

**Viola Zoltán**  
[violazoli@gmail.com](mailto:violazoli@gmail.com)

# A mau programozási nyelv

Verziószám: **16**

Release date: 2014.06.20

A mau programnyelv hivatalos oldala és fóruma:

<http://parancssor.info>

**M@U**  
„A hatékony krixkraxok és macskakaparások interpretere”  
"The programming language with efficient line noise"

  
...and scrawls...

## KÖSZÖNETNYILVÁNÍTÁS ÉS AJÁNLÁS:

Ezúton mondok köszönetet barátomnak, Ökrösy Gyulának, s ajánlom neki e könyvet és az egész mau programnyelvet (bár tudom előre hogy ez sose lesz a kedvence...) mert ő vezetett be engem a Linux világába, ő nyomta a kezembe sok évvel ezelőtt az első linuxos telepítő CD-t, s ettől eltekintve is mindig bátran fordulhattam hozzá minden linuxos vagy programozási kérdésben, s mindig roppant segítőkész volt, holott tudom jól, nemegyszer alaposan próbára tettem őt az értetlenségemmel és extravagáns igényeimmel/elképzeléseimmel!

## Tartalomjegyzék

A mau programnyelv fordítási opciói.....	4
Előszó a bevezetéshez.....	6
Bevezetés.....	7
1. fejezet - A mau nyelv változói.....	20
2. fejezet - Az aritmetikai kifejezések.....	26
3. fejezet - Értékadás.....	32
4. fejezet - A pontosvesszők.....	34
5. fejezet - A megjegyzések, kommentek.....	35
6. fejezet - Tömbök.....	35
7. fejezet - Stringek.....	37
8. fejezet - Stringtömbök.....	43
Stringtömbök névsorba rendezése.....	44
Stringek darabolása határolókarakter szerint.....	45
9. fejezet - Inkrementálás és dekrementálás.....	46
10. fejezet - Összevont utasítások.....	49
Egyenlőségjellel összevont utasítások.....	49
Előjelváltás.....	49
Gyorsfüggvények.....	50
Gyorsparancsok.....	50
11. fejezet - Kiíratás.....	51
12. fejezet - Vermek.....	55
13. fejezet - Mau rendszerváltozók és rendszerfüggvények.....	55
Alapértelmezetten #l (mau_l) értéket visszaadó rendszerfüggvények.....	56
Explicit módon casting operátorral jelölt rendszerfüggvények.....	58
#c értéket visszaadó rendszerfüggvények.....	58
#l értéket visszaadó rendszerfüggvények.....	59
#L értéket visszaadó rendszerfüggvények.....	60
#g értéket visszaadó rendszerfüggvények.....	60
#s értéket visszaadó rendszerfüggvények.....	60
14. fejezet - Vezérlési szerkezetek, azaz elágazások, ciklusok és „esetek”.....	66
UTF-8 kódok.....	66
Az if és a ha utasítás.....	69
Ciklusok.....	72
A „végtelen ciklus”.....	72
Hátultesztelő ciklus.....	73
Elöltesztelő ciklus.....	74
Előre rögzített fix számszor lefutó ciklus.....	75
String hosszától függő ciklus.....	80
Utasításblokkok.....	81
A switch-szerű vezérlési szerkezet.....	84
Elágazás keresési eredménytől függően.....	85
15. fejezet - Névterek.....	88
16. fejezet - File-kezelés.....	93
Input fájlok.....	93
Output fájlok.....	99
17. fejezet - Tartalomjegyzékek (directory) kezelése.....	103

18. fejezet - Mau nyelvű függvények.....	108
19. fejezet - Változó hosszúságú paraméterlista kezelése.....	119
20. fejezet - „Igazi” függvények és rekurzív függvények.....	123
21. fejezet - Bencsmarkok.....	125
22. fejezet - A mau interpreter és a shell kapcsolata.....	126
Shell parancs eredményének tömbbe olvasása.....	129
23. fejezet - A BRAINFUCK interpreter, avagy „ez itt a humor helye”.....	132
24. fejezet - Hasznos feladatokat ellátó mau nyelvű példaprogramok.....	136
A „maudir” program.....	136
File sorait névsorba rendező program.....	139
Állománylistázó program, olyasféle mint az "ls -l".....	140
Szótárprogi.....	142
Mau plugin készítése a Surf böngészőhöz.....	145
Mau nyelvű statusbarkezelő program a DWM ablakkezelőhöz.....	148
Parancssoros GMAIL watcher program mau nyelven.....	148
Menü a DWM ablakkezelőbe.....	150
25. fejezet - Stringkonstansok a programmemóriában.....	151
26. fejezet - Álfüggvények.....	154
27. fejezet - Közös kódú függvények.....	156
28. fejezet - A mau programozás csapdái.....	157
29. fejezet - Önmódosító programok készítése.....	159
30. fejezet - Pluginek készítése a mau interpreterhez.....	161
31. fejezet - A logolás egy mau programban.....	173
32. fejezet - Saját billentyűzetkiosztás készítése.....	174
33. fejezet - Esettanulmány: egy mau program felgyorsítása.....	186
34. fejezet - Hasznos X parancsok.....	190
A DMENU integráció.....	190
Kiiratás a statuszbarra.....	192
35. fejezet - A mau programnyelv fejlesztését segítő eszközök.....	192
36. fejezet - Mau megszakításrendszer.....	197
37. fejezet - A BETŰ és a JELSOR, vagyis az UTF-8 kódolású karakterek és stringek.....	200
A BETŰk.....	201
A JELSORok.....	208
A JELSOR-tömbök.....	217
38. fejezet - Bitmezők kezelése.....	220
39. fejezet - Egzisztenciafüggvények.....	226
40. fejezet - Streamkezelés.....	229
41. fejezet - Az ncurses integráció.....	232
Ncurses-sel összefüggő mau utasítások.....	232
Ncurses-sel összefüggő függvények.....	233

# A mau programnyelv fordítási opciói

A mau programnyelv lefordítása egyszerűen a

`make`

parancs kiadásával történik. Ekkor létrejön egy

`mau`

nevű futtatható bináris állomány, ez maga az interpreter, ezt kell valami olyan helyre bemásolnunk, ami könyvtár szerepel a rendszerünk \$PATH változója által definiált elérési útvonalak közt.

A mau interpreter beépítetten támogatja az ext fájlrendszerek speciális flagjainak lekérdezését is, emiatt igényli az ext2 header fájlkat. Ezek szokásosan abban a programcsomagban vannak fent, amelynek neve általában „e2fsprogs” szokott lenni a Linux operációs rendszerben (vagy valami nagyon hasonló). E programcsomag tehát úgymond a mau programnyelv „függősége”. Ám csak opcionális függőség, amennyiben ugyanis nem használunk ext filerendszert, lefordíthatjuk e nélkül az interpretert, s ehhez csak annyi kell, hogy a **vz.h** fájl első sorai közt található e sort:

```
#define EXT2ATTRIBUTUMOK
```

kommenteljük ki az eléje írt `//` jelekkel, vagy akár teljesen ki is törölhetjük. Esetben nem igényeli majd az ext2 header fájlkat, s így az e2fsprogs programcsomagot sem, ellenben ennek az az ára, hogy amennyiben netán mégis olyan mau programot óhajtanánk futtatni vele, mely e flageket szeretné beolvasni, akkor annak a programnak a futása e beolvasó utasítás elérésekor azonnal megszakad, a mau interpreter pedig leáll, e hibaüzenetet kiadva magából „elhalálózása” előtt:

```
LOG:> 2014.04.03 23:07:12 : E mau interpreter ext2 filerendszer támogatás nélkül lett  
lefordítva, emiatt nem képes beolvasni  
az ext filerendszer-specifikus flageket!
```

Továbbá, a mau interpreternek opcionális függősége az X szervert is. Ez azt jelenti, hogy a **vz.h** fájlban szerepel a

```
#define X11INTEGRATION
```

direktíva. Emiatt igényel bizonyos headerfájlokat az X egynémely funkcióinak kezeléséhez. Például ilyen a vágólap tartalmának beolvasása a mau programokba. Amennyiben e képességét nem igényeljük, akkor töröljük ki a **vz.h** fájlból a fent leírt sort, ezenfelül pedig a Makefile fájlban ezt a sort:

```
LIBS = -ldl -lX11 -lXmu
```

Írjuk át úgy, hogy a végéről hagyjuk az utolsó 2 paramétert, azaz maradjon belőle csak ennyi:

```
LIBS = -ldl
```

Ezenfelül, a mau interpreter fel van készítve többféle architektúra támogatására is. Legalábbis elvben... Gyakorlatilag jelenleg az X86 architektúra 32 és 64 bites verzióin működőképes. Mindegyiken lefordítható külön intézkedés nélkül, de a 64 bites verzióban a „long double” típusú változók 16 bájtól tárolódnak, a 32 bites verzióban azonban csak 12 bájtól. Ez azt a változótípust jelenti, ami a mau

programnyelvben a **#D** casting operátorhoz kapcsolódik. Ettől még működőképes kell legyen minden mau program ami e típust nem használja, sőt az is ami ezt használja, de ezen utóbbiak esetében e méretcsökkenés nyilvánvalóan azt eredményezi, hogy e típus számaábrázolási tartománya illetve pontossága csekélyebb, ezt tehát kéretik figyelembe venni!

Amennyiben valaki azzal kísérletezne, hogy e nyelvet egyéb architektúrákra portolja, például Windows vagy Mac alá, annak legfontosabb kezdeti lépése az kell legyen, hogy a **vz.h** fájlban keresse meg e részt (a fájl elején van, s a fájl különben is rövid):

```
//#define bit32

#ifdef bit32
#define mau_i    unsigned short int
#define mau_I    signed short int
#define mau_l    unsigned int
#define mau_L    signed int
#define mau_g    unsigned long long
#define mau_G    signed long long
#define mau_f    float
#define mau_d    double
#define mau_D    long double
#define mau_c    unsigned char
#define mau_C    signed char
#define mau_p    void *
#else
#define mau_i    unsigned short int
#define mau_I    signed short int
#define mau_l    unsigned int
#define mau_L    signed int
#define mau_g    unsigned long long
#define mau_G    signed long long
#define mau_f    float
#define mau_d    double
#define mau_D    long double
#define mau_c    unsigned char
#define mau_C    signed char
#define mau_p    void *
#endif
```

Mint látható, itt ki van kommentelve a

```
#define bit32
```

direktíva. Ennek Linux alatt jelenleg nincs jelentősége ezen interpreter számára, ellenben más architektúráknál ez lehetséges hogy fontos. Ezt hagyjuk így vagy távolítsuk el a kommentjelet ahogy nekünk tetszik, majd az alatta levő megfelelő blokkban a megfelelő mau számaábrázolási típusokat - mindegyik úgy kezdődik, hogy „**mau\_**” - állítsuk be úgy, hogy ezek bájtmérete megfelelő legyen az adott architektúrán! A következő méretek beállítása szükséges:

```
mau_c 1 bájt
```

```
mau_C 1 bájt
```

```
mau_i 2 bájt
```

```
mau_I 2 bájt
```

```
mau_l 4 bájt
```

```
mau_L 4 bájt
```

```
mau_g 8 bájt
```

```
mau_G 8 bájt
```

```
mau_f 4 bájt, lebegőpontos
```

```
mau_d 8 bájt, lebegőpontos
```

mau\_D mérete lehet architektúrafüggő, de nem nagyobb mint 16 bájt. Ez kell legyen a legnagyobb méretű lebegőpontos típus az adott architektúrán.

Azaz, e mau típusok után úgy kell megválasztani az adott architektúrán rendelkezésre álló C++ fordító típusait, hogy e bájtméretek legyenek beállítva e mau típusokra.

A mau interpreter bizonyos funkcióknál kijelzi a hét adott napjának nevét is, egy 3 karakteres rövidítéssel, ami a magyar napnévből származik alapértelmezés szerint, ez azonban átállítható mint fordítási paraméter. A megfelelő, azaz átírandó sorok a vz.h fájlban vannak, ezeket kell kicserélni igény szerint:

```
#define mau_vasarnap "V "  
#define mau_hetfo "H "  
#define mau_kedd "K "  
#define mau_szerda "Sze"  
#define mau_csutortok "Cs "  
#define mau_pentek "P "  
#define mau_szombat "Szo"
```

További pár fordítási opció megtekintéséhez lásd még az „A DMENU integráció” című fejezetet is!

## Előszó a bevezetéshez...

Ezen előszót abból az alkalomból írom, hogy úgy döntöttem, ezen... hm... „izémet” (Alkotásomat? Publikációmát? Dokumentumot? Könyvemet? Programnyelvet?... ) feltöltöm a MEK-be. Eddig is elérhető volt ugyan ingyen a mau programnyelvem, azaz letölthető volt a honlapomról (melynek linkje ott virít e doksi elején), de a MEK-be feltöltés mellett amiatt döntöttem, mert - ellentétben azzal, amit korábban a fórumomon beharangoztam - vélhetőleg nem fog sor kerülni rá, hogy a mau programnyelvet tovább fejlesszem ebben a formában, például kibővítem reguláris kifejezések kezelésének lehetőségével, illetve objektumorientált képességekkel. Ennek oka nem a lustaságom, s nem is időhiány. Bár nyaranta nemigen van időm komolyabb szellemi erőfeszítésekre a hobbym érdekében, telente lenne erre bőven mód. Nem is az az oka, hogy a mau nyelvet ne szeretném, s nem is az, hogy e nyelv rossz lenne, vagy hibás koncepció eredménye, alkalmatlan volna ilyesmire, de még csak nem is az, hogy e további fejlesztések nehezek volnának. (Ami az OOP megvalósítását illeti, kifejezetten könnyűnek tartom, a regexp sokkal nehezebb. De félig már az is megvan, nem volna egy ördögösség befejezni...)

Annak oka hogy így döntöttem, az, hogy

1. A mau jelenlegi formájában is már tökéletesen alkalmas az ÉN mindennapos feladataimhoz.
2. (ez a fő ok): Rájöttem, még ha ez egyesek szemében túlzott szerénységnak tűnik is, hogy az én képességeimhez nagyobb feladatok a valóak/méltóak! Szokásos bombasztikusan túlzó stílusomban úgy fogalmazhatok, hogy bár korábban nagy vágyam volt programnyelvet alkotni, de most hogy sikerült, rájöttem hogy ez is csak „kispályásoknak” való szellemi feladat, afféle „szellemi hátulgom-bolósaknak”... Ilyesmit hogy egy új programnyelv, egy szorgalmasabb utcalány is összedob pár hónap alatt, ha úgy dönt, hogy felhagy a kéjipari szakmával, s egy kicsit elmélyed a megfelelő tudnivalókban!

Hogy most mit akarok megcsinálni? Egy komplett virtuális számítógépet, aminek természetesen saját „gépi kódja” lesz, erre természetesen egy komplett, saját operációs rendszert kell írnom, természetesen ezen operációs rendszer alá szükségszerű lesz megvalósítani valamiféle magas szintű programnyelvet is... Ezen utóbbi nyilvánvalóan maga a mau nyelv lesz, vagy valami ahhoz nagyon hasonló. Ez természetesen jókora feladat, előre tudom hogy több évig fog tartani míg legalább a pre-alfa állapotot eléri, de akkor is erre vágyom, viszont ha ezzel foglalkozom, nem tudok foglalkozni a jelenlegi mau programnyelvvvel! Úgy illik azonban, hogy amit eddig alkottam belőle - s ami hitem szerint nagyon is hasznos lehet sokaknak, akár tanulási célra, akár mindennapos szkriptelési munkákra - azt közreadjam, így aki akarja, folytathatja ennek fejlesztését is, vagy egyszerűen csak örülhet neki, ötleteket meríthet belőle, szóval, ha valami újba kezdek, illik lezárni valamiképp az addig történt ténykedéseket... Ezt teszem most, a MEK-be feltöltéssel!

Ezen anyag megegyezik a honlapomon található legutolsó változattal ami a forráskódot illeti, s e dokumentáció is csak annyiban másabb, hogy ebben benne hagytam az Ncurses integrációval foglalkozó fejezetet, ami azonban afféle félig kész fejlesztés. Úgy értem, az ott leírt utasítások (reményeim szerint) működnek, de mégis félig implementált featurenak tekinthető csupán, mert számos olyan utasítás egyszerűen nem létezik, ami tulajdonképpen kéne ahhoz, hogy igazán azt mondhassuk, a mau támogatja az ncurses programozási stílust. Mégis meghagytam, mert minek irtsam ki amit már megcsináltam, s hátha valaki belefejleszti a hiányzóakat! Ez nem nehéz akkor se ha nem akar belenyúlni magába az interpreterbe, mert a mau nyelv igazán könnyen pluginezhető, amint ennek módszerét le is írja e dokumentáció egyik fejezete.

Amint a MEK könyvtárosai visszajelzik nekem, hogy a művem feltöltésre került hozzájuk, úgy értem, letölthető a MEK-ből, én magam azonnal le is törölöm a tárhelyemről az ottani csomagokat, s a mau forráskódja egyedül a MEK-ből érhető majd el. Teljesen felesleges ugyanis több példányt is meghagyni. A MEK jó és megbízható, sőt, mondhatnám előkelő hely.

## Bevezetés

A „mau” egy olyan programozási nyelv, amit én, Viola Zoltán alkottam meg. Ezt azért tettem, mert

1. Író vagyok, s a sci-fi sorozatom számára szükséges volt egy programozási nyelv, ami nem azonos egyik jelenleg létező nyelvvel sem, s amelyen írhatok majd néhány programféleséget vagy ilyesmiknek a részleteit a regényeimbe, a hangulat fokozása s a színvonal emelése céljából. Na most, nem akartam halandzsát írni, így megalkottam a szóban forgó nyelvet. Ez tehát a legkifejezettebben egy IRODALMI CÉLLAL létrehozott, mindazonáltal működő és (reményeim szerint) akár komoly feladatokra is használható PROGRAMOZÁSI nyelv. Amennyire tudom, semelyik író soha nem alkotott még meg programozási nyelvet a sorozata kedvéért, ezért ezzel egyedülálló vagyok a Világ (vagy legyek szerényebb: legalábbis a Föld...) összes jelenlegi vagy valaha létezett írója közt, s teljesen nyíltan elismerem hogy erre nagyon büszke vagyok, s ez rém jólesik nekem! Ha azonban

tévednék, s előfordult volna már hogy más író is alkotott az irodalmi művei számára/kedvéért programozási nyelvet, akkor az csak eggyel több ok a számomra, hogy ebben se maradjak el más íróktól!

**2.** Amúgyis érdekelt, képes vagyok-e egy effélének a megírására, azaz vonzott maga a kihívás. Mondhatni tehát hogy afféle „tudományos” vagy „intellektuális” virtus miatt is kedvem volt ezt megcsinálni. Szerintem az ilyesminek sokkal több értelme van, és nemesebb szórakozás, mint az idióta Valóságshowkat bámulni a tévében.

**3.** Teljesen nyíltan bevallom, szeretek dicsekedni, felválni, kérkedni a tudásommal, azaz hiú vagyok. S ebben még csak semmi rosszat se találok, mert mindenki hiú, csak egyesek ezt pofátlanul és képmutatóan letagadják. Én legalább bevallom. Na most, ha valaki megalkot egy új programnyelvet, az azért már elég jelentős valami ahhoz, hogy komoly mértékben növelje azt a „fogalmat”, amit a számítástechnikusok úgy neveznek, hogy az illető „E-penis mértéke”...

**4.** A programnyelvem konkrétan hasznos is lehet mindenfélre, amiatt, mert bár interpreter típusú, de meggyőződésem szerint ezen nyelvek közt jóeséllyel a leggyorsabb, emellett függősége semmi sincs a **g++** fordítóprogram kivételével (Az is csak akkor kell neki ha forrásból fordítjuk az interpreterét), erőforrásigénye elenyésző, emellett pedig teljesen biztos hogy ez a nyelv támogatja jelenleg a legislegjobban az **INDIREKCIÓT**, ugyanis a mau nyelvben szinte minden lehet tetszőleges aritmetikai (vagy string) kifejezés, még a változók NEVE is, vagy a meghívott függvények neve, az ugrások címkéi, a casting operátorok, meg mindenféle más akármik is.

Korábban készítettem egy leírást e nyelv megalkotásának FOLYAMATÁRÓL, aminek az a címe, hogy „**Hogyan írhat sz saját programnyelvet**” (letölthető a MEK-ből: <http://mek.oszk.hu/13000/13092/>), de annak érdekében hogy akit nem érdekel a megalkotás folyamata, csak maga a mau programnyelv, a szintaxis, a szabályok stb, hogy annak tehát ne abból a hosszú tanulmányból kelljen kimazsoláznia a végső változatot, itt most ezt közlöm az érdeklődőkkel. Emiatt aztán itt nem is írom le a programnyelvet megvalósító C/C++ nyelvű kódokat, bár mau nyelven írt példaprogramokat természetesen igen.

Fontos észben tartania a mau nyelvet felhasználóknak, hogy **NEM VÁLLALOK FELELŐSSÉGET SEMMIÉRT SEM**, azaz teljesen felelőtlen fickó vagyok! A kódot közreadom, INGYEN, de mindenki a maga felelősségére használja. Teljesen nyíltan bevallom, hogy bár tudásom a programozás terén minden bizonnyal tekintélyes, ha egyszer e nyelvet meg tudtam alkotni, de **NEM VAGYOK PROFI**, annyira nem, hogy nemcsak nem a programozásból élek, de még csak semmiféle hivatalos papírom sincs belőle, sőt még egy nyamvadt bármilyen **DIPLOMÁM SEM!** (nem programozási diplomám sincs tehát, teljesen diplomamentes vagyok...) Auto-didakta vagyok, műkedvelő, hobbyista. Azaz tökéletesen lehetséges, hogy valaki nálam „szakabb szakember” egyes funkciókat hatékonyabban tudna megvalósítani benne, desőt ami még szomorúbb, az is simán lehetséges (hovatovább roppant valószínű is!!!), hogy akadnak benne „bugok”, azaz hogy bizonyos esetekben a programnyelvem sajnos hibás eredményeket produkál, vagy egyszerűen leáll holmi hibajelzéssel. (Még rosszabb esetben hibajelzés nélkül... Brrr!) Ha így lesz, annak természetesen nem fogok örülni, de szégyenkezni sem, ugyanis ezt az egész hóbelevancot ha hiszitek ha nem, de úgy kb 3 hónap alatt dobtam össze, s nem is minden nap foglalkoztam vele ezen időtartam alatt. A tesztelés egészen minimális volt csak. Igazából **MOST** kezdődik majd a komoly tesztelés, miután



készen lett! Ez tehát legjobb esetben is csak egy „béta-verziós” valaminek mondható, semmi esetre sem „stable” kiadás, sőt, nagyonis jogos még abban is kételkedni, egyáltalán megérdemli-e legalább a béta-verzió besorolást!

Annyi bizonyos, hogy arra jó, amire nekem kell: irodalmi célra. Természetesen ha tudomásomra jut valami hiba, igyekszem javítani, de ismételten figyelmeztetek rá mindenkit, „keep in mind, please”, azaz tartsd észben, hogy ettől a programnyelvtől nem várhatsz el akkora stabilitást, robosztusságot, letisztultságot, mint egy olyantól amit már 20 éve fejlesztenek egyfolytában! Ez most született, s emiatt egészen természetes, hogy teli van „gyermekbetegségekkel”. Emellett, ez egy „one-man project”, azaz „egyemberes fejlesztés”, nem egy egész team végezte, nem is akartam hogy más is belekapcsolódjon, mert azt akartam hogy legyen egy olyan programnyelv amit ÉN MAGAM ALKOTTAM EGYSZEMÉLYBEN, ami emiatt az ENYÉM (mint szellemi alkotás), ami mindenben épp olyan amilyennek látni óhajtom! (Ennek nemcsak a hiúság és gőg az oka: az esetben ugyanis ha e nyelv fejlesztését társakkal együtt oldom meg, csapatmunkában, máris jogi aggályok merülhettek volna fel azt illetően, szabad-e felhasználnom e nyelvből ezt vagy azt valamely regényemben, pláne ha amaz regényből netán anyagi hasznom is származik... Nyilvánvaló, hogy ennek kockázatát nem vállalhattam fel, azaz ha nem akartam volna akkor is muszáj lettem volna ezt egyedül kifejleszteni). Ennek azonban értelemszerűen megvan az a hátránya, hogy nélkülözi a „több szem többet lát” előnyeit, hasznát, azaz ha ÉN nem vettem eddig észre benne egy hibát, azt más se vette észre mert nem voltak mások. Mostantól természetesen már fejlesztheti bárki, de a JELENLEGI állapota olyan amilyen, azaz semmiképp se javaslom hogy mondjuk e nyelven (az ő jelenlegi készütségi fokán) írják meg mondjuk egy atomerőmű biztonsági berendezéseit üzemeltető program létfontosságú funkcióit... (Különb en sem valósidejű folyamatok kezelésére van kitalálva).

Megemlítem azt is, hogy a forráskód átnézése során - ha valaki rászánja erre magát - simán találhat bizonyos „nem dokumentált funkciókat”, azaz olyan ficsőröket, például utasításokat, amikről egy árva szó se esik e leírásban. Ez nem azért van mintha valamit el akarnék titkolni, annyira nem, hogy ezek jó részéről részletes ismertető található a már említett másik doksiban, ami e nyelv megalkotásának történetét írja le. Viszont ezek olyasmik, amik egyáltalán nem biztos hogy benne maradnak a programnyelvem későbbi változataiban, vagy ha megmaradnak is de nem biztos hogy pontosan ugyanezen szintaxis szerint, stb. Szóval ezek csak afféle erősen kísérleti jellegű, fejlesztés alatt álló funkciók, vagy éppen „atavisztikus örökségek a nyelv digitális törzsfjlődésének korai stádiumaiból”, s emiatt nem is tekinthetők a „hivatalos”, „szabványos” mau programnyelv részeinek egyelőre (hogy ilyen fennköltten vagy akár beképzelten fejezzem ki magamat).

**KÜLÖN KIEMELEM MINT LÉTFONTOSSÁGÚ FIGYELMEZTETÉST**, hogy a nem dokumentált funkciók közül a legkiváltképpen a

m@u

és a

M@U

parancsokat ne használjuk semmiképp sem! Ezek nincsenek még készen éles felhasználásra! Hogy ezek mire valók, nem írom itt le, mert ugye akkor máris dokumentálva lennének, márpedig ezek ugyebár NEM dokumentált funkciók... Amennyiben te mégis ezekkel óhajtasz szórakozni a SAJÁT FELELŐSSÉGEDRE, akkor az a legminimálisabb előzetesen kötelező óvintézkedés, hogy ezeket nem a saját produktív rendszeredben teszteled, hanem:

**Fizikai követelmények:** Külön gépet veszel e célra, ami minimum Core i7 processzorral működik, és 16 giga RAM van benne, és a merevlemeze minimum 2 terabájtos, sőt e merevlemezből többel is rendelkezik, legkevesebb 8 példánnyal, RAID10-ba kötve. E gép nem szabad hogy bármiféle hálózatra is rá legyen kapcsolva. A processzora a túlmelegedés elkerülése érdekében legyen folyékony hélium hűtéssel ellátva. Ezenfelül fontos hogy a gép túlfeszültség elleni védelemmel is rendelkezze! Külön gondoskodjunk róla, hogy a számítógépet a fali konnektorhoz kapcsoló vezeték, ami ugye a tápkábel, az lehetőleg nyílegyenesen legyen lefektetve, de ha mégis kanyarog, akkor se legyen törött, szegletesen meghajlott, hanem csak lágyan kanyarogjon, finom ívekben, különben a parancs lefutása során biztos csak csupa nullabájtot kapsz eredményül, mert e nagyfeszültségű vezetéken haladó bitek közül az 1-es bitek nem tudnak eljutni a céljukig, hiszen sz(e/ö)gletes alakjuknál fogva fennakadnak a vezeték éles kanyarjaiban! Emellett ez nemcsak hibás eredményeket produkálna neked, de ha sokáig bekapcsolva hagyod így, a feltorlódó rengeteg 1-es bit előbb-utóbb a felfokozódó informatikai nyomás miatt szétpukkasztja neked a vezetéket, s a robbanás során minden a képedbe repül, amitől biteket látsz majd!

**Szoftverkövetelmények:** e gépen elindítasz egy Linux disztrót, ami kizárólag vagy Gentoo lehet, vagy LFS. (utóbbi az ajánlott). Erre felraksz valamely virtualizációs szoftvert, s azon belül elindítasz egy Windowst. A Windowson belül is elindítasz egy virtuális gépet, s azon belül kell fusson az a Linux disztró, amire a mau interpretet lefordítod. Itt kreálsz magadnak helyett egy álfelhasználót, mondjuk "testuser" néven, akinek kb semmihez sincs joga, s még így is readonly jogokat adsz csak a mau interpreter indítása előtt az összes könyvtárára és fájljára. Azt a partíciót is amin e testuser mappái vannak, mountoláskor readonly-ként csatolod fel a rendszerbe, természetesen!

Így sem ajánlatos kiadni e parancsokat, de enélkül kész katasztrófa. S természetesen a parancs begépelése után az Enter gombot bal kézzel kell lenyomnod, miközben jobb kezed mutatójával a gép kikapcsológombján nyugszik készenlétben, mert biztos ami biztos...

Ami a nyelv jogi helyzetét illeti: a mau programnyelv maga GPL licenc alatt van, méghozzá minden verziója pontosan azon verziójú GPL licenc alatt, ami az adott verziójú, azaz épp megjelent mau programnyelv megjelenésének pillanatában a legújabb GPL licenc. Ez azért van így, mert bevallom hogy síkhülye vagyok az ilyen jogi furmányosságokhoz, ellenben megbízom a GPL megalkotóiban, s olyan primitíven gondolkodom, hogy „az újabb GPL licenc biztos jobb valamiért mint a régi, hiszen azért találták ki”. Azaz a mau nyelv jelenlegi verziójának kiadására a jelenleg elérhető legfrissebb GPL licenc vonatkozik.

A doksik pedig amik a mau nyelvről szólnak, Creative Commons licenc alatt vannak, annak azon változata alatt, mely megengedi ezen írományok fordítását, sokszorosítását, sőt tőlem akár eladni is lehet őket, anélkül, hogy nekem emiatt jutalékot fizetnének. Nem szabad azonban őket MEGVÁLTOZTATNI. Az természetesen megintcsak engedélyezett, hogy idézzenek belőle, amúgy „normális határok közt”. És természetesen köteles minden leírásterjesztő meghagyni a doksiban a nevemet és email-címemet. (már amiatt se törölheti ki őket, mert ugye a leírás nem megváltoztatható...)

Kiemelem azonban, hogy a fenti meghatározások a GPL és a Creative Commons licencekkel kapcsolatban csak magára a forráskódra és a doksikra vonatkoznak, magára a „**mau**” NÉVRE már NEM! Ez azt jelenti, hogy bárki nyugodtan „forkolhatja” a programnyelvet, csinálhat belőle mindenféle változatokat a maga tetszése szerint, de ezeket csak akkor nevezheti „**mau**” programnyelvnek, ha én magam ehhez kifejezetten hozzájárulok! Elvem ugyanis az, hogy az a mau programnyelv, amit ÉN annak nevezek! Ez nekem azért ennyire fontos, mert mint fentebb írtam, e programnyelvet kifejezetten abból a célból hoztam létre leginkább, hogy a regényeimben majd szerepeltethessem, s emiatt nem tartanám jó ötletnek hogy mindenféle programnyelv-variánsok keringjenek a világban, mind mau néven, s így végül senki se tudhatná, melyik az „igazi” mau nyelv, amin a regényeimben netán szereplő programok istenbizony futtathatóak, s működni fognak! Kérek tehát mindenkit, hogy már becsületből és tisztességből se használja a mau nevet olyan variánsok azonosítására, amiket én nem hagyok jóvá. A kódot amit közreadtam nyugodtan, akár eredeti formájában akár módosítva, de maunak csak akkor nevezze ha ezt előbb egyeztetette velem! Szerintem ez igazán nem nagy kérdés. Mindazonáltal, aki valamely módosított verziót készít belőle, ha nem is nevezi maunak, de illendőnek tartom hogy valahol megemlítsse, hogy ennek alapja, „őse” az én mau nyelvem volt. Ezt a tényt nem muszáj agyonreklamozni, nem muszáj engem vagy ezen eredeti mau nyelvet az egekbe dicsérni, de valahol azért már mégiscsak legyen rá egy utalás... Köszönöm előre is!

Kiemelem továbbá, hogy a dokumentáció más nyelvekre lefordításához csak olyan formában járulok hozzá, hogy amennyiben a fordítás olyan nyelvre történik, mely nyelvben a személyes névmások nem-függőek (ilyen például az angol, német, orosz...), abban az esetben a mau interpreterre a fordításban a személyes névmás nőnemű alakjával utaljanak (azaz angolul például nem az "it" vagy "he", hanem a "she" névmással illessék őt!). Én ugyanis határozottan nőneműként gondolok rá. Ez logikus is, hiszen a mau jelképe egy macska, márpedig nem tudok olyan emberről, aki a macskákról ne úgy gondolkodna, hogy azok valamilyen értelemben „nőies” állatok. Ha ezt netán furcsállja mégis valaki, tekintse úgy, hogy nekem ez a mániám, amihez jogom van, mert miért is ne lenne, ha egyszer az angolok meg a hajókról vélekednek úgy, hogy ami "ship", az szerintük nőies! Szerintem meg a macskák nőiesek és kész.

Ehelyütt adok közre egy afféle FAQ-t vagy GyIK-ot, azaz előre válaszolok azon kérdésekre és kritikákra, amik szerintem úgyis felvetődnek majd. Már amiatt is, mert elég sok az ellenségem, igen, sajnos főleg épp Linuxos berkekben... Bevallom nyíltan, van olyan elég híres linuxos portál is ahonnan bannoltak, más helyről meg magam kértem hogy töröljenek. Tehát:

—Te hülye seggfej, hogy lehetsz olyan barom hogy a programkódba a megjegyzéseket/kommenteket nem angolul írod hanem magyarul! Sőt, még a HIBA-ÜZENETEKET IS!

—Ennek az az oka, hogy egyszerűen nem tudok annyira angolul, amennyire ehhez szükséges volna. Bár vélhetőleg össze tudnék hozni valami olyasmit amit egy angol megértene, de annyira helytelen lenne nyelvtanilag, hogy jobban kiröhögtetném magamat vele, mintha egyszerűen felvállalom, hogy az angol tudásom

ilyen téren a béka segge alatt van. Azonban még mindig jobb ha ott a magyar komment, mint ha nincs ott semmiféle komment.

Továbbá, e programnyelv irodalmi céllal született, a regényeimhez. A regényeim magyar nyelvűek. Vélhető hogy emiatt magyarok olvassák majd őket amúgy is. Ezen illetőknek, ha belenéznek netán a forráskódba érdekességképpen, kifejezetten jó ha a kommentek is magyarul vannak, mert ismerjük el, azért akik magyar nyelven regényeket olvasnak, azoknak mégiscsak csupán kisebbik hányada rendelkezik komoly angol nyelvismerettel, pláne olyannal, hogy azon a nyelven olvasva az neki szórakozás legyen és ne agyszikkasztó munka... Azaz, még csak nem is tartom okvetlenül „bugnak”, hogy a kommentjeim magyar nyelvűek.

#### —A forráskódod kinézete ocsmány.

—Ez ízlés kérdése. Tény hogy nem követtem semmiféle szigorú, előre kodifikált esztétikai elvet a programozás közben. Nekem így jó. Programot írtam, nem képzőművészeti alkotásnak szántam a kódot. Legfontosabb elvem az volt hogy ami rutinnál ez csak lehetséges, beleférjen egy képernyőnyi méretbe, mert akkor jól áttekinthető. Többnyire - bár nem mindig - e célokat el is értem. Ehhez képest minden más a számomra huszonhatodrangú jelentéktelen részletkérdés volt. Amúgy pedig az olyasféle fogalmak hogy „kinézet”, „elegancia”, az szerintem érdekelje a divatdiktátorokat és az úriszabókat. Akinek ez nem tetszik, tördelje át a kódot a neki tetsző formába. SZABAD. Azért GPL licencű...

#### —Használsz te valami verziókezelő rendszert?!

—Ezzel a kérdéssel szapultak eleinte a trollok állandóan, amikor először vettem fel az ötletet egy fórumon, hogy írnék egy programnyelvet. A válaszom az, hogy NEM, nem használok. Annak akkor van jelentősége ha többen is beleugatnak egy projectbe, de mint írtam ez eddig egyemberes fejlesztés volt. És nem kell atom-bombával lőni verébre. Nálam a verziókezelés abból állt, hogy amikor nagyobb „műtétekre” készültem a kódban, készítettem egy külön könyvtárat valami sorszámozott névvel, hogy mondjuk „mau84”, s ebbe átmásoltam mindent ami hozzá tartozik. Így ha elcsesztem valamit (amire különben alig volt példa, ezt büszkén vallom...) akkor ott volt kéznél a biztonsági másolat.

#### —Miféle fejlesztőeszközöket használtál?

—Ha furcsa is lesz a profiknak, a válaszom az, hogy egyáltalán semmit, kivéve az **mc** fájlkezelőt és ennek **mcedit** nevű szövegszerkesztőjét... Eredetileg a VIM szövegszerkesztőt akartam használni, amit tudok kezelni, könyvet is írtam róla (letölthető ingyen a Magyar Elektronikus Könyvtárból, e linkről:

<http://mek.oszk.hu/09600/09648/#>

) de igen hamar rájöttem, hogy még erre sincs okvetlenül szükségem. Na és hát a dolog úgy áll, hogy bár szeretem a VIM-et, de mert eleve szándékomban állt szintaxiskiemelő fájl is készíteni a nyelvhez, s ezt a nyelvemmel párhuzamosan fejlesztettem s előre is tudtam hogy azzal párhuzamosan fogom fejleszteni, továbbá tudom jól azt is hogy a legtöbben nem szeretik a VIM-et, így ezt az MC-hez akartam elkészíteni először, s így mert a fejlesztéshez az mc is elég volt nekem, maradtam amellet. Persze a jövőben várhatóan készítek majd azért szintaxiskiemelő fájl a VIM-hez is. Hacsak valaki meg nem előz ebben.

Azaz, a fejlesztés a lehető legősbibb, „legparasztosabb” módon, „nyers fapadossággal” történt: megírtam a kódot, majd a parancssorban kiadtam a „make” parancsot, s erre az megcsinálta nekem a binárist és kész. Nem használtam

semmiféle/mindenféle modern szírszart ami helyettem előállítja a kódot, nem rángattam ezért ikonokat a képernyőn, meg hasonlók. Nem is értek egyetlen efféle szoftverhez sem, de nem is vagyok hajlandó megtanulni ilyesmit mert nem bízom bennük.

—Komolyan úgy érted, hogy még egy francos debuggert se használtál?!

—Komolyan. Ha nagyon muszáj volt, egyszerűen beírtam valami sort a kód megfelelőnek tartott helyébe, ami kinyom nekem egy log üzenetet az stdout-ra, mondjuk hogy „Eddig oké!”, aztán kész. Amikor meg már nem kellett, természetesen töröltem e sort. Vagy legalábbis kikommenteztem... Ennyi nekem elég volt, különben meg elárulom, erre se gyakran volt szükségem. Igazából egyetlen hiba volt csak ami halálra „szopatott”, na azzal elment egy teljes napom amíg megtaláltam, ez igaz. De szerintem debuggerrel is ugyanannyi időbe tellett volna, sőt amíg egy debuggert megtanulok kezelni, eleve több időbe telik. Voltak más esetek is amikor nyomozgatnom kellett, de nem több mint talán fél tucat, s azokkal végeztem is egyenként mind kb negyed óra alatt.

—Miféle segítséget vársz a közösségtől?

—VÁRNI, na azt semmit. Sőt, teljesen pesszimista vagyok ezt illetően, de annyira, hogy el vagyok készülve rá, hogy kapok majd az égvilágon mindent, épp csak elismerést és segítséget nem, hanem annál több szapulást, gúnyt, meg minden más negatív dolgot ami csak képzelhető. Rém rossz tapasztalataim (s emiatt rossz véleményem) van(nak) a magyar linuxos közösségről. Igazából szerintem inkább örülni kéne annak, hogy van egy magyar fejlesztésű programnyelv is, mert nem tudok róla hogy létezne ilyen, még a viccből fejlesztett nyelvek közt se tudok ilyesmiről, de ha akad mégis, biztos nem olyan komolyan kidogozott mint ez, szóval ez akár mint „hungarikum” is tekinthető lenne. De inkább azt hiszem, mindenki majd azt akarja bizonygatni, ez miért szar, és nem jó, és miért kellett volna egészen máshogy megcsinálni, mert a kód nem követ ilyen meg olyan szabályokat, a nyelv szintaktikája nem ilyen kéne legyen hanem amolyan, meg stb. EZ LESZ.

Szerk.: A mau nyelv legelső, 0 verziószámú kiadása után pontosan ez is lett amit fentebb megjósoltam... Hogy én mennyire utálok, amikor ennyire igazam van!

Sokan megmondták már e népről, hogy itt e tejjel-mézzel (állítólag...) folyó Kánaánban az úgy megy hogy ha valaki kitalál valamit, azonnal elkezdik szapulni e fokozatokon át:

1. Lehetetlen.
2. Lehetséges volna de hülység az egész, semmi értelme, ilyesmi is csak egy hozzád hasonló ...(nem dicsérő jelzők halmaza)...-nak juthat az eszébe, tiszta égő hogy ezt javaslod, nem értesz a témához, beképzelt vagy hogy azt hiszed jobban tudod mint a Nagyok, hogy is merészeled ezt, nincs benned szerénység, és egyáltalán, dugulj már el!
3. Volna értelme a dolognak, de tulajdonképpen mégsem éri meg.
4. Megérné megcsinálni, de túl macerás volna, ezt csak más országokban lehetséges, mi ehhez szegények vagyunk, nincs forrásunk rá, meg időnk se, meg ... se.
5. Meg tudnánk csinálni mi is, de ez akkor is csak más országokban válna be igazán, a mi speciális magyar valóságunkban valamiért mégse, nekünk valamiféle „külön utakat” kell járni. Miért? Mert mi magyarok vagyunk ugyebár.

