**új** 54
unió 85
**út** 14
**utolsó** 63, 70
**utolsónak** 64, 70
**utolsónélküli** 63, 70
üres lista 69
**üres?** 64, 70
vágólap 16
**vagy** 17
**váramíg** 29
**várj** 28
**vége** 30
végtelenszer 22
vektor 72
**véletlenszám** 47
vezérlési szerkezetek 17
visszalépéses algoritmus 98
visszapattan 38, 52
vonszolható 38
webprojekt 43
**xor** 17
**xpozíció** 47
**xpozíció!** 47
**xypoz!** 41
**ypozíció** 47
**ypozíció!** 47

Ü

# Tartalomjegyzék

Rövid tartalomjegyzék 5
Kedves Olvasó! 7
A szóhasználatról 8
Programkészítés alapjai 9
Változó létrehozása, értékadás 9
Globális változók 9
Lokális változók 9
Adat be- és kivitel 11
Kiírás 11
Szöveg kiírása grafikus képernyőre 12
Bevitel 13
Átirányítás 14
Bevitel 14
Kivitel 16
Vezérlési szerkezetek 17
Szekvencia 17
Logikai müveletek 17
Szelekció 18
Egyágú kiválasztás 18
Kétágú kiválasztás 18
Többágú kiválasztás 19
Iteráció 20
Számlálásos ciklus, ciklusváltozó nélkül 20
Elöltesztelő ciklus 22
Feltétel nélküli, végtelen ismétlés 22
Számlálós ciklus, ciklusváltozóval 22
Halmazon végrehajtott ismétlés 26
Mókusok (NTTV 2001/02. 5-8. évfolyam 2. forduló) 27
Hátul tesztelő ciklus 28
Program futásának felfüggesztése 28

Várakozás adott ideig 28
Várakozás egy feltétel teljesüléséig 29
Időmérés stopperrel 29
Eljárás 30
Függvény 30
Rekurzió 30
Matematikai algoritmusok 31
Kiíratási gyakorlatok 34
Játékok készítése 35
Rali! 36
Hogyan készítjük el? 37
A pálya elemei 37
Teknőcök 38
Az eljárások 41
Önálló program készítése 43
Cápa az akváriumban 45
Szedjünk virágot! 49
Síeljünk! 51
Menjünk végig a labirintuson! 54
Tervező lap 54
A játék lap 55
Fogócska 57
Fej-írás játék 60
Funkcionális LOGO 61
Adattípusok 61
Egyszerű adatobjektumok, primitívek 61
Szó 61
Műveletek szavakkal 63
Feladatok szókezelésre 65
Lista 69
Műveletek listákkal 70
Feladatok listákkal 70

Listakezelés haladó szinten 72
Vektor létrehozása és kezelése 72
Kétdimenziós tömb létrehozása és kezelése 72
Programozási tételek 75
Elemi programozási tételek 75
Összegzés 75
Eldöntés 75
Kiválasztás 76
Lineáris keresés 77
Megszámlálás 77
Minimum megkeresése 78
Maximum keresése 78
Kiválogatás 79
Programozási tételek alkalmazása 79
Medián meghatározása 79
Prímtényezőkre bontás 80
Írjuk ki fordítva! 81
Nőjön mind! 81
Vonjuk le mindből! 81
Számzár ( NTTV 2001/02. 5-8. évfolyam 2. forduló) 83
Halmazmüveletek 85
Unió 85
Metszet 86
Szétválogatás 86
Fésüs egyesítés 88
Mégis, hány elemű? 89
Alapvető rendezések 89
Buborékrendezés 90
Javított buborékrendezés 91
Minimum elvű rendezés 91
Beszúró rendezés 92
Nevezetes feladatok 93

Rekurzió alkalmazása ... 93
Hanoi torony ... 93
Váltsunk! ... 95
Permutáció ... 95
Jó a rekurzió, de mindig? ... 96
Visszalépéses algoritmus ... 98
8 királynő probléma ... 98
Mohó algoritmus ... 101
Amikor lehetünk mohók ... 101
Fizessünk hármas pénzzel! ... 101
Amikor nem lehetünk mohók ... 102
Vegyes gyakorló feladatok ... 102
Programozási hibák ... 103
Igaz vagy hamis? ... 103
Emeljük a tétet! ... 104
Most akkor hány kell?! ... 106
Valós számok? ... 108
Emelt szintű érettségi Imagine környezetben ... 109
2005. május 19. Lottó ... 109
2005. október 27. Vigenére tábla ... 114
2006. február 28. Telefonszámla ... 118
2006. május 17. Fehérje ... 122
2006. november 3. Zenei adók ... 127
Felhasznált és ajánlott irodalom ... 133
Források a weben ... 133
Ajánlott irodalom LOGO-hoz ... 134
Tárgymutató ... 135
Tartalomjegyzék ... 139

**Ára: 1860 Ft**

***Kedves Olvasó!***

Ha a LOGO-val szemben támasztott előítéleteit néhány óra erejéig félre tudja tenni, akkor ígérjük, hogy e könyvön keresztül egy olyan világba engedünk bepillantást, ahol a más nyelven már jól programozókat is izgalmas kalandok várják. Emellett reméljük, hogy az eddig megismert programozási nyelvek jobb megértésében is segíthetünk.

E könyv közel száz mintaprogramon és százötven feladaton keresztül mutatja be az Imagine lehetőségeit, amelyek le is tölthetőek a regisztráció után.

***Miért Imagine?***

Mert...

... ingyenesen letölthető a közoktatásban tanulók és tanítók számára a logo.sulinet.hu oldalról.

... tartalmaz rajzoló és animáció készítő programot is.

... segítségével böngészhetünk a weben, hallgathatunk, szerkeszthetünk zenét, nézhetünk filmet, és mindezt akár egyszerre is!

... képes hangutasítások végrehajtására.

... hálózati, párhuzamos és osztott végrehajtású alkalmazásokat is fejleszthetünk vele.

... a Comenius LOGO-ban elkészített programjaink kis átalakításokkal futtathatóak.

... bármely parancsszó átdefiniálható.

... érettségizni lehet a közeljövőben Imagine környezetben.

... magyar parancsszavakat használhatunk.

A könyv háttéranyaga a **www.abax.hu/imagine** honlapon található. Ugyanitt megrendelhető a sorozat többi tagja is.

ISBN 978-963-06-2496-1