6. Reméljük valaki majd megcsinálja nekünk. Megérdemelnénk hogy végre valakinek megessen rajtunk a szíve, hiszen „megbűnhődte már e nép a múltat s jövőt” ugyebár. Az ötlet tényleg jó, most már csak kéne valaki elszánt, önzetlen, elhivatott személy, aki helyettünk, a maga erejéből...
7. MICSODA?! Hogy ezt épp TE akarod?! Na eredj a búsba, NEKED ez biztos nem fog soha se sikerülni!
8. Elkészült? Hm, nézzük csak, valamiért ez tuti hogy nem jó, hiszen egyrészt és leginkább mert épp **te** csináltad, másrészt különben is, ez nem nyugati termék. Magyar embernek a magyar termék nem lehet jó, mert az nem elég divatos, nem trendi, nem geek, nem lehet vele villogni. Hm... Tényleg! Persze hogy nem jó. Ugyanis... - itt jön egy csomó érv amik még akár jogosak is lehetnek hiszen vado-natúj dologról van szó, s a prototípusok mindig teli vannak kis tökéletlenségekkel - valamint azért se jó, mert... - itt jön egy rakás légből kapott indok, amik belemagyarázások és semmi közülük a termékhez, vagy ahhoz amiért az létrejött.
9. Micsoda?! Társakat keresel a sorozatgyártáshoz?! Meg a továbbfejlesztéshez?! Megvesztél?! Hogy mi ebben segítsünk NEKED?! Hogy neked HASZNOD legyen ebből?! Hiszen ez még erkölcsi elismerést se érdemel!
10. MICSODA????!!! Kiment külföldre?! És ott eladta?! A MAGYAR találmányt?! Az IDEGENEKNEK?! Hiszen ez HAZAÁRULÓ! Különben is, joga se volt hozzá, a ter-méke biztos hogy egy csomó olyasmit tartalmaz amit idehaza lesett el, azaz magyar találmány, ő semmi újat nem talált ki, és most a becsületes, itthon ma-radt magyarok nyakán élősködik, azokból nyeréskedik! Különben is, én magam is mondtam rég, hogy pontosan ezt kéne csinálni, erre a feladatra megoldás kell és az a megoldás efféleképp kéne hogy megalkottassék! Rengetegszer mondtam! Igaz akiknek mondtam már nem élnek, meg le se írtam, de ez akkor is az én ötletem, csak ellopták a... (itt jön egy csomó konteós elmélet, ultranacionalista hangokkal megfűszerezve).
11. A Magyar Kormány finanszírozási szerződést kötött az EU-val, XXXX millió euró kölcsönt vett fel a Világbanktól, azért, hogy bevezessék és elterjesszék orszá-gunkban is a ZZZZ terméket amit a ZZZ-UnitedCompany inc. New York-i konszern gyárt. Ez amiatt is fontos, mert e termék kidolgozása magyar vonatkozásokkal bír, amennyiben az alapötletét az egyik legnagyobb magyar zseni, XY találta ki, ő alkotta meg a prototípust is.
12. Igenis XY az egy MAGYAR volt, akkor is ha kivándorolt az USA-ba, már 20 évesen, s 25 éves korától USA-i állampolgárként élt, le is mondott a magyar állampolgárságáról, haza se látogatott többet, mert mint mondta elüldözték őt a honfitársai, és nem becsülték meg a zsenijét! Ő akkor is MAGYAR! És mi büszkék vagyunk rá, hogy a népünk és ez a föld ilyen zseniket szül! Nekünk ő a példa-képünk!
13. Hogy miért nem kellett a találmánya az akkori magyaroknak?! Te buta, ezt se tudod?! Hát mert akkor olyanok voltak a KÖRÜLMÉNYEK, amik nem tőlünk függnének sosem. Mi mindig megbecsüljük a lángeszeket, a tehetségeket, meg min-denkit, mi egy nagyon barátságos, segítőkész és befogadó nép vagyunk, csak minket mindig ELNYOMNAK, mások, a Világbank, a mindenféle más nagyobb népek, az EU, az USA, és agyelszívás van, meg környezetszennyezés, meg min-denféle idegenszívűek kollaborálnak az elnyomóinkkal, akik persze csak hamis magyarok, és... Szóval, a lényeg hogy mi sosem vagyunk hibásak abban, hogy nekünk rosszul megy. Mindig hibás mindenki más, de mi természetesen sosem.



Szóval én úgy vagyok ezzel hogy fel vagyok készülve nem ám a segítségre, hanem arra hogy jönnek majd rámozdulva az irigyek és a rosszakarók, mint dögre a legyek. Hogy is ne épp ez következne be, tapasztaltam hogy már amiatt is kiutáltak mindenhol, mert egy extravagáns és ritkán használt disztribúciót szeretek, a GoboLinuxot, aztán azért is kiutáltak mert hogy merek én könyvet írni a VIM-ről, meg mert én csakis egy agyilag zanza barom lehetek amiért általában véve nem kedvelem a grafikus felületű programokat hanem a parancssor a kedvencem, vagy maximum az ncurses felületű programok, aztán meg amiatt lettem gúny tárgya, hogy végigcsináltam az LFS-t, de azt is egyénien, hogy a GoboLinuxhoz hasonló, de azzal se teljesen azonos fájlrendszer-hierarchiát alakítottam ki saját szkriptekkel, most is azt a „disztrót” használom... Mi lesz most hogy még programnyelvet is írtam! Ha tudnák a lakcímemet, hétszentség hogy még interkontinentális rakétákat is kilőnének a fejemre!

Ha netán mégis akadna valaki aki szívesen segítene (bár ebben abszolút nem hiszek), annak azért leírom, mi az amit szívesen vennék:

1. Mindenekelőtt a doksik lefordítását más nyelvre, elsősorban természetesen angolra, beleértve a forráskód kommentjeit és a hibaüzeneteket is.
2. Bugreportot. Lehetőleg azonban ne olyan stílusban, hogy „te hülye faszi, ez se működik jól”...
3. Tippeket további ficsőrökre. Esetleg akár komplett rutinokkal együtt, azaz az ötletet rögtön meg is valósítva (bár ez természetesen nem követelmény).
4. A már elkészült rutinok némelyikének hatékonyabb (például gyorsabb) változatának megvalósítását, azaz refaktorálását.
5. Mau nyelvű programok írását és közzétételét mindenféle feladatokra, azaz a nyelvem HASZNÁLATÁT. Nyilvánvaló ugyanis, hogy ez a legjobb reklám neki. Továbbá, ez a legjobb módszer arra is, hogy kiderüljenek az esetleges hibái, meg az is, mik azon funkciók amik nagyon kellenének bele, de még nincsenek implementálva.
6. Mindenféle más módokon is a nyelvem népszerűsítését.
7. A mau nyelv portolása más architektúrákra, pld Windows alá, OSX alá, meg a satöbbi...
8. Plugin/extension készítése Firefoxhoz, hogy a böngészőben is lehessen mau programokat futtatni, mint ahogy azt most a javascriptekkel is meg lehet tenni. Ugyanez LibreOffice alá is valami módon.
9. Szintaxiskiemelő fájl készítése VIM-hez, hogy ne nekem kelljen ezzel időt tölteni.
10. Bencsmarkok készítése, azaz a nyelv összehasonlítása mindenféle szempontok szerint, de különösen sebességtesztben más nyelvekkel, különböző feladattípusokban. Mind compiler, mind interpreter típusú nyelvekkel. Különösen a következő nyelvekkel való összevetés érdekel:
  - bash (mert az van mindenütt)
  - C (mert az a kedvencem)
  - Perl (mert annak a szintaxisa hasonlít leginkább a mauéra, amennyiben első pillantásra ugyanolyan érthetetlen)
  - Python (kizárólag amiatt mert azt utálok, s jólesne ha valamiben lekörözhetné őt a mau nyelv)
  - PHP (mert ez se a kedvencem, bár annyira nem berzenkedem tőle mint a Pythontól)
  - Javascript (mert manapság mindenki attól van elájulva hogy az a hűdeszuper és az isten, mindenre azt használják, bár fogalmam sincs miért...)

—Nem igaz hogy ne lenne magyar ember által megalkotott programnyelv, a Basic is az, a <http://hu.wikipedia.org/wiki/BASIC>-ről illik tudni hogy magyar fejlesztés, és elég jól ki van dolgozva, meg elterjedt is (volt), irtó ciki hogy erről sincs fogalmad!

Erre a válaszom a következő:

1. A wikis leírás szerint is az a magyar illető nem egymagában fejlesztette ki hanem egy angol illetővel (legalábbis a neve alapján minden bizonnyal nem magyarral) közösen, azaz erre azt mondani hogy magyar fejlesztés, túlzás (szerintem). 50%-ig magyar, legfeljebb.
2. Nem is Magyarországon történt a fejlesztés hanem egy külföldi egyetemen. Na most ha még csak külföldön történt volna hagyján, de az hogy abban a Dartmouth College-ben, az felveti bennem a gyanút, hogy ugyan ki mindenki foglalkozott még vele akkoriban...
3. Kemény János matematikus 1940-ben vándorolt ki szüleivel az USA-ba, vagyis a BASIC megalkotásakor nem volt magyar állampolgár.

—A helyesírásod rossz. Szégyen egy írótól!

—Szerintem a helyesírásom kiváló, az átlagemberekhez képest, de pláne a számítástechnikai szakmát űzőkkel összevetve... Akik még azt a pofátlanságot is elkövetik, hogy ekezet nélküli betűkkel irnak forumokra meg sok mas helyre is... Holott épp egy programozó vagy egy rendszergazda igazán illene hogy be tudja állítani magának a magyar billentyűzetkiosztást! Akkor is, ha a klaviatúrája épp angol. Elárulom, az enyém is az, aztán mégse gond egyetlen ékezetes betű sem... Pedig épp a számítástechnikában az ékezetnélküliségből rém nagy zűrök támadhatnak — állítólag megtörtént, hogy egy főkönyvelő belső vizsgálatot rendelt el, mert mit is keres egy villanyszereléssel foglalkozó cég raktárában 2000 méter fókabél?! Aztán persze kiderült, hogy a „fokabel” nem fókabelet jelent, hanem fókábelt...

Abban az esetben ha valamely kötekedő nyelvtannáci a saját portáján óhajtana söprögetni, de nem tudja miként állítsa be magának az angol vagy akármilyen billentyűzeten a neki tetsző billentyűzetkiosztást, amin lehetnek nemcsak a magyar karakterek, de héber, görög, orosz vagy akármi más karakterek is, annak jószívvel felajánlom „népgazdasági hasznosításra” az e témakörben írt cikkemet, konkrét példákkal, mindennel:

[http://parancssor.info/dokuwiki/doku.php?id=sajat\\_billentyuzetkiosztas\\_ujra](http://parancssor.info/dokuwiki/doku.php?id=sajat_billentyuzetkiosztas_ujra)

Továbbá, ellenségeim állandóan ezzel a primitív érveléssel jönnek, hogy a helyesírásom. Holott alig tudnak felmutatni olyasmit, ami hiba volna. Szinte kizárólag olyan dolgokat pécéznak ki (mert más nincs) mint az egybeírás-különírás, meg a vesszőhasználatom, valamint hogy az „emdash” nagyköötőjelet merem gondolatjelként használni, holott azt szerintük csak az angoloknak szabad, s ők sem ebben a funkcióban használják. Nos amiket ezekkel kapcsolatban mondanak, elárulom, azt TUDOM magamtól is. Egyszerűen LESZAROM e nézeteiket, mert nekem más a véleményem, teccikérteni?! Messze jobban ismerem a helyesírási szabályzatot mint egyesek hiszik, csak sok dologgal ami ott van egyrészt nem értek egyet, másrészt az a véleményem hogy e kérdéskör amúgy is túlszabályozott. Szóval húzzon el minden nyelvtannáci a búsba mert ballisztikus ívben tojok a véleményükre!

Két konkrét példát is megemlítek, mi az amivel nem értek egyet a helyesírásban. Ott van mondjuk a „mammut” szó. Én ezt így tanultam gyerekkoromban, 2 darab



„m” betűvel. Nekem ezen állat neve mammut. TUDOM, hogy most elvileg mamutnak kéne írni, de NEM ÉRDEKEL. Nem fogom újratanulni a helyesírást X évenként, mert valami seggfej holmi nyelvészeti bizottságban kitalálta hogy mostantól Y darab szót másképp kell írni valami elvont elv miatt! Ha nem a mau programnyelvről írnék doksit hanem valami őslénytani kérdésről, direkt és szánt-szándékkal és csak azért is mammutnak írnám ezen állat nevét! Nem dachból, még csak nem is lázadásképpen, hanem egyszerűen mert ENGEM MAGAMAT zavarna és idegesítene hogy úgy látom viszont e szót leírva a szövegemben, olyan írásképpel, amit én magam igenis HELYTELENNEK érzek! Berzenkedne ellene a szépérzésem, sőt a nyelvérzésem is.

Vagy ott van például a "végülis" szó. Őszinténszólva, ez esetben bevallom, nem tudom, ezt a gyerekkoromban hogy kellett volna „helyesen” írni. Most biztos külön kell, így: „végül is”. Azért biztos, mert a szövegszerkesztőm állandóan azzal idegesített hogy amikor egybeírtam, ő kijavította különírt formára. Nos, PRÓBÁLTAM megszokni a különírt változatot, istenbizonny! De nem ment. Végül megparancsoltam a szövegszerkesztő proginak, hogy ezt a mániáját felejtse el, nekem ez egybeírva igenis jó. Azért jó, mert NEKEM igenis azt súgja a NYELVÉRZÉKEM, hogy ez EGY szó, egy fogalom, nem 2 szó és nem 2 fogalom. Ezzel az erővel ugyanis akár az is lehetne „helyes” íráskép a „Helyesírási Szabályzat” szerint, hogy:

"csak nem" a "csaknem" helyett,

"igen is" az "igenis" helyett,

"még is" a "mégis" helyett...

Meg más effélék is még. (Sőt, ki tudja, nem úgy van-e, hogy e gúnyból írt példák némelyike talán valóban a "szabványos"-nak minősül már manapság, vagy legalábbis (vagy "legalább is"?) ezen íráskép bevezetését fontolgatják némelyek egy új Szabályzat bevezetésének ürügyén...)

És RENGETEG ilyesmi van a magyar nyelvben, ahol nekem igenis mást sugall a nyelvérzésem, mint ami úgymond „helyes” lenne a helyesírás-náci nyelv-inkvizítorok szerint. És én vagyok annyira öntelt hogy úgy véljem, az én nyelvérzésem is ér annyit, mint az övék, és punctum!

### —Nem szakmai a stílusod!

—DE, nagyonis az! Épp csak nekem nem a számítástechnika a szakmám, hanem az írás. Egy írónál meg épp az a követelmény hogy a stílusa élvezetes, humoros legyen, és ne száraz, unalmas. Amúgy meg nem is hiszek abban, hogy egy dokumentációnak okvetlenül arra kell törekednie, hogy minél rövidebb legyen, s annyira élvezetes, mint egy 7 éves kisgyereknek ha a másodfokú egyenlet megoldóképletének levezetésével találkozik egy még marhára nem neki való matematika-könyvben... Akinek ez nem tetszik, ne olvassa, az ő dolga! Vagy írjon másik dokumentációt a mau nyelvről ami szerinte „szakmaibb”. Nem tol ki velem ha ez utóbbit teszi, mert még örülni is fogok neki...

### —Miért épp a „mau” nevet adtad neki?

—Azért, mert korábban alkottam egy emberi kommunikációra szánt/alkalmas mesterséges nyelvet (olyasmit tehát mint az eszperantó), azt is a regényeimhez, és annak is mau a neve. Na most a sorozatomban azon mau nyelv, s ez a programnyelv szoros kapcsolatban lesz egymással mindenféle okokból, főleg mert a sorozatomban létezik egy „mau” nevű nép is, mely természetesen az előbb említett mau nyelvet beszéli, (és a Mauia bolygón lakik...), és túlnyomórészt ezek nevéhez fűződik majd a regényeimben e programnyelv megalkotása. Logikus tehát, hogy a

programnyelv neve is mau legyen. Azt utólag tudtam meg, hogy létezik egy egyiptomi macskafajta is mau néven, de ha így esett, sebaj, jó lesz a nyelvem jelképének, kabalaállatnak, pláne mert teljesen véletlenül úgy alakult már korábban a sorozatom, hogy e mau nép kifejezetten kedveli épp a macskaféléket. Azaz ez szerencsés véletlen nekem e szempontból.

—Mi ez a baromság hogy a leírást ODT doksiban teszed közzé, miért nem a wikidbe tetted?

—Természetesen igazad van, ennek a wikiben volna a legjobb helye. De először: jól jöhet ez az offline elérhetőség is annak aki meg akarja tanulni. Másrészt, egyszerűen nincs időm rá, meg unalmasnak is találom az efféle szöszmötölést. Lehet hogy majd valamikor megcsinálom, de nem ígérem, és ha mégis, az sokára lesz. Esetleg valaki akinek tetszik a mau nyelv, ezt bevállalhatná helyettem...

—Mik azok, amikben semmiféleképp sem igényled a közösség segítségét?

—Semmi szükségem például olyasmire, hogy azt vagdalják a fejemhez, hogy a mau nyelv szintaxisa nem olyan közérthető első pillantásra, mint mondjuk a Pascal vagy akármi más nyelv. TUDOM. Teljesen tisztában vagyok vele! Ez azonban engem baromira hidegen hagy. Megvan az oka hogy miért ilyennek alakult ki, ott van leírva a leírásban ami a megalkotásáról szól. Tessék elolvasni. Másrészt, inkább még örülök is neki hogy ilyen, mert nem először említem hogy IRODALMI céllal jött létre elsősorban, s ha egy regénybe beleteszek valami mau nyelvű programrészletet, akkor oda kifejezetten HASZNOS és ELŐNYÖS ha a kód kissé titokzatosan néz ki, mert fokozza a regény sci-fis hangulatát! Sietek megjegyezni, ez NEM azt jelenti, hogy DIREKT írtam volna nehezen érthetővé e nyelvet. NEM. Sőt ellenkezőleg: tölem tellően igenis törekedtem a megalkotása során az egyszerűsége és érthetősége, épp csak nem ez volt a legfontosabb szempont. De ha egyszer ilyen lett, hát akkor ilyen, és ezt egyáltalán nem bánom. Tulajdonképpen magam is örülnék neki ha kissé olvasmányosabb lenne, csak arra akarok kilyukadni, hogy ennek hogy ilyen, ennek is megvan a maga előnye, emiatt tehát nem óhajtok erőfeszítéseket tenni a „felhasználóbarátabbá tételére”. Azt is megjegyzem, számomra nem tűnik különösebben nehézkesnek. Tanulni kell ez tény, de a VIM-et is tanulni kell, sokan mégis rajongnak utána, és nagyonis hatékony!

—Nem óhajtok tehát vitát folytatni a jelenleg megvalósított funkciók szintaxisáról. Kész vagyok ilyesmire olyan funkciókat illetően, amik MÉG nincsenek implementálva, azaz jövőbeli fejlesztések lehetnének, de ami MÁR készen van, az nem vita tárgya a szememben. (Mármint az nem ami EBBEN a doksiban van leírva. Azok amiket „nem dokumentált funkcióknak” nevezek, amik például a másik, a megalkotásról szóló doksiban benne vannak de e doksiban nem, azok nem állnak efféle „védelem” alatt, azokról lehet vitatkozni).

—Nem érdekelnek a kritikák arra vonatkozóan, hogy kinek mennyire tetszik a stringek névsorbarendezését megvalósító rutinom sorrendi szabálya. **NEKEM** ÍGY TETSZIK. Akinek nem, az írjon saját rendezőrutint magának. SZABAD.

—Nem érdekel a kritika arra vonatkozóan sem, miért az UTF-8 a nyelvem szabványa. Azért az, mert ez a jövő szerintem és kész, pontosabban, a nemzeti kódlapoknak nincs jövője, emiatt kell egy nemzetközi, és nullánál is kevesebb esélyét látom annak, hogy e nemzetközi ne az UTF-8 legyen. Igazából - elárulom - UTÁLOM... Én egészen másképp oldottam volna meg e kérdést. De ez van, és széllel szembe pisilni lehet, csak nem érdemes... Ha viszont az UTF-8 DE FACTO

a szabvány a karakterkódolásra nemzetközi téren, akkor nyugodtan felhasználhatom annak bármely karakterét máris. Nem izgat, hogy ezek némelyike nem érhető el default az angol, magyar, amhara, héber, zulu, gudzsarátí vagy akár-melyik másik billentyűzetkiosztáson, mert könnyedén lehet saját billentyűzetkiosztást készíteni Linux alatt amire azt definiálunk amit csak akarunk, erre is van ragyogó tutorial, én írtam, itt a link rá:

[http://parancssor.info/dokuwiki/doku.php?id=sajat\\_billentyuzetkiosztas\\_ujra](http://parancssor.info/dokuwiki/doku.php?id=sajat_billentyuzetkiosztas_ujra)

A Windowst használók sirámai pedig nem hatnak meg, mert e mau nyelv egyelőre úgyis csak Linux alá érhető el, másrészt nyilván meg lehet oldani a saját billentyűzetkiosztást valahogy Windows alatt is. Nem tudom hogy hogyan mert én nem azt használom, de valami mód rá tuti hogy van.

—A legislegkevésbé az a trollkodás érdekel, hogy úgymond felesleges dolgot műveltem e nyelv megalkotásával. Ez ugyanis egyszerűen baromság és hazugság. NYILVÁNVALÓ, hogy nem felesleges amit alkottam, hiszen leírtam, hogy nekem ez KELL a következő regényemhez, sőt feltehetőleg nem is csak 1 regényemhez. Eddig már 58 regényt írtam, és számtalan novellát meg kisregényt ezek mellé, így nem lehet kétséges hogy azon újabb regényt is képes leszek megírni. (Elárulom, a fele máris készen van... pont e programnyelv kedvéért hagytam félbe átmenetileg az írását...) Azaz semmi esetre se felesleges, mert NEKEM kell egy jól meghatározott célra. Továbbá, nagyonis könnyen el tudom képzelni hogy e nyelv másoknak is nagy hasznára lehet, ugyanis amióta csak linuxozom, rengetegszer volt olyan esetem, hogy nagyon kellett volna nekem valami nem erőforrás-zabáló szkriptnyelv, ami azonban tudja ugyanazokat a numerikus típusokat kezelni mint a C nyelv (és a stringeket is), és nem volt ilyen! A Bash szkriptek mint tudjuk tetűlassúak és a bash nem is típusos nyelv. (kérdéses előttem, egyáltalán méltó-e a programnyelv névre...) A Python erőforrásszabáló, és tényleg akkora mint egy óriáskigyó, emellett gyűlölöm és utálok azt az idiótaságát, hogy nem lehet akárhova akármennyi whitespace karaktert raknom. (Hétszentség hogy ezt csak valami teljesen IQ-negatív idióta találhatta ki ilyenné, mély alkoholmámorban, amikor még jól be is kábítószerezett emellé...) Szóval tényleg könnyedén el tudom képzelni, hogy a mau nyelv nagyonis hasznos lehet másoknak is, függetlenül az irodalmi céljaimtól. Nekem máris az a napi linuxos feladataimban.

—Én már a C nyelvet se szeretem, s ez a „mau” még annál is rosszabbnak tűnik.

—Semmi baj, nem te vagy e nyelv célközönsége, ennyi az egész. Használj Basicet...

—Gratulálok, újra feltaláltad a C-64 Basicjét a dolláros hexa jelöléssel...

—Szerintem fuss neki még egyszer a C-64 megismerésének, mert nemhogy a mau nyelvet, de még a C-64-et se ismered. Nyilván mert az is túl nagy szellemi erőfeszítés neked... A C-64 Basicjében a stringváltozók voltak \$ jellel postfixálva. A mau nyelvben a hexadecimális konstansok vannak e jellel prefixálva... Nem tudom, mi lenne e kettőben közös?! Vélhetőleg azzal kevered, hogy arra a gépre létezett pár úgynevezett „monitorprogram”, például a RatMon64 nevű (nekem az volt a kedvencem), s ott tényleg olyasféleképp jelölték a hexa számokat mint a mau nyelvben. Csakhogy az már nem a C-64 Basicje volt, hanem annak az assemblyje, ami óriási különbség...

—A szlogenedben a „krixkrax” szót nem x betűvel kell írni hanem ksz-szel, s különben is kötőjellel!

—Nos, bevallom fogalmam sincs róla, hogyan lenne e szó „szabályos” az ekkori vagy akkori kiadású Helyesírási Szabályzat szerint. (mert ugye, azt is állandóan variálják). Ami engem illet, konkrétan ezt is leszárom. A Google ad találatot e szó mindenféle variációira is:

- krixkrax: 11600 találat
- krix-krax: 8890 találat
- krixkrax: 6150 találat
- kriksz-kraksz: 36500 találat

(Mindegyiket idézőjelek közé téve kerestettem a Google-val).

Ennek fényében, elvileg a „kriksz-kraksz” írásmód lenne a „helyes”, ha azt az elvet követjük hogy az a helyes amit a többség ír, de tudjuk jól, hogy a Helyesírási Szabályzat nem mindig osztja ezt a nemes elvet... Mindenesetre, az általam követett „krixkrax” változat ezen „helyes” (???) alak gyakoriságának eléri majdnem a 32%-át, mondható tehát hogy egyharmada annak, eképp azért szerintem mondható minimum „tájnyelvi változatnak” vagy ilyesminek, azaz helytelennek tekinteni igazán túlzás, pláne mert egy ritka szóról van szó. Meg aztán, látható, hogy mindenképp ez legalábbis a „második leghelyesebb” alak a gyakoriság szerint.

De mondom, különben is tojok rá, mi a „helyes” a tojásfejűek szerint. Azért ezt használok, mert:

1. Az „x” titokzatosabban néz ki, mint a „ksz”, emiatt jobban illik a mau nyelv „szellemiségéhez”.
2. Az x-es változat 2 karakterrel rövidebb, ezért tömörebb, s ez is jobban illik a mau nyelv tömörségéhez.
3. Hülyeségnek tartom hogy ha egyszer van a magyar nyelvben „x” betű, akkor ne használjuk azon néhány esetben sem amikor egy szóban nagy ritkán előfordul a „k+sz” hangkapcsolat. Mi a búbánatos francért van x betűnk, ha nem merjük használni?!
4. A kötőjeles írásmódot is ugyanazért nem használok, amit már írtam: kötőjel nélkül tömörebb, s jobban illik a mau nyelv tömörségéhez.

## 1. fejezet - A mau nyelv változói

Aki egy mau nyelvű programra ránéz, első pillanatban valószínűleg elszörnyed, mert teljesen érthetetlennek tűnik számára a sok krixkrax. Leginkább még talán a Perl nyelvre emlékeztet, de annál is rosszabb. E benyomás túlnyomórészt azonban csak egyetlenegy dologból fakad: abból, hogy a mau nyelv a változókat tényleg meglehetősen egzotikus módon kezeli/nevezi, különben annak érdekében, hogy a kód végrehajtása minél gyorsabban történhessen. Hogy a sebességnek és a változók nevének mi köze van egymáshoz, azt azonban itt nem részletezem, akit érdekel olvassa el a *Bevezetésben* említett tanulmányt mely e nyelv megalkotásának folyamatáról szól. Mindenesetre semmiképp se lehet vitás, hogy a mau nyelv messze a legnagyobb nehézségét a megértésben és elsajátításában a változó-kezelés megtanulása jelenti. Aki ezt „túléli”, annak a többi már könnyű. A mau nyelvnek ugyanis számos előnye van, például nem kell benne mutatókkal bűvészkedni már egy nyamvadt stringkezelés kedvéért is, mint a C nyelvben (mely különben kedvenc programnyelvem, a mau nyelv interpretere is C/C++ nyelven van megírva). A mau nyelv ugyanis számos beépített magas szintű adattípussal rendelkezik, nemcsak stringekkel, de stringtömbökkel, vermekkel, sőt még külön

„directory” adattípusa is van! E nyelv tehát valójában nagyon könnyű, mondhatnám hogy egyenesen KÉNYELMES. Kivéve a változók nevét, igen, sajnos, gomen nasay, sorry és excuse me...

Ami tehát a változókat illeti: Minden mau program (pontosabban egy mau program minden külön NÉVTERE...) pontosan 256 különböző nevű változóval rendelkezhet, semmi esetre sem többel, egyáltalán soha és semmikor, tényleg, kivéve amikor mégis. (hehehe...) E változók azonban nem egy konkrét számértéket jelölnek, hanem az hogy pontosan miféle érték is tartozik hozzájuk, az attól függ, milyen kontextusban hívják meg őket. Például minden rendszerfüggvény meghatározott típusú input paraméter(ek)e)t vár el, ezért a megfelelő típusú adatot olvassa ki a változóból. Természetesen létezik casting operátor is, amivel ezt befolyásolni tudjuk.

A fentiek megértéséhez szükséges észben tartani, hogy tehát egy változó, mondjuk az amit a **@A** karakterpárossal jelölünk (minden változó neve okvetlenül a **@** jellel kezdődik, mindig, kivéve amikor mégsem...) tulajdonképpen egy 16 bájt méretű memóriatartományt címez meg, amely unionként van felosztva. Ezen union struktúrája a következő:

```
union V { // Az univerzális változó

unsigned char    c[16]; // Mérete 1 bájt
signed char     C[16]; // Mérete 1 bájt
unsigned short  int i[ 8]; // Mérete 2 bájt
signed short   int I[ 8]; // Mérete 2 bájt
unsigned int    l[ 4]; // Mérete 4 bájt
signed int     L[ 4]; // Mérete 4 bájt
float          f[ 4]; // Mérete 4 bájt
unsigned long long g[ 2]; // Mérete 8 bájt
signed long long G[ 2]; // Mérete 8 bájt
double        d[ 2]; // Mérete 8 bájt
void *        p[ 2]; // Mérete 8 bájt
long double D; // Mérete 16 bájt

};
```

```
// A V unionban elhelyezkedő mezők egymáshoz viszonyított pozíciói:
```

```
// | c[0] | c[1] | c[2] | c[3] | c[4] | c[5] | c[6] | c[7] | c[8] | c[9] | c[10] | c[11] | c[12] | c[13] | c[14] | c[15]
// | C[0] | C[1] | C[2] | C[3] | C[4] | C[5] | C[6] | C[7] | C[8] | C[9] | C[10] | C[11] | C[12] | C[13] | C[14] | C[15]
// | i[0] | | i[1] | | i[2] | | i[3] | | i[4] | | i[5] | | i[6] | | i[7] |
// | I[0] | | I[1] | | I[2] | | I[3] | | I[4] | | I[5] | | I[6] | | I[7] |
// | | l[0] | | | | | | | | l[2] | | | | | l[3] |
// | | L[0] | | | | | | | | L[2] | | | | | L[3] |
// | | f[0] | | | | | | | | f[2] | | | | | f[3] |
// | | | g[0] | | | | | | | | | g[1] |
// | | | G[0] | | | | | | | | | G[1] |
// | | | d[0] | | | | | | | | | d[1] |
// | | | p[0] | | | | | | | | | p[1] |
```

A fentiekben kékkel jelölt „p” nevű void\* típusú mezőfelosztással nem kell törődnie a mau nyelven programozónak, ezt csak pár rendszerrutin használja „ott lenn a mélyben”, csak a teljesség kedvéért van itt feltüntetve.

Amikor tehát az interpreter egy aritmetikai kifejezés kiértékelése közben belefut mondjuk egy **@A** nevű változóba, akkor az hogy ennek hatására miféle értéket szolgáltat nekünk, attól függ, épp milyen TÍPUSÚ adatot óhajt előcsalogatni magának. Ha az mondjuk unsigned char típusú, akkor az **A** változó unionjának **c[0]** rekeszében található értéket olvassa ki. Ha azonban neki mondjuk egy unsigned int érték kell, akkor **NEM** azt csinálja hogy a **c[0]** rekeszben levő, az előbb említett egybájtos értéket castolja 4 bájtos unsigned int értékké, hanem az **l[0]** rekeszben található mind a 4 bájtot kiolvassa, s ezt adja vissza mint egy unsigned int értéket!

A mau nyelvben kivétel nélkül minden függvény és utasítás esetén a legisleg-szigorúbban meg van határozva, hogy miféle típusú adatot vár a megfelelő helyen, pozícióban. Az **if** után például unsigned char érték kell jöjjön. (A részleteket lásd majd az **if**-ről szóló fejezetben). Ez nem jelenti azt, hogy más típusú adat nem adható meg azon a helyen, amennyiben azonban azon adat alapértelmezett típusa nem az amit azon a helyen az interpreter elvárna, külön ki kell tennünk a „casting operátor” jelét, ami jelzi az interpreternek, hogy most épp miféle adatot is akarunk vele „megkajáltatni”. Azaz, a **@A** egy egész csomó mindenfélét jelenthet, attól függően, hogy épp miféle környezetben áll. Ez egy „nem egyértelmű” neve a változónak, mert csak azt határozza meg, épp melyik 16 bájtos unionról van szó, de azt nem, hogy annak melyik mezőjéről. Ezt a jelölésmódot nevezzük majd e könyvben úgy a továbbiakban, hogy a változó **„rövid neve”**.

Természetesen van mód a változó teljesen konkrét meghatározására is, ez történik a már sokszor emlegetett „casting operátorral”. Egy efféle meghatározás, amit a továbbiakban a változó **„teljes nevének”** nevezünk, így néz ki:

**#c@A**

A **"c"** karakter helyén állhatnak más karakterek is, amik a típusra utalnak. A lényeg az, hogy e karakter közvetlenül a **#** jel után kell álljon, nem lehet köztük whitespace. A típusjelölő karakter (e példában a **"c"**) és a **@** jel közt azonban már állhat akárhány whitespace karakter is. Általában, a mau nyelvben ahol állhat 1 whitespace karakter, ott állhat akármennyi is.

A mau nyelv a 33-nál kisebb ASCII kódú karaktereket tekinti whitespace-nak. Ezek közül az ismertebbek:

**0**

**9 (=TAB)**

**10 (=újsor karakter)**

**11**

**12**

**13**

**32 (=SPACE)**

A mau nyelvben a **#** karakter (tehát a casting operátor) után a következő karakterek állhatnak, e jelentéssel (ezek ugyanazok, vegyük észre, mint a fenti union-ban):

**c: unsigned char**

**C: signed char**

**i: unsigned short int**

**I: signed short int**

**l: unsigned int**

**L: signed int**

**g: unsigned long long**

**G: signed long long**

**f: float**

**d: double**

**D: long double**

**s: string**



Vegyük észre, hogy a nem lebegőpontos típusok esetén az előjeles (signed) típusokat jelölő karakterek az előjel nélküliek nagybetűs párjai, így ez könnyebben megjegyezhető!

Akadnak a fentiekén kívül még egyéb típusok is, de azok csak speciális esetekben használhatók, ilyenek az inputfile, outputfile, directory, veremtár, benchmark... ezekről később lesz szó. Bár igazából a stringtípus is olyasmi, aminek ezen unionhoz nincs köze, de mégis jó dolog megemlíteni itt, mert az azért mégis majdnem bárhol használható, ahol a numerikus változók (legalábbis közülük a nem lebegőpontosak).

Megjegyzendő továbbá, hogy immár 2 stringfajta is létezik a mau nyelvben: az egyik a „szokásos”, a másik egy speciális, amit **JELSOR**-nak nevezek a megkülönböztetés végett, s ez nem bájtokból áll, hanem **BETŰK**ből... E disztinkció oka, hogy ez utóbbi 2 micsoda (a **BETŰ** és a **JELSOR**) teljesen az UTF-8 támogatására van „kihegyezve”. Ezek casting operátorai:

A BETŰé: **#u**

A JELSORé: **#U**

Azért kis és nagy „u”, mert ugye **UTF-8** kódolású... Ezekről soká-soká nem fogsz e doksiban többet olvasni, legfeljebb elvétve pár szót, hanem egy sokkal későbbi fejezet teljes egészében ezekről szól majd!

Amennyiben tehát azt írjuk hogy **#c@A**, az az A változó unionjának **c[0]** rekeszét jelöli mint unsigned char szám. A **#l@A** pedig az A változó **l[0]** rekeszét jelenti, ami 4 bájt, s unsigned int értékként értelmeztetik.

Amennyiben mi kifejezetten épp az A változó **c[0]** rekeszének 1 bájtnyi értékét akarjuk megkapni, de igenis unsigned int számként és nem unsigned char értékként, akkor azt kettős castolással tudathatjuk az interpreterrel:

**#l#c@A**

Lehetne nemcsak kettős, hanem hármas, négyes, akárhányas castolást is alkalmazni, azaz akárhány casting operátort egymás után írni. Ugyanis nagyon fontos megjegyezni, hogy a casting operátor nem olyasmi, ami speciálisan csak egy változóra vonatkozik, hanem A TELJES, ŐUTÁNA KÖVETKEZŐ ARITMETIKAI KIFEJEZÉSRE! Az hogy egy változóból miféle adatot szed elő az interpreter, attól függ, épp azon a helyen mit vár el alapértelmezettként; - kivéve ha van valami casting operátor közvetlenül a változó előtt. Ha ugyanis van, akkor az annak megfelelő adatot szedi elő az unionból. Ha van ettől balra másik casting operátor is, akkor a már előszedett adatot arra castolja. Ha még másik is van, akkor megint castol, és így tovább. Ha már nincs több, akkor legvégül arra castol, ami típus neki kell az adott művelethez, s elvégzi az így kapott adattal azt, ami ott a dolga.

Amennyiben nem tudod biztosan, hogy egy rövid névvel jelzett változót azon a helyen a programban épp miféle típusként akarna kezelni az interpreter, akkor nem kell elkeseredned, egyszerűen tedd ki elé a konkrét casting operátort, ami neked ott épp megfelelő, és kész! Semmi baj nem történik ugyanis, ha a casting operátor olyan helyen szerepel, ahol tulajdonképpen feleslegesen van ott. Ez mindössze 2 plusz bájt a programkódban, s esetenként egy plusz függvényhívás. Nemigen jelent érzékelhető mértékű programlassulást általában, hacsak nem egy 10 milliószor lefutó ciklus belsejében van.

A fenti jelöléstípus azonban csak az union nullás indexű mezőit tudja visszaadni. Ha olyan mező kell aminek az indexe nem nulla, azt a következőképp adhatjuk meg:

**@A:3**

Természetesen a „3” szám helyén nemcsak a 3-as szám állhat. Hogy mennyi, az attól függ, az adott típusú adatból mennyi fér el a 16 bájtos unionban. Korábban megadtam az union felépítésének ábráját, abból láthatóak a maximális indexek. Ez az unsigned és signed char típusnál lehet a legnagyobb, tudniillik 15. A long double értéknél azonban semmilyen se lehet, mert az egymaga 16 bájt hosszú.

Most már gondolom érthető, miért viccelődtem korábban azzal, hogy a mau nyelvben csak 256 különböző változó lehet, és semmiképp sem több, kivéve amikor mégis... E 256-os maximális érték ugyanis csak az unionokra vonatkozik. A long double típust kivéve azonban a többi adattípusból több is belefér egy 16 bájt hosszú unionba, azaz simán megcsinálhatjuk, hogy mondjuk a **@A** nekünk egyszerre több változót is jelentsen:

```
#f@A:0 // ez egy float szám, 4 bájt a sizeof mérete
#i@A:2 // unsigned short int, 2 bájtos sizeof mérettel
#C@A:6 // signed char, 1 bájtos sizeof mérettel
#C@A:7 // signed char, 1 bájtos sizeof mérettel
#g@A:1 // unsigned long long, 8 bájtos mérettel.
```

A fentiek mind a **@A** változóban vannak, egyidejűleg, és „nem bántják egymást”. Azaz, végeredményben 256-nál jócskán több változónk lehet, egyedül akkor van éppen pontosan csak 256 változónk, ha kizárólag long double változókat használunk. De olyan eset alig is képzelhető el. Ráadásul majd e leírás későbbi részeiben láthatjuk, hogy akad még ezeken felül is sok további lehetőségünk mindenféle változókat használni az itt most bemutatottakon túl is, tudniillik veremtárakban, vagy tömbökben, vagy álfüggvények névterében levő változókkal bűvészkedni, vagy azokat használni amiket a függvényünk az őt hívó programból ad át, vagy a szülőnévtér változóit használni... s akkor még nem is beszéltünk arról, hogy egy függvényen belül akárhány elkülönített névteret is létrehozhatunk, ráadásul e névtereket tetszőleges mélységben egymásba skatulyázva... Szóval nem kell aggódnia annak, aki netán úgy véli, a 256 lehetőség a változók nevére túl kevés! Bár megjegyzem, én úgy találtam hogy többnyire nemhogy 256 elegendő, de az is amit kizárólag az angol ABC kis- és nagybetűivel tudok megvalósítani. Ha több változó kell egy függvényen belül, úgyis tömböket használok.

Továbbá, a fenti példák amiket felsoroltam, hányféleképp használhatjuk egyidejűleg a **@A** nevet változóazonosításra, amiatt se teljeseek, mert például a stringet jelölő **#s@A** változónak semmi köze sincs ezen unionhoz, az tehát egy további plusz lehetőség!

Na most, fontos tudni, hogy a mau programok azért tűnnek bonyolultnak, mert a **@A:3** jelölésben az „A” és a „3” helyén is állhat tetszőleges (alapértelmezésként unsigned char típusú) aritmetikai kifejezés... Tehát más változók is, sőt, akármiféle tetszőlegesen bonyolult kuttyulmány! Muszáj is hogy ez így legyen, mert változóból (azaz inkább változó-unionból) 256 fajta lehetséges (egy névtéren belül), s ennyit nem lehet megnevezni mindössze az angol ABC kis- és nagybetűivel.



Az aritmetikai kifejezések a mau nyelvben mindazonáltal nem különösebben bonyolultak. (A változónevektől eltekintve. Bár tulajdonképpen azok is csak akkor bonyolultak, ha nagyon egymásbaskatulyázzuk őket).

Ami a konstansokat illeti, azokra a következő szabályok érvényesek:

—Minden számsorozat ami a 0-9 karakterek valamelyikével kezdődik, DECIMÁLIS SZÁMKÉNT van értelmezve. (Igen, a nullával kezdődőek is, azok sem oktálisnak hanem decimálisnak vannak tekintve!)

—A **\$** jellel kezdődnek a hexadecimális, azaz a 16-os számrendszer beli számok. Ezek nem lehetnek lebegőpontosak. A 9-nél nagyobb értékű számjegyek megadhatóak kis- és nagybetűkkel is, azaz az a-f és A-F betűkkel egyaránt.

—A **%** jellel kezdődnek a bináris számok. Ezek is csak egész típusúak lehetnek.

—Amennyiben az interpreter egy „o” vagy „O” betűbe szalad bele, megnézi, ezt valamely oktális számjegy követi-e, vagyis a 0-7 számjegyek valamelyike. Ha igen, akkor beolvassa azt a számsorozatot oktálisán, s ez lesz az adott konstans értéke. Ha nem követi a kis vagy nagy o betűt oktális számjegy, akkor viszont a betű (vagyis az „o” vagy „O” karakter) ASCII kódját adja vissza.

—Tetszőleges nem whitespace karakter esetén, ha az nem „\$”, „%”, „#”, „?”, „o”, „O” vagy „'” azaz aposztróf jel, nem is unáris operátor vagy netán más olyasmi ami valamely speciális jelentéssel bír a mau programnyelvben (ezt azért írom ilyen óvatosan, mert ugye lehet ám hogy a nyelv bővílni fog...) ekkor tehát az adott karakter ASCII kódját adja vissza. Az angol ABC kis- és nagybetűit illetően egészen biztos hogy ez a helyzet, s ez meg is marad, kivéve az „o” és „O” betűket, ahol ez attól is függ, követi-e őket oktális számjegy, ezt fentebb már írtam.

—Az aposztróf jel esetén („'”) az aposztróft követő bájt ASCII kódját adja vissza, akármilyen legyen is az, akkor is ha az valamely whitespace karakter, vagy nem kiiratható karakter.

Egy konstans addig olvastatik be, azaz a karaktereket addig tekinti az interpreter a konstans részének, ameddig olyant nem talál, mely nem esik bele az adott számrendszer értelmezési tartományába.

Ennek megfelelően az alábbi jelölések mind ugyanazt a változót határozzák meg:

```
@A
@'A
@65
@$41
@%1000001
@o101
@0101
```

Nyilvánvaló természetesen, hogy e jelölésmódok közül a **@A** típusú a leggyorsabban feldolgozható, leghatékonyabb, sőt, a legkönnyebben olvasható is.

A mau nyelvénél SZABVÁNY, azaz alapvető előírás, hogy minden létrejövő névtér esetén a névtér LÉTREJÖTTEKOR azonnal rendelkezésre áll minden fentebb felsorolt változótípusból mind a 256 darab, azaz nem kell őket semmiféle módon előre deklarálni, meghatározni, előírni, definiálni, stb, másrészt kivétel nélkül mindegyiknek a kezdeti értéke éppen pontosan 0 azaz NULLA és nem más!

Amennyiben valamikor valamely mau változat (release) esetén ez nem így volna, az igenis nem feature, hanem ordas nagy BUG, méghezzá EXTRÉM KRITIKUS minősítéssel!

Figyelem! A névtér LÉTREJÖTTEKOR lesz nulla a változók értéke, nem amikor a névtérbe belépünk, tudniillik függvényeknél a változók utoljára használt értéke megmarad a következő belépés számára!

## 2. fejezet - Az aritmetikai kifejezések

Az aritmetikai kifejezés egy olyan szimbólumsorozat, aminek a végeredménye valami szám. Hasonlóképp, a stringkifejezés olyan szimbólumsorozat, aminek a végeredménye egy string. A két fogalom azonban nem különül el egymástól élesen, mert a stringek többnyire megfeleltethetők bizonyos számoknak (különösen, ha számjegyek sorozatából állnak...), meg amiatt is, mert gondoljunk csak bele, ha rész-stringet kell képezni, meg kell mondani számmal, hogy a rész-string az eredeti string hányadik karakterénél kezdődjön, s e szám már nemcsak konstans lehet, hanem akármilyen aritmetikai kifejezés is... Arról nem is beszélve, hogy egy szám is átalakítható stringgé, s persze ezesetben sem csak konstans számról lehet szó, hanem tetszőleges aritmetikai kifejezésről...

Mindezen dolgok miatt az aritmetikai vagy string kifejezések akár rémségesen összetettek is lehetnek. Egy ilyen kifejezés úgynevezett „tagokból” áll, melyeket műveleti jelek, vagy általánosabban fogalmazva **operátorok** kapcsolnak össze egymással, de egy tag maga is lehet több tag kombinációja. Egy tag például egy változó, vagy egy konstans, vagy egy beépített mau rendszerfüggvény, vagy egy a Felhasználó által megvalósított mau nyelvű függvény hívása, egy címke, egy általános veremtárra utalás, egy speciális veremtárra utalás amiben függvény input paramétere van, valami speciális objektum által biztosított adat például amikor egy megnyitott input fájlból lekérjük a következő karaktert, vagy amikor egy tartalomjegyzék-változótól lekérdezzük hogy benne hány szimbolikuslink-bejegyzés van, vagy valami fontos rendszerváltozó értéke, vagy a mau forrásprogram egy karaktere, vagy valami rendszerflag értéke... szóval, rengeteg minden „micsoda” lehet egy kifejezésben, ami mind egy-egy „tag”. Na most, ezeket operátorok kapcsolják össze, ezek egy része ismerős lehet más programnyelvekből, például az összeadás (+), a kivonás (-), szorzás (\*) és osztás (/) operátora, de van sok más is.

Az operátorok kiértékelésének sorrendjére más programnyelvekben bonyolult precedenciaszabályok vonatkoznak. A mau nyelvben ilyesmi egyszerűen NINCS. Itt az aritmetikai kifejezések alapértelmezés szerint jobbról balra értékelődnek ki. NEM TÉVEDÉS! Az utasításfeldolgozás természetesen balról jobbra halad továbbra is, de mert a kifejezéskiértékelő rutinok minden operátornál önmagukat rekurzívan meghívják, emiatt legelsőként a legutolsó operátor lesz végrehajtva. Azaz:

$6*3+10$  értéke 78 lesz, nem 28, ahogy azt a matematikában általában megszoktuk. Ezt azonban előírhatjuk zárójelekkel:

$(6*3)+10$

Természetesen akárhány zárójel is egymásba ágyazható.

Ami a precedenciaszabályokat illeti, ott ugyanis nem sok értelmét láttam annak, hogy ilyesminek a megvalósításával görcsöljek. Akad pár más programnyelv is, ha nem is a legismertebbek, melyekben nincsenek precedenciaszabályok, sőt, még ilyen zsebszámológépek is vannak. Továbbá, még talán rá is szánám magamat erre, ha csak arról lenne szó, hogy a „+” és „-” jelek alacsonyabb precedenciájúak mint a „\*” és a „/”, de hát ugyebár van ezeken kívül még egy egész rakás más operátorunk is, no és ezek közt cseppet se egyszerű meghatározni valami logikus precedenciát! Ott van példának okáért a C nyelv is, tele van operátorokkal, és ezeknek rengeteg szintje létezik. Ember legyen a talpán, aki fejből vágja, melyiknek épp mi a precedenciaszintje! Még én magam is úgy vagyok vele hogy nagy ívben tojok az egészre, és ha kicsit is bonyolultabb esetről van szó, egyszerűen úgy zárójelezem a kifejezést, hogy az teljesen egyértelmű legyen. Holott talán senki se vitatja hogy tényleg tudok C nyelven programozni, hiszen magát a mau nyelvet is javarészt C nyelven írtam (és kisebbrészt C++ nyelven). Ennek ellenére, ez a véleményem a C és C++ precedenciaszabályairól. Az, hogy ha picit is bonyolultnak tűnnek, zárójelezni kell és kész. Minden más megoldás kizárólag arra jó, hogy rengeteg tévedésre adjon lehetőséget!

Arról már nem is beszélve, hogy más programnyelvekben ugyanazon operátorok precedenciája teljesen eltérő is lehet... Gyakorlatilag annyi precedenciaszabály létezik, ahány programnyelv. Teljesen felesleges e zűrzavart fokoznunk azzal, hogy mi is bevezetünk valami megint újabb rendszert a mau nyelvbe. Mi a szösznek, hogy még mi magunk is belekeveredjünk?!

Nálunk, a mi mau nyelvünkben ilyesmi tehát nem fordulhat elő. Egyetlen szabály van csak, amit emiatt rém könnyű megjegyezni: MINDIG jobbról balra értékelődik ki a kifejezés, s bármi más sorrendet óhajtunk, azt zárójelezéssel érhetjük el! Ez világos és egyértelmű.

Ebből azonban következik pár dolog, amit észben kell tartanunk. Tegyük fel, az A változó nulladik unsigned char mezőjének értékéhez hozzá akarunk adni egyet. E mező jele:

**#c@A:0**

De mert alapértelmezés szerint úgyis a nulladikat tekintjük ha nem jelöljük explicite a mezőindexet, így elég ennyit írunk:

**#c@A**

Adjunk hozzá egyet:

**#c@A+1**

Nos, a fenti kifejezés szintaktikailag helyes, sajnos azonban nem az **A** változó értékének eggyel megnövelt értékét adja vissza nekünk, hanem a **B** változó aktuális (meg nem növelt) értékét! Ugyanis a **@** karakter után ő egy aritmetikai kifejezést vár el, ami meghatározza neki a kiértékelendő változó NEVÉT, és az **A+1** a mau nyelvben egy tökéletesen érvényes aritmetikai kifejezés, ami azt a számot jelenti, ami az „A” karakter ASCII kódjánál eggyel nagyobb érték. Az pedig azonos a „B” karakter ASCII kódjával ugyebár...

Azaz, amit akarunk, azt zárójelezéssel érhetjük el. Ezt kell írunk:

**(#c@A)+1**

Vagy ezt:

**#c(@A)+1**

A mau nyelv operátorai (jelen pillanatban...):

<i>Kétooperandusú operátorok</i>	
+	összeadás (stringekre is)
-	kivonás (értelmezve van az is, amikor stringből vonunk ki egész számot, ennek magyarázatát lásd a stringek leírásánál...)
*	szorzás
/	osztás
&&	logikai ÉS
	logikai VAGY
^^	logikai KIZÁRÓ VAGY (EXOR)
>>	maradékképzés (csak egész számoknál)
<<	bitenkénti eltolás balra
>>	bitenkénti eltolás jobbra
==	EGYENLŐ összehasonlító művelet
<	KISEBB MINT összehasonlító művelet
>	NAGYOBB MINT összehasonlító művelet
<=	KISEBB VAGY EGYENLŐ összehasonlító művelet
>=	NAGYOBB VAGY EGYENLŐ összehasonlító művelet
!=	NEM EGYENLŐ összehasonlító művelet
<>	NEM EGYENLŐ összehasonlító művelet (teljesen azonos a != művelettel)
&	indirektművelet-operátor, a magyarázatát lásd kicsit később a 7. fejezetben a stringeknél

Az összehasonlító műveletek természetesen stringek esetén is értelmezettek.

Fontos megemlíteni, hogy míg a C nyelvben létezik külön & és &&, valamint | és || operátor, mert az a nyelv megkülönbözteti egymástól az aritmetikai ÉS meg VAGY kapcsolatot a logikai értékek közt képzett efféle viszonyoktól, addig a mau nyelvben ilyen disztinkció nincs, az összehasonlító kifejezések által visszaadott eredményeket ugyanúgy a && és a || operátorokkal hozhatjuk ÉS illetve VAGY kapcsolatba, mintha valami bitműveleteket hajtanánk végre. Ez azért tehető és engedhető meg, mert a mau nyelvben kivétel nélkül minden összehasonlító művelet garantálja, hogy az eredménye KIZÁRÓLAG **0** vagy **1** lehet: 0 a hamis, és 1 az igaz esetben. Márpedig amennyiben csak 1 bitnyi értékekről van szó, mint nálunk az összehasonlító műveleteknél, úgy mindegyiknek az eredménye azonos:

```
1&1 = 1&&1 = 1,
1&0 = 1&&0 = 0,
0&0 = 0&&0 = 0,
1|1 = 1||1 = 1,
1|0 = 1||0 = 1,
0|0 = 0||0 = 0
```

Vagyis nálunk a logikai operátor lehet ugyanaz, mint a bitműveletekre szolgáló. E célra a dupla jeleket választottam, a kettős & és | jeleket, mert jobban olvashatóak a forráskódban. Figyelemfelkeltőbbek.

Egyoperandusú azaz „unáris” operátorok is léteznek a mau nyelvben, melyek a tőlük jobbra eső teljes aritmetikai kifejezésre vonatkoznak. Ezekből azonban nincs sok. Ilyen mindenekelőtt maga a casting operátor, tehát a **#c**, **#C**, **#i**, **#s** stb

karakterpárosok. Ilyen aztán a közismert „-” azaz mínuszjel, ami természetesen a mau nyelvben is az öt követő szám (pontosabban aritmetikai kifejezés értékének) mínusz egyszeresét jelenti. Ilyen a ~ jel, ami egész számok esetében bitenkénti negációt jelent, string esetén pedig azt, hogy a stringet megfordítja bájtanként, azaz az utolsó karakterből lesz az első, az utolsóelőttiből a második, stb. Ügyeljünk rá, hogy ez a több-bájtos kódolású karaktereket (pld UTF-8 kódolás esetén) elrontja! Van aztán még a ! jel, ami string esetén a string hosszát adja vissza bájtokban mérve.

Van aztán a !! jel is, azaz a dupla felkiáltójel, ami a string hosszát BETŰKBEN adja vissza! Ezen fogalom alatt természetesen UTF-8 kódolású karaktereket kell érteni. Minthogy az UTF-8 kódolás rém bonyolult valami, itt a mau interpreter rém egyszerűen jár el, ami a karakterek számolásának algoritmizálását illeti: feltételezi, hogy az a string egy SZABÁLYOS string, azaz csupa érvényes UTF-8 kódolású karakterből áll! Azaz, úgy tekinti, hogy abban nincsenek más bájtok, csak karakterek, s azok mind szabályos UTF-8 kódolásban leledzenek. Ez azt jelenti, hogy az ő bájtokban mért hosszuk megállapítható az első bájtjuk alakjáról, mert az az első bájt vagy egy érvényes ASCII karakter, vagy ha annak kódja 128-nál nagyobb (vagy azzal egyenlő), akkor az abban levő bitekből kikövetkeztethető, még hány bájt tartozik a karakterhez. Amennyiben valami olyan stringet „kajáltatunk” meg vele ami nem ennek megfelelően „működik magát”, akkor e függvény (vagy minek nevezzem) hibás eredményt fog produkálni... Íme egy kis példa a használatára:

```
#s@s="Macska";
#s@g="Álomkór";
#s@q="„ez”";
#s@k="miáúéi";
"Unsigned char:\n"
?s @s; " hossza = "; ?c !#s@s; /;
?s @s; " UTF-8 hossza = "; ?c !!#s@s; /;
?s @g; " hossza = "; ?c !#s@g; /;
?s @g; " UTF-8 hossza = "; ?c !!#s@g; /;
?s @q; " hossza = "; ?c !#s@q; /;
?s @q; " UTF-8 hossza = "; ?c !!#s@q; /;
?s @k; " hossza = "; ?c !#s@k; /;
?s @k; " UTF-8 hossza = "; ?c !!#s@k; /;
"Unsigned int:\n"
?s @s; " hossza = "; ?l !#s@s; /;
?s @s; " UTF-8 hossza = "; ?l !!#s@s; /;
?s @g; " hossza = "; ?l !#s@g; /;
?s @g; " UTF-8 hossza = "; ?l !!#s@g; /;
?s @q; " hossza = "; ?l !#s@q; /;
?s @q; " UTF-8 hossza = "; ?l !!#s@q; /;
?s @k; " hossza = "; ?l !#s@k; /;
?s @k; " UTF-8 hossza = "; ?l !!#s@k; /;
```

Eredménye:

```
Unsigned char:
Macska hossza = 6
Macska UTF-8 hossza = 6
Álomkór hossza = 9
Álomkór UTF-8 hossza = 7
„ez” hossza = 8
„ez” UTF-8 hossza = 4
miáúéi hossza = 9
miáúéi UTF-8 hossza = 6
Unsigned int:
Macska hossza = 6
Macska UTF-8 hossza = 6
Álomkór hossza = 9
Álomkór UTF-8 hossza = 7
```

```

„ez” hossza = 8
„ez” UTF-8 hossza = 4
miáúéi hossza = 9
miáúéi UTF-8 hossza = 6

```

Mint látható a fentiekből, a **!** és a **!!** unáris operátor eredménye teljesen megegyezik azon stringek esetében, melyek nem tartalmazznak mást csak csupa ASCII karaktert.

Van aztán a dupla **~** jel, azaz a **~~** operátor: Ez csak stringek esetén értelmezett, ez is mint a szimpla hullámvonal, megfordítja a stringet, de ez már az UTF-8 kódolású karaktereket nem rontja el! Csupa ASCII karaktert tartalmazó stringek esetén az eredménye természetesen ugyanaz mint amit a szimpla hullámvonal, azaz az egyszerű **~** operátor esetén kapnánk. Egy kis példa a használatára, ahol szépen megmutatkozik a szimpla hullámvonal és a dupla hullámvonal használata közti különbség:

```

#s@s="Macska";
#s@g="Álomkór";
#s@q="„ez”";
#s@k="miáúéi";

```

```

?s @s; /;
?s ~@s; /;
?s ~~@s; /;
"---\n";
?s @g; /;
?s ~@g; /;
?s ~~@g; /;
"---\n";
?s @q; /;
?s ~@q; /;
?s ~~@q; /;
"---\n";
?s @k; /;
?s ~@k; /;
?s ~~@k; /;
"---\n";

```

Eredménye:

```

Macska
akscaM
akscaM
---
Álomkór
r³ÄkmoLÄ
rókmoLÄ
---
„ez”
âzeâ
”ze,,
---
miáúéi
iöúáÄim
iëúáim
---

```

Gyakori az is, hogy egy stringnek éppen pontosan az utolsó karakterére vagyunk kíváncsiak, ezt szolgáltatja nekünk a **!\$!** operátor. Ha ez nem állna rendelkezésünkre, akkor csak bonyolult módon hivatkozhatnánk rá, lekérdezve a string hosszát, majd abból egyet kivonva. Ezen operátor azonban jelentősen megkönnyíti a dolgunkat, mint ezen alábbi példaprogramon látható. Ha nem értenéd,

ne keseredj el, majd a stringekről szóló fejezetben „fel leszel homályosítva” arról, mi mit jelent benne:

```
#!mau
#s@s="Ez egy mau string";
?s @s; /;
#c@c=#s@s[--(!@s)]; // Ez az első variáció
?c @c; " = " ?k @c; /;
#c@c=|s|@s; // Ez a második variáció
?c @c; " = " ?k @c; /;
XX
```

Eredménye:

```
Ez egy mau string
103 = g
103 = g
```

Van továbbá **!** alakú unáris operátorunk is. (Felkiáltójel-Aposztróf). Ez egy **BETŰ**, azaz **#u** típusú aritmetikai kifejezés előtt használható mint unáris operátor, és minthogy egy BETŰ nem más mint egy UTF-8 kódolású karakter, s egy efféle nem okvetlenül 1 bájtot foglal el (hanem maximum 6-ot), ezért ez az unáris operátor visszaadja a beolvasott BETŰről azt az információt, hogy ez épp hány bájtot igényel tárolásra az UTF-8 kódolás szerint. Példa erre:

```
#!mau
#s@s="álmós";
#u@u=#s@s; ?u @u; /; ?c !'@u; /;
#u@u=#c g; ?u @u; /; ?c !'@u; /;
#u@u="„ez”"; ?u @u; /; ?c !'@u; /;
XX
```

Eredménye:

```
á
2
g
1
”
3
```

Akad még pár unáris operátor az eddig említetteken kívül is:

```
|<|
|>|
|^|
|!|
|!|
```

de ezek mind szintén az UTF-8 kódolású **#U** stringekkel vagy az ezeket alkotó **#u** BETŰkkel állnak összefüggésben, emiatt ehelyütt teljesen felesleges tárgyalni őket, ezekről részletes információ az ezen **#u** és **#U** típusokat bemutató fejezetben található.

Nem kizárt természetesen, hogy a mau nyelv későbbi kiadásában az unáris operátorok száma is bővülni fog.

Itt kell megemlítenem, hogy az előző fejezetben taglalt casting operátoroknak is létezik természetesen INDIREKT változata! Ez nem más, mint a

```
#(
```

Ami azt jelenti, hogy a **#** jelet közvetlenül kell kövesse egy nyitó kerek zárójel, nem állhat köztük whitespace. Ezesetben az interpreter beolvassa az ezután következő aritmetikai kifejezést mint unsigned char értéket, s elvárja hogy ezt egy csukó kerek zárójel zárja le. A beolvasott kifejezés eredményéül kapott bájtot próbálja meg értelmezni mint típusjelölő karaktert.



Mint azt nyilván mindenki sejti, a mau nyelvben számos olyan utasítás, függvény stb van, melyek paramétert várnak el, sőt, egynél több paramétert is igen gyakran. E paraméterek természetesen tetszőleges aritmetikai (vagy string) kifejezések. Tekintve hogy a mau nyelvben szinte bármi lehet aritmetikai kifejezés, még a változók neve is, ezért ha egy utasítás vagy bármi más egynél több paramétert vár el, s e paraméterek többszörös mélységben egymásba vannak ágyazva, ilyenkor cseppet se magától értetődő első pillantásra, hogy hol kezdődik az egyik paraméter, és hol végződik a másik! A paraméterek elválasztására nem használhatjuk a pontosvessző karaktert, mert az maga is utasítás, mint opcionális utasítás-szeparátor. A mau nyelv emiatt rendelkezik egy másik szeparátorral, a paraméter-szeparátorral, s ez a ",", azaz a vessző karakter. EZ IS OPCIONÁLIS, azaz nem kötelező kitenni, de kitehető bárhová ahová mi azt jónak látjuk, minden olyan helyre, ahol az interpreter éppen egy új aritmetikai kifejezést szeretne elkezdni beolvasni. Azaz, nemcsak ilyesmi formában használhatjuk:

a, b, c,

Hanem még műveleti jelek után is, például:

```
#c@a=(@b)+,(@c)
```

Ugyanis a „+” jel után ő egy aritmetikai kifejezést vár, s ha azt vár akkor ott szerepelhet vessző is. Azt ugyanis egyszerűen átugorja. Ez azonban már nem megengedett hanem syntax error-t eredményez:

```
#c@a=(@b),+(@c)
```

Ugyanis ahol operátort vár el, ott ugyebár nem aritmetikai kifejezést vár el, márpedig a vessző kizárólag azok előtt állhat.

### 3. fejezet - Értékadás

Az értékadás a mau nyelvben is az egyenlőségjellel történik, például:

```
#c@A=#c@B
```

A fenti esetben unsigned char változóról van szó, ezt jelzi a **#c** casting operátor. Kivétel nélkül minden értékadó utasítás a **#** casting operátorral kell kezdődjék (van pár „nem dokumentált” utasítás amire ez nem igaz, de azokat jobb ha nem használjuk...) mert ebből tudja az interpreter, miféle típusú változónak kell értéket adnia. Azonban a fenti példa második casting operátorának szerepeltetése tulajdonképpen felesleges ebben az esetben, mert ha az értékadó utasítás a **#c** casting operátorral kezdődik, akkor már amúgy is unsigned char értékként akarja kiértékelni az egyenlőségjel utáni aritmetikai kifejezést alapértelmezés szerint. Azaz oda csak akkor kötelező casting operátor, ha valami olyasmit óhajtunk ott szerepeltetni, melynek típusa nem azonos azzal, ami a balérték típusa. Például:

```
#c@A=#i@K:3
```

Itt a K változó 3-adik unsigned short int értékét castoljuk unsigned char értékké, s ezt rakjuk az A változó nulladik unsigned char rekeszébe.

A jobbérték természetesen tetszőleges aritmetikai kifejezés lehet.

Felmerülhet esetleg a kérdés Olvasóimban (bennem legalábbis felmerült...) hogy mivel a változóknak a neve lehet valami számmal megadva is, azt vajh' decimális vagy hexadecimális számmal érdemesebb-e megadni, hasonlóképp pedig ha a változónak egy konstans számot adunk értékül, érdemes-e azt hexában megadni, hogy ezzel gyorsítsuk a programunk futását!



Nem sajnáltam a fáradságot rá, és kimértem több variációt egy egyenként 3 milliószor lefutó ciklussal. A méréshez a mau nyelv benchmarkolási lehetőségét használtam fel, ami e könyv egy későbbi fejezetében ismertetik. Két programot készítettem, mert ugyebár a decimális számjegyek lehetnek akár 2, akár 3 karakteresek is, hasonlóképp az egy változó nevét meghatározó hexa számkonstansok is lehetnek 1 vagy 2 karakteresek (a \$ prefixumukat nem számolva). A két program ez lett:

Első program:

```
#!mau
#g@g=3000000;
#t@a;
{|#g@g;
#c@255=255;
|}
#t@b;
{|#g@g;
#c@255=$ff;
|}
#t@c;
{|#g@g;
#c@$ff=255;
|}
#t@d;
{|#g@g;
#c@$ff=$ff;
|}
#t@e;
#g@a=#t@a; #g@b=#t@b; #g@c=#t@c; #g@d=#t@d; #g@e=#t@e;
"#c@255=255; variáció ideje: " ?g (@b)-(@a); /;
"#c@255=$ff; variáció ideje: " ?g (@c)-(@b); /;
"#c@$ff=255; variáció ideje: " ?g (@d)-(@c); /;
"#c@$ff=$ff; variáció ideje: " ?g (@e)-(@d); /;
```

XX

Második program:

```
#!mau
#g@g=3000000;
#t@a;
{|#g@g;
#c@10=10;
|}
#t@b;
{|#g@g;
#c@10=$a;
|}
#t@c;
{|#g@g;
#c@$a=10;
|}
#t@d;
{|#g@g;
#c@$a=$a;
|}
#t@e;
#g@a=#t@a; #g@b=#t@b; #g@c=#t@c; #g@d=#t@d; #g@e=#t@e;
"#c@10=10; variáció ideje: " ?g (@b)-(@a); /;
"#c@10=$a; variáció ideje: " ?g (@c)-(@b); /;
"#c@$a=10; variáció ideje: " ?g (@d)-(@c); /;
"#c@$a=$a; variáció ideje: " ?g (@e)-(@d); /;
```

XX

Nem baj ha nem értesz mindent még e programokból, lényeg az, hogy az jött nekem ki, hogy ha a változó neve is és a neki értékül adandó konstans is decimális számmal van megadva, az valóban lassabb mintha mindkettő hexában van megadva. Ez egyezett előzetes sejtésemmel. Gyakorlatilag azonban a különbség cseppet se lényeges, mert mindössze 1 ezrelék az időkülönbség...

Ellenben igenis jelentős az időkülönbség akkor, ha azt vetjük össze, mennyi időbe telik ha mindkettő ugyanabban a számrendszerben van megadva, illetve ha egyik így, másik úgy! Ha ugyanis mindkettő hexában van megadva, vagy ellenkezőleg, decimálisán (tehát a konstans is meg a változó neve is), akkor akár 20%-kal is GYORSABBAN fut, mint ha az egyik így, a másik meg úgy van megadva, tehát vegyesen!

Ez a felismerés, bevallom, engem is megdöbbszentett. Nem tudom pontosan, mi lehet ennek az oka, csak tippjeim vannak - esetleg összefügghet a mai számítógépprocesszorok bonyolultságával, amik már többszázúak, s „előredolgoznak”, azaz lesik, mikor melyik függvényt kell majd meghívniuk. Na és ha ugyanúgy a decimális kiértékelőt kell meghívniuk a konstans kiszámolásához is mint korábban (nem sokkal azelőtt) a változónév-kiértékeléshez, akkor az már be van töltve nekik valamiféle gyorstárba, s ez hogy nem kell újratöltögetniük adatokat/címeket, sokkal többet számít a processzornak, mint hogy miféle számrendszer szerint kell számolnia. Mindenesetre tehát ha mau programot írunk, nem az a legfontosabb hogy a konstansainkat miféle számrendszerben jelöljük, hanem hogy a számrendszereket NE KEVERJÜK, mert ez akár 20% időnyereséget is hozhat nekünk!

## 4. fejezet - A pontosvesszők

A mau nyelvben az egyes parancsokat, utasításokat többnyire nem kötelező lezárni pontosvesszővel. Gyakorlatilag nemcsak „többnyire” nem kötelező lezárni őket, hanem soha sem, kivéve azokat az eseteket, amikor nem lenne egyértelmű, hol végződik az egyik parancs, és hol kezdődik a másik. Ez egyrészt olyankor fordulhat elő, amikor valamely olyan karaktert használunk utasításként, mely mondjuk aritmetikai kifejezés operátoraként is értelmezett — ez többnyire (bár nem kizárólag) a „/” jel szokott lenni, ami utasításként alkalmazva egy sorvége-karaktert ír ki az stdoutra. Másrészt vannak olyan utasítások is a mau nyelvben, sőt ilyen „beépített rendszerfüggvények” is, amelyeknél nem okvetlenül kell mindig megadni minden lehetséges paramétert, mert a paraméterlista egy része (a jobboldali, azaz utolsó paraméterek közül egy vagy több) elhagyható, s ilyen esetben a nem megadott paramétereket „alapértelmezés szerinti” értékűnek tekinti az utasítás vagy a függvény. Ezen esetekben, ha elhagyunk egy vagy több paramétert, az utasítást kötelező lezárni éppen pontosan a pontosvessző karakterrel, innen tudja ugyanis az interpreter, hogy nincs már több paraméter, s a többit tekintse alapértelmezés szerintinek.

A mau nyelvben tehát a pontosvessző nem utasításlezáró jel, hanem SZEPARÁTOR, még hozzá majdnem mindig csak opcionális. A pontosvessző e nyelvben lényegében egy külön utasítás, ami olyan, hogy a funkciója mindössze abból áll, hogy előlépteti az utasításszámlálót (a programmutatót mely a követke-

ző feldolgozandó karakterre mutat) a következő whitespace karakterig. Eképp ha egymagában áll, vagy szóköz, tab stb követi, semmit se csinál - ha azonban valami kulcsszó előtt áll, akkor azt nem veszi figyelembe az interpreter, azaz semmit se értelmez, ami pontosvesszővel kezdődik! Ügyeljünk erre, mert eszerint a mau nyelvben nem lehet olyasmit művelni mint a C esetében, hogy tömören egymás után vannak írva az utasítások, köztük egy-egy pontosvesszővel - Itt ha kiteszünk egy pontosvesszőt, azt okvetlenül kell kövesse legalább 1 whitespace karakter is, különben ami utána van utasítás, azt nem veszi figyelembe az interpreter!

## 5. fejezet - A megjegyzések, kommentek

A mau nyelv ismeri mindkét C stílusú megjegyzés-szintaxist, a `//` stílusút és a `/* ... */` stílusút is. Pontosan ugyanezen karakterekkel ráadásul. Amennyiben tehát valahol egy `//` karakterpárhoz ér, akkor azonnal a következő sorra ugrik, tehát a sor további részén levő karaktereket egyáltalán nem veszi figyelembe a végrehajtás szempontjából.

Ami a `/*` utasítást illeti, itt is az a helyzet mint a C nyelv esetében: a megfelelő `*/` karakterpárosig semmit se vesz figyelembe, azaz a köztük levő parancsokat átugorja. Ezesetben a `//` jeleket se veszi figyelembe! Viszont számolja a `/*` keresése közben az esetleg fellelt `/*` karakterpárosokat, azaz a `/* ... */` megjegyzésrészek egymásba ágyazhatóak többszörösen.

Akadnak azonban olyan speciális részek a mau programnyelvben, amikor a `/* ... */` megjegyzéstípus nem használható. Ez függvényhívásoknál fordul elő, egy függvénynek azon részében ahol az input paramétereket fogadó változók vannak felsorolva, illetve ahol a visszatérési értékek céljából feltüntetett aritmetikai kifejezéseket soroljuk fel. Ezen helyeken csak a `//` típusú kommentelés megengedett. (Legalábbis egyelőre. Lehet hogy később majd megengedett lesz `/* ... */` kommentelés is, de ha igen, az nem mostanában következik be, mert ezt igazán nem tartom fontos funkciónak). Ennek részleteiről a függvényekről szóló fejezetben olvashatsz többet.

## 6. fejezet - Tömbök

A mau nyelvben a „tömb” tulajdonképpen egy speciális adattípus. Illetve nem EGY, hanem annyi, ahány fajta adattípust támogat e programnyelv a „single”, azaz nem tömb változók esetében is. A tömbváltozók nevére teljesen ugyanazon szabályok vonatkoznak, mint a közönséges változókéra, azt kivéve, hogy esetükben nem lehet „mezőindexet” megadni a „.” segítségével. Eképp minden fajta adattípus tömbjéből külön-külön pontosan 256 lehetséges nevű szerepelhet egy névtérben. Onnan lehet tudni egy változóról hogy az nem „single” hanem tömb, hogy a mau nyelvben kivétel nélkül minden változó és utasítás ami tömbbel kapcsolatos, valamiképp kötődik a szögletes (vagy szegletes?) zárójelekhez. (Viszont ami ezen zárójelekkel kapcsolatos, még nem feltétlenül bizonyos hogy tömb - tudniillik e zárójelfajttal kapcsolatosak a veremtárak is, meg természetesen a stringek is... meg még sok minden más is...)

Ahhoz hogy egy tömböt használjunk, természetesen „inicializálni” kell, azaz lefoglalni neki kellő nagyságú memóriaterületet. Természetesen a mau interpreter van annyira értelmes, hogy nem kell bájt méreteket számolgatnunk, mint a C nyelv *malloc* utasításánál, hanem a memóiafoglalás nálunk inkább a C++ nyelv „new” operátorára emlékeztet, amennyiben egyszerűen azt kell megadnunk, az adott típusú számokból hány darab számára foglalunk memóriaterületet. Ez efféleképp történik:

```
[#c@t=100] // Memóiafoglalás 100 darab unsigned char értéknek a "t" nevű tömbbe
```

Természetesen a **#c** casting operátor helyén bármely korábban megismert más típusú casting operátor is szerepelhet, az meg végképp természetes gondolom, hogy a fenti példa „100”-as száma helyén nemcsak más szám is állhat, de tetszőleges aritmetikai kifejezés is! Sőt, a „t” változónév helyén is állhat tetszőleges aritmetikai kifejezés, amit ökelme a végén unsigned char értékre fog castolni. Emiatt lehet olyan aljas trükköket is művelni, mint e példában, amit egy konkrét, számomra hasznos dolgot végző mau nyelvű programból idézek ide:

```
{! #s@Z; if (#l (?#l "T#" t, ?#c "{!}")>0) T [#c@(?#c "{!}")=?#l "T#" t, ?#c "{!}"];  
!} // E ciklusban foglalunk le memóriát annyi darab unsigned char értéknek a megfelelő nevű  
// tömbbe, ahány tartalomjegyzéki bejegyzés van abból a típusból. Ez egy fix számszor lefutó  
// ciklus, annyiszor fut le,  
// amennyi a string hossza. A ?#c "{!}" adja vissza a string aktuális karakterét.
```

E fenti példában persze hemzsegnék az olyan utasítások melyek ezen ismertetőben még nem lettek elmagyarázva, ezekkel most ne is törődjünk, a lényeg belőle e rész:

```
[#c@(?#c "{!}")=?#l "T#" t, ?#c "{!}"]
```

Ez végzi egy tömb számára a memóiafoglalást. Ugye, szegletes (szögletes?) zárójelek közt van. Ott a **#c** casting operátor: azaz unsigned char értékeknek foglal területet. Az egyenlőségjel után van meghatározva hogy hány darabnak, ez egy aritmetikai kifejezés ami a ciklusváltozó értékével függ össze (de nem csak azzal), most még ne törődjünk vele hogy ez mi. Hanem a kukacjel (@) után és az egyenlőségjel közt szerepel egy másik kifejezés, na ez határozza meg a tömbváltozó NEVÉT, aminek a memória lefoglaltatik:

```
(?#c "{!}")
```

Ez egy unsigned char típusú, a mau interpreter által „beépítve” szállított rendszerfüggvény, majd később lesz elmagyarázva a jelentése. (A ciklusokról szóló fejezetben). A lényeg hogy látható, a mau nyelvben tényleg lehet jószerivel minden egy aritmetikai kifejezés, még a változók vagy TÖMBVÁLTOZÓK NEVE is, még (vagy már?) akkor is amikor memóriát foglalunk le neki!

Sőt, még a **#c** casting operátor helyett is írhattunk volna ilyesmit:

```
#(@W)
```

ami azt jelentené, hogy azt, hogy ebben az esetben miféle típusú tömbnek is foglalunk le memóriaterületet, azt a **@W** unsigned char típusú (azaz teljes nevén **#c@W**) változó (alapértelmezés szerinti azaz nullás rekesze) határozza meg.

Illik tudni, hogy a mau nyelvénél GARANTÁLT, hogy ha egy akármilyen típusú tömb számára memóriát allokalunk, akkor azon tömb összes változójának kezdeti értéke éppen pontosan 0. Azaz nem szükséges külön inicializálnunk őket nulla értékkel. Amennyiben valamely mau verzióban ez nem így volna valamely típus

esetén, akkor az igenis nem feature, hanem BUG, még hozzá EXTREM KRITIKUS minősítéssel!

Tömb valahányadik elemének természetesen a C nyelvhez hasonlóan, a sz(e/ö)gletes zárójelek által indexelve adhatunk értéket:

```
#c@t[#i@i]=#i@i
```

A fenti példában a „t” nevű unsigned char típusú tömb azon elemébe, aminek indexét az „i” nevű unsigned short int (ezt a **#i** casting operátor jelzi) típusú változó tartalmazza, értékül adjuk épp azt a számot szintén, ami maga a tömb-index akkor, vagyis ugyanezen „i” nevű unsigned short int típusú változó értékét! Ebből persze baj lenne ha ezen változó értéke nagyobb lenne mint 255, de ugye előfordulhatnak olyan esetek amikor biztosan nem nagyobb. Amúgy ha nagyobb, akkor se fog az interpreter kiabálni, castolja unsigned char értékűvé, valami szám majd lesz belőle, oszt' jóóóó'van... A mau interpreter csak akkor jelez hibát, ha végképp lehetetlen kitalálnia bármiféle értelmes tevékenységet az adott utasítás kontextusából. Abból indul ki, hogy ő a hülye és nem a programozó. Azaz, mau nyelven programozva észnél kell lenni de nagyon ám, mert itt aztán maximálisan lehetősége van a programozónak extrém örült baromságok elkövetésére is! Ez az ára annak, hogy másfelől abszolút szabadsága van. Igen, ezen abszolút szabadság vonatkozik az abszolút hülyeségek elkövetésére is.

A tömbváltozók értelemszerűen szerepelhetnek jobbértékben is, ugyanezen szintaxissal:

```
#c@k=@t[#i@i]
```

A fenti példában mint látjuk nem tettük ki a tömbváltozó elé a casting operátort. Ebben az esetben, ha a balérték eleve egy unsigned char változó, akkor amúgy is unsigned char értékű tömbből próbál olvasni. Pontosabban, az egyenlőségjel utáni aritmetikai kifejezést olyan típusként próbálja értelmezni, amilyen típusú a balérték. Természetesen azonban nyugodtan kitehetjük a megfelelő casting operátort tömbváltozó elé akkor is, ha jobbértékként szerepel.

A tömbnek lefoglalt memóriaterületet fel is szabadíthatjuk így:

```
[#c@t]
```

Természetesen itt is állhat a „t” helyén aritmetikai kifejezés, és a **#c** helyett is használhatunk bármilyen casting operátort. A tömb memóriaterületének felszabadítása után természetesen lefoglalhatunk neki új memóriaterületet, akár más mérettel is.

## 7. fejezet - Stringek

A stringváltozókból is 256 különböző lehetséges minden külön névtérben, s ezen nevekre is ugyanazon szabályok érvényesek, mint a többi változók neveire. Természetesen e nevek is lehetnek tetszőleges aritmetikai kifejezések. A stringek „stringségét” a **#s** casting operátor jelzi, amit minden egyes stringváltozó elé mindig oda kell írni, kivéve természetesen amikor mégsem... Értelemszerűen akkor nem kötelező ezt kitenni, amikor alapértelmezés szerint azon a helyen

amúgyis stringet vár el az interpreter. Ha mégis kiteszük egy olyan helyre ahová nem kötelező, akkor azonban nem történik semmi baj.

A mau stringekről nagyon fontos észben tartani, hogy más nyelvekkel ellentétben a stringek karaktereit a mau nyelv nem signed, hanem **UNSIGNED** CHAR értékeként kezeli „alapból”, azaz alapértelmezés szerint! Amennyiben mást várunk el tőle, castolnunk kell a karaktert. Továbbá, a stringek itt is a nulladik karaktertől számozódnak, ellenben minden string automatikusan lezáratik a CHR\$(0) bájjal, azaz a 0 ASCII kódú karakterrel, mint az amúgy a C nyelvben is szokásos. Ez a lezáró nullabájt nem számítódik bele a string hosszába, amit közöl velünk a rendszer, amikor lekérdezzük tőle a string hosszát.

Azaz: Ha van mondjuk egy ABC karakterekből álló stringünk, akkor ennek hosszára a 3 számot kapjuk meg, s a karakterek indexe balról jobbra a 0, 1 és 2 lesz. Nem kapunk azonban hibajelzést a 3 index alkalmazására sem, mert ott is van még karaktere a stringnek: a záró nullabájt. Mindazonáltal, nem kényszerülünk rá, hogy kizárólag ennek vizsgálatával dolgozzuk fel a string összes karakterét, mert minden string esetén eltárolja a programnyelv nekünk a string hosszát is, így nem kell félnünk attól, hogy ha sokszor kérdezzük le a hosszat, mondjuk egy cikluson belül, akkor ezért mindig végigvizsgálja a string minden karakterét, a záró nullabájtot keresve, ami nagy idővesztés volt.

Egy string maximum annyi bájt hosszú lehet, amekkora szám belefér egy unsigned int értékbe.

A mau interpreter garantálja, hogy kezdetben mind a 256 string máris létezik, természetesen mint üres string, aminek a hossza nulla, s eképp kizárólag a záró nullabájtot tartalmazzák, más egyebet nem, e nullabájt pedig természetesen a string nulladik karaktere.

Konstans stringek stringváltozó értékéül a C nyelvben szokásos módon, idézőjelek közt adhatóak meg:

```
#s@a = "macska"
```

Természetesen ez esetben is kiegészül a string a lezáró nullabájttal a végén.

String hosszára így hivatkozhatunk:

```
!@a
```

Vagy így:

```
!#s@a
```

Jó tudni, hogy a string hosszánál a BÁJTBAN és nem a KARAKTERBEN mért hosszat számolja, illetve adja vissza nekünk... Ez egyáltalán nem mindegy, ha a karakterek több-bájtos kódolásban vannak, pld az UTF-8 kódolás esetén...

Aztán, Escape szekvenciákat is megadhatunk neki, amikor konstansokkal inicializálunk egy stringváltozót. Példa:

```
#s@a = "Egy \"jó\" macska"
```

A mau nyelv jelenleg a következő Escape szekvenciákat ismeri:

`\v` : sorvég karakter (újsor), ez Linux alatt az ASCII 10 karakter

`\n` : NL (LF) karakter (újsor), azaz ez is az ASCII 10 karakter Linux alatt



`\0` : ASCII 0 karakter  
`\t` : ASCII 9, tabulátor karakter  
`\b` : ASCII 8, backspace karakter  
`\r` : ASCII 13, CR karakter  
`\f` : ASCII 12, FF karakter  
`\a` : ASCII 7, csengő karakter (alert)

Stringeket a `+` operátorral összeadhatunk, akár konstansokkal vegyítve is:

```
#s@c = (@a) + " az kérlekálásson egy " + (@b);
```

Mint azt korábban az unáris operátoroknál már említettük, stringet meg is fordíthatunk a `~` operátorral:

```
#s@c = ~@c;
```

Viszont, ismétlem, ez elront minden több bájtos kódolású karaktert, azaz UTF-8 kódolás esetén minden ékezetes karaktert...

String egy karakterére a szegletes/szögletes zárójelek által indexelve hivatkozhatunk:

```
#s@c[2]='2;  
#s@c[3]=64;  
#c@c=#s@s[4];
```

Természetesen a sz(e/ö)gletes zárójelek közt tetszőleges (amúgy unsigned int típusú) aritmetikai kifejezés állhat.

Ha egy numerikus változónak adunk egy stringet értékül, akkor a mau interpreter megpróbálja a string elejét számként értelmezni, s amit így talál, azt adja értékként a numerikus változónak. Nézzük az alábbi példát:

```
#s@s="234kutya";  
#c@c=#s@s;
```

A fenti 2 sor esetén a `@c` nevű unsigned char típusú változó értéke 234 lesz.

Jó ha tudjuk, hogy amikor egy string egy karakterére hivatkozunk, akkor az nemcsak egy string változó esetén lehetséges, hanem egy egész stringkifejezésnél is működik! Például nézzük ezt:

```
"52cd"+"efghijk"[#l@i]
```

A fenti példában nem arról van szó, hogy az "52cd" stringhez hozzáadjuk azt a karaktert, amit az "efghijk" konstans stringből kijelöl a `@i` nevű unsigned int típusú változó, hanem az interpreter képez egy összegstringet mindkét, idézőjelek közti stringkonstansból előbb, ugyanis balról jobbra halad az utasításfeldolgozásban. Ezután rájön, hogy hoppá, nincs már több string, amit hozzáadhatna az eddigi stringek összegéhez! Van ellenben sz(e/ö)gletes zárójel, aminél arra gyanakszik, hogy ez valami indexelés lesz. Megvizsgálja, látja hogy szintaktikailag helyes, ez tényleg elfogadható indexelésnek. Oké, akkor amit eddig kapott eredménystringet, abból kiszedi azt a karaktert, amit az indexelés kijelöl neki.

Stringekből kivonhatunk számot is! Ez a string esetén azt jelenti, hogy a végéről annyi darab karaktert hagyunk, ahány számot kivonunk belőle. Példa:

```
#s@s="234kutya";  
#s@e=(@s)-2;
```

E fenti példában a `#s@e` eredménye ez lesz:

```
234kut
```

Rész-stringek képzése a `[,]` operátorral történik. (Jobb ötlet híján aggatom rá erre az izére az „operátor” elnevezést). Példák a használatára:

```
#s@s="101kiskutya"; #s@g="film"; #s@r=[,5]@s; // Az r eredménye: 101ki
#s@r=[,13](@s)+(@g); // Az r eredménye: 101kiskutyafi
#s@r=[0,4]@s; // Az r eredménye: 101k
#s@r=[5,]@s; // Az r eredménye: skutya
#s@r=[,]@s; // Az r eredménye: 101kiskutya
#s@r=[4,9](@s)+(@g); // Az r eredménye: iskutyafi
#s@r=[4,59](@s)+(@g); // Az r eredménye: iskutyafilm
#s@r=[700,59](@s)+(@g); // Az r eredménye üres string
#s@r=[2,0](@s)+(@g); // Az r eredménye üres string
```

Vagyis a fentiekből látható a [,] operátor működése: mindkét paramétere opcionális (és unsigned int típusú). Az első paraméter azt mondja meg, hányadik karakter a stringben a legelső, mely a képzett rész-string első karaktere lesz. A második paraméter azt mondja meg, összesen hány karakter hosszú kell legyen a rész-string (maximum). Ha az első paraméter nincs megadva, akkor a nulladik karakternél kezd. Ha a második paraméter nincs megadva, az eredeti string legvégéig mindent a rész-stringhez tartozónak tekint. Amennyiben az első paraméter nagyobb, mint az eredeti string hossza, vagy ha a második paraméter nulla, akkor üres string lesz a végeredmény. Ha az első paraméter és a második paraméter összege hosszabb mint az eredeti string hossza, de az első paraméter még benne van az eredeti string belsejében, akkor egyszerűen a string végéig pakol mindent a rész-stringbe.

A mau nyelv külön szépséges egzotikuma, hogy többszörösen is indexelhetünk egy stringkifejezést, előlről is és hátulról is! Nézzük csak meg e kis programot:

```
#s@s="0123456789";
#c@c=#s[3,5]@s[2];
"Kód: " ?c @c; /;
"Karakter: " ? @c; /;
```

Ennek eredménye ez lesz:

```
Kód: 53
Karakter: 5
```

Magyarázat: unsigned char értéknek adunk értéket. A jobbértéket egy stringből akarjuk kiszedni, ezért az egyenlőségjel után kell a #s casting operátor. Ezután megadunk egy rész-stringet meghatározó „operátort”, ami azt mondja meg, az utána következő stringkifejezés melyik része kell nekünk. Ezután jön a stringkifejezés, ami most csak egy szimpla stringváltozó. Ezt beolvassa, és kivágja belőle a megfelelő részt. Ez a rész ugyebár a 34567 karaktersorozat lesz. Ezután jön még egy indexelés, a sor végén, ami meghatározza, az iménti string (amely a stringváltozó által tárolt stringnek csak egy része) hányadik karakterére van nekünk szükségünk. Ez a második karakter. Miután a karakterek a képzett rész-stringnél is nullától kezdve számozódnak, ezért e karakter itt az '5' lesz.

Természetesen többször is vagdoshatjuk a stringet ha efféle perverzióra támad kedvünk:

```
#s@s="0123456789";
#c@c=#s[1,8][3,5]@s[2];
"Kód: " ?c @c; /;
"Karakter: " ? @c; /;
```

Ennek eredménye:



Kód: 54  
Karakter: 6

Előbb ugyanis kivágtuk az eredeti stringből a **34567** részt, majd ebből kéne kivágni az 1 indexű karaktertől kezdve 8-at, de mert nincs benne annyi karakter, emiatt ennek eredménye a **4567** karaktersorozat lesz. Ennek 2-es indexű karaktere pedig a **6**.

String készíthető másik string megsokszorozásával is:

```
#s@S="ABC"*3; // Ennek eredménye ABCABCABC lesz
```

Ilyet is lehet csinálni:

```
#s@w=(" " *3)+"<=";
```

Ilyet viszont NEM:

```
#s@w=(3*" ")+ "<=";
```

Azaz ügyeljünk rá, hogy stringet csak jobb oldalról szorozhatunk számmal, azaz jobboldalra írva a szorzótényezőt...

Érdekes tudnunk, hogy amennyiben egy stringnek annyiadik elemére hivatkozunk a sz(e/ö)gletes zárójellel, azaz tömbindexszel, amennyiedik elem momentán nem is létezik az ő esetében, mert a string jelenlegi hossza kisebb nála, akkor kibővíti a stringet olyan hosszúságúra, hogy létezzék annyiadik karaktere is. Eközben a string régi tartalma nem vész el, hanem megőrződik! Az új, megnövelt hosszúságú string esetében is garantált, hogy a legvége egy nullabájttal lesz lezárva, de amennyiben ez az automatikus stringhossznövelés olyankor következik be, amikor a stringet jobbértékként szerepeltetjük, akkor bizony az így visszaadott karakter tartalma véletlenszerű hogy micsoda, mert ott tetszőleges memóriaszemét lehetséges. Továbbá, ha az így kibővített string eredeti hossza x, s kibővítjük y hosszúságúra azzal, hogy az y-odik elemre hivatkozunk, akkor az x és y indexű elemek közti karakterek értéke nem garantált, az is véletlenszerű hogy mi lesz!

Számot stringgé természetesen a casting operátorok segítségével alakítunk:

```
#s@d = #c@a;  
#s@d = #C@a;
```

A **char** típusú értékek egyformán 4 karakter hosszú számstringgé vannak alakítva ilyenkor, melynek első helye szóköz a pozitív értékek esetén, illetve egy „-” negatív előjel a negatív számok esetén. Elvárom egy szám stringgé konvertálásakor ugyanis, hogy ne kelljen nekem vizsgálgatni azt a programban, hogy hát jajistenkém, ez vajon most épp milyen hosszú, s nekem kell emiatt „varázsolgatni”, hogy a helyiértékek szépen olvashatóan egymás alá kerüljenek... Francokat! Ezt intézze el a program. Úgy értem, maga a programNYELV, nem az a program amit én írok. Én a magam feladatának speciális problémáival akarok foglalkozni programírás közben, nem efféle mindennap felbukkanó, hétköznapi, piszlicsári problémákkal. Minden szám olyan hosszú stringbe legyen pakolva, amilyen hosszú kell az adott számtípusba beférő legislegnagyobb elképzelhető értéknek. Ha ugyanis annál sokkal kisebb számok is elegendőek nekünk, használjunk más számtartományt lefedő számtípust.

Ez a stringgé castolási lehetőség létezik a mau nyelvben az összes fixpontos számra. Íme a táblázat, hogy melyik típus hány karakteres stringgé alakul:

Numerikus típus	A keletkező string hossza
#c és #C, azaz unsigned és signed char	4
#i és #I, azaz unsigned és signed short int	6
#l és #L, azaz unsigned és signed int	11
#g és #G, azaz unsigned és signed long long	21

És mindegyik fenti esetben az van, hogy a string balelső karaktere szóköz, ha a szám pozitív (ez biztos így van minden unsigned típusnál), illetve egy „-” jel ha a szám netán negatív. A szám maga pedig jobbra igazítva helyezkedik el a stringben, azaz ezeket egymás alá kiírva, a helyiértékek is szépen egymás alatt lesznek. A „nem használt” helyiértékeken pedig szintén szóközök lesznek.

Lebegőpontos számok esetén, tehát a float, double és long double esetében a mau nyelv azt csinálja, hogy stringgé alakítás előtt mindegyiket átalakítja long double típusra, majd ezt alakítja át stringgé, olyanná, ami fix hosszúságú, 42 bájt hosszú, a string 21-es indexű pozícióján van a tizedespont, a nulladik pozíción szóköz van ha a szám pozitív, és egy '-' jel azaz mínuszjel ha a szám negatív, a vezető nullák helyén szóközök vannak, ellenben a törtrésznél az utolsó értékes számjegyet követő helyiértékek nem szóközökkel hanem nullákkal vannak kitöltve. A pontosság természetesen a szám nagyságától és a numerikus típus számbábrázolási tartományától függ.

Itt említem meg a mau nyelv egy rendkívül speciális operátorát, az indirekt-művelet-operátort. Ennek jele ez:

&|

Ez az operátor önmagában nem értelmezhető, hanem csak az utána következő stringkifejezéssel együtt. Ez az operátor ugyanis azt jelenti, hogy helyette azt az operátort kell végrehajtani, amit az utána következő stringkifejezés tartalmaz... Ez legjobban egy konkrét példából érthető meg. Íme:

```
#c@c=10;
"c=" ?c @c; /
#c@d=2;
"d=" ?c @d; /
#c@e=(@c)+(@d);
#c@f=(@c)-(@d);
"c+d=" ?c @e; /;
"c-d=" ?c @f; /;
#s@p="+";
#s@m="-";
#s@s="*";
"Meg1: " ?c (@c) + (@d); /;
"Meg: " ?c (@c) &|(@p), (@d); /;
"Ből: " ?c (@c) &|(@m) (@d); /;
"Szor: " ?c (@c) &|(@s) (@d); /;
```

E programocska eredménye:

```
c=10
d=2
c+d=12
c-d=8
Meg1: 12
Meg: 12
Ből: 8
Szor: 20
```

## 8. fejezet - Stringtömbök

Amint a numerikus típusú változóknak van tömb-párjuk, aképp a stringek számára is létezik a mai nyelvben tömblehetőség, azaz vannak stringtömbjeink is. Természetesen ezekből is 256 lehetőség adatik nekünk minden névtérben, a már unalomig ismerős névadási szabályok szerint, s ezeket is csak akkor használhatjuk, ha előbb memóriát foglalunk le nekik. A memória lefoglalásakor nem az számít, a leendő stringek külön-külön vagy összesen milyen hosszúak lesznek, nem kell ilyesmivel foglalkoznunk, mindössze azt kell megadnunk, maximum hány darab string lesz abban a nevű stringtömbben!

A stringtömbökre ugyanúgy a **#s** casting operátorral hivatkozunk mint a közönséges stringekre, mindössze ezek dupla sz(e/ö)gletes zárójelekkel indexelnek. Íme, miként is megy egy efféle memórialefoglalás, meg adattal feltöltés:

```
[@t[[4]]]; // lefoglalok területet 4 string részére egy t nevű tömbbe
#s@t="Ez egy sima string, nem tömb!\n"
// Feltöltöm értékekkel a tömböt
#s@t[[0]]="kutya";
#s@t[[1]]="macska";
#s@t[[2]]="vadbarom";
#s@t[[3]]="cica";
```

Mindez gondolom világos. Látható is a fenti példán, hogy a **@t** single stringváltozónak a világon semmi köze a szintén **@t** nevű stringtömbhöz!

A „t” nevű stringtömbnek korábban lefoglalt memóriaterületet ezen utasítással szabadíthatjuk fel:

```
[@t[[]]];
```

Ha ez a szintaxis talán fura is első pillantásra, érthetővé válik ha belegondolsz, hogy ez ugyanaz mint amikor memóriát foglalsz le neki, csak akkor megadod a számot (akár egy aritmetikai kifejezéssel) hogy mennyi stringnek foglalsz le területet. Most nem adsz meg semmit, akkor pedig nyilván volt már lefoglalva korábban terület, amit most szabadítasz fel!

Stringtömb egy tagjának szerepeltetése jobbérték-helyzetben:

```
#s@s=@t[[1]];
```

Azaz ebben semmi különös nincs, egyszerűen kettős sz(e/ö)gletes zárójelekkel kell indexelni.

Stringtömb egy tagját persze tovább indexelhetjük, hogy megkapjuk valamely karakterét:

```
#c@c=#s@t[[#i@i]][3];
#s@t[[1]][3]='@';
```

Ügyeljünk rá, hogy a fenti példákban mindenféleképpen kell a **#s** casting operátort használnunk, mert különben nem a string, hanem az unsigned char értékek közt próbál adatot keresgetni az interpreter, és syntax error-t dob nekünk!

## Stringtömbök névsorba rendezése

A mau nyelv olyan fejlett beépített szolgáltatásokkal is rendelkezik, megkönnyítendő a programozó életét, hogy egyetlen utasítással névsorba rendezhetjük a tömböt:

**QS t, 4;**

A fenti példában a QS a rendezőutasítás (amúgy ez a **Q**uick**S**ort rövidítése), és itt bukkan fel az első olyan eset, amikor egy változónál nem kell kitenni a név elé a **@** jelet: egyszerűen megadjuk hogy „t”, ami itt a rendezendő stringtömb neve. Tudniillik felesleges karakterszaporítás lenne kitenni a **@** jelet ezesetben, mert minek? Itt egyszerűen egy tömb neve kell szerepeljen, mert a **QS** utasítás semmi mást nem tud rendezni, CSAK tömböt! Mármint stringtömböt. Ezután pedig meg kell adni azt is, a tömb első hány darab elemét rendezi. Természetesen a **QS** utasítás mindkét paramétere lehet tetszőleges aritmetikai kifejezés. Valamint, ezek közé a vessző nem kötelező, csak opcionális szeparátorjel.

Na most, muszáj kitérni rá, a névsorbarendezés miféle sorrendi szabályokat követ. Mindenekelőtt elárulom, az összes string-utasítás feltételezi, hogy amennyiben ékezetes karakterekről is „szó van” a stringben, akkor azok az UTF-8 kódolási rendszer szerint vannak kódolva. Természetesen a rendezőrutin nem veheti figyelembe az UNICODE tábla mind a sok tízezer egzotikus karakterét, de legalábbis a magyar és német ékezetes karaktereket felismeri azért, s ezeket egy bizonyos (nekem tetsző, azaz okvetlenül helyes, logikus, értelmes és szép...) sorrendbe rendezi. Ez pedig NEM a „hivatalos” magyar sorrend, amit már picike általános iskolás koromban is k!b@\$ottnagy idiótaságnak tartottam!

Mert az bizony így hangzik, idézem a magyar Wikipédiából a „szabályzatot”:

**d) A magánhangzók rövid és hosszú változatát jelölő betűk (a – á, e – é, i – í, o – ó, ö – ő, u – ú, ü – ű) a kialakult szokás szerint mind a szavak elején, mind pedig a szavak belsejében azonos értékűnek számítanak a betűrendbe sorolás szempontjából. A magánhangzó hosszú változatát tartalmazó szó tehát meg is előzheti a rövid változatút.**

Na most e „szabályzattól” én hideglelést kapok, herótom és hányingerem van tőle, s azóta is hogy valamikor gyermekkoromban megtudtam hogy ez lenne a magyar „szabály” a névsorra, azóta is ha csak ez az eszembe jut, idegrángás környékez, és közepes súlyosságú agyérgörcs! Ez a rendezési sorrend nem más a szememben, mint a magyar kultúra SZÉGYENE, és az emberi logikátlanság diadala! Szerintem ezt csak valaki tökéletesen IQ-negatív idióta találhatta ki, valami génsérült, akit születése után rögvést azonnal fejre is ejtettek, s ez is csak akkor, miután a maradék eszét is elitta félig. Számomra már az i-i is teljesen külön betű, de az a-á aztán pláne! Engem ZAVAR egy szöszedetben ez a keveredés. Ha vége az a-val kezdődő szavaknak s jön egy á-val kezdődő, azt hiszem nincs is több az a-val kezdődő szavakból, s erre kiderül hogy ja de mégis! Megesküdtem rá, hogy az ÉN programnyelvemben nem ilyen zagyva összevissza ~~rend~~ OCSMÁNY RENDETLEN-SÉG lesz. KERÜL AMIBE KERÜL programozásra fordított időben, „szopásban”, munkában, kínlódásban, energiában, tárméretben és sebességlassulásban, de NEM ez lesz! Sőt: nálam jöjjenek előbb a NAGY betűk, aztán a kicsik, nem vegyesen! Azaz, efféle sorrendet akartam felállítani:

**AaÁáÄäBbCcDdEeÉéFf...** azt hiszem érthető.

Ez a rendezés semmiféleképp sem zavarhatja a nyelvem esetleges angol felhasználóit, mert nekik teljesen mindegy, az ő nyelvükben nincsenek ékezetes karakterek. Ugyanakkor e rend jó szerintem a németeknek is. Ettől természetesen bárki megírhatja a maga rendezőrutinját más szabályok szerint ha neki ez nem tetszik, de a mau nyelv ezt „alapból” azaz „default” támogatja.

Ugyanezen szabályok szerint történik a mau nyelvben a stringek összehasonlítása is, mondjuk az **if** utasításnál, de az összehasonlítások, feltételek, ugrások stb nem ennek a fejezetnek a témája.

## Stringek darabolása határolókarakter szerint

Stringtömbökkel függ össze az a mau utasítás is, ami egy stringet szétvagdál egy megadott határolókarakter mentén, és a rész-stringeket elhelyezi egy string-tömbben. Ezen utasítás szintaxisa:

```
||| S, t, c
```

vagy:

```
||| S, t;
```

Ez tehát az S stringet szétdarabolja úgy, hogy határolókarakternek a „c” karaktert tekinti. A részstringeket a „t” stringtömbbe pakolja, mindet nullabájttal lezárva. Az „S” egy stringkifejezés lehet, a „t” egy **#c** típusú kifejezés ami a stringtömb nevét határozza meg, a „c” meg természetesen szintén egy **#c** típusú aritmetikai kifejezés. A határolókarakter nem kerül letárolásra a részstringeknél! A „t” tömb régi tartalma elvész. Minden rész-string természetesen automatikusan lezáratik a határoló nullabájttal a végén.

Az f.inputfilesordb rendszerváltozóba berakja hogy hány darabra vágta a stringet, azaz hány eleme van a t tömbnek. Ezen érték a **?n** rendszerváltozóval kérdezhető le.

Ha az S hossza 0, semmit se csinál. Ezesetben nem törli a t tömböt sem.

A második esetben kötelező a pontosvessző az utasítás végére, és c=SPACE az alapértelmezés!

Egy példaprogram:

```
#!mau
#s@p="Ez egy sok szóból álló szöveg!";
"Szóközre darabolva:\n"
||| @p, t;
{| ?n;
?s #s@t[[]?n-?|]]; /;
|}
"A 'z' betűre darabolva:\n"
||| @p, t, z;
{| ?n;
?s #s@t[[]?n-?|]]; /;
|}
XX
```

Eredménye:

```
Szóközre darabolva:
Ez
egy
sok
szóból
álló
```

```
szöveg!
A 'z' betűre darabolva:
E
    egy sok s
óból álló s
öveg!
```

Persze, a példaprogram így is írható:

```
#!mau
#s@p="Ez egy sok szóból álló szöveg!";
"Szóközre darabolva:\n"
||| @p, t;
{| ?n;
?s #s@t[[]]; /;
|}
"A 'z' betűre darabolva:\n"
||| @p, t, z;
{| ?n;
?s #s@t[[]]; /;
|}
XX
```

Na most őszintén, Tisztelt Olvasóm, neked talán nem tetszik ez a szintaxis, hogy  
|||  
?

Mert szerintem ez nagyonis hüen, vizuálisan asszociálja a darabolást, mintha 3 függőleges késvágással metszenénk részekre valamit...

Ez pedig:

```
[[[]]]
```

igazán gyönyörűen szimmetrikus, esztétikus... A mau nyelv egészen egyszerűen SZÉP...

## 9. fejezet - Inkrementálás és dekrementálás

Aki ismeri a C és/vagy a C++ nyelveket, az biztos ismeri e nyelvek -- illetve ++ operátorait is, melyek eggyel csökkentik vagy növelik a változó értékét. Ilyesmi van a mau nyelvben is, természetesen, csak itt ezek bizonyos esetekben vonatkozhatnak akár egy teljes aritmetikai kifejezésre is!

Inkrementálásra példák:

```
#c++@c:2;
#c++@c;
```

Dekrementálásra példák:

```
#c--@c:2;
#c--@c;
```

Persze, nem csupán unsigned char értékekkel tehetjük meg ezt, amit most a #c casting operátor jelez, hanem bármely numerikus típussal. Sőt, ilyet is eljátszhatunk:

```
#s--@a; // A hosszából le kell vonnunk 1-et, mert másképp a file nevéhez tartozónak tartaná a 10-es karakterkódú
// sorvégjelet is.
```

A fenti példa egy konkrét mau programomból egy pici részlet. Világosan mutatja a komment, milyen hasznos tud lenni e lehetőség. Ugye, ez a dolog „szabályosan” azaz inkább hagyományosabban így nézne ki:

```
#s@a=(#s@a) - 1;
```

Na de minek írjuk ki kétszer a stringváltozó nevét... Meg minek hívjunk meg egy komplett, monstruózus aritmetikai kifejezés kiértékelő rutint, ami minden elkép-

zelhető rafináltságra fel van készülve, amikor úgyis tudjuk jóelőre, hogy amit beolvas, az a szám éppen pontosan 1 lesz és nem más, minek atombombával lőni verébre... Az előző megoldás sokkal gyorsabb, sőt maga a forráskód is rövidebb. Azaz, ez leahagyja a string utolsó karakterét.

E dekrementáló megoldás működik olyan stringeknél is, melyek egy stringtömbben vannak:

```
#s--@t[[1]];
```

Ilyesmit is csinálhatunk, azaz egy string egy karakterét is inkrementálhatjuk vagy dekrementálhatjuk:

```
#s++@s[2];
```

```
#s--@s[4];
```

Sőt egy stringtömbben levő string egy karakterét is (in/de)krementálhatjuk:

```
#s--@t[[1]][2];
```

```
#s++@t[[1]][2];
```

Na most a fenti példák mind megváltoztatják az eredeti (numerikus vagy string) változó tartalmát! Azonban, ha a dupla mínusz vagy plusz jel jobbjében szerepel, akkor semmiféle változótartalmat nem módosít, hanem egyszerűen kiszámolja az interpreter az ezen operátortól jobbra eső teljes aritmetikai kifejezés értékét, és ezen eredményt aztán csökkenti vagy növeli eggyel, attól függően hogy dekrementáló vagy inkrementáló operátor van előtte. Ezekből ráadásul egymás után többet is szerepeltethetünk:

```
#c@c="52cd"+"efghijk"[#l@ -- --k]
```

A fenti példában ugyebár a jobbjék egy stringkifejezés indexelésével kapott karakter. E stringkifejezés indexét a

```
#l@ -- --k
```

rész határozza meg.

A **#1** egy unsigned intet jelző casting operátor, majd utána a **@** jel azt mondja hogy egy változóról van szó. Ennek nevét a

```
-- --k
```

kifejezés határozza meg. Ez nem más mint a „k” karakter ASCII kódja, aminek értékéből 2 alkalommal vonunk le egyet, azaz összesen kettővel csökkentjük azt.

Itt említek meg egy rém speciális, stringekkel összefüggő utasítást is, nem mintha inkrementálás volna, de amiatt mert a szintaxisa az inkrementáló/dekrementáló utasításhoz hasonlít. Arról van szó, hogy ugyebár stringek karaktereinek az index-operátorral adhatunk értéket. Na most, tegyük fel van egy több karakterből álló stringünk, mondjuk az "abcdefgh", s ennek valahányadik - mondjuk 3-adik - pozíciójába beírunk valami értéket. S tegyük fel ez épp a 0 kódú karakter, ami a string végét jelzi normális esetekben... De ettől még a stringünk hossza, amit külön tárol el az interpreter, nem változik meg!

Az ilyesmi lehet hogy jó nekünk, de az is lehet hogy nem, mert szeretnénk, ha ilyenkor az interpreter tudomásul venné, hogy mostantól a string hossza nem annyi mint máskor, s megjegyezné az új hosszat. Ezért ilyen „műtétek” után kiadhatjuk a

```
#s!@k;
```

parancsot, természetesen a „k” helyén bármi állhat ami érvényes stringnév. Itt egy kis példaprogi ilyesmire:

```
#s@s="ABCDEF"
```

```
?s @s; /;
```

```
"hossza: " ?l !#s@s; /;
```

```
"Most a #s@s[3]-ba 0-t rakok!\n"
```

```
#s@s[3]=0;
```

```
?s @s; /;
```

```
"hossza: " ?l !#s@s; /;
```



```
"Most ezt jön: #s!@s\n"
#s!@s;
?s @s; /;
"hossza: " ?l !#s@s; /;
```

Eredménye:

```
ABCDEF
hossza: 6
Most a #s@s[3]-ba 0-t rakok!
ABC
hossza: 6
Most ezt jön: #s!@s
ABC
hossza: 3
```

Ugyanez igaz olyan stringekre is, amik egy stringtömb valahányadik elemei:

```
[@t[[4]]]; // lefoglallok területet 4 string részére egy t nevű tömbbe
// Feltöltöm értékekkel a tömböt
#s@t[[0]]="kutya";
#s@t[[1]]="macska";
#s@t[[2]]="vadbarom";
#s@t[[3]]="cica";
"Kiíratom a 4 elemű stringtömb mindegyik értékét:\n"
#i@i=0; { | 4 ?i @i; ". = " ?s @t[[#i@i]]; /; #i++@i; |}
"A tömb 2. tagja:\n"
?s @t[[2]]; /;
"hossza: " ?l !#s@t[[2]]; /;
"Most a #s@t[[[2]][3]-ba 0-t rakok!\n"
#s@t[[2]][3]=0;
?s @t[[2]]; /;
"hossza: " ?l !#s@t[[2]]; /;
"Most ezt jön: #s!@t[[2]]\n"
#s!@t[[2]];
?s @t[[2]]; /;
"hossza: " ?l !#s@t[[2]]; /;

"Felszabadítom a stringtömb memóriaterületét.\n"
[@t[[[]]]];
```

Eredménye:

```
Kiíratom a 4 elemű stringtömb mindegyik értékét:
0. = kutya
1. = macska
2. = vadbarom
3. = cica
A tömb 2. tagja:
vadbarom
hossza: 8
Most a #s@t[[[2]][3]-ba 0-t rakok!
vad
hossza: 8
Most ezt jön: #s!@t[[2]]
vad
hossza: 3
Felszabadítom a stringtömb memóriaterületét.
```

Végeredményben persze tényleg tekinthetjük ezt az utasítást speciális értelemben vett dekrementáló utasításnak, hiszen általa csökken(het) a string eredeti hossza. Ezen utasítás szintaxisát pedig megjegyezhetjük könnyen abból, hogy itt a felkiáltójel szerepel, s ugyanazt a jelet használjuk unáris operátorként a string hosszának lekérdezésére is. Mert a mau nyelv igenis logikus, legfeljebb ez nem látszik rajta az első pillantásra!

## 10. fejezet - Összevont utasítások

### Egyenlőségjellel összevont utasítások

E címszó alatt többnyire olyasmiket értek, melyek teljesen ismerősek lesznek a C és C++ nyelvet tudók számára: a

**`+=, -=, *=, /=`**

utasításokat. Ezek egészen hasonlóak is e C nyelvű társaikhoz, muszáj is hogy úgy legyen, mert bevallom az ötletet ezekhez épp a C nyelvből csórtam én is. Például, itt van, egy string egy karakterével miként alkalmazzuk ezt:

**`#s@s[#i@i]+=2;`**

Természetesen a numerikus értékekkel is megtehetjük ezt, single és tömbváltozókkal egyaránt:

**`#c@t[#i@i]-=2;`**

**`#c@t-=2;`**

Magától értetődően ugyanez igaz a **`*`** és **`/`** utasításokra is. Működnek a

**`~~=`**

**`><=`**

**`<<=`**

**`>>=`**

**`&&=`**

**`||=`**

összevont utasítások is, a single változókra, a stringek karaktereire, a stringtömbök stringjeinek a karaktereire, és a numerikus tömbök értékeire is — de természetesen csak a fixpontos számokra, azaz a **`#f`**, **`#d`**, **`#D`** típusokra **nem**. A stringek karaktereire amiatt működnek, mert azok ugye unsigned char értékek, s emiatt fixpontos számoknak minősülnek.

### Előjelváltás

Előjelváltásra természetesen felhasználhatjuk a mínusz 1-el való szorzást, de van rá sokkal gyorsabb módszer is. Ez a **`±`** utasítás. Ez az a karakter, aminek UTF-8 kódja a 194,177 bájt páros. Ennek szintaktikája:

**`±x k`**

ahol az **`x`** a casting operátoroknál ismert előjeles típusok valamelyike lehet, tehát C, I, L, F, d, D, a **`k`** pedig a változó nevét meghatározó unsigned char típusú aritmetikai kifejezés. Ez a művelet csak a változók nulladik indexű mezőjének képes megváltoztatni az előjelét!

Az utasítás megadható így is:

**`±(y) k`**

ez esetben az **`y`** a casting operátor karakterét meghatározó unsigned char típusú aritmetikai kifejezés.

A **`±`** és a casting operátort meghatározó „x” között nem állhat whitespace karakter!

## Gyorsfüggvények

Bár a mau egy szkriptnyelv, s azoknál nem a gyorsaság a legfontosabb szempont, tudniillik mert sebességkritikus alkalmazásokra nem szkriptnyelveken szoktak programokat írni, de ELŐFORDULHAT, hogy azért számít a sebessége a programoknak! Még ha maga az egész program nem is sebességkritikus, de egyes részei - például valami sokszor végrehajtandó ciklus belsejében - már lehetnek időérzékenyek. Ezért a mau nyelv biztosít lehetőséget némely műveletek jelentős felgyorsítására. Ezek a „gyorsfüggvények”. Ezek olyankor használhatóak, ha a **+, -, \*, /** műveletek valamelyikét akarjuk elvégezni két olyan single numerikus változó közt, melyeknek a típusa megegyezik, továbbá mindegyiknek csak a nulladik mezőjén végezhető művelet így. A gyorsfüggvény nem utasítás, hanem ténylegesen egy függvény, s alakja a következő:

**xxmab**

ahol az x a casting operátornál megadható valamelyik típus karaktere, tehát a **cCiIlLgGfdD** valamelyike, az „m” a műveleti jel, az „a” és a „b” pedig a változókat azonosító karakterek. Egyik karakter közt se lehet itt whitespace! Azért nem, mert minden beolvasandó bájtt lassítja a programfutást, márpedig a „gyorsfüggvény” direkt azért van kitalálva, hogy minél gyorsabb legyen! Továbbá, az „a”-val és „b”-vel jelölt karakterek a fenti példában CSAK karakterek lehetnek nem kifejezések, és teljesen mindegy miféle bájtok, mindenképp az azon a helyen álló bájtt ASCII kódja lesz értelmezve, akármi is legyen az!

Használatára egy példa:

**#d@p=(@p)+#d/sn**

E fenti sor egyenértékű a következővel:

**#d@p=(@p)+(@s)/(@n)**

A **▣** karakter az az UTF-8 karakter, aminek bájt kódja a 194,164 bájtsorozat.

## Gyorsparancsok

A gyorsparancs, az olyasmi mint a gyorsfüggvény, épp csak nem ad vissza semmi-féle értéket, hanem egyszerűen csinál valamit „gyorsan”.

Efféle gyorsparancsok:

Inkrementálás:

**++CX**

A fenti parancs egyes karakterei közt nem állhat whitespace. A „c” a casting operátoroknál megadható numerikus típusokat reprezentáló valamely karakter lehet, tehát a c,C,i,I,l,L,g,G,f,d,D karakterek valamelyike. Az x pedig a névtér valamely single numerikus változóját azonosító KARAKTER. E változó nulladik mezőjének értékét fogja a gyorsparancs megnövelni 1-el, azaz ez **inkrementáló** parancs.

**+1cx**

Ez a parancs megegyezik az előző paranccsal, a **++CX** -val.

**--CX**

Mint a **++cx** parancs, csak a fenti gyorsparancs **dekrementál**, azaz CSÖKKENTI a változó értékét 1-el.

**-1cx**

Teljesen megegyezik a fenti parancs a **--cx** paranccsal.

**+2cx**

E fenti parancs szinte ugyanaz mint a **++cx** parancs, csak ez nem 1-el hanem rögvest 2-vel inkrementálja a megfelelő változót.

Fontos tudni a fenti parancsoknál, hogy az „x” itt minden esetben egy konkrét bájt ASCII kódját jelenti, tehát ennek a helyén mondjuk a „3” számjegy nem a 3-as indexű változót jelenti, hanem az 51-es indexűt, mert a „3” karakter ASCII kódja 51...

## 11. fejezet - Kiiratás

A közismert „Hello World!” vagy magyarul a „Helló Világ!” program így néz ki mau nyelven:

```
"Hello World!\n"
```

Illetve:

```
"Helló Világ!\n"
```

És ezek után még valaki azt meri mondani, hogy a mau nyelv BONYOLULT?! Hát már hogy is lehetne ezt ennél egyszerűbben megoldani?!

A fentiek gondolom mindenkinek világosak: A mau nyelvben az (angol) idézőjel maga is egy utasítás, ami egyszerűen mindent kiír a következő idézőjelig. Kivéve, ha azon idézőjel előtt egy **\** jel van, mert akkor az idézőjelet is kiírja. Azaz, megadhatóak itt is Escape szekvenciák, mint a stringeknél. Látjuk is az egyiket a fenti (rémségesen komplex...) programokban: a végén a **\n** jel jelenti az „újsor” karaktert.

A fentieket így is írhattuk volna azonban:

```
"Helló Világ!"; /;
```

Ennek eredménye ugyanaz lett volna. A **/** is egy utasítás ugyanis a mau nyelvben, és annyit csinál, hogy egy újsor karaktert ír ki. Ha ezt egy olyan utasítás után adjuk ki, mely aritmetikai kifejezésre végződik, akkor ne felejtünk el pontosvesszőt tenni az előző utasítás végére, nehogy az interpreter a per jelünket osztás-jelként akarja értelmezni...

Létezik egy speciális Escape szekvencia is, ami képernyőtörlést eredményez, ez mau specialitás:

**\z : Képernyőtörlés. Egyenértékű a következő C nyelvű utasítással:**

```
printf("\x1b[2J\x1b[H");
```

Azaz egy konkrét példán bemutatva:

```
"\z"
```

A fenti printf utasításnál bemutatott Escape-szekvenciasorozat valószínűleg jó minden szabványos Linux-terminálon, de annak érdekében hogy a mau interpretert hozzáigazíthassuk (forrásból fordítás esetén...) valamely egzotikus terminálhoz vagy terminálemulátorhoz, e szekvencia átállítható (fordítás előtt) a vz.h fájl ezen sorának megfelelő átírásával:

```
#define CLEARSCREENSTRING "\x1b[2J\x1b[H"
```

Alternatív kiíró utasításként alkalmazhatjuk a **?** jelet is:

```
? "Helló Világ!\n"
```

Ennek így persze nem sok értelme van, mert minek tegyük oda a kérdőjelet, ha nem muszáj! Egyedül amiatt létezik ez, mert a kérdőjel segítségével irathatunk ki másfajta adatokat is, amennyiben a **?** után odaírjuk a következő karakternek azt a karaktert, amit a **#** casting operátor után is szoktunk írni típusjelölőként. Azaz:

A **?c** utasítás egy unsigned char értéket ír ki, mint számot.

A **?C** utasítás egy signed char értéket ír ki, mint számot.

A **?i** utasítás egy unsigned short int értéket ír ki, mint számot.

És így tovább.

Ezenfelül, a **?k** utasítás az utána következő unsigned char értéknek megfelelő karaktert írja ki, MINT KARAKTERT tehát, azaz nem mint a bájt számértékét! Ugyanez lesz az eredménye az egyedülálló **?** karakterből álló utasításnak, vagyis annak, ha a kérdőjelet egy whitespace karakter követi.

Ügyeljünk a szintaxisra! Ugyanis:

Ez:

```
? "65cdefghijk" ; /;
```

kiírja magát az egész stringet.

Ez azonban:

```
? ("65cdefghijk") ; /;
```

azt a KARAKTERT írja ki, aminek az ASCII kódját az utána következő unsigned char típusú aritmetikai kifejezés határozza meg! Az egy zárójeles kifejezés. Azon belül egy konstans string van. Ezt az interpreter kötelességszerűen átalakítja unsigned char értékké, ami úgy megy, hogy kiolvassa, miféle numerikus konstans van a string első valahány bájtján. Ott „65” van, ezt adja vissza. Tehát az a karakter íródik ki, aminek az ASCII kódja 65, ez pedig az „A” karakter.

A képernyőtörléshez nem muszáj idézőjelek közé zárt Escape szekvenciával kínlódnunk, elég ennyit írni:

```
?z;
```

Fontos megemlíteni, hogy e kérdőjellel való kiíratások csak a stringekre és a fixpontos numerikus típusokra vonatkozóan pontosak! A lebegőpontos számok esetén, azaz a **?f**, **?d**, **?D** utasításoknál ugyan kiír valami körülbelüli értéket, de ezt jobb ha nem használjuk, azaz ezen utasítások léteznek ugyan, de inkább csak tesztelési-debuggolási lehetőség gyanánt. Lebegőpontos számok kiíratását mindenkinnek magának kell megoldania, olyan pontossággal és formátumban, ahogyan jónak látja. A mau nyelv valamelyik következő verziójához feltehetőleg

készíték majd egy printf-szerű kiíró függvényt, ami tud majd valami effélét, de egyelőre ez egy nem implementált funkció még.

Továbbá: A casting operátoroknál mint tudjuk, megadható a castolási kódot jelző karakter aritmetikai kifejezésként, ha a **#** karaktert közvetlenül követi egy zárójel. Ugyanez eljátszható a **?** esetében is:

**?(valamilyen\_kifejezés) Amit\_ki\_akarunk\_iratni**

A mau nyelv beépített lehetőséget biztosít a Linux konzolra történő SZÍNES kiíratásra is! Szerepel ugyanis a megfelelő include fájlban e néhány direktíva:

```
#define COLOR_RED          "\x1b[31m"
#define COLOR_GREEN        "\x1b[32m"
#define COLOR_YELLOW       "\x1b[33m"
#define COLOR_BLUE         "\x1b[34m"
#define COLOR_MAGENTA      "\x1b[35m"
#define COLOR_CYAN         "\x1b[36m"
#define COLOR_WHITE        "\x1b[37m"
#define COLOR_BRIGHT       "\x1b[01m"
#define COLOR_UNDERLINE     "\x1b[04m"
#define COLOR_RESET         "\x1b[0m"
#define COLOR_BLACK        "\x1b[30m"

#define BACKGROUND_COLOR_DEFAULT "\x1b[49m"
#define BACKGROUND_COLOR_BLACK  "\x1b[40m"
#define BACKGROUND_COLOR_RED    "\x1b[41m"
#define BACKGROUND_COLOR_GREEN  "\x1b[42m"
#define BACKGROUND_COLOR_YELLOW "\x1b[43m"
#define BACKGROUND_COLOR_BLUE   "\x1b[44m"
#define BACKGROUND_COLOR_MAGENTA "\x1b[45m"
#define BACKGROUND_COLOR_CYAN   "\x1b[46m"
#define BACKGROUND_COLOR_LIGHTGRAY "\x1b[47m"
#define BACKGROUND_COLOR_DARKGRAY "\x1b[100m"
#define BACKGROUND_COLOR_LIGHTRED "\x1b[101m"
#define BACKGROUND_COLOR_LIGHTGREEN "\x1b[102m"
#define BACKGROUND_COLOR_LIGHTYELLOW "\x1b[103m"
#define BACKGROUND_COLOR_LIGHTBLUE "\x1b[104m"
#define BACKGROUND_COLOR_LIGHTMAGENTA "\x1b[105m"
#define BACKGROUND_COLOR_LIGHTCYAN "\x1b[106m"
#define BACKGROUND_COLOR_WHITE "\x1b[107m"
```

Ezek segítségével a megfelelő színkódokat a mau interpreter újrafordítása során könnyen hozzá tudjuk igazítani az aktuális terminál színértelmezéséhez. E színekre pedig a következő utasításokkal hivatkozhatunk:

```
Sd // innentől kezdve a default (azaz alapértelmezett) színnel ír (COLOR_RESET)
Sv // világosság (bright) (COLOR_BRIGHT)
Su // aláhúzás (COLOR_UNDERLINE)
Sr // vörös (COLOR_RED)
Sg // zöld (COLOR_GREEN)
Sy // sárga (COLOR_YELLOW)
Sb // kék (COLOR_BLUE)
Sm // magenta (COLOR_MAGENTA)
Sc // türkiz (COLOR_CYAN)
Sw // fehér (COLOR_WHITE)
Sf // fekete (COLOR_BLACK)
```

A háttérszínek állítása:

```
Bd // alapértelmezett háttérszín (BACKGROUND_COLOR_DEFAULT)
Bf // fekete (BACKGROUND_COLOR_BLACK)
Br // vörös (BACKGROUND_COLOR_RED)
Bg // zöld (BACKGROUND_COLOR_GREEN)
Bs // sárga (BACKGROUND_COLOR_YELLOW)
Bb // kék (BACKGROUND_COLOR_BLUE)
Bm // magenta (BACKGROUND_COLOR_MAGENTA)
Bc // türkiz (BACKGROUND_COLOR_CYAN)
Bl // sötétszürke (BACKGROUND_COLOR_DARKGRAY)
BL // világosszürke (BACKGROUND_COLOR_LIGHTGRAY)
```

```
BR // világospiros (BACKGROUND_COLOR_LIGHTRED)
BG // világoszöld (BACKGROUND_COLOR_LIGHTGREEN)
BY // világossárga (BACKGROUND_COLOR_LIGHTYELLOW)
BB // világoskék (BACKGROUND_COLOR_LIGHTBLUE)
BM // világosmagenta (BACKGROUND_COLOR_LIGHTMAGENTA)
BC // világostürkiz (BACKGROUND_COLOR_LIGHTCYAN)
Bw // fehér (BACKGROUND_COLOR_WHITE)
```

A színeket ráadásul meg lehet adni indirekt módon is, így:

Írásszín esetén:

**S(x)**

Háttérszín esetén:

**B(x)**

Ahol az „x” egy unsigned char típusú aritmetikai kifejezés, és az értéke (eredménye) a megfelelő karakter kell legyen, amit fentebb leírtam az írásszín illetve a háttérszín esetén, tehát ami amúgy következne a parancs **S** illetve **B** karaktere után. Fontos tudni, hogy ezen esetekben NEM állhat semmiféle whitespace karakter az **S** vagy a **B** karakter és a nyitózároljel között!

Példaprogram a színekre:

```
Bw; "Szines!"; Bf; "Másik!\n"
Br; "Szines!"; Bf; "Másik!\n"
BR; "Szines!"; Bf; "Másik!\n"
Bg; "Szines!"; Bf; "Másik!\n"
BG; "Szines!"; Bf; "Másik!\n"
By; "Szines!"; Bf; "Másik!\n"
BY; "Szines!"; Bf; "Másik!\n"
Bb; "Szines!"; Bf; "Másik!\n"
BB; "Szines!"; Bf; "Másik!\n"
Bb; "Szines!"; Bf; "Másik!\n"
Bm; "Szines!"; Bf; "Másik!\n"
BM; "Szines!"; Bf; "Másik!\n"
Bc; "Szines!"; Bf; "Másik!\n"
BC; "Szines!"; Bf; "Másik!\n"
Bf; "Szines!"; Bf; "Másik!\n"
Bl; "Szines!"; Bf; "Másik!\n"
BL; "Szines!"; Bf; "Másik!\n"
Bd; "Szines!"; Bf; "Másik!\n"
; /;
Bw; "Szines!"; Bd; "Másik!\n"
Br; "Szines!"; Bd; "Másik!\n"
BR; "Szines!"; Bd; "Másik!\n"
Bg; "Szines!"; Bd; "Másik!\n"
BG; "Szines!"; Bd; "Másik!\n"
By; "Szines!"; Bd; "Másik!\n"
BY; "Szines!"; Bd; "Másik!\n"
Bb; "Szines!"; Bd; "Másik!\n"
BB; "Szines!"; Bd; "Másik!\n"
Bb; "Szines!"; Bd; "Másik!\n"
Bm; "Szines!"; Bd; "Másik!\n"
BM; "Szines!"; Bd; "Másik!\n"
Bc; "Szines!"; Bd; "Másik!\n"
BC; "Szines!"; Bd; "Másik!\n"
Bf; "Szines!"; Bd; "Másik!\n"
Bl; "Szines!"; Bd; "Másik!\n"
BL; "Szines!"; Bd; "Másik!\n"
Bd; "Szines!"; Bd; "Másik!\n"
"Indirekt:\n"
#c@d=d;
#c@c=w;
B(@c); "Szines!"; B(@d); "Másik!\n"
#c@c=r;
B(@c); "Szines!"; B(@d); "Másik!\n"
```



## 12. fejezet - Vermek

A mau nyelv fontos részét képezik a verem. Többféle verem is létezik e nyelvben, itt most azokról írok, amelyeket a programozó teljesen a maga tetszése szerint kezelhet. Ezekből természetesen szintén 256 -féle létezik egy névtérben, nevükre ugyanazon szabályok érvényesek, mint a közönséges változókra. A vermet a legelső használata előtt természetesen el kell látni megfelelő mennyiségű memóriával, azaz memóriát kell lefoglalni neki. Ennek mértékét bájtban kell meghatározni, s ez így megy:

```
[@x[2000]] // 2000 bájtot lefoglalunk az „x” nevű verem számára
```

A veremnek lefoglalt memóriaterületet fel is szabadíthatjuk, ekkor persze minden elvész ami benne volt:

```
[@x[]] // Törölöm a vermet
```

Egy verembe bármilyen típusú numerikus adatot belerakhatunk, STRINGET AZONBAN NEM. (Elképzelhető, hogy erre is lesz valami módszer a nyelv későbbi kiadásában. Most azonban még nincs).

Adatnak verembe rakása esetén tudnia kell az interpreternek, milyen típusú a bele rakandó adat, ezekből ilyenkor a balérték-szerepben levő veremre a szokásos casting operátorokkal kell hivatkozni, s azt hogy veremről van szó, a két csukó sz(e/ö)gletes zárójel jelzi, azzal, hogy nincs köztük semmi se megadva. Azaz:

```
#c@x[] = 4      // Beteszünk az „x” nevű verembe 4-et  
#c@x[] = 5      // Beteszünk az „x” nevű verembe 5-öt  
#c@x[] = 18     // Beteszünk az „x” nevű verembe 18-at  
#c@x[] = 20     // Beteszünk az „x” nevű verembe 20-at  
#i@x[] = $fce2 // Beteszünk a verembe egy 2 bájtos unsigned short int hexa számot is, ez a  
decimális 64738
```

Jobbértékként szerepeltetve, például kiíratásnál:

```
?c @x[] ; / // Itt kiírom a hexa fce2 2 bájtja közül a felső bájtot, értéke 252  
?c @x[] ; / // Itt kiírom a hexa fce2 2 bájtja közül az alsó bájtot, értéke 226  
?c @x[] ; / // Kiírom a 20-at  
?c @x[] ; / // Kiírom a 18-at
```

Persze nem csak unsigned char értéket olvashatunk ki belőle:

```
?i @x[] ;
```

## 13. fejezet - Mau rendszerváltozók és rendszerfüggvények

A mau nyelv rendelkezik bizonyos „rendszerváltozókkal”, melyek szerepelhetnek aritmetikai kifejezésekben. Ezeknek 2 fő csoportjuk van: az egyiknél nem kell megadni külön explicit módon a típust, mert amúgy is tudja az interpreter, hogy mi a típus - pontosabban, ezek mind unsigned int (tehát **maul\_1**, vagyis **#1**) értékként érkeznek vissza „hozzánk”. A másik csoportba azok tartoznak, ahol ki kell tennünk a hívás során a megfelelő casting operátort mindenféleképpen.

Az első csoportba tartozó változókat a

**?x**

szintaxis szerint kérdezhetjük le, ahol az „x” helyén valami karakter áll, lehet épp maga az „x” is, de egy csomó minden más is, mindegyik különböző rendszer-változókat reprezentál. Itt egy lista arról, melyik mit jelent, de e lista a magyarázat szempontjából nem teljes. Egyrészt, ezek száma minden bizonnyal jócskán bővül majd a mau nyelv későbbi kiadásában, másrészt, azok sincsenek itt mind elmagyarázva, amelyek már benne vannak e nyelv jelen változatában, s amiatt, mert számos ezek közül olyasmire vonatkozik, ami még e könyv előző fejezeteiben nem lett megemlítve, márpedig azon ismeretek nélkül nem érthető úgyse e rendszerváltozó jelentése. Ilyenek például bizonyos olyan rendszerváltozók, melyek egyes ciklusfajtákkal vagy más vezérlési szerkezetekkel függnek össze. Ezek a maguk helyén, a könyv későbbi fejezeteiben lesznek elmagyarázva. Ezeknek itt csak a jelét írom le, de magyarázat helyett csak egy nagykötojelet írok (ezt: „—”), ezzel jelzem hogy létezik ilyen nevű rendszerváltozó, de az máshol lesz elmagyarázva. Ha pedig a magyarázat helyett ezt látod: „...”, az azt jelenti, hogy az a változó részletesebb magyarázatot igényel, s így a táblázat után külön lesz elmagyarázva a jelentése, de még ebben a fejezetben.

Szóval íme a „nem teljes” lista:

## **Alapértelmezetten #1 (mau\_l) értéket visszaadó rendszerfüggvények**

<b>?_k</b>	—
<b>??</b>	—
<b>?!</b>	—
<b>?.</b>	—
<b>?-</b>	—
<b>?0</b>	—
<b>?1</b>	<b>Az épp végrehajtott mau utasítás kódjának első karaktere</b>
<b>?2</b>	<b>Az épp végrehajtott mau utasítás kódjának második karaktere</b>
<b>?a</b>	<b>A mau programnak megadott parancssori paraméterek száma (argc)</b>
<b>?b</b>	<b>— (A BRAINFUCK flag értéke, ezt is lásd később).</b>
<b>?d</b>	—
<b>?e</b>	—
<b>?f</b>	—
<b>?h</b>	<b>A mau programfájl nevének hossza</b>
<b>?m</b>	—
<b>?n</b>	—
<b>?p</b>	<b>A mau program számára lefoglalt programmemória hossza</b>
<b>?s</b>	...
<b>?u</b>	—
<b>?v</b>	...
<b>?x</b>	...
<b>?D</b>	—
<b>?E</b>	—
<b>?F</b>	—
<b>?H</b>	<b>A betöltött (épp végrehajtott) mau programfájl hossza (bájtban).</b>

- ?M** A mau interpreter verziószáma.
- ?P** Az aktuális (épp végrehajtott) utasítás címe a programban.
- ?S** ...
- ?T** ...
- ?V** Az a karakter, amit a rendszer a sorvég jelölésére használ. Linux alatt ez tehát 10.
- ?X** ...
- ?Y** —
- ?I** —

Részletesebb magyarázatok:

**?s** Ez a függvény visszaadja (egész számra kerekítve) a szabad memória százalékát a rendszerünkben. Ezt például az e dokumentáció egy későbbi fejezetében bemutatott calc.mau programmal könnyen kiirathatjuk efféleképpen:

```
vz@Csiszilla /Releases/2014/U/Common/vz/MAU=>./mau calc.mau
mau> ?l ?s; "%"; /
95%
```

Na most, a visszakapott értékről jó ha tudjuk hogy nemcsak hogy egy egész számra kerekített érték, de az is kérdéses, miként értelmezzük. Ugyanis a jelenlegi operációs rendszerek memóriakezelése rém összetett. Például, szabad memóriának tekintjük-e azt, ami nem okvetlenül kell a programoknak, de azért a rendszer lefoglalja cache-nak azaz gyorsítótárnak, ám ha valami új alkalmazást indítunk aminek memóriára van szüksége, akkor ezen cache egy részét felszabadítja?! Még még rengeteg efféle kérdés felmerül ezzel kapcsolatban. Én a számításra a következő képletet alkalmaztam:

```
szabadmemoriaszazalek=((double)MemFree+(double)Cached)/(double)MemTotal)*100;
```

Tapasztalataim szerint ez meglehetősen jól visszaadja épp azt az értéket amit a „free” parancs ad ki nekünk. Mindenesetre, valamiféle támpontot azért ad ez az érték arra vonatkozóan, mennyire „elfoglalt” a masinánk.

**?T** Ez az összes memória mennyiségét adja vissza, kilobájtban, pont azt az értéket tehát amit így nevez a „free” parancs is. Ez valamivel kevesebb mint a rendszerünkbe installált összes RAM mérete, úgy tudom amiatt mert ebből lefoglal magának valamennyit állandó használatra a kernel. Ha ez nem igaz és hülyeséget írtam itt, kérem küldjön nekem erről „bugreportot” valaki e témában tájékozott szaki!

**?v** Ezt paraméterrel kiegészítve kell használni, így:

**?v x**

Ahol az „x” egy veremváltozó neve. (azaz az a karakter ami a @ jel után adandó meg). Természetesen az x lehet tetszőleges aritmetikai kifejezés is. A **?v** rendszerváltozó értéke akkor lesz 0, ha a verem üres, azaz nincs benne semmi, ellenkező esetben a visszatérési érték 1.

**?x** Paraméterrel együtt kell használni:

**?x L**

ahol az L egy unsigned char típusú aritmetikai kifejezés lehet. Az L-edik logflag értékét adja vissza. A mau interpreter ugyanis rengetegféle logolási lehetőséget támogat. A részletekkel kapcsolatban lásd azt a fejezetet, melynek címe: „A logolás egy mau programban”.

**?S** Ezt is paraméterrel együtt kell használni, hasonlóan mint a **?v** rendszerváltozót:

**?S x**

ahol az „x” itt is egy olyan (unsigned char típusú) aritmetikai kifejezés, mely egy veremváltozó nevét határozza meg. A rendszerváltozó visszaadja a verem számára lefoglalt (maximális nagyságú) memóriaterület méretét (bájtban).

**?X**

A mau interpreter a log üzeneteket egy „**stdlog**” nevű fájlba írja. Ez azonban átállítható, lehet ugyanis az **stderr** is vagy az **stdout**. Na most a **?X** rendszerfüggvény értéke 1, ha **stdlog==stderr**, és 0, ha **stdlog==stdout**. Minden egyéb esetben az értéke 2. További részletekért olvasd el azt a fejezetet melynek címe: „A logolás egy mau programban”.

## Explicit módon casting operátorral jelölt rendszerfüggvények

**Azon rendszerváltozók, melyek esetén meg kell adni a casting operátort is, a következő módon hívhatóak:**

**?#x S esetleges\_paraméterek**

Ahol az x természetesen a változó által visszaadott érték típusára utaló karakter, az S pedig egy string(kifejezés), ami tulajdonképpen a rendszerváltozó nevét határozza meg. Ez az „S” megadható természetesen konstans formában is, így:

**"név"**

Ez a fajta rendszerváltozó-csoport már nem annyira rendszer„változó”, hanem sokkal inkább afféle „beépített rendszerfüggvény”. Mindenesetre a határ a két fogalom közt a mau nyelvben meglehetősen homályos.

A jelen kiadásban megvalósított efféle rendszer(változók/függvények): (A leírásnál a „—” jelentése ugyanaz mint fentebb, azaz máshol lesz elmagyarázva a függvény/változó jelentése):

### #c értéket visszaadó rendszerfüggvények

**?#c "c" – Lásd később**

**?#c "r" – Lásd később, a regexpről szóló fejezetben.**

**?#c "8" x**

ahol az x egy unsigned char típusú aritmetikai kifejezés. E függvény visszaadja azt a számot, mely megmondja, hogy amennyiben az x egy UTF-8 kódolású karaktert jelentő bájt sorozat első karaktere volna, akkor ahhoz a karakterhez összesen hány bájt tartozik. Ez természetesen minden ASCII karakter esetén 1. Példa a használatára:

**"a " ?c ?#c "8" a; /;**

**"á " ?c ?#c "8" #c"á"; /;**

**„ „ ?c ?#c "8" #c"„"; /;**

Eredménye:

**a 1**

**á 2**

**„ 3**

**?#c "P" index**

ahol az „index” egy unsigned int érték. E függvény visszaad egy (unsigned char típusú) bájtot, ami az épp végrehajtott mau program index-edik bájtja. Ha így hívjuk meg:

**?#c "P" ^ index**

akkor az épp végrehajtott mau függvényt hívó programból olvassa ki a megfelelő index-edik bájtot. Ezzel a függvénnyel lehet például olyan programot írni, mely kilistázza önmaga kódját vagy annak egy részletét. Lesz is e könyvben később erre konkrét példa bemutatva.

**?#c "S" S**

ahol az S egy stringkifejezés. Ez az S egy csatolási pontot (mount-pontot) kell meghatározzon. A függvény visszaadja az ahhoz a ponthoz csatolt fájlrendszeren levő szabad hely százalékos mennyiségét. Legkönnyebben lekérdezhetjük ezt az ezen dokumentáció egy későbbi fejezetében ismertetett **calc.mau** nevű kis kalkulatorprogramot elindítva, s beleírva utasításként efféle sorokat:

vz@Csizsilla /Releases/2014/U/Common/vz/MAU=>./mau calc.mau

mau> ?c ?#c "s" "/"; "%"; /

40%

mau> ?c ?#c "s" "/Mount/Zene"; "%"; /

79%

mau> ?c ?#c "s" "/Releases/2014/Mount/Zene"; /

79

mau> ?c ?#c "s" "/Mount/Lubuntu"; /

36

**?#c "(")**

A fenti függvény addig vár, amíg le nem ütünk egy karaktert a konzolon. Tulajdonképpen a standard inputról olvas be egy karaktert, s annak a kódját adja vissza. Azaz ezt kell használnunk, ha a mau programmal egy pipe-on keresztül küldött adatfolyamot akarunk feldolgozni.

**?#c "{!" – Lásd később**

**?#c "L?" – Lásd később**

**?#c "EXE" – Lásd később**

## **#I értéket visszaadó rendszerfüggvények**

**?#l "[" – Lásd később**

Az épp végrehajtott mau függvény sorszámát adja vissza, lásd részletesebben később.

**?#l "T#" – Lásd később**

**?#l "FSIZE"**

Visszaadja az „F” struktúra méretét bájtban. Ez tulajdonképpen csak nekem kell vagy általánosabban szólva a mau nyelv fejlesztőinek — az F ugyanis egy nagyon fontos struktúra a mau programnyelvben, aminek kb mindenhez köze van.

**?#l "PGMSIZE"**

Visszaadja az „PGM” objektum méretét bájtban. Ez tulajdonképpen csak nekem kell vagy általánosabban szólva a mau nyelv fejlesztőinek — a PGM ugyanis az az objektum, amiben a mau programok le vannak tárolva.

## #L értéket visszaadó rendszerfüggvények

**?#L "POS" S G**

Visszaadja azt a pozíciót, ahonnan kezdve az S stringben megtalálta a G stringet. Az S és a G egy-egy tetszőleges stringkifejezés. Ha nem találta meg a G-t az S-ben, akkor -1 lesz a visszaadott érték.

Példa a függvény használatára:

```
#s@g="Ebben a stringben keresek!";  
#s@h="ing";  
  
#L@L=?#L "POS" @g,@h;  
"Első string, amiben keresek: " ?s @g; /;  
"Ezt keresem benne: " ?s @h; /;  
ha(#L(@L)<0) "Nincs benne a stringben!\n"  
E "A stringbeli pozíció: " ?L @L; /;  
"Vége a proginak!\n"
```

## #g értéket visszaadó rendszerfüggvények

**?#g "FM" – Lásd később**

**?#g "?|" – Lásd később**

**?#g "2" S – Lásd később**

## #s értéket visszaadó rendszerfüggvények

**?#s "in"**

Beolvas 1 sort az stdin-ről, azaz a standard inputról (de legfeljebb 65535 bájtot...) és visszaadja stringként. A sor végét lezáró '\n' karaktert kicseréli nullabájtra.

**?#s "1" s**

Itt az „s” egy stringkifejezés lehet. Ez a függvény megjeleníti a dmenu-höz hasonlóan a képernyő tetején azt az egyetlen sort amit ez a stringkifejezés tartalmaz. Ha Entert ütünk visszaadja stringként ugyanezt a sort, ha Esc-t, akkor egy egyetlen szóközből álló stringet kapunk vissza. Csak akkor működik, ha a mau interpreter X11 integrációval lett lefordítva, azaz ha a **vz.h** fájlban be van állítva az

```
#define X11INTEGRATION
```

direktíva.

E függvény tulajdonképpen amiatt létezik, mert így könnyen kiirathatunk mindenféle üzeneteket/információkat a grafikus képernyőre.

Példa a használatára: Vegyük fel a \$HOME/.xbindkeysrc fájlba e sorokat:

```
# (CapsLock + m) A moc-cal épp játszott zene címet kiírja a DWM ablakkezelő tetejére  
"mau /_/P/Mau/-/maubin/mocin.mau `mocp -i 2>/dev/null | grep "File:"`"  
Mod3 + m
```

Ezután ez a mau szkript (amit a fent megadott útvonalra kell elhelyeznünk) kiírja a moc által épp játszott szám nevét a képernyő tetejére a megfelelő billentyű-kombináció lenyomása esetén:

```
#!mau
#s@s="A MOC zenelejátszó nem fut!";
ha (?a)>=3 #s@s=?#s "ARGV" 3;
#s@v=?#s "1" @s;
XX
```

Nálam e billentyűkombináció a jobbshift+m, mert a jobb shiftet átmappeltem egy Mod3 módosítóbillentyűvé.

```
?#s "STDIN"
```

Ez a függvény beolvassa az stdin-ről az összes sort, és a dmenu-höz hasonlóan megjeleníti a képernyő tetején őket. Ugyanúgy mint a dmenu-ben, mozoghatunk köztük a kurzorbillentyűvel, majd ha Enter-t ütünk, visszaadja azt egy stringként. Amennyiben Esc-vel kilépünk ebből választás nélkül, úgy a visszaadott string egyetlen szóközt tartalmaz csak. Csak akkor működik, ha a mau interpreter X11 integrációval lett lefordítva, azaz ha a **vz.h** fájlban be van állítva az

```
#define X11INTEGRATION
```

direktíva.

```
?#s "D" t
```

A „t” itt egy mau\_c típusú aritmetikai kifejezés, egy stringtömb nevét határozza meg. Ez a függvény a dmenu-höz hasonlóan megjeleníti a képernyő tetején a stringtömb összes stringjét. Ugyanúgy mint a dmenu-ben, mozoghatunk köztük a kurzorbillentyűvel, majd ha Enter-t ütünk, visszaadja azt egy stringként. Amennyiben Esc-vel kilépünk ebből választás nélkül, úgy a visszaadott string egyetlen szóközt tartalmaz csak. Csak akkor működik, ha a mau interpreter X11 integrációval lett lefordítva, azaz ha a **vz.h** fájlban be van állítva az

```
#define X11INTEGRATION
```

direktíva.

```
?#s "C"
```

A fenti függvény visszaadja stringként a vágólapra másolt tartalmat. Csak akkor működik, ha a mau interpreter X11 integrációval lett lefordítva, azaz ha a **vz.h** fájlban be van állítva az

```
#define X11INTEGRATION
```

direktíva. Ez alapértelmezésben be van állítva. Ha azonban valaki kitörli, akkor ezen függvény minden meghívása egy olyan stringet ad vissza, aminek ez a tartalma:

```
"Ez a mau interpreter X11 támogatás nélkül lett lefordítva, emiatt nem képes beolvasni a vágólap tartalmát!\n"
```

(idézőjelek nélkül, természetesen).

```
?#s "A" – Lásd később
```

Tartalomjegyzék-bejegyzés ext2 specifikus flagjainak attributumstringjét adja vissza, lásd részletesebben később.

```
?#s "H"
```

Beolvassa és visszaadja stringként a „hostname” értéket.

```
?#s "I"
```



Az épp végrehajtott mau függvény nevét adja vissza stringként, lásd részletesebben később.

**?#s "S" x,b,k,m,g**

A fenti függvény esetében az „x” egy stringkifejezés, a b,k,m,g pedig egy-egy unsigned char típusú kifejezés lehet. E függvény visszaadja az x stringet, de úgy, hogy a jobbról számított minden 3-adik pozícióba beilleszti azt a színstringet, amit a megfelelő b,k,m,g karakterek valamelyike határoz meg. Ez arra jó, hogy amennyiben az x egy számjegyekből álló string, akkor a különböző helyiértékek ezres csoportjait eltérő színekkel (vagy háttérszínekkel) tudjuk megjelentetni. Tipikusan nagyon jó alkalmazási terület erre, ha egy tartalomjegyzék-listázó programban az állományméreteknél külön színekkel szeretnénk jelölni a bájtokat, kilobájtokat, megabájtokat és gigabájtokat. A fenti példában is e színeket jelölő változók nevét úgy választottam meg, hogy mutassa, melyik paraméter melyik nagyságrend színét határozza meg:

**b** - bájtok, **k** - kilobájtok, **m** - megabájtok, **g** - gigabájtok.

E fenti paraméterek esetén a színstringek azok lesznek, amik a vz.h include fájlban vannak letárolva, ezeknek kell megadni a sorszámát. A sorszám egy 0-35 közti szám. A jelentésének megértéséhez legegyszerűbb, ha idemásolom a megfelelő tömböt a programból:

```
struct szinek szin[] = {
/* 0 */ {" ??HIBÁS SZÍNKÓD!?? "},
/* 1 */ { COLOR_BRIGHT},
/* 2 */ { COLOR_UNDERLINE},
/* 3 */ { COLOR_RED},
/* 4 */ { COLOR_GREEN},
/* 5 */ { COLOR_YELLOW},
/* 6 */ { COLOR_BLUE},
/* 7 */ { COLOR_MAGENTA},
/* 8 */ { COLOR_CYAN},
/* 9 */ { COLOR_WHITE},
/* 10 */ { COLOR_BLACK},
/* 11 */ { COLOR_BRIGHT COLOR_WHITE},
/* 12 */ { COLOR_BRIGHT COLOR_RED},
/* 13 */ { COLOR_BRIGHT COLOR_GREEN},
/* 14 */ { COLOR_BRIGHT COLOR_YELLOW},
/* 15 */ { COLOR_BRIGHT COLOR_BLUE},
/* 16 */ { COLOR_BRIGHT COLOR_MAGENTA},
/* 17 */ { COLOR_BRIGHT COLOR_CYAN},
/* 18 */ { BACKGROUND_COLOR_DEFAULT},
/* 19 */ { BACKGROUND_COLOR_BLACK},
/* 20 */ { BACKGROUND_COLOR_RED},
/* 21 */ { BACKGROUND_COLOR_GREEN},
/* 22 */ { BACKGROUND_COLOR_YELLOW},
/* 23 */ { BACKGROUND_COLOR_BLUE},
/* 24 */ { BACKGROUND_COLOR_MAGENTA},
/* 25 */ { BACKGROUND_COLOR_CYAN},
/* 26 */ { BACKGROUND_COLOR_DARKGRAY},
/* 27 */ { BACKGROUND_COLOR_LIGHTGRAY},
/* 28 */ { BACKGROUND_COLOR_LIGHTRED},
/* 29 */ { BACKGROUND_COLOR_LIGHTGREEN},
/* 30 */ { BACKGROUND_COLOR_LIGHTYELLOW},
/* 31 */ { BACKGROUND_COLOR_LIGHTBLUE},
/* 32 */ { BACKGROUND_COLOR_LIGHTMAGENTA},
/* 33 */ { BACKGROUND_COLOR_LIGHTCYAN},
/* 34 */ { BACKGROUND_COLOR_WHITE},
/* 35 */ { COLOR_RESET}
};
```

Látható, nulla kód esetén színstring kiadása helyett egy hibaüzenetet másol be a megfelelő helyre. Ugyanez lesz az eredménye különben annak is, ha a színkód-paraméter értéke nagyobb, mint 35. Azaz nem lesz emiatt programleállás, hibaüzenet - a hibaüzenet maga az, hogy nem más színnel jelenik meg az adott szöveg, hanem beszűrődik oda ez a figyelmeztetés. Ez igazán figyelemfelkeltő, s rögvest megmutatja, hol a hiba... Egy példa a használatára a listázóprogramomból:

```
?s ?#s "S" ([#c@:1,] ?#g "FM" t,#l@i,R),4,14,12,11; // A fileméret, különböző színekkel  
// kiírva a bájtokat, Kbyteokat, Mbyteokat és gigabájtokat
```

Természetesen ha a string hossza kevesebb mint ami minden megadott szín alkalmazására elegendő lenne, akkor nem csinál olyan ökörséget hogy mégis beszúrja a felesleges színekódokat, hanem csak annyit, amennyi kell a létező helyiértékeknek.

**?#s "(") " x**

Ez a függvény egy stringet olvas be a standard inputról (a konzolról általában). Maximum x karaktert. Az „x” egy unsigned int típusú aritmetikai kifejezés. A string végét jelző újsor karaktert nullabájtra cseréli.

**?#s ",,"" – Lásd később**

**?#s "\$x" b**

Az adott „b” unsigned char típusú bájtot adja vissza 2 karakteres hexadecimális stringként, azaz a stringhossz 2 lesz, és nincs „\$” jel az elején! A 9-nél nagyobb értékű karakterek kisbetűsen jelennek meg, azaz a-f formátumban.

**?#s "\$X" b**

Az adott „b” unsigned char típusú bájtot adja vissza 2 karakteres hexadecimális stringként, azaz a stringhossz 2 lesz, és nincs „\$” jel az elején! A 9-nél nagyobb értékű karakterek NAGYbetűsen jelennek meg, azaz A-F formátumban.

**?#s "BG" k**

Visszaad egy színstringet, aminek a kódja „k” (ami egy unsigned char kifejezés), és amely színstring alkalmas az írás háttérszínének meghatározására. A „k” karakter ugyanaz kell legyen mint a kiíratásról szóló fejezetben a „B”-vel kezdődő parancsok második karaktere. Vagyis:

K értéke	szín
d	Alapértelmezett háttérszín
f	fekete
r	vörös
g	zöld
y	sárga
b	kék
m	magenta
c	türkiz (cián)
l	sötétszürke
L	világosszürke
R	világospiros
G	világoszöld
Y	világossárga
B	világoskék
M	világosmagenta
C	világostürkiz (világoscián)
w	fehér

**?#s "NL" – Lásd később**

**?#s "d" – Lásd később**

**?#s "SB" string hossz**

Visszaad egy stringet fix hosszúságú mezőbe balra igazítva. Az üres helyek szóközökkel lesznek kitöltve. Ha a string hosszabb mint a megadott hossz, akkor NEM csonkít! A hossz egy unsigned int érték.

**?#s "SP" string**

Visszaadja a stringet úgy, hogy levágja a bal oldaláról az összes, esetlegesen ott levő whitespace karaktereket.

**?#s "ST"**

Ez az aktuális rendszerdátumot és rendszeridőt adja vissza stringként, a következő formátumban:

2014.02.26 10:07:38 Sze

Gondolom, érthető a szintaxis... A legvégén a hét adott napja van 3 karakteres rövidítéssel jelölve, amik a következők:

H - Hétfő  
K - Kedd  
Sze - Szerda  
Cs - Csütörtök  
P - Péntek  
Szo - Szombat  
V - Vasárnap

**?#s "Tg" – Lásd később**

**?#s "Tn" – Lásd később**

**?#s "Tt" – Lásd később**

**?#s "UE" neve hanyadik x – Lásd később**

**?#s "UM" neve hanyadik x – Lásd később**

**?#s "UV" neve hanyadik x – Lásd később**

**?#s "ENV" x**

Visszaadja valamely fontos operációsrendszer-változó stringértékét. Az „x” egy stringváltozó, mely megmondja melyik oprendszer-stringet kell visszaadnia.

Például: **?#s "ENV" "EDITOR"**

**?#s "{-}" – Lásd később**

**?#s "JOG" – Lásd később**

**?#s "PID" x c**

Visszaadja a processz id -nek megfelelő értékét stringként, a megfelelő hosszra igazítva/csonkolva. Az „x” egy unsigned char érték, a stringhosszat határozza meg, c pedig szintén egy unsigned char érték, az a karakter, amivel a string üres helyeit feltölti majd.

**?#s "ARGV" x**

Visszaadja az **argv** megfelelő értékét stringként. Az „x” egy unsigned char érték, azt határozza meg, hányadik parancssori paraméterstringet akarjuk megkapni.

**?#s "LINK" – Lásd később**

**?#s "SZIN" k**

Visszaad egy színstringet, azaz azt a karaktersorozatot (a mau nyelvben megszokott string típusként), amit a kiíratásról szóló korábbi fejezetben bemutattam mint #define direktívát. A „k” egy unsigned char típusú érték, ami a megfelelő színstring kódja. E karakter természetesen ugyanaz, ami a megfelelő „S” karakterrel kezdődő, színbeállító mau parancs második karaktere. A jelentése a „k” karakternek tehát a következő (ilyen szint előállító színstringet ad vissza):

d	COLOR_RESET	"\x1b[0m"	default szín
r	COLOR_RED	"\x1b[31m"	vörös
g	COLOR_GREEN	"\x1b[32m"	zöld
y	COLOR_YELLOW	"\x1b[33m"	sárga
b	COLOR_BLUE	"\x1b[34m"	kék
m	COLOR_MAGENTA	"\x1b[35m"	magenta
c	COLOR_CYAN	"\x1b[36m"	türkiz
w	COLOR_WHITE	"\x1b[37m"	fehér
v	COLOR_BRIGHT	"\x1b[01m"	világosság
u	COLOR_UNDERLINE	"\x1b[04m"	aláhúzás
f	COLOR_BLACK	"\x1b[30m"	fekete

**?#s "MAUDATE"**

Visszaadja stringként **eeee.hh.nn** formában azt a dátumot, ami az épp futó interpreter forráskódjának kiadási dátuma. Például

2014.04.06

**?#s "<<<" eredeti keresem erre**

Az „eredeti”, a „keresem” és az „erre” egyaránt egy-egy tetszőleges stringkifejezés. Az „eredeti” stringben megkeresi a „keresem” string minden előfordulását, és kicseréli mindegyiket az „erre” stringre. A „keresem” és „erre” hossza eltérő is lehet (az „erre” lehet akár üres string is). Az eredmény a megváltoztatott string lesz. Az „eredeti” stringet nem bántja. Ha az „eredeti” vagy a „keresem” string üres string (nulla hosszú), akkor az eredményül visszkapott string is üres string lesz. Példa:

```
#!mau
#s@e="0123abc4567abc8901234abc34"; // eredeti string, ebben keresünk
#s@k="abc"; // Ezt a stringet keressük
#s@c="!cseréltem!"; // Erre cseréljük ki
#s@u=?#s "<<<" @e,@k,@c;
"Eredeti string : " ?s @e; /;
"Kerestem benne : " ?s @k; /;
"Cseréltem erre : " ?s @c; /;
"Eredmény      : " ?s @u; /;
XX
```

A fenti program gondolom önmagát magyarázza.

**?#s "n", InterFace, LE, FEL**

A fenti függvény visszaad egy olyan stringet, ami megmutatja nekünk az aktuális nethasználati statisztikát a megfelelő interfészen. Az InterFace, LE és FEL egy-egy stringkifejezés lehet. Az InterFace definiálja a lekérdezett interfészt, azaz lehet például az értéke "eth0", "eth1", "wlan0", "wlan1" stb, a „LE” egy olyan string amit a letöltési sebesség számértéke elé ír ki, a „FEL” pedig egy olyan string amit a feltöltési sebesség számértéke elé ír ki.

Példa a használatára az ezen doksi egy későbbi fejezetében ismertetett calc.mau program segítségével:

```
vz@Csiszilla /Releases/2014/U/Common/vz/MAU=>./mau calc.mau
mau> ?s ?#s "n" "eth0" "D: " " ^"; /
D: 90 K ^2 K
mau> ?s ?#s "n" "eth0" "D: " " ^"; /
D: 110 K ^73 K
mau> ?s ?#s "n" "eth0" "D: " " ^"; /
D: 90 K ^2 K
mau> ?s ?#s "n" "wlan0" "D: " " ^"; /
D: 0 K ^0 K
```

Amennyiben a sebesség a kilobájtos nagyságrendbe esik, akkor a számok kilobájtban értendők, s utánuk a K betűt írja ki, amennyiben megabájtos nagyságrendűek úgy megabájtban értendők, és M betűt ír utánuk.

## 14. fejezet - Vezérlési szerkezetek, azaz elágazások, ciklusok és „esetek”

A mau programnyelv egyik legislegnagyobb erőssége, hogy rengeteg pompás lehetőséget biztosít különböző vezérlési szerkezetek megvalósítására, azaz számtalan különböző fajta ciklustípust és esetvizsgálatot támogat.

Nézzük először a közösleges ugróutasításokat, azaz a feltétel nélküli szubrutin-hívást illetve ugrást! Ezekhez meg kell ismerkednünk a „címkék” fogalmával.

Címkék más programnyelvekben is vannak. A mau nyelvben ezekről annyit kell tudni, hogy mindig 2 karakterből állnak, de a második karakter lehet szóköz vagy más whitespace karakter is. Ezenkívül a címke kivétel nélkül mindig egy paragrafusjellel kell kezdődjön, ezzel: §, ez ugyanis az „előtétkarakter”, ebből tudja az interpreter, hogy egy címke következik.

Na most itt térek ki rá, hogy a mau programnyelv utasításaiban szerepel néhány olyan „egzotikus” karakter, melyek nem részei az ASCII kódtáblának. Ilyen a paragrafusjel is. A mau interpreter emiatt feltételezi, hogy a fájl ami a feldolgozandó mau forráskódot tartalmazza, UTF-8 kódolású. Ez nem hinném hogy nagy megkötés lenne, mert az UTF-8 egy nemzetközi kódolás, igencsak elterjedt, s még jobban is el fog terjedni. Természetesen a mindenféle magyar, német, héber, stb karaktereket is UTF-8 kódolás szerint kezeli a mau interpreter, ezt azért kell megemlíteni, mert effélék nemcsak megjegyzésekben szerepelhetnek nála, de akár függvények nevében is, stb, erről majd később a függvényeknél lehet olvasni részletesebben. Címke esetén azonban jobb ha megmaradunk az ASCII karakterek mellett, mert itt az interpreter 2 darab BÁJTOT olvas be a § jel után, ezekből kalkulál ki egy sorszámot a címkének. A § jel UTF-8 bájt-reprezentánsa a 194, 167 sorozat. Minden magyar ékezetes karakter, valamint a mau interpreter számára fontos egzotikus karakter UTF-8 kódjai megtekinthetők ezen alább következő táblázatban, meg még pár olyan is ami most nem fontos neki, de „szemezek” velük, s emiatt nem kizárt hogy bekerülnek e nyelv későbbi kiadásába is valamiféle funkcióban:

### UTF-8 kódok

Karakter	decimális bájtsorrend	hexa bájtsorrend	Nemzetközi elnevezés	Egyéb nevek
Á	195 129	c3 81	Aacute	
á	195 161	c3 a1	aacute	
Ä	195 132	c3 84	Adiaeresis	nagy umlaut
ä	195 164	c3 a4	adiaeresis	umlaut
É	195 137	c3 89	Eacute	
é	195 169	c3 a9	eacute	
Í	195 141	c3 8d	Iacute	
í	195 173	c3 ad	iacute	

Karakter	decimális bájtrend	hexa bájtrend	Nemzetközi elnevezés	Egyéb nevek
Ó	195 147	c3 93	Oacute	
ó	195 179	c3 b3	oacute	
Ö	195 150	c3 96	Odiaeresis	
ö	195 182	c3 b6	odiaeresis	
Ő	197 144	c5 90	Odoubleacute	
ő	197 145	c5 91	odoubleacute	
Ú	195 154	c3 9a	Uacute	
ú	195 186	c3 ba	uacute	
Ü	195 156	c3 9c	Udiaeresis	
ü	195 188	c3 bc	udiaeresis	
Ű	197 176	c5 b0	Udoubleacute	
ű	197 177	c5 b1	udoubleacute	
– (nagykötőjel)	226 128 148	e2 80 94	emdash	(kvirtminusz)
(nem törhető szóköz)	194 160	c2 a0		
§	194 167	c2 a7	section	paragrafusjel
»	194 187	c2 bb	guillemotleft	
«	194 171	c2 ab	guillemotright	
– (gondolatjel)	226 128 147	e2 80 93	endash	félkvirtminusz
„ (nyitó idézőjel)	226 128 158	e2 80 9e	doublelowquotemark	
” (záró idézőjel)	226 128 157	e2 80 9d	rightdoublequotemark	
ß	195 159	c3 9f	ssharp	sárfesz-esz
… (három pont)	226 128 166	e2 80 a6	ellipsis	
×	195 151	c3 97	multiply	
¢	194 162	c2 a2	cent	
±	194 177	c2 b1	plusminus	
÷	195 183	c3 b7	division	
£	194 163	c2 a3	sterling	font, pound
¤	194 164	c2 a4	currency	
¿	194 191	c2 bf	inverted question mark	questiondown
¡	194 161	c2 a1	inverted exclamation mark	exclamdown
·	194 183	c2 b7	middle dot	periodcentered

Annak érdekében hogy aki mau nyelven akar programozni, annak ne magának kelljen szerencsétlenkednie egy saját billentyűkiosztás kitalálásával, amin szerepel az összes szükséges karakter, leírom ennek módszerét is e könyvben! Ezt már korábban közreadtam a wikimben, s e doksiban is szerepel annak a wiki cikknek a linkje, ez:

[http://parancssor.info/dokuwiki/doku.php?id=sajat\\_billentyuzetkiosztas\\_ujra](http://parancssor.info/dokuwiki/doku.php?id=sajat_billentyuzetkiosztas_ujra)  
de e doksiban is közreadom „offline”, abban a fejezetben, aminek címe:

**Saját billentyűzetkiosztás készítése.** Amennyiben tehát ez neked problémát okozna, ugorj arra a fejezetre!

A címkével akkor jelölünk meg egy részt a mau forráskódban, ha a címkéhez tartozó **§** jel egy sor legislegelső karaktere! Nem lehet előtte semmi, **whitespace sem** (kivéve természetesen az előző sor CHR\$(10) karakterét, hiszen abból tudja az interpreter, hogy sor vége volt, s itt kezdődik egy új sor). Ez azért fontos, mert a címkék esetén a forráskód beolvasása során dől el, hogy hozzájuk miféle memóriacím tartozik amit ő jelöl, s ha nem sor legelején találkozik a **§** jellel az interpreter, akkor úgy véli, itt a címke nem azért van mert itt jelöli meg a forráskód egy adott helyét, hanem azért van itt, mert itt hivatkozunk a címkére, mondjuk egy ugróutasítás céljaként megadva azt!

A feltétel nélküli ugróutasítás, az, ami más nyelvekben goto-nak hívatik, a mau nyelvben így néz ki:

» **x**

aholis az „x” egy unsigned int típusú aritmetikai kifejezés, ami egyszerűen a mau forráskód valamelyik bájtját jelenti, attól kezdve folytatódik az utasításfeldolgozás. Fontos tudni, hogy először is, eszerint itt e nyelvben minden ugróutasítás „számított”, azaz relatív, amennyiben nem biztos hogy legközelebb is ugyanoda ugrik e sorra érve, mert ugyebár lehet hogy az „x” kifejezés akkor már más értéket határoz meg. Másodszor, az interpreter kizárólag azt ellenőrzi ilyenkor, az ugrás célja nem lépi-e túl a mau programnak lefoglalt programmemória határát, de azt például nem ellenőrzi, hogy az ugrás megadott célja netán nem egy aritmetikai kifejezés kellős közepébe, vagy egy **//** jeles megjegyzés utánra, vagy egy **/\* ... \*/** közé ékelt megjegyzésblokk belsejébe mutat-e! Efféle ellenőrzés lekodolása iszonyatosan bonyolult valami volna, továbbá rémségesen lelassítaná a program futását. Sőt, még abban se vagyok biztos, nem okozna-e inkább több kárt mint hasznot, mert el tudok képzelni olyan eseteket - bár extrém körülmények közt - amikor egy ilyen lehetőség akár hasznos is lehet.

Na most a fenti példában az „x”, mint írtam, aritmetikai kifejezés. Aritmetikai kifejezés tartalmazhat azonban címkét is, amit ugyanúgy kell megadni, mint ahogy a sor elején megjelölünk vele egy helyet a kódban, azaz itt is ki kell tenni eléje a **§** jelet. A címke visszaadott értéke ezesetben, amikor tehát aritmetikai kifejezésben, általában véve jobbérték-szerepben használjuk, pontosan az az unsigned int szám, ami a címke által jelölt index a programkódban. Többnyire a címke természetesen egymagában áll egy ugróutasítás után, de ez nem szabály, mert nyugodtan végezhető vele bármi olyan művelet, ami egy akármiféle másik unsigned int típusú változóval - épp csak értéket nem lehet neki adni, azaz ez egy „readonly”, kizárólag „input” változónak tekintendő.

Azaz ha van egy így megjelölt rész a kódban:

**§be // Itt kezdődik a beolvasás a fájlból**

akkor erre így ugorhatunk:

**»§be // ugrás a beolvasórutinra**



Indirekt ugrást úgy a legegyszerűbb csinálni, ha a címkét belerakjuk egy változóba, a program egyik részénél az egyik, másikonál a másik címke lesz a változó értéke, aztán ahol kell, ugrunk rá. Persze mert a címke unsigned int méretű adatot tárol, a változó is ilyen típusú illik legyen:

```
#l@c=$be;
```

Aztán valahol ugrunk rá:

```
»#l@c;
```

A lehetőségek végtelenek.

A szubrutinhívás, az ami a más nyelvekben a „gosub” nevű, épp ezen „gosub” szóra asszociálva a **G** utasítás által valósul meg a nyelvünkben. Épp úgy működik mint a » utasítás, egyszerűen ugrás előtt elmenti egy veremtárba a visszatérési címet. A szubrutinból való visszatérés (ami más nyelvekben a „return” nevű utasítás) itt a « utasítás által hajtatik végre. Ez az utasítás nem ad vissza semmiféle paramétert, tudniillik ez itt kizárólag vezérlési szerkezet, ennek semmi köze a névterekhez, itt még minden változó globális. Bár VAN RÁ MÓD, hogy egy szubrutin belsejében külön névteret (sőt többet is, nemcsak egyet, s akár egymásba ágyazva is) definiáljunk, sőt ehhez szubrutinba menni se kell, de az egy egészen külön lehetőség, aminek tulajdonképpen semmi köze a szubrutinokhoz, általában a vezérlési szerkezetekhez, ezért azzal mi is külön fejezetben foglalkozunk majd.

## Az **if** és a **ha** utasítás

Minden programnyelvre jellemző, hogy elágazási utasítások előtt többnyire valamiféle összehasonlító művelet, azaz feltételvizsgálat áll. A mai nyelvben többfajta feltételvizsgáló lehetőség is rendelkezésünkre áll, ezek közt van természetesen a leggyakrabban használt, a jól ismert „őreg harcos” és „nagy munkás”, az **if** is.

Ennek formája nagyon egyszerű:

```
if x
```

ahol az „x” egy egészen közönséges, tetszőleges unsigned char típusú aritmetikai kifejezés. Még csak az se muszáj hogy ezt az „x”-et zárójelek közé tegyük, bár megtehetjük, sőt, ez ajánlatos is a kód jobb olvashatósága kedvéért. De nem kötelező. Amennyiben az „x” kifejezés „alapból” nem unsigned char típusú, az se baj, majd az **if** castolja azzá a maga számára, de ezesetben az „x” az őrá jellemző casting operátorral kell kezdődjen.

Na most miután az **if** kiszámolta az „x” kifejezés értékét, s ha szükséges volt castolta unsigned char értékévé, ezután ezt megvizsgálja, és egy speciális változóban - aminek az a neve hogy „ifflag” - eltárol egy 0 számot, ha az „x” értéke 0 volt, vagy eltárol 1-et, ha az „x” értéke nem nulla volt. SEMMI MÁST SE CSINÁL az **if**, nem ugrik sehova, esze ágában sincs.

Az ugrás ugyanis nem az **if** dolga, ő azt se tudja mi az hogy ugrás. Az ugrásokat más utasítások végzik. Ezek megvizsgálják az ifflag aktuális állapotát, s attól függően ugranak vagy nem ugranak.

A legfontosabb két efféle utasításunk a „**T**” és az „**E**”. Ezek nem véletlenül kapták e neveket, mert ezek felelnek meg annak - legalábbis nagyjából - ami más nyelvekben a „then” és az „else”. A következőképp működnek:

A **T** csinálja azt, hogy leellenőri ezen ifflag flaget, s ha ez nem nulla, akkor semmit se csinál. Ha ellenben nulla, akkor... Nos, akkor ugyanazt csinálja mint a **//** utasításunk, azaz elugrik a következő sor elejére! Ez nyilván azzal jár, hogy ha a feltétel igaz volt, végrehajtódik a **T** utáni programrész. Ha ellenben hamis volt, az utasításvégrehajtás a következő sor elején folytatódik. Persze, oda is tehetünk nyugodtan egy újabb **T** utasítást, meg a következőbe is, akárhova akármennyit, mindaddig, míg egy újabb **if** parancsot végre nem hajt a programunk! (Vagy egy **ha** parancsot, amit picit később ismertetünk). Az **E** utasításunk meg pont ugyanaz mint a **T**, csak akkor ugrik, ha az ifflag értéke 1.

A mau nyelv garantálja, hogy minden névtér létrejöttékor (tehát a program indulásakor is) az ifflag értéke kezdetben 0.

Ehhez kapcsolódik két rendszerváltozó. A **?0** egy konkrét unsigned char típusú KARAKTERT ad vissza, a "0" vagy az "1" karaktert, ahol természetesen az "1" felel meg az IGAZ értéknek, vagy lekérdezhetjük ezt a **?F** szimbólummal is, amely rendszerváltozó (vagy rendszerfüggvény?) egyszerűen visszaadja e flag szám-kódját, azaz egy CHR\$(0) vagy CHR\$(1) kódú karaktert.

E flag értékét „direkt” módon is beállíthatjuk a következő utasításokkal:

**/0**  
A fenti utasítás 0 értékűre állítja az ifflag-ot.

**/1**  
A fenti utasítás 1 értékűre állítja az ifflag-ot.

**/F**  
A fenti utasítás pedig invertálja az ifflagot.

Példa az **if** használatára:

```
if (?a)<3 T #s@A="."; // Ha nem adtunk meg parancssori paramétert, az aktuális könyvtárat listázza
```

A fenti sor egy konkrét programomból egy részlet.

Egy másik példa (ennek semmi értelme csak illusztráció):

```
if 3 T "Igaz" /;  
? ?0 /;  
"ifflag számértéke: " ?c ?F /;  
T "Ez is igaz" /;  
E "Nem igaz" /;  
T »$ug  
"Ezt nem írom ki" /;  
$ug "Ide ugrott" /;
```

Stringek összehasonlítása:

```
#s@s="Álom";  
#s@f="álom";
```

```

"s=" ?s @s; " f=" ?s @f; /;
if(#s((@f)==(@s))) T "Egyenlőek!\n"
E "Nem egyenlőek!\n"
if(#s((@f)==(@s))) T "Egyenlőek!\n"
E "Nem egyenlőek!\n"
if(#s((@f)!=(@s))) T "f != s !\n"
E "f == s !\n"
if(#s((@f)<(@s))) T "f != s !\n"
E "f == s !\n"
if(#s((@f)>(@s))) T "f>s !\n"
E "f<=s !\n"
if(#s((@f)<(@s))) T "f<s !\n"
E "f>=s !\n"
if(#s((@f)<=(@s))) T "f<=s !\n"
E "f>s !\n"
if(#s((@f)>=(@s))) T "f>=s !\n"
E "f<s !\n"

```

Eredménye:

```

s=Álom f=álm
Nem egyenlőek!
Nem egyenlőek!
f != s !
f != s !
f>s !
f>=s !
f>s !
f>=s !

```

Na most, nyilván felmerül a kérdés a Tisztelt Olvasóban, miért van ilyen „bonyolultan” megadva az összehasonlítás az „**if**” után. Nem bonyolult pedig az, csak tisztában kell lenni nyelvünk alapelveivel... Ez pedig az, hogy minden utasítás megszabja, miféle típusú paramétert vár el. Az **if**, az egy unsigned char számot. Ha tehát mi stringeket akarunk összehasonlítani, kell a **#s** casting operátor, ami azt mondja meg, hogy egy stringkifejezés jön, azt kell kiértékelnie. E stringkifejezés természetesen egy stringet ad vissza eredményül, mi mást is amikor épp azért STRINGkifejezés... de mert összehasonlító műveletről van szó e stringkifejezésben, ezért az eredmény egy olyan string lesz, ami csak 2 bájtból áll: az **[1]** indexű bájt mindig a stringzáró nullabájt, a **[0]** indexű pedig egy CHR\$(0) bájtot tartalmaz ha az összehasonlítás eredménye HAMIS, és CHR\$(1) bájtot, ha az összehasonlítás IGAZ. E bájtot aztán a kifejezéskiértékelő rutin annak rendje s módja szerint unsigned char értékké alakítva adja vissza az **if** utasításunknak, ami ennek megfelelően cselekszik.

A fenti példában az **if**-et követő legelső zárójel és a párja tulajdonképpen felesleges, nem szükséges, de szerintem nem árt kitenni, mert akkor már igenis szükséges, ha további összehasonlítások is vannak, amiket netán az **&&** vagy a **||** operátorokkal kombinálni akarunk.

A **ha** utasítás tulajdonképpen semmi más, mint egy „egybeépített” **if** és **T** utasítás. Egyszerűen arról van szó, hogy az esetek óriási többségében az **if** után megadott feltétel után egy **T** szokott következni. Emiatt erre célszerűnek látszott kitalálni valami egyszerűsítést, mely így gyorsítja a programot, de meg különben is, a C nyelven programozni tudó emberként előnyösnek tarthatják, ha nem kell mindig kirakosgatni a **T** karaktert. Azaz, a **ha** utasítás beolvassa az aritmetikai kifejezést, megvizsgálja, eltárolja az ifflagba amit kell, éppúgy jár el tehát mint az **if**, majd rögvést megcsinálja azt is amit a **T** tenne. Semmiben se tér el attól, mintha helyette az **if** és a **T** lenne kiírva, tudniillik a **ha** utasítás programo-

zástechnikai megvalósítása ténylegesen nem is áll semmi másból, mint hogy egyszerűen meghívjuk egymás után előbb az **if**, majd a **T** utasítást megvalósító rutinokat. Egy példa a használatára:

```
ha(#L(@i)<0) "Nincs benne a stringben!\n"  
E "A stringbeli pozíció: " ?L @i; /;
```

## Ciklusok

### A „végtelen ciklus”

A legegyszerűbb ciklusfajta a mai nyelvben a VÉGTELEN CIKLUS. Ennek nagyon sok lehetséges alkalmazási területe van, tudja ezt minden programozó! Ebből természetesen valamely külön utasítással kell kiugrani, az ő belsejéből. E ciklusfajta szerkezete a következő:

```
{(  
// itt van a ciklusmag  
)}
```

Tartsuk észben, hogy mind a **{** mind a **}** egy-egy külön mai utasítás, ha úgy tetszik parancs, azaz az ezeket alkotó zárójelek közt nem lehet semmiféle más karakter, whitespace sem!

A ciklusok esetében a mai interpreter a visszatérési címet (azaz azt hogy hol kezdődik a ciklus) egy speciális veremtárban tárolja. Ezekből, nagyon de nagyon **ROSSZ ÖTLET** egy efféle ciklusból egyszerűen csak úgy kiugrani egy » utasítással (vagy bármi más módon, ami nem olyan módszer amit itt mindjárt ismertetünk). Ezesetben ugyanis benne marad a visszatérési cím feleslegesen a veremtárban... Na most, ez addig nem gond amíg csak a főprogram van és benne egyetlen ilyen ciklus. Abban a pillanatban azonban, amint egy efféle ciklust egy másik ciklus belsejébe ágyazunk bele, s a legbelsőből kiugrunk szabálytalanul, gond lesz de nagy, amint a külső ciklus végéhez ér az interpreterünk! Akkor ugyanis a belső ciklus veremben maradt visszatérési címét szedné ki, s oda ugrana vissza, holott már hátra nem oda kéne érkeznie...

Hogyan is kell akkor egy efféle végtelenciklusból (örökciklusból) szabályosan „eltávozni”? Nos, valahogy így:

```
{( // Kezdődik a ciklus  
// Valamiféle utasítások vannak itt  
if ( /* itt valami feltétel áll */ ) TT »$in // Ha a feltétel igaz, kiugrik  
// Valamiféle utasítások lehetnek itt is  
}) // Ez itt a ciklus vége  
$in  
"Vége!" /;
```

Azaz, egy efféle ciklusból a **TT** utasítással ugorhatunk ki - vagy a párjával, aminek **EE** a neve. Ezek teljesen megegyeznek a már ismert **T** illetve **E** utasításokkal, azt kivéve, hogy amennyiben az **if** által korábban letárolt ifflag értéke olyan, hogy végre kell hajtani az utánuk álló utasítást, azaz nem kell a következő sorra ugraniuk, akkor (és csak akkor!) egyrészt törlik a veremtárból a visszatérési címet, másrészt SZIGORÚAN ELLENŐRZIK, hogy a következő végre-hajtandó utasítás (az esetleg előtte levő whitespace karaktereket nem számítva) éppen pontosan a » utasítás-e! Mást ezek nem fogadnak el. Ebből következik, hogy ezek nem lehetnek többsorosak.

Azaz, míg a **T** és **E** lehet többsoros, addig a **TT** és **EE** már egyáltalán nem, mert gondoljuk csak el, mi történne, ha többsorosak volnának, és mindegyik megkísérelne törölni a verem tetejéről valahány bájtot, mint szerinte már felesleges visszatérési címet... Természetesen így se lehet kiküszöbölni minden hibalehetőséget, de olyan programnyelv még nem született, amely eleve lehetetlenné tesz minden hibaelkövetést. Ez amit ide beleépítettem, nem igazán erőforrásigényes, ugyanakkor mégis nyújt valamekkora biztonságot.

Amennyiben tehát kiugrás előtt még végre akarunk hajtani bizonyos más mindenféle tevékenységeket is, azt így oldjuk meg:

```
if ( /* itt valami feltétel áll */ ) T // valami utasítások
T // itt is valami utasítások állnak
T // itt is valami utasítások állnak
TT »$in // Ha a feltétel igaz, kiugrik a $in címére (aminek illik a cikluson kívül lennie...)
```

Azaz minden olyasmit ami nem kiugrás, a **T** utasítással prefixáljunk, s legutoljára az ugrásnál használjuk csak a **TT** parancsot. Mindez ugyanígy érvényes az **E** és **EE** utasításokra is, természetesen.

## Hátultesztelő ciklus

Abban az esetben, ha nem örökciklust akarunk megvalósítani, hanem afféle „hagyományos”, „rendes” ciklust, ami azonban „hátultesztelő”, azaz legalább 1 alkalommal mindenképpen lefut, akkor egészen hasonlóan kell eljárunk a fentiekhez (az örökciklushoz), mert annak vázlata így néz ki:

```
{( // Kezdődik a ciklus
// Valamiféle utasítások vannak itt
)}( /* itt valamiféle feltétel van */ )
// Ez már nem a ciklus része
```

A fenti tehát egy hátultesztelő ciklus váza, amiről 2 dolgot kell tudni:

**1.** Akkor folytatja a ciklus végrehajtását, ha a végén megadott feltétel IGAZ. Ha ugyanis nem igaz, kiugrik a ciklusból. A feltétel nem kell hogy okvetlenül valami összehasonlító művelet eredménye legyen, simán megadható oda bármi, ami egy unsigned char értékke konvertálható, s ha ez nem 0, akkor igaznak tekinti. Ellenkező esetben hamisnak. Azaz e szempontból a C nyelvben megszokott módon működik.

**2.** A ciklus végét lezáró **}}** karakterpáros és a feltételt bevezető nyitózárójel, a „(” között EGYÁLTALÁN SEMMIFÉLE karakter se lehet, whitespace sem! Az inter-

preter ugyanis épp abból tudja hogy nem örökciklussal van dolga, hogy a ciklus végét jelző karakterpárost rögvest követi egy nyitó kerek zárójel. Ebből máris következik, hogy a ciklus kilépési feltételére vonatkozó kifejezést itt MUSZÁJ zárójelek közé rakni mindenképpen.

Egy példa a ciklus végére, miképp is nézhet ki ez konkrétan:

```
)}((#i@K) <= 8) // Ha (K <= 8) folytatja a ciklust
```

Íme egy **hasznos** kis példa a hátultesztelő ciklus alkalmazására:

```
{( #c@c=?#c "("; ? @c; )}((@c)!=10); /;
```

A fenti progi egyszerűen addig várja a karaktereket amíg Entert nem nyomunk, s mindet visszaírja a képernyőre. (Az Enter után meg még kiír egy újsort is);

## Elöltesztelő ciklus

**Az előltesztelő ciklusnál** mindenképp kell címkét alkalmaznunk, tudniillik mert lehet hogy annak ciklusmagját egyszer se kell végrehajtania, s ezért tudnia kell, az esetben hova ugrik, azaz hol a ciklus vége. Lássunk is először egy példát, aztán jön a magyarázat:

Egy konkrét példa, két egymásbaágyazott ciklusra:

```
#i@W = 1 // unsigned short int W=1

{( // Kezdődik a külső ciklus. Ez hátultesztelő

#i@K = 5 // unsigned short int K=5

// Ez itt egy előltesztelő ciklus
{($ki ((#i@K) == 8) // Ha (K == 8) kiugrik a ciklusból a „$ki” címkére
// Kezdődik a belső ciklus

"W=" ?i @W " , K=" ?i @K " , W*K=" ?i (@W)*(@K) /; // Kiíratás

#i++@K // K++ (Azaz K+=1, vagy még hagyományosabban írva: K=K+1)

}} // Itt végetért a belső ciklus, ami előltesztelő volt
$ki // Erre a címkére kerül a vezérlés az előltesztelő ciklus után

#i++@W // W++

)}((#i@W) <= 5) // Ha (W <= 5) folytatja a külső ciklust
```

Magyarázat: A belső ciklus mint látjuk így kezdődik:

```
{($ki ((#i@K) == 8) // Ha (K == 8) kiugrik a ciklusból a „$ki” címkére
```

Azaz, a **{** karakterpáros után meg van adva a címke, amire majd ugrik, ha el kell hagynia a ciklust. (vagy ha bele se kell mennie egyszer se a ciklusba). Fontos tudni, hogy itt NEM adható meg tetszőleges aritmetikai kifejezés a címke helyett, hanem kifejezetten csak és kizárólag a **\$** jellel azonosított konkrét címke, valamint a **{** és a **\$** jel közt itt se állhat semmiféle karakter, whitespace SEM! Az interpreter ugyanis épp abból tudja, hogy itt előltesztelő ciklussal van dolga és nem hátultesztelővel, hogy a ciklus kezdetét jelölő **{** karakterpáros után közvetlenül egy paragrafusjel található.

A címke után egy aritmetikai kifejezés áll, ennek eredményétől függően (ha nem 0, akkor tekinti igaznak, különben hamisnak) ugrik ki a ciklusból, vagy hajtja végre azt.

A ciklus végét természetesen ugyanúgy a `}}` karakterpáros zárja le, mint a hátultesztelő ciklusok esetében. Ezúttal azonban nem kell itt feltételt megadnunk, mert mi a csudának ha a feltételt már megadtuk a ciklus fejrészénél! Mindazonáltal SZABAD azért megadnunk feltételt ITT IS, ha nagyon akarunk különködni, pontosan ugyanúgy megadhatunk feltételt mintha ez egy „normális” hátultesztelő ciklus volna. Azaz a belső ciklus vége lehet akár ilyen is:

```
}}((@K)<=6) // Itt végetért a belső ciklus, ami előltesztelő volt
```

A fenti esetben tehát elől IS és hátul IS tesztel a ciklusunk, utóbbi esetben természetesen ugyanúgy csak akkor folytatódik a ciklus, ha a ciklus végén kiértékelt feltétel eredménye igaz.

## Előre rögzített fix számszor lefutó ciklus

Nagyon gyakoriak azonban a programokban a ciklusok azon alkalmazásai, amikor előre pontosan lehet tudni, hányszor kéne annak a ciklusnak lefutnia! Ezek a „fix darabszámszor lefutó ciklusok”. Sőt, megkockáztatom az állítást, hogy talán ezek a ciklusok leggyakoribb alkalmazási területei... Nos, a mau nyelv kínál erre egy rendkívül könnyű, kényelmes, előkelő megoldást! E ciklusfajtának a vázlata:

```
{| darabszám  
// itt állnak az utasítások  
|}
```

A fenti példában a „darabszám” helyén természetesen tetszőleges (amúgy unsigned long long típusú) aritmetikai kifejezés állhat, amit csak egyszer értékel ki, amikor legelőször ugrik be e ciklusba. Azután soha nem foglalkozik már vele. Azaz ha a „darabszám” helyén egy változó van, annak értékét is csak egyszer olvassa ki, utána már ha az a változó megváltozik, az se számít neki, nem befolyásolja hogy hányszor hajtódik végre a ciklus.

Fix számszor lefutó ciklust is „megpatkolhatunk” egy, a végén tesztelendő feltétellel. Azaz e ciklus is mint a hátultesztelők, egyszer mindenképpen végrehajtódik. És itt se lehet whitespace a `|}` és a feltétel kezdetét jelölő nyitó „(” jel közt. Ügyeljünk azonban rá, hogy az előbbi hátultesztelő ciklusfajta akkor ugrik vissza egy újabb „menetre”, ha a végén megadott feltétel IGAZ volt - e most tárgyalt fix számszor lefutó ciklusnál azonban az a helyzet, hogy a végére elhelyezett feltétel IGAZ volta esetén KIUGRIK a ciklusból! Azaz: e ciklusfajta - amennyiben van a végén megadva feltétel - LEGFELJEBB annyiszor hajtódik végre, ahányszor azt a ciklus fejrészében specifikáltuk neki, de egyszer mindenképpen. Íme egy példa rá:

```
#c@c = 26 // c=26, ennyiszor hajtódik majd végre a ciklus - MAXIMUM.  
{| #c@c // c-darabszámszor fut le majd ez a ciklus  
? (@c)+'@; // kiiratás
```



```
#c--@c // c-=1
|}(((@c)+'@) == S) // kilépünk 'S'-nél
// Ciklus vége
/; // Üres sor kiírása
"Itt a vége fuss el véle!" /;
```

Gyakran hasznos lehet, ha le tudjuk kérdezni a fix darabszámszor lefutó ciklus esetén, épp mennyi a ciklusváltozó értéke - még ha nincs is ez egy külön változóval jelölve a ciklusfejrészben! Íme ennek használata:

```
{| 6
?l ?l; /;
|}
```

Eredménye:

```
6
5
4
3
2
1
```

Mint látható, a ciklusváltozó értéke rendre CSÖKKEN, erre ÜGYELJÜNK! Első értéke konkrétan az amit megadtunk neki, s a nulla értéket sosem éri el!

Valamint jó tudni, hogy bár a **|** után a ciklus fejlécében egy unsigned long long értéket tárol el a kis aranyos, de a **?l** rendszerváltozó ezt nekünk simán unsigned int értékként adja vissza... Ez többnyire nem baj, általában ezen értéktartomány is megfelelő nekünk. Ha azonban a típushelyesség fontos, akkor sincs semmi gond, lekérdezhetjük unsigned long long értékként is, azaz a maga eredeti típusában, a következő módon:

```
{| 6
?g ?#g "?l"; /;
|}
```

Eredménye természetesen ugyanaz, mint az előzőnek. Azaz, a ciklusváltozó értékét típushelyesen, unsigned long long értékként a

```
?#g "?l"
```

rendszerváltozó szolgáltatja nekünk, unsigned int értékként pedig simán a

```
?l
```

rendszerváltozó.

Rendelkezésünkre áll e ciklusok esetén a

```
?-
```

rendszerváltozó is, ami telibe pontosan ugyanaz mint a **?l** épp csak annál **eggyel KISEBB** értéket ad vissza. Tapasztalataim szerint ugyanis nagyon gyakori, hogy a ciklusváltozó értékénél eggyel kisebb értékre van szükség. (Amiatt, mert a ciklusváltozó sosem éri el a 0 értéket).

Amennyiben mégis úgy vagyunk kíváncsiak a ciklusváltozó értékére hogy az számunkra lekérdezés közben igenis nullától növe legyen visszakapva, akkor a **|l** jelet használjuk a lekérdezésre:

```

{ | 5;
?c ?|; /;
|}
"Másik:\n"
{ | 3;
?g {|}; /;
{ | 5;
?g {|};
|}
/;
|}

```

Eredménye:

```

5
4
3
2
1
Másik:
0
01234
1
01234
2
01234

```

Ami a **{|}** által visszaadott ciklusváltozó-érték TÍPUSÁT illeti, az „transzparens” - azaz, bár a veremtárból annak eredeti, **#g** típusaként olvassa ki, de azonnal castolja éppen arra a típusra, amit a lekérdezés helyén elvár a mau program. Azaz nyugodtan használhatjuk bármiféle numerikus típus helyén, bár természetesen ha az értéke kellően nagy, mondjuk 1000, akkor már nem fér bele mondjuk egy **#c** értéktartományba, emiatt a progink furcsa dolgokat produkálhat, azaz erre illik ügyelnünk...

Továbbá, lekérdezhetjük egymásba ágyazott ciklusok esetén nemcsak a legbelső ciklus ciklusváltozójának értékét (ez utóbbit csinálja ugyebár a **{|}** változó) de a külső ciklusok valamelyikének ciklusváltozóját is - igaz, legfeljebb 10 mélységben (bár szerintem itt a „magasságban” szó helyénvalóbb volna). Itt egy példa majd jön a magyarázat:

```

#!mau
{|3
{|2
?c {1}; " " ?c {|}; /;
|}
|}
XX

```

Eredménye:

```

0 0
0 1
1 0
1 1
2 0
2 1

```

A dolog lényege a **{1}** jelsorozat. Ez egyetlen szimbólum, azaz az „1” helyén itt nem állhat aritmetikai kifejezés, és szóköz se lehet e 3 karakter közt. Ellenben az „1” helyén állhat a **0123456789** karakterek bármelyike. Mindegyik egy ciklus-szintet reprezentál. A „0” jelenti az aktuális ciklust, azt tehát amiben épp lekérdezzük e jelsorozat által a ciklusváltozót, azaz a **{0}** teljesen ekvivalens a **{|}** szimbólummal. A **{1}** jelenti az eggyel kijebbi levő ciklust, azt tehát amibe az aktuális be van

ágyazva, a **{2}** jelenti az ezen kívülit, és így tovább, mint látható maximum 9-ig mehet ez a számozás, ami szerintem bőségesen elegendő.

Minthogy a **{1}** és a **{0}** ekvivalensek, felmerülhet a kérdés, mi az értelme annak, hogy két külön szimbólum is van ugyanarra a feladatra! Hát, nem sok. Elvileg a **{1}** gyorsabb, mert ezesetben rögvest tudja az interpreter hogy a legbelső ciklusról van szó, és nem kell így neki ciklus-szintet számolnia a középen levő számjegy ASCII értékéből, majd ezzel nem kísérel meg megszorozni a veremmutatókat, stb. Na most, kimértem ezt a sebességkülönbséget benchmarkokkal, több tesztben is, és úgy találtam, hogy ez az elmélet teljesen igaz a gyakorlatban is - mindazonáltal a különbség a két variáció közt mindössze mintegy 1 EZRELEK! Bár eléggé sebességmániás vagyok, de őszinténszólva ennek még én se tulajdonítok nagy jelentőséget. De mert HÁTHA fontos lehet mégis valakinek, végülis ciklusról van szó, hát hadd maradjon. Továbbá, a **{1}** jelsorozat *alakja* is jobban utal szerintem a funkcióra, minthogy ebből rögvest látszik hogy itt a fix számszor lefutó ciklussal lehet ez az izémizé valami összefüggésben.

Akit érdekel hogy effélet miként lehet kimérni benchmarkkal a mau nyelvben, annak itt a tesztprogram:

```
#!mau
#g@g=30000000;
#t@a;
{|#g@g;
#g@G={|};
|}
#t@b;
{|#g@g;
#g@G={0};
|}
#t@c;
#g@a=#t@a; #g@b=#t@b; #g@c=#t@c;
"{|} 1. variáció ideje: " ?g (@b)-(@a); /;
"{0} 2. variáció ideje: " ?g (@c)-(@b); /;
XX
```

Ügyeljünk azonban arra az esetre, amikor egy belső ciklus inicializálásához a külső ciklus ciklusváltozóját óhajtjuk felhasználni, mert alattomos csapdába szaladhatunk bele... Nézzük csak ezt a kis példát:

```
#!mau
{|5
{| {|}+1;
?c {1} " " ?c {|}; /;
|}
|}
XX
```

Eredménye:

```
0 0
1 0
1 1
2 0
2 1
2 2
3 0
3 1
3 2
3 3
4 0
4 1
4 2
4 3
4 4
```

Na most, e programból ez a sor az érdekes:

```
{| |}+1;
```

Első pillanatra ez valami ökörségnek tűnik. Ugye a **|** jel után az jönne, hányszor fusson le a ciklus. E ciklust mi annyszor akarjuk lefuttatni, amennyi épp a külső ciklus ciklusváltozójának aktuális értéke, pontosabban annál eggyel többször, mert ugye ha az épp nulla, akkor reklamálna az interpreterünk hogy ő nem tud nullaszer lefuttatni fix számszor lefutó ciklust. Emiatt van az hogy hozzáadunk egyet. Így is írhattuk volna emiatt azt a sort:

```
{| ++{|};
```

Ez tehát rendben van, oké, hozzáadunk egyet. De miért nem a **{1}** szimbólum van ott a **{|}** helyett, holott a **{1}** kéne utaljon a külső ciklusra, és a belső ciklusban a ciklusmagon belül a kiíró utasításban tényleg ezzel hivatkozunk a külső ciklus számlálójára?!

Nos azért nem a **{1}** áll ott, mert amikor azt a kifejezést kiértékeli az interpreter, akkor még nincs belső ciklus, azaz, akkor még javában a látszatra „külső” ciklus az aktuális! Az a „belső”... Gondoljuk csak el, amint elér a **|** jelhez, akkor kezd csak megcsinálni azt a belső ciklust. Ehhez az kell hogy beolvassa, hányszor kell majd végrehajtania azt. E beolvasott értéket kell aztán elmentenie egy veremtárba (sőt 2 külön veremtárba is de efféle technikai részletek most mellékesek). Azaz javában „dolgozik” az új ciklus megcsinálásán, az tehát még NINCS KÉSZEN, azaz NEM LÉTEZIK! Ezen pillanatokban tehát a **{|}** jel (vagy a **{0}** jel) egyszerűen nem vonatkozhat annak a ciklusnak a ciklusváltozójára ami ciklus még nem létezik, nem jelentheti azt a ciklusváltozót amit ki se tudna olvasni a veremtárból, mert még be se lett oda rakva semmi... Tehát hiába áll az interpreter programmutatója a forráskódban a **|** jelek után, valójában még a „külső” ciklus dolgozik ekkor, egészen addig amíg a **|** utáni aritmetikai kifejezést, mely a leendő új ciklus darabszámát meghatározza, ki nem értékeli. Azaz, az ilyesmire **NAGYON DE NAGYON** ügyeljünk...!!!!!!!!!!!!!!!!!!!!

**Nagyon fontos észben tartani** e ciklusfajtánál, hogy az e típusú (azaz a fix számszor lefutó) ciklusból való kiugráshoz nem jó a **TT** és **EE** utasítás, mert itt nemcsak a visszatérési címet kellene törölni a veremtárból, hanem a ciklusváltozó értékét is. Emiatt ha egy fix számszor lefutó ciklusból idő előtt ki akarunk ugrni, folyamodjunk ahhoz a megoldáshoz, hogy a végére beteszünk valami feltételt mint azt korábban bemutatam. Amennyiben ez se megfelelő mert okvetlenül a ciklus közepén kell kiugrani belőle, akkor csináljunk valami efféle trükköt:

```
{| 100 // E ciklus 100 szor fut le legfeljebb
// valami utasítások
if( /* itt a feltétel ami miatt azonnal ki kell ugrani a ciklusból */ ) T #c@i=20; »$ki
// valami utasítások
$ki |}{(#c@i)==20) // kiugrik ha ez a @c változó 20
```

Megjegyzem, ha rászorolunk a fenti trükkre, akkor erős a gyanúm, hogy ott valami rossz programozási technikáról, rossz szervezésről van szó.

A fentinel előkelőbb megoldás (bár ismétlem az efféle rendkívüli kiugrásokkal „alapból” nem szimpatizálok mert rossz programszervezést sejtet bennem...) hogy a rendkívüli kilépéshez a közönséges **T** vagy **E** utasításokat használjuk, vagy úgy általában véve abszolút bármiféle ugróutasítást amit csak akarunk (de a **TT** és az **EE** utasításokat **NE!**) – csak közvetlenül a kiugrás előtt adjuk ki ezt a speciális utasítást is:

^^

Ez semmi mást nem csinál, mint hogy mindössze törli a veremtárból épp a fent említett ciklusváltozó értékét (amit emiatt a továbbiakban természetesen nem is kérdezhetünk le), valamint a visszatérési címet is. Ezt például a következő módon használhatjuk, hogy egy konkrét példán is bemutassam:

```
#c@c=g;
{| 10
?(@c) ?|; " ... " ?(@c) ?-; /;
if(?!==5) T ^^ »$ki
|}

$ki
"Vége!\n"
```

A fenti kis szösszenet futási outputja:

```
10 ... 9
9 ... 8
8 ... 7
7 ... 6
6 ... 5
5 ... 4
Vége!
```

## String hosszától függő ciklus

Van a mau nyelvnek olyan ciklusa is, ami annyiszor fut le, ahány karakterből áll egy string. Ez a ciklusfajta így néz ki:

```
{! string
// valamilyen utasítások
!}
```

Vagy:

```
{! string
// valamilyen utasítások
!}(feltétel)
```

ahol a „string” természetesen egy stringKIFEJEZÉS is lehet, s a ciklus annyiszor fut majd le, ahány bájtól áll a string. (Ha a stringhossz nulla, akkor hibajelzéssel elszáll a program...)

A „feltétel” a végén meg természetesen arra vonatkozik, hogy ha annak értéke **IGAZ**, akkor kiugrik a ciklusból.

Természetesen a **?|** függvényünk a ciklus belsejében itt is a ciklusváltozó aktuális értékét adja vissza, ami ugye egy érvényes index a stringkifejezés valamely karakterére, de ez most nullától növekszik folyamatosan. A **?-** rendszerfüggvény is használható ezesetben, ami itt is a **?|** értékénél eggyel kisebb számot ad vissza, de ezzel ne nagyon bűvészkedjünk, mert ezesetben a **?|** amúgy is nulláról indul...

Továbbá, e ciklus belsején belül rendelkezésünkre áll a

```
?#c "{!"
```

rendszerfüggvény, ami pontosan azt a karaktert adja nekünk vissza, amire épp „mutat” a ciklusváltozó aktuális értéke, vagyis a fentebb említett **?! rendszer-**változó, azaz a ciklus „aktuális” karakterét kapjuk meg ezáltal.

Ebből a ciklusból is óvatosan kell azonban kiugranunk, ha nem várjuk meg míg lefut a megadott string teljes hosszára. Erre ugyanis érvényes mindaz, amit a fix darabszámszor lefutó ciklusnál elrebegettem mint problémákat, épp csak egyáltalán nem annyira súlyos a helyzet hanem még sokkal olyanabb, ugyanis a ciklus fejrésznél megadott stringkifejezést a drága elmenti egy ideiglenes string-változóba, s ennek címét egy külön veremben eltárolja. Rendkívüli kiugráskor tehát ezen külön veremből törölni kell e string címét, de még azelőtt fel is szabadítani a neki lefoglalt memóriaterületet...

Azaz: ha mindenáron ki akarunk ugrani e ciklusból a vége előtt, akkor ugyanazt kell tennünk mint a fix számszor lefutó ciklusnál, de nem a ^^ utasítást kell kiadnunk a kiugrás előtt, hanem ezt:

^^^

Azaz itt 3 db „felfelenyíl” vagy más néven „kalap” áll egymás mellett, és nem lehet köztük semmiféle whitespace. Példa a használatára:

```
#s@s="abcdefghijklm"
{! @s ? ?#c "{!";
if(?!==4) T ^^^ »$ki
!}
$ki /;
```

## Utasításblokkok

Az **if** és a **ha** utasítások esetén felmerülhetett a Tisztelt Olvasóban az elégedetlenség érzése, gondolván hogy ez mind szép és jó, de akkor is rút hiányossága a mau nyelvnek, hogy az **if** vagy **ha** utasítások utáni „then” vagy „else” ágak, amiket ugye e nyelvben a **T** illetve **E** utasítások valósítanak meg, nem ágyazhatóak egymásba többszörösen, holott ez minden „normális” programozási nyelvben rég megoldott!

Nos, ez olyasmi amit a mau valóban nem tudott a 14-edik kiadásáig bezárólag, de őszintén megvallva, nem is éreztem e funkció hiányát... De hogy „neve legyen a gyerekeknek”, a 15-ödik kiadásba már beletettem e ficsőrt. Ez pedig a következőképp néz ki - jöjjön előbb a példaprogram, majd a magyarázat:

```
#!mau
{|6
"Ciklusváltozó=" ?c {|}; /;
if({|}>=3);
-> "Most a ciklusváltozó " ?c {|}; /;
"Ez is az első blokk!\n"
if({|}>3);
-> "A ciklusváltozó nagyobb mint 3!\n"
if({|}>=5);
-> "A ciklusváltozó 5!\n"
<-
<-
"A ciklusváltozó nagyobb vagy egyenlő mint 3!\n"
<-
"Ezt mindenképp kiírja!\n"
|}
XX
```

Eredménye:

```
Ciklusváltozó=0
Ezt mindenképp kiírja!
Ciklusváltozó=1
Ezt mindenképp kiírja!
Ciklusváltozó=2
Ezt mindenképp kiírja!
Ciklusváltozó=3
Most a ciklusváltozó 3
Ez is az első blokk!
A ciklusváltozó nagyobb vagy egyenlő mint 3!
Ezt mindenképp kiírja!
Ciklusváltozó=4
Most a ciklusváltozó 4
Ez is az első blokk!
A ciklusváltozó nagyobb mint 3!
A ciklusváltozó nagyobb vagy egyenlő mint 3!
Ezt mindenképp kiírja!
Ciklusváltozó=5
Most a ciklusváltozó 5
Ez is az első blokk!
A ciklusváltozó nagyobb mint 3!
A ciklusváltozó 5!
A ciklusváltozó nagyobb vagy egyenlő mint 3!
Ezt mindenképp kiírja!
```

A dolog nyitja (sőt „zárása” is, hehehe...) a **->** és a **<-** utasítások. Érdekesképpen elárulom, hogy a **<-** utasítás megvalósítása programozástechnikailag életem „legnehezebb feladata” volt, ez az utasítás ugyanis SEMMIT SE CSINÁL... Tényleg, minden vicctől mentesen! Technikailag ez teljesen egyenértékű egy üres utasítással... Ez csak arra kell hogy ott legyen a programkódban, ez ugyanis igazából a **->** utasításnak fontos...

Tehát a szintaxis: A **->** utasítás megvizsgálja ugyanazt az ifflaget, mint amit a **T** vagy **E** utasítás is vizsgál. Na most ha ez igaz, azaz 1 értékű, akkor ő maga se csinál semmit, eképp ebből az következik, hogy végrehajtódik az a rész a kódban, ami őutána, a **->** után található. Ha ellenben az ifflag értéke nulla, akkor mindenekelőtt azonnal azt csinálja mint a **//** utasítás, azaz elugrik a következő programsor elejére. Ott megvizsgálja, hogy a sor első 2 karaktere micsoda. A lehetséges variációk amiket figyelembe vesz, az az, hogy ezen első két karakter egy újabb **->** utasítás, vagy egy **<-** utasítás, vagy akármi más. Ha akármi más, akkor megint azonnal elugrik a következő sor elejére. Ha egy **->** utasítás, akkor egy számlálót megnövel 1-el, s elugrik a következő sor elejére. Ha egy **<-** utasítás, akkor ezen számlálót csökkenti eggyel, kivéve ha az már eleve nulla lenne. Ha ugyanis nulla, akkor vége van ennek az egész játszadózásnak, és folytatja a program végrehajtását a megfelelő **<-** jel utáni résztől.

Mindebből a következő tudnivalók jegyzendők meg:

1. Efféle többszörösen egymásba ágyazott utasításblokkokból akármikor kiugorhatunk bármi nekünk tetsző módon, mert ennek a vezérlési szerkezetnek abszolút semmi köze nincs a vermekhez.
2. Az utasításblokkoknak a KEZDETE IS, és a VÉGE IS kizárólag, de TÉNYLEG KIZÁRÓLAG, csakis és egyedül csupán csak egy sor legislegelején állhat, tehát MUSZÁJ, hogy a **->** és a megfelelő **<-** utasítás is éppen pontosan egy programsor első két karaktere legyen! (Whitespace se állhat előttük!)



Tulajdonképpen, a `->` utasítás működik akkor is ha nem egy sor első két karaktere. De rém rossz ötlet nem oda tenni, mert ha módosítjuk a programunkat, s az a rész ahol ő szerepel bekerül egy másik `->` utasítással nyitott blokk belsejébe, akkor a külső `->` utasítás már nem fogja megtalálni e belsőnek a kezdetét, mert ugye ő csak a sorok elejét vizsgálhatja, s ebből eszméletlen bonyodalmak származhatnak... Hasonlóképp, természetesen a `<-` utasítás is működik bárhol, nemcsak sor elején, tudniillik miért is ne működne, neki könnyű a dolga, mert hiszen SEMMIT SE kell csinálnia! Igenám, de ha máshova tesszük, a `->` utasítás nem találja őt meg, s emiatt rosszul számolhatja a blokkok végét...

3. Ezen utasításblokkok annyi mélységben ágyazhatóak egymásba, amennyi belefér egy **#1** típusú változó értéktartományába. Minthogy ez 2 a 32-ediken nagyságú, azt hiszem e téren csak a RAM mérete szab nekünk határt...

Felmerülhet a kérdés, miért van ez ilyen „hülyén” megoldva, hogy a blokkokat csak a sorok elején lehet/szabad nyitni és zárni! Hiszen e doksi elején én magam dörögtem a python nyelv ellen, hogy milyen ökörség a kód szintaxisát a külalakhoz kötni...

Én valóban nem vagyok a híve az ilyesminek, de ez azért nem olyan zavaró, mint a pythonnál. Ott egy átrendezés teljesen összezagyválja a kódot. És ha az egyik szövegszerkesztő nem ugyanannyi space-ot feleltet meg egy tab-nak mint a másik, máris baj van. A maunál nem: az teljesen egyértelmű minden szövegszerkesztőnek, hol kezdődik egy sor! És akkor sincs baj ha bármely blokkba beírunk akárhány akármilyen sort, akár üres sort is vagy új blokkot. A kód továbbra is érvényes marad. És könnyű megállapítani, hol kezdődnek a blokkok, mert csak a sorok elején kell számolgatni a `->` és `<-` utasításokat.

Na és hogy miért így van kitalálva... A gyorsaság miatt! Ha az interpreter tudja, hogy a szabály az, hogy blokk csak sor elején végződhet vagy kezdődhet, akkor egy `->` utasítás után csak azt kell tesztelnie egy blokk végének keresésekor, hogy az adott karakter egy sorvég-e. Ha ugyanis nem sorvég, nem kell még külön tesztelnie arra is hogy egy újabb `->` vagy egy `<-` utasítás-e, s ez jelentősen meggyorsítja a blokkok határainak keresését.

Elárulom azt is, a szintaxis ilyenét kialakítása bizonyos értelemben „felkészülés a jövőre”. Fontolgom ugyanis az ötletet, hogy a gyorsítás érdekében beolvasáskor valamiképp letárolja az interpreter a sorok kezdetének a pointereit magának. Ez némi memóriapocséklás árán sok mindent jelentősen felgyorsítana... Mostanában biztosan nem foglalkozom majd ezzel az ötletemmel, mert rengeteg fejleszteni való van még a mau nyelven amit messze sokkal fontosabbnak tartok, de majd valamikor, esetleg... Na és ha ez megvalósul, rém kellemetlen lenne bejelenteni, hogy az ezutáni mau interpreterek szintaxisa nem kompatibilis az előzőekével, tessék átírni minden korábban született mau programot hogy az utasításblokkok sor elején kezdődjenek és végződjenek... Még én magam is anyáznám saját magamat amiért így szopatom önmagam, hát még mások mit gondolnának rólam... Jobb tehát már most az elején szigorúan kikötni az ilyesmit!

Hasonlóképp ágyazhatjuk egymásba az „else” ágakat is, azaz azokat, amiket az **ifflag** hamis állapota esetén hajtunk végre. Ennek a blokknak a „bevezető jele” a `<-` jel, („mínusz-kisebb”), a blokkot pedig a `>-` jellel („nagyobb-mínusz”) kell lezárni, és ezek is csak egy sor legislegelején állhatnak. Példaprogram (semmi értelme különben):

```

#!mau
if(2<3);
-> "2<3\n"
"Igen, 2<3\n"
if(4>1);
-> "4>1\n"
<-
-< "4>1 else ág\n"
>-
if(6<10);
-> "6<10\n";
<-
-< "6<10 else ág\n"
if(0<4);
-> "0<4\n"
<-
>-
<-
"vége!"; /;
XX

```

## A switch-szerű vezérlési szerkezet

A mau nyelv is rendelkezik olyan esetkezelési módszerrel, ami a C nyelv „switch” utasításához némileg hasonlatos. Ennek szintaktikája a következő:

```

?! c
...K1
...K2
...Kn
.....

```

ahol a c, valamint a K1, K2...Kn értékek egy-egy unsigned char típusú kifejezés. A rutin beolvassa a c kifejezést, majd sorra megvizsgálja a következő programsorok elejét. Ha az a ... jellel kezdődik, kiértékeli az utána következő K1, K2 stb kifejezéseket is mint egy-egy unsigned char értéket, és ha a c értéke ezzel megegyezik, a vezérlés a megfelelő K kifejezés utánra adódik. Ha nem egyezik meg, tovább keresi a megfelelő ... -al kezdődő sort ahol már esetleg megegyezik. A megfelelő érték keresése akkor áll csak le egyezés hiányában, ha olyan sorra lel, ami egymás után 2 db ... karakterrel kezdődik, ez a „default”, azaz ha ilyenre lel innen folytatja e 2 ... jel után a programvégrehajtást, akármi legyen is a c értéke. Amennyiben azonban 3 db ... jelet talál egymás után, akkor azután folytatódik a vezérlés. Azaz egy efféle vezérlési szerkezetet mindenféleképpen le kell zárunk a ..... jelsorozattal, különben a program végéig keresne!

A fentiek bemutatása egy kis példán:

```

?! g
...a "a\n";
...b "b\n";
...c "c\n";
...d "d\n";
...e "e\n";
"Ez még az e betű sora!\n"
...f "f\n";
...g "g\n";
..... "valami más!\n";
"Ez is a valami máshoz tartozik!\n"
..... "Itt a vége!\n"

```

Fontos megérteni a következőket:

A „három pont” nem 3 darab különálló pont, azaz nem 3 darab ilyen: ".", hanem az a "..." karakter, aminek a kódja az UTF-8 szerint a 226,128,166 bájt sorozat! Továbbá, a fenti elágazásszervezésben amint a **?!** parancs kiértékelte az utána következő aritmetikai kifejezést, azonnal abbahagyja a sor további feldolgozását, nem veszi figyelembe, ami utána van. Ezenkívül, azt meg lehet csinálni, hogy ne írjunk olyan sort, ami csak 2 db „háromponttal” kezdődik, azaz nem definiálunk „default” eseményt, amit „nem illeszkedés” esetén hajt végre az interpreter, azt azonban semmiféleképpen sem szabad elfelejtenünk, hogy legutoljára legyen egy

.....  
azaz 3 db „háromponttal” kezdődő sorunk, ami azt jelzi hogy itt ér véget ez az egész miskulancia. Azaz a ..... utáni (a 3 db „hárompont” utáni) utasítások már mindenféleképpen végrehajtnak majd. Amint ugyanis belép egy eseményágba az interpreter, azt elkezd végrehajtani, s amint egy 1 db „hárompont”-tal kezdődő utasítássorozatra lel, azt azonnal ÁTUGORJA, azaz rögvest a következő sor elejére ugrik. Ha ott is efféle hárompontot talál, azt is átugorja, és így tovább, míg csak olyan sorra nem lel, ami 3 db egymás utáni „háromponttal” nem kezdődik.

Az egyes eseményágak nem muszáj hogy beleférjenek egyetlen programsorba, látható hogy folytatódhatnak több soron keresztül is, és egyáltalán semmivel se muszáj jelölni a végüket, mert nekik az a vég, hogy egy újabb hárompont-utasításhoz érkeznek. Ellenben egy eseményen belül nem nyithatunk újabb efféle szerkezetet a **?!** utasítással... Azaz, ezek nem ágyazhatóak egymásba. Viszont nyugodtan kiugorhatunk belőle bármiféle nekünk tetsző módon, például a » utasítással, ennek az esetkezelésnek ugyanis semmiféle köze sincs a vermekhez.

## Elágazás keresési eredménytől függően

Előre tudható, hogy rendkívül gyakoriak lesznek a programokban az olyan feladatok, hogy ellenőrizni kell, szerepel-e egy karakter egy stringben, s ha igen, az hányadik karakter. Erre is ad beépített lehetőséget nekünk a mau programnyelv, és ez a következő:

**?? c S x**

ahol c egy unsigned char érték, az S pedig egy stringkifejezés, x pedig egy unsigned int érték. Amennyiben a c karaktert megtalálja az S stringben, annak x-edik karakterétől a string végéig levő tartományban, akkor az **f.kerdojelkerdojel-pozicio** értékét (ez egy speciális rendszerváltozó) beállítja arra a számra ahol a stringben megtalálta. Ez a szám abszolút értékű, a string elejétől számítható! Ezesetben ugyanakkor egy másik speciális rendszerflag, az **f.kerdojelkerdojel** flag értéke 1 lesz. Ha nem találta meg a stringben a c karaktert, vagy x>=S azaz az S string hossza, akkor a pozíciószám nulla lesz, ellenben a flag értéke is nulla. Ha ebben a formában adjuk meg az utasítást:

**?? c S;**

akkor az x értékének a nullát tekinti automatikusan. Ne feledjük, hogy ezesetben MUSZÁJ lezárni az utasításunkat egy pontosvesszővel!

Mielőtt bemutatnám a fent említett rendszerflagokat miképp használhatjuk, ideírom ezen utasítás egy másik alakját is:

??#t c T x

A fenti változat egy TÖMBBEN keres. A T tömbben. Ez tehát itt már nem lehet KIFEJEZÉS, csak **tömbváltó**! Viszont a **T** itt a tömbváltó **nevét** meghatározó unsigned char típusú KIFEJEZÉS... Az x szintén egy unsigned int érték, a tömb ennyiedik elemétől keres. A c adatot keresi, amely olyan típusú, amilyen típusra a **t** karakter utal. A **#** jel közvetlenül a **??** után kell álljon, nem lehet köztük whitespace, s hasonlóképpen a **t** karakternek is közvetlenül a **#** után kell állnia. A **t** karakter lehetséges értékei:

c,C,i,I,l,L,g,G,d,D,f,s

Ezenfelül, a **t** karakter lehet egy "(" karakter is, azaz egy rendes gömbölyű nyitózárójel. Ezesetben amint azt a casting operátorról írtam e könyv elején, az utána következő unsigned char típusú aritmetikai kifejezés határozza meg a típust, s e kifejezés egy csukózárójellel kell lezárattassék.

Megengedett a

??#t c ^ T x

alak is, ekkor a szülő névtér tömbjében keres. (hogyan ez mi, arról lásd később a névterekről írtakat). Ha ebben a formában adjuk meg:

??#t c T;

akkor az x értékének a nullát tekinti automatikusan. Ne feledjük, hogy ez esetben MUSZÁJ lezárni az utasításunkat egy pontosvesszővel!

Na most hogy ezt hogyan is használhatjuk. Hát először is: A fent emlegetett **kerdojelkerdojel** flag értékét a

??

rendszeráltozó adja vissza mint unsigned int értéket, a kerdojelkerdojelpozicio értékét pedig a

?.

rendszeráltozó adja vissza szintén unsigned int értékként. Használata egyszerű, például kiirathatjuk a találatot így:

"Tömb pozíció: " ?l ?.; /;

Tehát: először lefuttatjuk a **??** paranccsal a keresést, majd azután amíg csak újabb keresést nem futtatunk le, a **?.** függvény az utolsó keresés eredményét adja vissza nekünk. Ennek eredményét különben egy külön mezőben tárolja (az F típusú struktúrában, ezt azoknak írom akik belenéznek netán az interpreter forráskódjába) és épp ennek a neve az **f.kerdojelkerdojelpozicio**. Amennyiben a keresés sikertelen volt mert nem találta meg a megadott adatot, akkor ennek értéke nulla. Ez megtévesztő lehet, mert hogy is különböztessük meg akkor ezt az esetet attól, amikor nagyonis megtalálta az elemet, de a tömb nulladik pozícióján? Nos erre szolgál az **f.kerdojelkerdojelflag** mező, aminek értéke 0 ha nem találta meg, és 1 ha megtalálta. Ezt természetesen a **??** függvénnyel kérdezhetjük le, mint unsigned int számot (ahogy ezt fentebb írtam már).

Igen ám, de ha már ott az az érték, miért kéne külön összehasonlítani az **if** utasítással, azért, hogy utána ugorhassunk az eredménytől függően?! Messze okosabb, ha kreálunk valami speciális ugróutasítást, ami ugyanúgy működik mint a **T,T,E,EE** — csak éppen nem az **f.ifflag** állapotát veszi figyelembe hanem ezen másik flagét!

Ezek az új ugróutasításaink legjobb ha pontosan ugyanazt a nevet viselik mint a régiak, csak bővítsük ki a nevüket egyetlen karakterrel - legyen ez a "." azaz a pont karakter! Alakjuk tehát:

T.  
TT.  
E.  
EE.

Mindez valami efféleképp mutatkozhat meg egy konkrét programban:

```
?? @t @Z; E. "Ismeretlen bejegyzéstípus az ideiglenes fájlban!\n"  
E. "Esetleg kitörölted vagy megváltoztattad a sor első karakterét?\n"  
E. "A hibás típusazonosító karakterkódja: " ?c @t; " Maga a karakter: " ? @t; /; XX;
```

(A fenti példában az **XX** egy eddig még nem említett utasítás, a program végét jelzi).

Mint látható a fenti példán, nem is alkalmaztunk „then” ágat azaz **T.** utasítást, csak az „else” utasítás megfelelőjét. Így egyszerűbb és gyorsabb, mintha negálnánk a flag visszaadott értékét.

Egy rövid példaprogram annak bemutatására, miként keresgélhetünk tömbökben meg stringekben értékeket:

```
#s@S="abcdefgh";  
  
?? c @S;  
"Pozíció: " ?l ?.; /;  
  
[@t[[4]]]; // lefoglalok területet 4 string részére egy t nevű tömbbe  
#s@t="Ez egy sima string, nem tömb!\n"  
// Feltöltöm értékekkel a tömböt  
#s@t[[0]]="kutya";  
#s@t[[1]]="macska";  
#s@t[[2]]="vadbarom";  
#s@t[[3]]="cica";  
"Kiíratom a 4 elemű stringtömb mindegyik értékét:\n"  
#i@i=0; { | 4 ?i @i; ". = " ?s @t[[#i@i]]; /; #i++@i; |}  
  
??#s "vadbarom" t;  
  
"Tömb pozíció: " ?l ?.; /;  
  
[#c@t=4]; // lefoglalok területet 4 unsigned char szám részére egy t nevű tömbbe  
#c@t[0]=10;  
#c@t[1]=11;  
#c@t[2]=12;  
#c@t[3]=13;  
  
"Kiíratom a 4 elemű unsigned char tömb mindegyik értékét:\n"  
#i@i=0; { | 4 ?i @i; ". = " ?c @t[[#i@i]]; /; #i++@i; |}  
  
??#c 11 t 8;  
  
"Tömb pozíció: " ?l ?.; /;
```

A futási eredmény:

```
Pozíció: 2  
Kiíratom a 4 elemű stringtömb mindegyik értékét:  
0. = kutya  
1. = macska  
2. = vadbarom  
3. = cica  
Tömb pozíció: 2  
Kiíratom a 4 elemű unsigned char tömb mindegyik értékét:  
0. = 10  
1. = 11  
2. = 12  
3. = 13  
Tömb pozíció: 0
```

A mau interpreter garantálja, hogy minden névtér létrejöttekor e két fontos rendszerváltozó (a pozíció és a flag) értéke kezdetben egyaránt nulla.

## 15. fejezet - Névterek

Nem csodálnám, ha Olvasóm, ki ezen írás által meg óhajt ismerkedni a mau programnyelvvél, már a kezdetektől fogva aggódott volna amiatt, hogy mégiscsak nagyon kevés lesz az a 256 darab változólehetőség a programnyelvünkben! Még akkor is, ha egy változó neve egyszerre több tényleges változót is jelölhet, mert több mezőből áll. Nos, ezen a gondon jelentős mértékben segítünk ezúttal, még-hozzá ROPPANT ELŐKELOŐEN! Olyan lehetőséget mutatok be ezúttal, ami igazából számos programnyelvből hiányzik, bár a legelterjedtebbekben kétségtelenül benne van. Ez pedig a „névterek” lehetősége. Ez tulajdonképpen azt mondja meg, egy adott nevű változó meddig „él”, azaz létezik. Ez szoros összefüggésben van az olyan furmányos huncutságokkal is, mint a függvények hívásakor történő input paraméterátadás, meg szintén a függvények befejezésekor történő output paraméterek átadása.

A programnyelvek efféle kérdés szempontjából mint a névtér, illetve paraméterátadás, nagyjából a következő csoportokra oszthatóak:

1. Egyetlen névtér van, minden változó mindig mindenhol látható. Ilyen a Basic nyelv legtöbb változata, például a régi C-64 -es számítógépen megvalósított Basic is.
2. Több névtér van. Hogy hány, az az adott programnyelvtől függ. Némelyikben olyan rengetegsok, hogy cseppet se könnyű észben tartani, melyik változónak meddig terjed az élettartama.

Amikor a nyelvben több névtér van, az tipikusan azt jelenti, hogy egy meghívott függvény nem látja az őt hívó program változóit, ezért akár ugyanolyan néven használhat ő is változókat, melyeknek azonban lehet teljesen más jelentésük is, eképp más értékük is, s ezen változók értékének változása semmiféle módon nem befolyásolja a hívó program hasonló nevű változóinak értékét. Ez természetesen remek dolog, mert ha ezt mi megvalósítjuk, máris nem tűnik kevésnek az a változónév-mennyiség, amivel rendelkezünk!

Igenám, de ekkor merül fel az a gond, hogy ha a hívott program egyetlen árva változót se lát a hívó névterében szereplők közül, akkor miként is kaphat onnan input paramétereket! S ennek a gondnak a fordítottja is megjelenik: mikor befejezi a futását a hívott program, akkor oké hogy a legtöbb változója nyugodtan megsemmisülhet, de NÉHÁNY változó eredményét mégis vissza kéne juttatnia a hívónak, mert hát ugye nyilván azért hívtuk meg a hívott rutint hogy valami hasznos tevékenységet folytasson, s annak igényelnénk az eredményét!

Erre a gondra aztán az eddig elkészült programnyelvek a legkülönbözőbb technikákat fejlesztették ki. A legismertebb talán a C módszere, mely (kissé leegyszerűsítetten elmagyarázva a dolgokat) abból áll, hogy vannak úgynevezett

„globális” változók, amiket minden függvény láthat, ezen kívül azonban a függvény csak a maga változóit láthatja. Input paraméterként egy változólistát kap, aminek értékeit megfelelteti formálisan a maga bizonyos változóinak, érték szerint. Eredményül a függvény csak egyetlen értéket adhat vissza, máskülönben kénytelen a globális változókat használni, vagy pointerekkel bűvészkedni. Utóbbira a C++ nyelv ad némi „emberközelibb” megoldást, a „referencia típusú paraméterátadást”, de igazából az is csak pointer, csupán ezesetben a pointerekre való hivatkozás kissé el van rejtve és automatizálttá téve. Emellett pedig a függvények mind egyenrangúak, nem lehet olyasmit csinálni, hogy az egyik függvény belsejébe beépíték egy másik függvényt.

Na most, természetesen a mau nyelvben is az van hogy minden függvény külön névtérrel rendelkezik, de a függvényekről később lesz szó, másik fejezetben. Ehelyütt azt szeretném elmagyarázni, hogy egy függvényen belül (és természetesen az először elindított főprogramon belül is) definiálhatunk külön névtereket, melyek teljesen úgy viselkednek mintha valamely külön függvényben lennének, az egyetlen kivétel az, hogy ezen esetekben a címkék (azok a **S** jellel kezdődő „dolgocskák” tehát) azok továbbra is globálisak maradnak.

Egy külön névtér felépítése a következőképp néz ki:

**{** A dupla nyitó kapcsos zárójellel kezdődik a névtér meghatározása. Ekkor létrejön egy új „mau” nyelvű program adatterülete, azonban a programkód nem változik. Ez a gyakorlatban azt jelenti, hogy e dupla nyitó kapcsos zárójel után - mindaddig míg egy SZIMPLA, azaz NEM DUPLA csukó kapcsos zárójelet nem talál - minden teljesen ugyanúgy működik mint korábban, EGYETLEN kivételtől eltekintve: hogy az értékadó utasításunk, tehát ami így kezdődik hogy mondjuk

**#x@v=**

(ahol az x helyén természetesen a megfelelő típusra utaló karakter áll), tehát ez az utasítás és KIZÁRÓLAG ez!, — szóval tehát ez innentől kezdve úgy fog dolgozni, hogy a „jobbérték”-et amikor kiszámítja, tehát azt amit majd egy változó eredményül kell adnia, azt továbbra is úgy számolja, hogy a kifejezés kiértékelésénél minden adatot onnan vesz ahonnan eddig, azaz a „szülő” névtérből, de ezen adatokat, az eredményt már nem a szülő névtér változóiba teszi bele, hanem az új névtér változóiba, mert neki már az a „balérték”!

**}** Az egy darab csukó kapcsos zárójel az, ami az egész mindenség lényege: innentől kezdődik az új névtérben végrehajtódni a program! Ez tehát tulajdonképpen a „gyermekprocessz”. Vegyük észre ugyanis, hogy gyakorlatilag „forkoltuk” a programunkat: a végrehajtandó kód ugyanaz maradt, de az adatterület teljesen megváltozott, kivéve hogy néhány változóba beletöltöttünk input paramétereket. A többinek az értéke azonban nulla.

**{** Innentől kezdve - tehát a szimpla nyitó kapcsos zárójeltől kezdve - egészen addig míg egy dupla csukó kapcsos zárójellel nem találkozik, még mindig a „gyermekprocesszben” dolgozik, azaz az új névtérben, de úgy, hogy az értékadó utasításunk (amit fentebb említettem, meg ennek „nem dokumentált” párja) a kifejezés kiértékelésénél a változókat a maga új névtéréből veszi, ellenben az eredményt a szülő program névtérének változóiba pakolja.

**}}** Ez fejezi be az új névtérben folyó munkát, innentől visszatér minden a szülőprocessz névtérébe.



Kissé rövidebben megfogalmazva, más szavakkal, ez a következőt jelenti:

## SZÜLŐ NÉVTÉR

**{ INPUT PARAMÉTEREK ÁTADÁSA }** Ezt nevezhetjük a névtér vagy függvény fejlécének

**Program az új névtérben** Ezt a névtér fő része, itt folyik a „munka”, ez a függvény törzse.

**{ EREDMÉNYEK ÁTADÁSA }** Ez a névtér vagy függvény „lábléce”

## SZÜLŐ NÉVTÉR

A névtér nem őrzi meg a keletkezett változók tartalmát a belé való új belépés idejére!

A névterek tetszőleges mélységben egymásba ágyazhatóak. A névtér aktuális mélysége le is kérdezhető ezzel az (unsigned int értéket visszaadó) rendszerváltozóval:

**?Y**

Ennek értéke kezdetben (a főprogramban) természetesen 0.

Fontos **KIHANGSÚLYOZNI**, hogy a címkék ugyanolyan értékűek maradnak az új névtérben, azok értéke ugyanis a forráskód beolvasásakor dől el. (Ez természetesen nem vonatkozik azon címkékre amik igazi függvényekben vannak elhelyezve, ez nyilvánvaló, azonban itt most egyetlen függvényen (vagy épp a főprogramon) belül elhelyezett különböző névterekről van szó).

Ezenfelül, a fentiekből kifolyólag, ésszél kell lenni az efféle névterekben folyó ugrándozások esetén. Ugyanis képzeljük csak el, hogy valami rosszul specifikált » utasítás során (vagy máshogy) elhagyjuk a névteret, s azon kívülre kerül a vezénylés! A változók meg közben továbbra is azok maradnak, amik az új névtérben keletkeztek... Irtó nagy kavarodás lesz ám belőle!

Azaz, a mau nyelven való programozásnál ésszél kell lenni de nagyon ám. Itt hihetetlen módon lehet trükközni, mert a nyelv maximális szabadságot ad a programozónak, ugyanis majdhogynem assembly nyelven programozol ha a mau nyelvet használod. Maximális szabadságod azonban vonatkozik maximális idiótaságok, hülyeségek, baromságok és kolosszális hibák elkövetésének lehetőségére is...

És ezt rém komolyan kell venni tényleg. Hallottam olyan véleményt, mely szerint a C nyelv állítólag tulajdonképpen egy felcsicsázott Assembler csak. Na most ha e vélemény némileg túlzás is, de az talán nem, hogy a C nyelv a „magas szintű” programnyelvek közt talán a legalacsonyabb szintű... Azaz, az volt eddig. A mau nyelv azonban minden bizonnyal a C és az Assembly közt helyezkedik el e szempontból. Semmiképp sem érezném gúnyolódásnak ha valaki azt állítaná, hogy szerinte a mau programnyelv tulajdonképpen egy „interpreter stílusú/típusú assembly, sok jópofa makrószerűséggel megtámogatva”, vagy hogy a mau, az „az eddig létrehozott legmagasabb szintű assembly nyelv”. E megállapításokban ugyanis kétségtelenül sok igazság rejtezik.

Ezenfelül, tartsd észben, hogy valahányszor csak a program elér egy **{** utasításhoz, akkor kénytelen mindig és újra meg újra létrehozni egy új névteret, ami egyenértékű egy teljes „mau programobjektum” adatterületének „legyártásával”, azaz a változóknak, flagoknak meg a satöbbiknek memóriát foglalni, ezeket

lenullázni... Na és hát ez ugye IDŐ. Nem azt mondom hogy olyan rengetegsok, de azért itt ám akkor is súlyos kilobájtokról van szó, és emiatt baromira nem jó ötlet egy ciklus belsejében bűvészkedni ilyesmivel. Legalábbis ha az sokszor hajtódik végre, a program pedig sebességkritikus alkalmazás. Ha mindenáron külön névtérre van szükségünk arrafelé, akkor inkább tegyük azt, hogy ELŐBB hozzuk létre a névtérrel, s azon belül specifikáljuk a ciklust, és NE azt tegyük, hogy előbb jön a ciklus fejrésze, azután a névtér, majd a névtérből kilépés után a cikluslezáró utasítás! Mert aztán majd készítesz nekem erről bencsmarkokat, és panaszkodol hogy a mau nyelv „szar”, mert leahagyja őt sebességben még a ZX-Spectrum Basic-je is, holott te ezt egy sokmagos Core i7 processzoros gépen futtattad, sőt egyenesen valami katonai számítógépen, túlhúzott processzorokkal folyékony hélium hűtéssel... Ne felejtse el, MINDEN programnyelven lehet SZAR stílusban programozni, és nem hatékony kódot írni... Szerintem a mau nyelv kifejezetten előnyös azoknak, akik szeretnek a programjaikban „trükközni”, azoknak, akik „Igazi Programozók”! Ha nem tudod milyen az „Igazi Programozó”, elolvashatod az Interneten terjedő azon humoros írásból, aminek szintén „Az igazi programozó” a címe (itt van egy leírás róla: [http://www.szabilinux.hu/orlando\\_unix/igazi.html](http://www.szabilinux.hu/orlando_unix/igazi.html) ). Na most bár az a leírás kétségkívül humoros céllal készült és van benne sok túlzás — ám van benne rengeteg IGAZSÁG is... Itt van belőle e mondat például:

**„Az igazi programozó önmódosító kódot ír - különösen akkor, ha meg bír vele spórolni 20 nanoszekundumot egy kis ciklus közepén.”**

A mau nyelv nagyon alkalmas efféle „aljas húzások” elkövetésére, mert itt egy string maga is végrehajtható például (később lesz e lehetőség bemutatva e könyvben), azaz lehet önmódosító kódokat írni mauban, igen, de nem ezért idéztem e fenti mondatot, hanem mert ebből az a lényeg, hogy az „Igazi Programozó” igenis törődik az optimalizálással és nem fél azoktól az „aljas trükköktől”, amiket egyesek, akik kizárólag OOP-ben meg Haskellben meg más hátulgombolós nyelvekben tudnak gondolkodni, „ocsmány hack”-nek neveznek. Az „Igazi Programozó” is hacknak tartja ezeket természetesen, de SZÉP hacknak! Ő szeret optimalizálni, gyorsítani, kizárólag az optimalizálás kedvéért. Számára egy program egy MŰALKOTÁS, egy gyönyörű bit-építmény, ami akkor jó, ha a végletekig kihasználja az adott programnyelv minden apró lehetőségét. Nos, a mau nyelv effélékre majdnem annyi lehetőséget ad, mintha „nyers” assemblyben programoznál. Na de ahogy ott, úgy itt is rajtad áll minden...

Mindenesetre, egyvalami tutira nem igaz abból az írásból: hogy az „Igazi Programozó” Fortranban programozna. Az a kőkorszakban volt úgy, de már jó régóta az igazi programozók C nyelven programoznak. Szerintem. Most azonban, a mau nyelv megszületése után, ez úgy módosult, hogy az „Igazi Programozó” C nyelven programoz ha natívan futó bináris programot kell alkotnia s emiatt compiler típusú nyelv kell neki, ha azonban valahova jó egy gyors szkriptnyelven írt rutin is, akkor az „Igazi Programozó” azt kizárólag mau nyelven hajlandó „elkövetni”...

Visszatérve az utasítások ismertetéséhez: Egy konkrét példa erre az egészre, ahol ráadásul egymásba van ágyazva két névtér:

```
"Főprogram indul!" /  
#c@a:2 = 7  
#c@c:1 = 8  
#c@f:3 = 5
```

```

{{
  "Input adatok megadása az 1. szintű gyermekfolyamatnak:" /;
  "#c@a:2=@a:2 « CHILD-a = PARENT-a " #c@a:2=@a:2 ; /;
  "#c@c:1=@c:1 « CHILD-c = PARENT-c " #c@c:1=@c:1 ; /;
  } "Indul az 1. szintű gyermekprocessz!" /;
  "f értéke 0 kell legyen: f = " ?c @f:3; /;
  "a értéke 7 kell legyen: a = " ?c @a:2; /;
  "c értéke 8 kell legyen: c = " ?c @c:1; /;
  "Most az 1. szintű gyermekfolyamatban értékeket adunk az „a” és az „f” változóknak:" /;
  #c@f:3 = 9; "f értéke 9 kell legyen: f = " ?c @f:3; /;
  #c@a:2 = 50; "a értéke 50 kell legyen: a = " ?c @a:2; /;

  {{ "Indul a 2. szintű gyermekprocessz!" /;

  #c@a:2 = @a:2; "2.gyermekfolyamat „a” értéke = 1. gyermekfolyamat „a” értéke." /;
  }
  "Most vagyunk a 2-es gyermekfolyamat belsőjében." /;
  "Értéket adunk az „a” változónak." /;
  #c@a:2 = 111; "Az „a” értéke 111 kell legyen: a = " ?c @a:2; /;
  {
  "Most a 2-es gyermekfolyamat „a” értékét visszaadjuk az öt szülő 1-es folyamat „h” változójába!"
  /;
  #c@h:9 = @a:2
  }}
  "Végetért a 2-es gyermekfolyamat" /;
  "Most az 1-es gyermekfolyamatban vagyunk. A változók értékei:" /;
  "az „a” 50 kell legyen: " ?c @a:2 ; /;
  "a „c” 8 kell legyen: " ?c @c:1 ; /;
  "az „f” 9 kell legyen: " ?c @f:3 ; /;
  "a „h” 111 kell legyen: " ?c @h:9 ; /;

  {
  "Itt adjuk vissza az input adatokat az 1. szintről a 0 szintre:" /;
  "#c@a:2 = @a:2 « = PARENT-a = CHILD-a" #c@a:2 = @a:2 ; /;
  }}
  "Végetért az 1 szintű gyermekprocessz!" /;
  "Az „a” értéke szabad csak hogy megváltozott legyen:" /;
  "f értéke 5 kell legyen: f = " ?c @f:3; /;
  "a értéke 50 kell legyen: a = " ?c @a:2; /;
  "c értéke 8 kell legyen: c = " ?c @c:1; /;
  "h értéke 0 kell legyen: c = " ?c @h:9; /;
}

```

## Eredménye:

```

Főprogram indul!
Input adatok megadása az 1. szintű gyermekfolyamatnak:
#c@a:2=@a:2 « CHILD-a = PARENT-a
#c@c:1=@c:1 « CHILD-c = PARENT-c
Indul az 1. szintű gyermekprocessz!
f értéke 0 kell legyen: f = 0
a értéke 7 kell legyen: a = 7
c értéke 8 kell legyen: c = 8
Most az 1. szintű gyermekfolyamatban értékeket adunk az „a” és az „f” változóknak:
f értéke 9 kell legyen: f = 9
a értéke 50 kell legyen: a = 50
Indul a 2. szintű gyermekprocessz!
2.gyermekfolyamat „a” értéke = 1. gyermekfolyamat „a” értéke.
Most vagyunk a 2-es gyermekfolyamat belsőjében.
Értéket adunk az „a” változónak.
Az „a” értéke 111 kell legyen: a = 111
Most a 2-es gyermekfolyamat „a” értékét visszaadjuk az öt szülő 1-es folyamat „h” változójába!
Végetért a 2-es gyermekfolyamat
Most az 1-es gyermekfolyamatban vagyunk. A változók értékei:
az „a” 50 kell legyen: 50
a „c” 8 kell legyen: 8
az „f” 9 kell legyen: 9
a „h” 111 kell legyen: 111
Itt adjuk vissza az input adatokat az 1. szintről a 0 szintre:

```

```
#c@a:2 = @a:2 « = PARENT-a = CHILD-a
Végetért az 1 szintű gyermekprocessz!
Az „a” értéke szabad csak hogy megváltozott legyen:
f értéke 5 kell legyen: f = 5
a értéke 50 kell legyen: a = 50
c értéke 8 kell legyen: c = 8
h értéke 0 kell legyen: c = 0
```

## 16. fejezet - File-kezelés

Fájlból alapvetően kétféle létezhet: amit olvasni akarunk, s amit írni. Nem bölcs dolog vegyíteni a két típust... (Szerintem). Emiatt két külön beépített típus létezik a mau nyelvben a fájlokra: az egyik neve az lesz e leírásban, hogy **„inputfile”**, a másiké hogy **„outputfile”**. Természetesen mindegyikből 256 változónk lehetséges, és e változótípust az inputfile esetében a **„B”** karakter jelöli, mert ez olyasmi, amit **Beolvasunk**. Nem amiatt ez a karakter a típusazonosító, hogy magyarkodjunk, hanem mert az „i” és „I” karakterek amik az „input” szóra utalnának, már foglaltak más változótípusnak, ugye.

### Input fájlok

Íme rögvest egy kis mau program az input fájlok kezelésére. Előbb közlöm a teljes progit, majd utána soronként a magyarázatot, ahol szükséges:

```
#B@b="ido.cpp"; // Megnyitjuk a fájlt
if(#B@b) T "A file sikeresen meg lett nyitva!\n"
E "A file nem megnyitható!\n" XX

#l@m=#B@b; // A file mérete
"A file mérete = " ?s #l@m; " bájt\n"
{| 30 ; // beolvassuk az első 30 karaktert
#L@c=#B@b;
if(#L((#L@c)==-1)) T "\nItt a file vége!\n" XX
if(#L((#L@c)==-2)) T "\nOlyan fájlból próbáltál olvasni, ami nem is lett megnyitva!\n" XX
? @c; // és kiírjuk
|} // Ciklus vége
/;
"Most lezárjuk a fájlt\n"
[#B@b] // Itt zártuk le, ezzel az utasítással
"Most megnyitjuk a fájlt újra\n"
#B@b="ido.cpp"; // Megnyitjuk a fájlt újra
[#B@b=6] // Elugrunk a 6-odik bájtjához
{| 14 ; // beolvassuk innentől az első 14 karaktert
#L@c=#B@b;
? @c; // és kiírjuk
|} // Ciklus vége
/;
"Vége!\n"
XX
```

Magyarázat:

```
#B@b="ido.cpp"; // Megnyitjuk a fájlt
```

Mint látható, a fájl a „b” nevű változó által lett megnyitva, azaz ezen a változón keresztül hivatkozhatunk rá ezentúl. Itt egy konstans string adja meg neki a fájl nevét, de stringváltozón át is megadhatnánk azt, eképpen:

```
#s@b="ido.cpp";
#B@b=#s@b; // Megnyitjuk a fájlt
```

E fenti két sor is ugyanúgy megnyitná. Sőt, ennyi is elég volna:

```
#s@b="ido.cpp";  
#B@b=@b; // Megnyitjuk a fájlt
```

Ugyanis egy **#B** típusú változónak kizárólag stringet adhatunk értékül, amit ő meg akar majd nyitni, na most miután amúgy is stringet vár, ezért a **@b** által jelölt változót mindenképp stringként próbálja értelmezni, akkor is ha nincs előtte explicit módon megadva a **#s** casting operátor. A fentiekből az is látszik, hogy bár a stringváltozónknak is meg a fileváltozónknak is egyaránt **@b** a neve, de e két változónak a világon semmi köze egymáshoz.

```
if(#B@b) T "A file sikeresen meg lett nyitva!\n"  
E "A file nem megnyitható!\n" XX
```

E két fenti sorból az látszik, hogy az inputfile változóink átkonvertálhatóak unsigned char értéké is. (Az **if** ugyanis azt várja el). Ezen esetben az inputfile típus eredménye mindig egy 0 vagy egy 1 értékű bájt - nulla, ha a változóhoz nem tartozik megnyitott állomány, és 1, ha tartozik hozzá, azaz ha a fájl épp meg van nyitva. Az **if** sorát így is írhatnám, zárójelek nélkül:

```
if #B@b T "A file sikeresen meg lett nyitva!\n"
```

Csak szerintem zárójelezve az inputfile változót olvashatóbb a kód.

```
#l@m=#B@b; // A file mérete  
"A file mérete = " ?s #l@m; " bájt\n"
```

A fenti két sorból az látszik, hogy az inputfile típus castolható unsigned int értéké is, ez esetben pedig egyszerűen az épp megnyitott fájl méretét adja vissza (bájtokban). Ezen értéket aztán itt kiíratjuk stringként a **?s** utasítással.

```
{| 30 ; // beolvassuk az első 30 karaktert
```

Itt fentebb egy fix számszor lefutó ciklust indítottunk.

```
#L@c=#B@b;
```

E fenti sorban történik a fájlból a beolvasás, pontosan 1 bájt beolvasása. E beolvasás azonban nem signed vagy unsigned char, hanem signed int értéként történik, mert - amint az a következő sorban látható - le kell tudjuk tesztelni valamiképp azt is, hogy elértük-e az állomány végét.

```
if(#L((#L@c)==-1)) T "\nItt a file vége!\n" XX
```

Itt fentebb teszteljük hogy nem EOF-ot kaptunk-e, azaz nem értük-e el az állomány végét. Az **if** utáni **#L** casting operátor nélkülözhetetlen, mert az **if** alaphoz olyan kiértékelő rutint hív ami unsigned char értéket ad eredményül, de nekünk az kell, hogy a változónkat és a mínusz 1 számot még mint signed int értékeket hasonlítsa össze! Ellenben lehet mindezt kevesebb zárójellel is írni, ez is jó lenne:

```
if #L (#L@c)==-1 T "\nItt a file vége!\n" XX
```

Viszont, bár a **#L** casting operátor használata mulhatatlanul szükséges, nem kell kétszer kitenni, elég egyszer is, így:

```
if #L (@c)==-1 T "\nItt a file vége!\n" XX
```

Ugyanis eképp az egész, utána következő kifejezésre vonatkozik, s emiatt a **@c** változót is signed int-ként értelmezi. Az nem lenne jó, ha így szerepelne:

```
if (#L@c)==-1 T "\nItt a file vége!\n" XX
```

ugyanis ekkor a **@c** változót igaz hogy signed int-ként értelmezné, de a TELJES aritmetikai kifejezésre az unsigned char vonatkozna, mert az **if** azt várja el, emiatt aztán a „-1” értéket unsigned char -ként akarná beolvasni, annak viszont nem lehet előjele. Így megakad már a mínuszjelnél, annak ASCII kódját adja vissza. Ezt összehasonlítja a **#L@c** változó értékével, visszaadja az eredményt az **if**-nek, ami eltárolja ahova illik, majd menne tovább a parancsértelmezésben, de a következő karakter neki az „1”, mert csak az előtte álló mínuszjelet értékelte ki előzőleg, s mert az „1” karakterre nem definiáltunk még parancsot, így *syntax error* hibajelzéssel megáll.

Az is látható a fenti sorokból, hogy ha elértük a fájl végét, csak nagy lazán kilépünk a programból (az **XX** utasítással), nem tökölödünk olyasmivel, hogy a fájl lezárása. Az efféle rutinmelőtt előzőeken elvégzi nekünk maga az interpreterünk: amint ugyanis kilépünk a programból, pontosabban, amint bármiért is de megsemmisülnek az inputfile típusú változóink, előbb még - ha volt hozzájuk rendelve még le nem zárt fájl - azt lezárja. Ezt természetesen azért teszi, mert az inputfile osztály destruktora így lett megírva.

```
if #L (@c)==-2 T "\nOlyan fájlból próbáltál olvasni, ami nem is lett megnyitva!\n" XX
```

A fenti sor azt mutatja meg, hogy azt is tesztelhetjük, hogy nem épp egy olyan fájlból akarunk-e olvasni, ami meg se lett nyitva.

```
? @c; // és kiírjuk
```

A fenti sor amilyen rövid, olyan érdekes. Ugye, a beolvasás a **#L@c** változóba történt. Ide egy signed int került. Amennyiben azonban normális beolvasás történt, s nem valami hibajelzés, akkor ezen signed int érték valójában teljesen megegyezik egy közönséges unsigned char értékével - azzal a karakterrel, amit várunk a fájlból! S tekintve hogy elég bölcsek voltunk ahhoz, hogy a különböző single, azaz nem tömb változóinkat unionban helyezzük el, emiatt ezen **#L@c** signed int változónk első bájtja épp azonos a **#c@c** unsigned char változóval, de mert a „?” kiíró utasításunk amúgy is unsigned char értéket vár el, ezért ezen változónévből még a **#c** típusjelölőt se muszáj kiírnunk, elég a változó neve!

```
} // Ciklus vége
```

```
;
```

```
"Most lezárjuk a fájlt\n"
```

```
[@B@b] // Itt zártuk le, ezzel az utasítással
```

Gondolom, a fenti utasítás szintaxisa nem szorul különösebb magyarázatra: egyszerűen a két szögletes zárójel közt meg kell adnunk az inputfile változót. Itt kötelező használni a **#B** típusjelölőt!

```
"Most megnyitjuk a fájlt újra\n"
```

```
#B@b="ido.cpp"; // Megnyitjuk a fájlt újra
```

```
[@B@b=6] // Elugrunk a 6-odik bájtjához
```

A fenti utasításnál is kötelezően használandó a **#B** típusjelölő! Az egyenlőségjel után megadandó pozíció mindig a fájl legelejétől számolódik, a fájl legelső karaktere természetesen a nulladikként van sorszámozva, hasonlóan mint a stringeknél.

```
{ 14 ; // beolvassuk innentől az első 14 karaktert
```

```
#L@c=#B@b;
```

```
? @c; // és kiírjuk
```

```
} // Ciklus vége
```

```
;
```

```
"Vége!\n"
```

```
XX
```

Ez is itt fent már mind világos kell legyen az eddigiek alapján.

A fentiek bár működnek, de kissé mintha nehézkesnek tünnének a sok tesztelgetés miatt! Szerencsére azonban, az input fájllokból való beolvasást meg a mindenféle vizsgálatokat megoldhatjuk sokkal elegánsabban is. Mindenekelőtt, rendelkezésünkre áll egy **BN** nevű utasítás, ami olyasféle mint a „**ha**”, csak épp az utána következő inputfile változót teszteli le, hogy sikerült-e megnyitni a fájlt. (Ha sikerült, az **ifflag** értéket nullára állítja, ha nem sikerült, 1-re.) Aztán van nekünk egy **EOF** nevű utasításunk is, ami meglehetősen komplex feladatot lát el:

—Beolvassa a következő bájtot a fájlból. Ezt eltárolja egy speciális változóban (az **F** struktúrában) mint unsigned char értéket. De meg is vizsgálja, nem EOF-e. Ha EOF, az **ifflag**et 1-re állítja, ha nem EOF, nullára. Ezután pedig azt csinálja amit a **T** utasításunk.



Ezenfelül, a

```
?#c "c"
```

függvény visszaadja unsigned char értéként a fájlból utoljára beolvasott karaktert. (azt nem, amit konzolról olvastunk be, csak azt, amit input fájlból...) Ennél mindig a legutolsó beolvasás számít, függetlenül attól, melyik inputfile változóból történt a legutolsó beolvasás.

Mindez egy példán bemutatva:

```
#B@b="proba.txt"; // Megnyitjuk a fájlt
BN b "A file nem megnyitható!\n" XX
E "A file sikeresen meg lett nyitva!\n"

#l@m=#B@b; // A file mérete
"A file mérete = " ?s #l@m; " bájt\n"
{| 500 ; // beolvassuk a karaktereket
EOF b "A file végéhez értünk!";
T ^^ »$ki; // Ha eof volt, ki is ugrunk a ciklusból
? ?#c "c"; // Ha nem eof volt, kiírjuk az utoljára beolvasott bájtot
|} // Ciklus vége
$ki /;
"Most lezárjuk a fájlt\n"
[#B@b] // Itt zártuk le, ezzel az utasítással
"Vége!\n"
XX
```

Persze a fenti megoldás rémségesen ocsmány, tisztességes mau programozó aki ad magára valamit, ilyen programot egyszerűen nem ad ki a kezéből, inkább szeppukut követ el szégyenében! Ez tehát kizárólag az utasítások bemutatását szolgálja. Ha arról van szó hogy egy fájlt be kell olvassunk a végéig és ki óhajtjuk írni, azt valahogy így csináljuk inkább:

```
#B@b="proba.txt"; // Megnyitjuk a fájlt
BN b "A file nem megnyitható!\n" XX
E "A file sikeresen meg lett nyitva!\n"

#l@m=#B@b; // A file mérete
"A file mérete = " ?s #l@m; " bájt\n"
{($ki (?e b) // Beolvassuk a karaktert, s egyben teszteljük is EOF-ra
? ?#c "c"; // ki is írjuk a beolvasott karaktert
)} // Ciklus vége
$ki /;
"A file végéhez értünk!\n";
"Most lezárjuk a fájlt\n"
[#B@b] // Itt zártuk le, ezzel az utasítással
"Vége!\n"
XX
```

A fenti példában fix darabszámszor lefutó ciklus helyett egy előltesztelő ciklus alkalmaztatik, aminek fejlécében egy eddig még nem ismertetett függvény bújik meg, ez:

```
?e b
```

Ebből a „b” természetesen az az unsigned char típusú aritmetikai kifejezés, mely az aktuális inputfile változó nevét határozza meg, azt, amire a **?e** függvény vonatkozik. Na most, e függvény mint a korábban ismertetett **EOF**, beolvas egy bájtot a fájlból. Elmenti a megfelelő rendszerváltozóba (ugyanoda ő is mint az **EOF**), ő is megvizsgálja hogy **EOF**-e a beolvasott érték, s ha igen, beállítja az **ifflag**-et 1-re, ha pedig nem **EOF**, akkor nullára. Aztán visszaadja az **ifflag** értékét unsigned int értéként. Amit itt e példában a ciklusunk használ fel, átkonvertálja



magának unsigned char értékke, és ettől függően megy bele a ciklusba vagy ugrik a végére.

Ez már majdnem mondható „civilizált emberhez méltó” fájlkezelésnek. Ám a mau nyelv ennél is előkelőbb megoldást ad nekünk input fájlok kezelésére! Nézzük csak e kis példaprogramot, aztán jön a magyarázat:

```
{ } "proba.txt" $fi; // A file minden karakterére végrehajtjuk a „fi” szubrutint  
T "A file nem megnyitható!\n" XX
```

```
"A file mérete = " ?s ?m; " bájt volt\n"  
"Vége!\n"  
XX
```

```
$fi // fájlfeldolgozó szubrutin kezdete  
? ?#c "c"; // ki is írjuk a beolvasott karaktert  
« // vége a szubrutinnak
```

Na hát ez már tényleg nem bonyolult, ugye?! Nem kell szarakodnunk az input fájlunk se a megnyitásával, se a lezárásával, sőt, még azzal se hogy hozzárendeljünk valami inputfájlváltozóhoz! Desőt még azzal se hogy konvertálgassuk a beolvasott int értéket, meg hogy ellenőrizgessük nem EOF-e véletlenül, és mindezek tetejébe még ciklust se kell szerveznünk! Annyit kell csak tudnunk, hogy mi a fájlunk neve. Ezt megadjuk valami nekünk tetsző stringkifejezéssel a {} karakterpáros után (ez egyetlen utasítástoken, nem lehet whitespace a kapcsos zárójelek között!) és megadjuk utána azt a címet a programunkban, ahol a fájlfeldolgozó szubrutinunk „elkezd magát”. Ezt persze leglogikusabb egy címkével megadni. Mint minden szubrutint, természetesen ennek végét is a « utasítás jelzi. A mau interpreter erre megpróbálja megnyitni a fájlt. Ha nem sikerül, az **ifflag** értékét 1-re állítja, s hasonlóképp akkor is, ha sikerült ugyan a fájl megnyitása, de a mérete nulla. Ezen esetekben nem is ugrik bele a szubrutinba. Ezért van az, hogy a {} utasítás után illik letesztelni az **ifflag** állapotát, ezt láthatjuk is a fenti példaprogramban a T utasítással.

Ha azonban a fájlme nyitás sikerült, egy ciklussal végigolvassa a fájlt, s minden egyes bájt beolvasása után meghívja (végrehajtja) a megadott címkénél kezdődő mau szubrutint. Ezen szubrutin belsejében a fájlból beolvasott aktuális karakterre mindig a

```
?#c "c"
```

függvénnyel hivatkozhatunk. Egyben rendelkezésünkre áll e szubrutinban a fájl mérete, amit megkaphatunk a ?m rendszerváltozóval, ami unsigned int értéket ad vissza. Ezt e példában nem használtuk fel a szubrutin belsejében, de a {} utasítás után igen, mert a fájl feldolgozása után is rendelkezésünkre áll ez az infó, egészen a következő ilyen utasítás végrehajtásáig.

És természetesen miután a fájl minden bájtjára végrehajtotta a szubrutint a program, azután rögv est be is zárja azt, azzal se kell törődnünk.

És még ezután meri valaki azt állítani, hogy a mau programnyelv BONYOLULT?!

A fájl minden bájtjára végrehajtandó cikluson belül rendelkezésünkre áll (mert a mau nyelv kényelmes és előzékeny is...) a ?D rendszerfüggvény, ami (unsigned int értéként) visszaadja mindig azt a sorszámot, ahányadik karakterrel foglalkozunk épp a fájlban.

Továbbá, hasonló funkció létezik arra is, hogy egy szubrutint annyiszor hajtsunk végre, ahány sor van egy fájlban! Nézzük csak ezt a kis progit, ami névsorba rendezi egy fájl sorait, s kiírja azt egy új fájlba:

```
#!/mau // Egy file sorainak névsorba rendezése
// rendez.mau
if (?a)<3 T "Nem adtad meg a rendezendő állomány nevét!\n" XX;
    E #s@A=?#s "ARGV" 2; // A rendezendő file neve

<? @A; // Megszámoljuk a sorait
"A file sorainak száma: " ?l ?n ; /;
"A file leghosszabb sora " ?l ?!; " bájt hosszú.\n";
if(#l ?n>0) [@t[?n]]; // lefoglalok területet annyi string részére, ahány sor van a fájlban, a
t nevű stringtömbbe

{-} @A, $su;
T "Az input file nem megnyitható!\n" XX

#t@a; // inicializálunk egy időváltozót
QS t,?n; // névsorba rendezzük a tömböt
#t@b; // inicializálunk egy másik időváltozót
#g@e=(#t@b)-(#t@a);
"A rendezés ennyi milliomed másodpercig tartott: " ?g @e; /;
#K@o=(@A)+".rendezett"; // Megnyitjuk írásra az output fájlt
if #K@o E "Az output file nem megnyitható!\n" XX
{| ?n; // A ciklus annyiszor fut le, ahány sor van a fájlban
<s @o, @t[?n-?|]; // Kiírjuk a sort a fájlba
|} // vége a ciklusnak
[#K@o]; // Lezárom az output fájlt

XX // vége a programnak

$su // A szubrutin
#s@t[?D]=?#s "{-}"; // Beolvassuk a sort, de max. annyi karaktert, amennyi a leghosszabb sor a
fájlban.
«
```

A fenti progiban van pár olyan utasítás ami még nem került ismertetésre, ezekkel most ne törődjünk; a lényeg ez a sor:

```
{-} @A, $su;
```

Ez azt csinálja, hogy megnyitja az utána következő (itt a **@A**) stringkifejezés által megnevezett input fájlt, és megszámlolja, hány sor van benne, s ezek közül melyik a leghosszabb, annak hány bájtja van. Ezután pedig az utána megadott címke által specifikált szubrutint hívja meg annyiszor, ahány sor van a fájlban. Majd a végén bezárja a fájlt. Ha így adjuk meg a parancsot:

```
{-} @A, $su, db;
```

akkor a db azt mondja meg neki (mint unsigned int kifejezés) hogy maximum hány bájtot tekintsen egy sorhoz tartozónak. Ebből máris következik hogy az előző változatnál ahol nincs megadva a db paraméter, KÖTELEZŐ az utasítást lezárni pontosvesszővel!

A szubrutinon belül két fontos rendszerfüggvényt érhetünk el:

```
#s@t[?D]=?#s "{-}"; // Beolvassuk a sort, de max. annyi karaktert, amennyi a leghosszabb sor a
```

Mint a fenti sor példáján látható, a **?D** függvény unsigned int értéként ez esetben azt a számot adja vissza, ami az aktuális sor indexe, azaz hogy hányadik sornál tartunk a fájlban épp. A sorok nullától kezdve számozódnak! A **?#s "{-}"** függvény pedig stringként szolgáltatja nekünk az aktuális beolvasott sort.

Igenám, de előre tudható, rendkívül gyakori lesz az olyan feladat, hogy egy teljes fájlt be kell olvasnunk soronként egy tömbbe! Természetesen stringtömbbe. Ezesetben pedig nehogy már a szerencsétlen mau programozónak kelljen bohózkodnia olyasmivel, hogy megvizsgálja hány sorból áll a fájl, memóriát allokáljon neki, meg ciklust szervezzon a beolvasásra és hasonlók... A mau nyelv KÉNYELMES, emiatt ezt nagyon egyszerűen el lehet intézni, amint azt megleshetjük e fenti program megfelelőképpen módosított alábbi változatából:

```
#!mau // Egy file sorainak névsorba rendezése
// rendez.mau
if (?a)<3 T "Nem adtad meg a rendezendő állomány nevét!\n" XX;
    E #s@A=?#s "ARGV" 2; // A rendezendő file neve

BE @A, t; // Beolvasom a fájlt a „t” nevű stringtömbbe
T "Az input file nulla méretű!\n" XX
"A file sorainak száma: " ?l ?n ; /;
"A file leghosszabb sora " ?l ?!; " bájttal hosszú.\n";

#t@a; // inicializálunk egy időváltozót
QS t,?n; // névsorba rendezzük a tömböt
#t@b; // inicializálunk egy másik időváltozót
#g@e=(#t@b)-(#t@a);
"A rendezés ennyi milliópercig tartott: " ?g @e; /;
#K@o=(@A)+".rendezett"; // Megnyitjuk írásra az output fájlt
if #K@o E "Az output file nem megnyitható!\n" XX
{| ?n; // A ciklus annyiszor fut le, ahány sor van a fájlban
<s @o, @t[[?n-?|]]; // Kiírjuk a sort a fájlba
|} // vége a ciklusnak
[#K@o]; // Lezárom az output fájlt

XX // vége a programnak
```

Látható a lényeg: a teljes beolvasás, mindennel ami csak kellhet hozzá, lerövidült erre az egyetlen sorra:

```
BE @A, t; // Beolvasom a fájlt a „t” nevű stringtömbbe
```

Gondolom a szintaxis világos: az első paraméter egy string, ami a beolvasandó fájl nevét adja meg, a másik pedig a stringtömb neve, amibe beolvassuk. Ebben természetesen pontosan annyi string lesz, ahány sor van a fájlban. Ennél egyszerűbben igazán nem lehet beolvasni egy fájlt soronként... Ja: ha olyan stringtömbbe olvastatjuk be, ami nem üres, mert már korábban valamiért lett neki memória allokálva, abban az esetben beolvasás előtt azt a memóriaterületet automatikusan felszabadítja. Ez természetesen a régi tartalom elvesztését/megsemmisülését jelenti.

A kiírás azonban mintha még mindig bonyolultnak tűnne, ugye?! De nem kell félni, lesz az ennél sokkal egyszerűbb is, de az a következő fejezetben lesz kitérve.

## Output fájlok

Ami az output fájlok kezelését illeti, azt is egy példaprogrammal magyarázom el. Az alábbi program beolvas egy input fájlból pár karaktert, s ezt kiírja egy output fájlba, majd visszaolvassa:

```

#s@b="proba.txt"; // Az input file neve
#s@o="mau_output_file.txt"; // Az output file neve
#B@b=@b; // Megnyitjuk az input fájlt
if #B@b T "Az input file sikeresen meg lett nyitva!\n"
E "Az input file nem megnyitható!\n" XX
#l@m=#B@b; // Az input file mérete
"Az input file mérete = " ?s #l@m; " bájt\n"
#K@o=@o; // Megnyitjuk az output file-ot.
if #K@o T "Az output file sikeresen meg lett nyitva!\n"
E "Az output file nem megnyitható!\n" XX
<s @o "Ezt a szöveget írom a fájl első sorába!\n";
{| 50 ; // beolvassuk az input file első 50 karakterét
#L@c=#B@b;
if #L (@c)==-1 T "\nItt az input file vége!\n" XX
if #L (@c)==-2 T "\nOlyan fájlból próbáltál olvasni, ami nem is lett megnyitva!\n" XX
<c @o @c; // és kiírjuk az imént beolvasott karaktert az output file-ba.
|} // Ciklus vége
<c @o 10; // Kiírunk egy sorvég-karaktert az output fileba
"Most lezárjuk az input fájlt!\n"
[#B@b] // Itt zártuk le, ezzel az utasítással
<s @o "Ezt a sort az output file legvégére írom ki!\n";
"Lezárom az output fájlt!\n";
[#K@o]; // Lezárom az output file-ot ezzel az utasítással
"Most megnyitjuk az imént lezárt output fájlt újra, ezúttal olvasásra\n"
#B@b="mau_output_file.txt"; // Megnyitjuk a fájlt újra
{| #l#B@b // beolvasóciklus indul, annyit olvasunk belőle ahány bájt a file mérete
#L@c=#B@b;
? @c; // és kiírjuk
|} // Ciklus vége
/;
"Vége!\n"
XX

```

Remélem, a sok komment miatt a program jórészt önmagát magyarázza; néhány kiegészítést fűznék csak hozzá:

```
<s @o "Ezt a szöveget írom a fájl első sorába!\n";
```

A fenti sor mutatja, hogy a „<” jel a fájlba kiíró utasításunk. Közvetlenül ezután kell álljon a típusjelölő. (nem állhat köztük whitespace). Ha ez az „s”, akkor stringet jelent. Természetesen nem csak konstans stringet adhatunk meg, hanem akármiféle stringváltozót is. Amennyiben a karakter nem „s” hanem „c”, akkor unsigned char értéket vár el, más típusjelölő karakterek esetén pedig más típusokat. Viszont a numerikus típusokat NEM STRINGKÉNT fogja kiírni, hanem úgy ahogy azok a memóriában le vannak tárolva, bájttonként! Például az unsigned short int típus decimális számok stringjeként akár 5 karaktert is elfoglalhat a képernyőn, mert lehet akár 65535 is egy ilyen szám értéke, mindazonáltal ha mi ezt a

```
<i @o 65535;
```

utasítással írjuk ki az „o” nevű outputfile-változónkba, akkor a fájlba csak 2 bájt kerül, s mindegyiknek az értéke 255 lesz.

Megjegyzendő még, hogy a fájlba kiírató < utasításunk esetén a típust meghatározó karaktert természetesen megadhatjuk unsigned char típusú aritmetikai kifejezéssel is, azaz indirekt módon, efféle formában mondjuk:

```
<(@Y:2) @o @m;
```

A fenti esetben a < utáni nyitózároljel természetesen közvetlenül a < jel után kell álljon, nem lehet köztük whitespace sem. És a zárójelek közt tetszőleges unsigned char értéként értelmezhető aritmetikai kifejezés állhat. Ennek eredménye határozza meg, miféle típusként értelmezi majd a @m változót (vagy úgy általában véve azt a kifejezést ami a @m helyén áll). A @o mindenképp unsigned char kifejezésként lesz kiértékelve, mert ez kell megmondja azt neki, hogy mi a neve annak az outputfile-változónak amire a kiírás vonatkozik.

```
[#K@o]; // Lezárom az output file-ot ezzel az utasítással
```

A fenti sor a fájl-lezárást mutatja be. Nem kell szórakoznunk a „C” nyelvben megismert „fflush” utasítással, azt magától is megcsinálja ilyen esetben, automatikusan.

```
{| #l#B@b // beolvasóciklus indul, annyit olvasunk belőle ahány bájt a file mérete
```

A fenti sor egy érdekes trükköt mutat be. Ugye, itt egy „fix számszor lefutó” ciklust alkalmazunk. A ciklus fejlécében szereplő számot, mely megmutatja hány-szor kell lefusszon, az interpreter csak egy alkalommal olvassa be (illetve értékeli ki az ezt meghatározó aritmetikai kifejezést). Mi ide azt a számot óhajtjuk írni, ami a fájl mérete. Na most, ezt nekünk nem kell külön elmentenünk egy plusz változóba, ha amúgy nincs rá szükségünk, mert mi a csudának?! Egyedül arra kell ügyelnünk, hogy az inputfile-változónk az esetben adja vissza ezen méretet nekünk, ha unsigned int értékre castoljuk. Emiatt kell elé a **#1** unáris casting operátor.

Itt mutatok be egy olyan utasítást s két függvényt, amiknek nincs közük az input-file osztályunkhoz, de a fájlkezeléshez azért igen. Ugye sokszor jó lenne tudni, hány sor van egy fájlban, mert soronként akarjuk feldolgozni! Azt se ártana tudni esetleg, hány bájt hosszú a leghosszabb sor belőle. Nos, ehhez sajnos muszáj végigolvasni a fájlt. Azaz megnyitjuk, végigolvassuk, megszámloljuk amit kell, majd bezárjuk. Milyen jó is lenne ezt automatizálni... Íme:

```
<? s
```

A fenti utasítás megszámlolja, hogy egy fájlban hány sor van, s ezek közül vissza-adja a leghosszabbnak a hosszát bájtokban számolva. Az „**s**” egy stringkifejezés, ami a fájl nevét határozza meg (a teljes elérési útvonallal együtt). A program ahhoz hogy ezen adatokat megtudja, természetesen kénytelen a fájlt megnyitni, végigolvasni, majd lezárni. A tartalmát nem módosítja. Ezen utasítás után a két adat a következő rendszerfüggvényekkel érhető el:

```
?n // visszaadja a file sorainak a számát unsigned int értékként
```

```
?! // visszaadja a file leghosszabb sorának hosszát bájtban számolva, unsigned int értékként
```

Egy kis példa erre:

```
<? "filesorokszama.mau";  
"A file sorainak száma: " ?l ?n ; /;  
"A file leghosszabb sora " ?l ?!; " bájt hosszú.";  
/;
```

Ami a visszaadott eredményeket illeti, azok a **<?** utasítás következő kiadásáig nem módosulnak.

Ezenfelül e fejezethez tartoznak azok az utasítások, melyekkel egy fájl vagy könyvtárat törölhetünk illetve átnevezhetünk:

Fájl törlése:

```
RM s
```

ahol az s egy stringkifejezés, ami a törlendő fájl nevét tartalmazza az elérési útvonalával együtt.

Fájl vagy könyvtár átnevezése:

```
RN regineve ujneve
```

ahol a regineve és az ujneve egy-egy stringkifejezés, amik természetesen kell hogy tartalmazzák az elérési útvonalat is.

Könyvtár törlése rekurzívan, azaz minden benne levő fájlal és alkönyvtárral együtt:

**RD s**

ahol az s egy stringkifejezés, ami kell hogy tartalmazza az elérési útvonalat is.

Az előző, input fájlokról szóló fejezetben bemutattam egy kis progit, ami egy fájlt soronként beolvas, névsorba rendezi a sorokat, majd kiírja a rendezett fájlt egy másik állományba. Akkor ígértem, hogy a kiíró rész lehet sokkal egyszerűbb is... Íme, ugyanaz a program, némileg kényelmesebb formában bemutatva:

```
#!/mau // Egy file sorainak névsorba rendezése
// rendez.mau
if (?a)<3 T "Nem adtad meg a rendezendő állomány nevét!\n" XX;
    E #s@A=?#s "ARGV" 2; // A rendezendő file neve

BE @A, t; // Beolvasom a fájlt a „t” nevű stringtömbbe
T "Az input file nulla méretű!\n" XX
"A file sorainak száma: " ?l ?n ; /;
"A file leghosszabb sora " ?l ?!; " bájt hosszú.\n";

#t@a; // inicializálunk egy időváltozót
QS t,?n; // névsorba rendezzük a tömböt
#t@b; // inicializálunk egy másik időváltozót
#g@e=(#t@b)-(#t@a);
"A rendezés ennyi milliionod másodpercig tartott: " ?g @e; /;
#K@o=(@A)+".rendezett"; // Megnyitjuk írásra az output fájlt
if #K@o E "Az output file nem megnyitható!\n" XX

ki @o, t; // Kiírjuk a tömböt a fájlba

[#K@o]; // Lezáróm az output fájlt

XX // vége a programnak
```

A beolvasó rész nem változott - azt nem lehet egyszerűbben. A kiírás azonban mint látjuk, jócskán leegyszerűsödött: eltűnt a ciklus! E sor:

**ki @o, t; // Kiírjuk a tömböt a fájlba**

elvégzi a tömb kiíratását. Az első paramétere természetesen a megnyitott output-file változója, a másik pedig az az unsigned char típusú aritmetikai kifejezés, mely a kiíratandó stringtömb nevét határozza meg.

De lehet még ennél is egyszerűbben! Íme:

```
#!/mau // Egy file sorainak névsorba rendezése
// rendez.mau
if (?a)<3 T "Nem adtad meg a rendezendő állomány nevét!\n" XX;
    E #s@A=?#s "ARGV" 2; // A rendezendő file neve

BE @A, t; // Beolvasom a fájlt a „t” nevű stringtömbbe
T "Az input file nulla méretű!\n" XX
"A file sorainak száma: " ?l ?n ; /;
"A file leghosszabb sora " ?l ?!; " bájt hosszú.\n";

#t@a; // inicializálunk egy időváltozót
QS t,?n; // névsorba rendezzük a tömböt
#t@b; // inicializálunk egy másik időváltozót
#g@e=(#t@b)-(#t@a);
"A rendezés ennyi milliionod másodpercig tartott: " ?g @e; /;

KI (@A)+".rendezett", t; // Kiírjuk a „t” tömböt a fájlba

XX // vége a programnak
```



Amint látjuk, a különbség az előző változathoz képest annyi, hogy nem kell output fájlt se megnyitnunk, se lezárnunk, ellenben a kiíróutasítás itt nagybetűvel van megadva, s az első paramétere a leendő output fájl neve, mint string-kifejezés. (Ha már volt ilyen nevű fájl, akkor a régi elvész!) Azaz ez az utasítás a lényege az egésznek:

```
KI (@A)+".rendezett", t; // Kiírjuk a „t” tömböt a fájlba
```

Ennél egyszerűbben igazán nem lehet...

## 17. fejezet - Tartalomjegyzékek (directoryk) kezelése

Ez egy meglehetősen szövevényes és bonyolult ügy. Arról van szó, hogy az Unix/Linux tartalomjegyzék-struktúrája nem épp az egyszerűségéről híres! Rengetegsok mindenféle adata, értéke, attribútuma van egy-egy fájlnek ami a tartalomjegyzékben szerepel, már amiatt is, mert egy tartalomjegyzék nemcsak fájlokat tartalmazhat, hanem mindenféle más izémizéket is, ami lényegében csak a nevében „fájl”, valójában azonban cseppet se, mert lehet például maga is tartalomjegyzék, vagy szimbolikus link (ami lehet „törött” vagy létező fájlra mutató), lehet közönséges fájl, lehet holmi „socket”, meg egy rakás egyéb mindenféle bizbasz is még. Meg vannak mindenféle attribútumok amik meghatározzák, hogy mit csinálhat vele a tulajdonos, a csoport vagy mindenki más, van az állományoknak méretük, meg mindenféle időbélyegeik, és... és még rengeteg minden.

Gusztustalan és undorító mindezt kezelni, főleg mert nehéz ennek a sokmindennek a megjelenítéséhez egy egyszerű, átlátható, konzisztens rendszert tervezni, amiben minden benne van amit épp látni akarunk, és ki is fér a képernyőre... Ez gyakorlatilag lehetetlen is, mert hiszen már egyetlen fájlnev is lehet akár 255 karakter hosszú, s hol van még az elérési útvonal?! És symlinknél azt is illő kijelezni, mire mutat.

Mindez tényleg kb teljességgel megoldhatatlan, s erre bizonyíték, hogy maga az „ls” parancs is többféle módon képes listázni, hogy az épp felmerülő igényekhez alakítsuk a megjelenítési formáját. Kár, hogy az „ls -l” formán kívül a többi kb semmire se jó, s még ez a forma is csak alig alkalmas valamire...

Na most, mindezen dolgok megfontolása miatt a mau nyelv természetesen NEM rendelkezik holmi beépített tartalomjegyzék-listázó rutinnal, mert mindenkinek más lenne a megfelelő. E könyvem végén bemutatok majd a példaprogramok közt egyet, ami nekem jó, de nem hiszek benne hogy más is épp ezt tartaná a „tuti”-nak, szóval az csak illusztráció inkább. Ellenben a mau nyelv igenis tartalmaz beépített adattípust a tartalomjegyzékben rejlő adatok könnyű lekérdezésére! Akit ez mélyebben érdekel, az a forráskódban az „**ADAT**” nevű osztály metódusainak nézzen utána. Ebben van egy tartalomjegyzék-bejegyzés minden adata, tehát a fájlnev, típus, jogosultságok, időadatok, stb. Ezen ADAT típusú bejegyzések egy-egy tömbben szerepelnek, típusonként különválogatva, azaz külön tömbben vannak az altartalomjegyzékeket tartalmazó ADAT értékek, külön tömbben a symlinkekre vonatkozó ADAT értékek, külön tömbben a közönséges fájlokra



vonatkozó ADAT értékek, stb. Ezen tömbök együttesen alkotnak egy **MAPPA** nevű objektumot. És természetesen programnyelvünkben e MAPPA nevű adattípusból is 256 különböző lehetséges változónk van, amiket a **#T** típus jelöl. (Mert ugye **T**artalomjegyzék). Amint egy ilyen változónak értéket adunk, amely egy tartalomjegyzék útvonala, ő azonnal beolvassa a megfelelő tömbökbe az összes, azon típusnak megfelelő bejegyzést, azaz az ADAT értékeket, sajnos azonban ahhoz hogy tudja melyikből hány van, előbb végigjárja a teljes tartalomjegyzéket. Ezután tudja melyikből mennyi kell, lefoglalja nekik a kellő memóriát, sőt egy picit többet a biztonság kedvéért, hátha épp most nyit a directoryban valami másik program egy fájl... Ezután újra beolvassa a tartalomjegyzék-struktúrát, de most már teljes részletességgel, s elhelyezi a megfelelő tömbökben az adatokat, azaz rögvesszétsszortírozza őket típus szerint. Majd névsorba is rendezi őket. (a stringeknél bemutatott sorbarendezi szabályok szerint, természetesen). Ezek után már a mi dolgunk, a „mau” nyelven programozóké, hogy miként listázzuk ki e kutyulmányt, természetesen a különböző adatok lekérdezésére vannak mindenféle függvényeink.

Illetve... Vannak függvényeink a legfontosabb dolgok lekérdezésére, igen. De azért nem mindenre mégsem. Amikre nincs, azokat is beolvassa a „rencer”, de a lekérdező függvényeket még nem írtam meg, „eccörűjen” időhiány okának miatta s belőle kifolyólagosan. A mau nyelv későbbi kiadásában e hiány természetesen pótolva lesz. (nem olyan marha nehéz ezek után már...) Ami azonban eddig készen van az máris a lelke, a lényege a dolgoknak, nézzük is át, mivel rendelkezünk eddig!

**#T@t=".";** // Az aktuális tartalomjegyzéket jelölő string

A fenti sor azonnal megnyitja a tartalomjegyzéket, azaz beolvasásra kerül az egész trutumó a „t” nevű tartalomjegyzék-változóba. Ez jelen példánknál az „aktuális” tartalomjegyzék, megadható azonban akármi más teljes elérési útvonal is, természetesen nemcsak konstans hanem tetszőleges stringkifejezés is. Lehetne például az is, hogy

**#T@t="/home/CommonUsers/Walacky/Musics/NonHungarian/Mystical/Shaman";**

A tartalomjegyzék-változónknál (miután már lett belé beolvasva valami) lekérdezhettük azt, hogy az egyes típusú bejegyzésekből épp hány szerepel benne. Ezt tudjuk megoldani a

**?#l "T#" t D**

függvénnyel. Ennek mint látjuk 2 paramétere van, amelyeket itt a „t” és a „D” betűkkel jelöltem, természetesen mindegyik lehet tetszőleges unsigned char típusú aritmetikai kifejezés. A „t” a tartalomjegyzék-változónk nevét határozza meg, a „D” pedig azt, hogy milyen típusú bejegyzésre vonatkozik a darabszám-lekérdezésünk. Az érvényes típusjelölő karakterek és jelentéseik:

**D** : altartalomjegyzék (subdirectory)

**L** : szimbolikus link

**R** : közönséges (reguláris) fájl

**B** : blokk-device

**C** : karakteres eszköz

**S** : socket

**F** : fifo

**M** : Minden egyéb

Amennyiben BÁRMI olyan karaktert adunk meg e darabszámlekérdező függvénynek ami nem szerepel a fenti listában, akkor a tartalomjegyzék-változóba beolvasott tartalomjegyzék ÖSSZES bejegyzésének a számát adja vissza.

A lekérdezett darabszámot például a következőképp hasonlíthatjuk össze:

```
if (#l (?#l "T#" t D)==0) T »$DV; // Ha nincs altartalomjegyzék, nem listáz
```

Azaz ez a fenti egy összehasonlító művelet. Nem illik ugyanis hogy akkor is megpróbálkozzunk kiíratással (vagy bármi egyébbel), ha netán nincs mit kiírni, mert épp egyetlen subdirectory sincs a tartalomjegyzékben. Az hibát eredményezne. Na most, az **if** egy unsigned char értéket vár el, ezért külön jelezni kell a **#1** casting operátorral, hogy mi most unsigned int értékeket hasonlítunk majd össze. A belső zárójelen belüli rész adja meg, hogy épp hány dir típusú bejegyzés van a beolvasott tartalomjegyzékben. E kifejezés részletes jelentése:

**?#1** : unsigned int típusú rendszerfüggvényt hívunk meg

**"T#"** : A függvény neve. Arra utal, hogy tartalomjegyzék tételszáma után tudakozódunk.

**t** : A „t” nevű tartalomjegyzék-változót kérdezzük le.

**D** : A tartalomjegyzék-változónk bejegyzései közül a directorykra vagyunk kíváncsiak.

Egy tartalomjegyzék változó valahányadik bejegyzésénél lekérdezhetjük a nevét is természetesen, arra ez a függvény szolgál:

```
?#s "Tn" t, #l@i, D
```

Mint látható az elején rögvest, ez természetesen egy stringet ad vissza. 3 paramétere van, az első amit itt a „t” jelöl, unsigned char típusú kifejezés lehet és a tartalomjegyzék-változónk nevét határozza meg. A második paraméter amit itt a **#l@i** jelöl, egy unsigned int típusú kifejezés lehet, s azt adja meg, hányadik tétel neve után tudakozódunk, a harmadik paraméter pedig épp az mint az előző fenti példánál, azt mondja meg melyik típusú tartalomjegyzék-bejegyzések közül vagyunk kíváncsiak az előző paraméterben meghatározott sorszámúnak a nevére.

Annak érdekében hogy kiderítsük, létezik-e az adott tartalomjegyzék-változónál a megfelelő típusból egy bizonyos nevű bejegyzés (azaz olyan nevű közönséges fájl, vagy symlink, vagy subdirectory, stb), nem kell külön bonyolult ciklusokat szerveznünk! A mau nyelv igenis KÉNYELMES, és fejlett szolgáltatásai vannak! Erre a feladatra kész „kincstári” függvényünk van, ami így néz ki:

```
?#L "?" t D S;
```

Aholis a „t” és „D” jelentése ugyanaz mint fentebb az előző függvény ismertetésénél, az „S” pedig egy stringkifejezés, mely azt az állománynevet határozza meg, aminek a léte felől tudakozódunk. Na most ha az S hossza netán nulla, akkor a függvény visszatérési értéke **-2**, azaz mínusz kettő. Ellenkező esetben, ha létezik az adott tartalomjegyzék-változó megadott „D” típusú bejegyzései közt olyan, aminek a neve S, akkor a visszatérési érték egy pozitív szám vagy nulla, mely ezen bejegyzés indexét adja vissza (azaz hogy hányadik tétel a beolvasott tartalomjegyzék-struktúrában, az adott tartalomjegyzék-változónál). A tételek ugyanis nullától számozódnak. Az így visszakapott indexet tehát felhasználhatjuk arra, hogy lekérdezzük e fájl vagy akár micsoda mindenféle adatait ha akarjuk. Amennyiben nem létezik ilyen nevű bejegyzés, a visszatérési érték **-1**, azaz mínusz egy.

Itt van erre egy kis példaprogram:

```
#!mau
#s@s="main.cpp";
#T@t=".";
#L@L = ?#L "?" t R @s;
ha(#L (@L)>=0) "Van ilyen nevű fájl! Sorszáma: " ?L @L; /;
E "Nincs ilyen nevű fájl!\n"
XX
```

Az állományméret lekérdezésére e függvény szolgál:

```
?#g "FM" t, #l@i, R
```

E függvény nevét könnyű megjegyezni: „**F**ile **M**éret”... Ennek paraméterei ugyanazok mint az előző függvéynél ami a nevet adja vissza, de ez a fájl méretet mondja meg nekünk, unsigned long long típusként. Mint látható, itt a példában nem a **D** hanem az **R** karaktert használtam, mert fájl méretet többnyire a reguláris fájlok esetében nézünk. Ezt az adatot célszerűen stringként jó kiíratnunk, jobbra igazítva:

```
?s [#c@c:1,] ?#g "FM" t, #l@i, R;
```

Mit is csinál ez? Ugye, a **?s** parancs eleve stringet akar kiírni, tehát a sokféle kifejezésértékelő függvényünk közül azt hívja meg, ami stringet akar készíteni mindenből, amiből csak tud. Ez belefut abba a kutyulmányba, ami a szögletes zárójelek közt található. Ebből tudja, hogy neki nem az egész stringet kell visszaadnia amit majd beolvas, hanem annak csak egy részét. Első paramétere egy változó, azt mondja meg, hányadik karaktertől. Ezt megjegyzi. Aztán jön a vessző, majd semmi, ebből tudja hogy innen a hátralevő teljes hosszat kell vennie. Oké. Ezután beolvassa ami hátra van. Igen ám, de ezután nem stringkifejezés jön, hanem egy unsigned long long típusú szám! Nem jön zavarba a drága, rögvést tudja hogy ez maximum 21 karakter hosszú decimális számsorozat lehet, azonnal átalakítja arra. Majd az eredményül kapott stringet csonkolja úgy, ahogy a szögletes zárójelek közt előírtuk neki. A csonkolást azért tartom fontosnak, mert azért egy fájl méret többnyire sokkal kevesebb karakterbe is belefér ám, mint 21...

A szimbolikus linkek kiírása meg minden egyéb izémizéje is a fentiekhez hasonlóan kezelendő, csak esetükben a típusjelölő karakter az **L**. Van azonban náluk egy plusz lehetőség, azon fájl nevének kiírása, amire a link mutat. Az e névnek megfelelő stringet e függvény adja vissza nekünk:

```
?#s "LINK" t, #l@i
```

Látható, ennek csak 2 paramétere van, a tartalomjegyzék-változó neve (e példában a **t**), és a sorszám. Típust itt nem kell megadni, mert értelemszerűen csak link mutathat más fájlra, más típus nem.

Symlinkek esetén lekérdezhethetjük azt is, a link „törött”-e, azaz olyan-e hogy valamely nem létező fájlra vagy könyvtárra mutat! Ezt a következő függvény valósítja meg nekünk:

```
?#c "L?" t, sorszám, L
```

ahol a „sorszám” természetesen tetszőleges (unsigned int típusú) aritmetikai kifejezés lehet. E függvény visszatérési értékei:

**0** ha nem linkről van szó, **1** ha élő a link, és **2** ha törött link.

Ezt mondjuk efféleképp használhatjuk:

```
if((?#c "L?" t, #l@i, L)==2) T "A link törött!\n" »$ki;
E Sc; ?s @c; Sd; Sm;
```

Van függvény arra is, hogy lekérdezzük a bejegyzés jogosultságait. Ez természetesen 4 karakteres oktális stringként visszaadja a fájl jogait. Igaz „Kocka” ugyanis kizárólag oktálisán szereti látni a jogosultságokat, nem afféle helypocskoló kezdő hátulgombolós dedósoknak való übergagyi stílusban, hogy „rwxrwxrwx”... Íme a függvény:

```
?#s "JOG" neve hanyadik x
```

ahol „neve” a tartalomjegyzék-változó neve (ami unsigned char típusú kifejezés lehet), a „hanyadik” az, hogy hanyadik bejegyzés abból a típusból, az „x” pedig azon karakterek valamelyike lehet, amik a bejegyzéstípusra utalnak, s amelyek listáját fentebb már korábban megadtam. (D, R, L, stb).

Azt is lekérdezhettük, a bejegyzés egy végrehajtható fájl-e. Ennek lehetőségét e függvény nyújtja nekünk:

```
?#c "EXE" neve hanyadik x
```

Ez természetesen egy unsigned char értéket ad vissza, ami végrehajtható fájlok esetében 1, különben nulla. A paraméterek ugyanazok mint az előző függvény esetében.

A tartalomjegyzék-bejegyzés tulajdonosának nevét tartalmazó string e függvénnyel csalogatható elő:

```
?#s "Tt" neve hanyadik x
```

Ez pedig azt a stringet adja vissza ami „csoport”-hoz tartozik a bejegyzés:

```
?#s "Tg" neve hanyadik x
```

Az időadatok lekérdezése: Úgy találtam, a fájloknak 3 időadatuk van. Az utolsó elérés ideje (ez egyértelmű hogy mi), az utolsó változás ideje és az utolsó módosítás ideje. Ezek közt bevallom, nem tudom mi a különbség... Mindenesetre, mindhárom lekérdezhető immár! Mindhármát egy stringként kaphatjuk meg, melynek formátuma épp olyan, mint amit a rendszer idejének lekérdezésekor kapunk. Azaz e string hossza 23 karakter lesz. A megfelelő függvények:

Utolsó elérés ideje:

```
?#s "UE" neve hanyadik x
```

Utolsó módosítás ideje:

```
?#s "UM" neve hanyadik x
```

Utolsó változás ideje:

```
?#s "UV" neve hanyadik x
```

A paraméterek ugyanazok mint fentebb.

```
?#s "A" neve hanyadik x
```

A fenti függvény esetében a paraméterek jelentése ugyanaz mint az előző függvényénél. Maga a függvény pedig visszaad egy 19 bájt hosszúságú (a záró nullabájttal együtt tehát 20 bájt hosszú) attributumstringet, ami a megadott tartalomjegyzék-bejegyzés ext2 specifikus flagjainak megfelelő karaktereket tartalmazza a megfelelő pozíciókban, épp úgy, ahogy az az **lsattr** parancs esetén szokásos.

További mindenfélék lekérdezési lehetőségei tartalomjegyzékekkel kapcsolatban „coming soon”, azaz jönnek majd a mau nyelv következő verzióiban!

## 18. fejezet - Mau nyelvű függvények

A „mau nyelvű függvény” fogalma alatt sok mindent értünk. Nagy vonalakban ez valamiféle olyan program vagy programrészlet, amit szeretett mau programnyelvünkben írtunk meg, s ezt meg akarja hívni valamely másik program, hogy hasznos munkát végezzen neki. Na most, ennek azért van ám egy rakás variációja, s azoknak a esetei... Mert kezdjük azzal, hogy az se teljesen világos, mármint nincs rá elfogadott terminológia, mi is az, hogy „függvény”!

Sokan ugyanis a szubrutinokat is függvényeknek tartják. Ilyesmiket mi már tudunk gyártani a meglevő eszközeinkkel, s ezzel simán elértük a régi C-64 -es számítógép BASIC-jének színvonalát és használhatóságát. Ha ehhez hozzávesszük, hogy a szubrutinon belül képesek vagyunk külön névteret generálni, aminek paramétereket is átadhatunk, sőt e névterek tetszőleges mélységben egymásba is ágyazhatóak - nos, akkor meg bőven túl is haladtuk az említett nyelv színvonalát már! Általában ugyanis azért úgy vélekednek a programozók közt a legtöbben, hogy csak az függvény, ami saját névtérrel rendelkezik. Igen ám, de a névtér fogalmába ők bele szokták érteni a függvényen belüli címkéket is, amik nálunk viszont globálisak! Továbbá, igazi függvényeknek azért szokott ám neve is lenni, ami nálunk eddig még nincs. Címkéink vannak, függvényneveink még nincsenek.

Mindez még tovább bonyolódik azzal, hogy egy „függvény”, az legalább 2 nagyon különböző valami lehet: mert lehet egyrészt ugyanabban a fájlban amiben a főprogram, vagy, általánosabb megfogalmazásban, az a program ami őt a függvényt meg óhajtja hívni, s az is lehet - legalábbis illik hogy megoldjuk ennek lehetőségét - hogy a hívott függvény egy teljesen külön forrásprogram. Utóbbi esetre nem jó megoldás az hogy a shellen keresztül hívjuk meg, mert úgy csak stringeket adhatnánk át neki paraméterül, visszatérési értéként meg jószerivel semmit se fogadhatnánk tőle, ha csak fájlba nem menti az eredményeket. Ami azonban rém durva dolog volna. És természetesen a meghívott függvénynek is tudnia kell hívni más függvényeket, olyanokat amik vele egy fájlban vannak, s olyanokat is amik külön fájlban vannak. Szóval az ügy cseppet se egyszerű. Akit érdekel hogy mindez hogyan lett megoldva programozástechnikailag, nézzen utána a Bevezetésben említett tanulmányomnak, itt most csak a szintaxis ismertetésére szorítkozom.

Nos, e szintaxis a következő:

Mindenekelőtt: szigorúan a főprogram, a „main” kell legyen a programfájl legisleg-elején. Őt nem muszáj elnevezni (sőt, nem is szabad...), de ettől még neki akkor is, automatikusan lesz név adva, s ez a „BIGBOSS” név lesz, ami magyarul azt jelenti: „NAGYFŐNÖK”. Ez különben fordítási paraméterként állítható, ennek cseréjéhez a vz.h fileban kell megkeresni e sort és kicserélni a nekünk tetszőre:

```
#define MAINPROGRAMNEVE "BIGBOSS"
```

Aztán, e main függvény illik hogy valahol végetérjen egy

XX



utasítással, ez adja vissza a vezérlést a shellnek, ez nálunk az „exit” parancs. S ezután jöhetnek a mindenféle, fájlban belüli függvények. Ezek alakja a következő példa szerinti:

```
„Hányados és maradék megjegyzésekkel” // Két számnak adja vissza a hányadosát és a maradékát
#i@e, // első paraméter - az osztandó
#i@m // második paraméter - az osztó
#i@h=(@e)/(@m); // hányados
#i@M=(@e)>@m; // maradék
\ // Itt jön az output paraméterek átadása
// Ez egy többsoros megjegyzés!
#i@h, // Első output paraméter - a hányados
#i@M; // Második output paraméter, a maradék
xx
```

Mint látható, a függvény neve egyszerűen a magyar szabványoknak megfelelő idézőjelek közt áll, azaz alul nyitó, felül záró idézőjelek közt. Jelen esetben a függvény neve az, hogy **„Hányados és maradék megjegyzésekkel”**. Az idézőjelek nem tartoznak a függvény nevéhez. A bal oldali, alsó nyitó idézőjel a címkéinkhez hasonlóan szigorúan csak egy sor legislegelső karaktere lehet!

Mint már a fenti példából is sejthető, a címke neve abszolút bármiféle (UTF-8 kódolás szerinti) karaktert tartalmazhat, ékezeteseket is, kivéve a 0 kódú, azaz „szakmaian” írva a **CHR\$(0)** bájtot, ami amúgy is stringek végét jelzi, valamint a magyar nyitó- és záró idézőjeleket tehát a „ és ” karaktereket, meg az „angol” idézőjelet, tehát a " jelet.

Utóbbi, az angol idézőjelet tulajdonképpen tartalmazhatja elvileg a címke neve, nincsen azzal semmi baj, épp csak ezesetben rém macerás lesz a használata, mert speciális módon lehet csak megadni stringekben, szóval ettől inkább óvakodjunk. Teli van sok ezer karakterrel az UTF-8 kódolású UNICODE táblázat, vannak benne mindenféle gyönyörűséges vesszők, aposztrófok meg más határolókarakterek, minden mi szem-szájnak ingere, erről az egy jelről nyugodtan lemondhatunk. Ismétlem azonban nem kötelező erről lemondani, mindenki szívathatja magát vele ha akarja, de minek ha nem muszáj... Esetleges rossz-szándékú kritikusaimnak megjegyzem, függvényeim elnevezése még e korlátozás mellett is messze nagyobb szabadságot nyújt mint a C, C++, Pascal vagy más nyelvek elnevezési szabályai, mert ugyan melyikben is tehető ékezetes karakter is a függvény nevébe, még szóköz sem... a mau nyelvben ez mind SZABAD, még kínai, héber, koreai, japán vagy akármi más karaktereket is! Azaz, ha tegyük fel valami olyan szubrutint készítünk, ami a legendás izraeli énekesnő, Ofra Haza élettörténetéről ír ki információkat, vagy valamit az ő dalaival kapcsolatban végez, akkor az e feladatot ellátó függvénynek nyugodtan adhatjuk az ő nevét úgy, ahogy az eredetileg írva van, héber betűkkel:

```
„עפרה חזה” // Ofra Haza függvénye
```

Vagy ha esetleg hinduk számára készítünk vallásos tárgyú programot, ott esetleg hasznos lehet ha a Szaraszvatí istennővel kapcsolatos dolgokat kiírató függvény neve épp az, hogy Szaraszvatí, de természetesen dévanagári írással és karakterekkel, mert ugye India nagy részén az a legelterjedtebb, azaz ekkor így nézhet ki a függvény kezdete:

```
„सरस्वती” // Szaraszvatí függvény
```

Vagy ha netán tibeti nyelvű dolgokkal kapcsolatos programot írunk, esetleg jó ötletnek tűnhet egy függvény nevének az „Om Mani Padme Hum” közismert mantra nevét adni, persze úgy, ahogy azt ott ejtik, azaz „Om Ma Ni Pe Me Hung”, s mindezt tibeti karakterekkel írva:

ཨོཾ་མ་ཎི་པཌ་མུ་ཨུཾ་ // Om Ma Ni Pe Me Hung

Mindez szerintem tök jópofa dolog. Természetesen azonban nem muszáj élni e káprázatos lehetőséggel, a múltbaragadt fosszilis ókonzervatívok megmaradhatnak az angol ABC 26 karaktere mellett is, senki se tiltja meg nekik.

A függvény meghívása pedig a következő szintaxis szerint történik:

(„Hányados és maradék megjegyzésekkel” 26, 7 \ #i@H, #i@M); // Függvény meghívása, 2 input és 2 output paraméterrel

Azaz, a függvényt meghívó utasítás tulajdonképpen a kerek nyitó zárójel és az utána következő alul nyitó („magyar”...) idézőjel. Ezután nem muszáj hogy egy angol idézőjelek közt levő függvénynév álljon, mert állhat ott tetszőleges string-kifejezés is, például olyasmi, hogy

#s@f

de nyilvánvaló, hogy a legtöbbször egyszerűen ki szeretnénk írni oda a függvény nevét, s nem változóban eltárolva meghívni azt. Na most amennyiben ki akarjuk írni, akkor azt angol idézőjelek közt kell megtegyük, mert az azok közti karakterláncot alakítja át az interpreterünk stringgé. Most már gondolom érthető, miért nem jó ötlet ha a függvénynek a nevében is szerepel angol idézőjel...

A függvény nevét meghatározó stringkifejezés után egy záró magyar idézőjelnél kell állnia. Ezután hogy mi jön, attól függ, milyen a meghívandó függvény: akarunk-e neki paramétereket átadni, s várunk-e tőle visszatérési értékeket. Ha ezek egyike se áll fenn, egyszerűen egy csukózárójel kell következzen.

Ha adunk át input paramétereket, akkor a csukóidézőjel után kell felsorolni azon aritmetikai vagy stringkifejezéseket, melyek a paraméterek. Ezek érték szerint lesznek átadva, azaz ha itt változókat adunk meg, azok tartalmát nem módosítja a meghívott függvény. A paramétereket egymástól elválaszthatja vessző, de ez nem kötelező.

Ahogy végetért az input paraméterek listája, következik egy \ jel, azaz egy „visszaperjel”, angolul backslash. Ez az a baromság amit mindig is utáltam a Windows alatt, az, ami ott az egyes könyvtárakat elválasztja egymástól. Linux alatt szerencsére megszabadultam ettől az idiótaságtól, ott a könyvtárszeparátor a normális per-jel. Most azonban szükségem volt valami speciális jelre, amit nem nagyon használok máshol, főleg nem aritmetikai vagy string kifejezésekben műveleti jelként, s így felhasználtam e backslash jelet, mert itt még elviselhető a jelenléte, tudniillik nem lesz nagyon gyakran szükség a használatára feltehetően. Szóval ez a jel jelzi azt, hogy nincs több átadandó input paraméter, s következnek az úgynevezett „output paraméterek”. Amennyiben azok nincsenek, akkor a backslash jelet természetesen a csukózárójel követi, sőt, az esetben magára a backslash jelre sincs szükségünk, jöhet az input paraméterek után rögvést a csukózárójel!

Na most azonban ha *vannak* output paraméterek, pontosabban „visszatérési értékek”, ezeket természetesen a meghívott függvény fogja szolgáltatni, ezekből a



hívási helyen a `\` jel után már nem állhat aritmetikai vagy string kifejezés, hanem kizárólag csak VÁLTOZÓK, még hozzá olyan változók, amik nem a rövidített nevükön szerepelnek, hogy mondjuk `@d`, hanem teljes alakjukban, kitéve eléjük a casting operátor jelet is, hogy például `#i@d`, tudniillik másképp nem tudja majd az interpreter, a hívott függvény visszatérési értékét milyen típusú változóba helyezze el!

Természetesen az output paramétereket is elválaszthatja egymástól vessző, de ez se kötelező. Az utolsó output paramétert kell kövesse a függvényhívást lezáró kerek csukózárójel, a `)`.

A hívott függvény a nevének végét követő csukó azaz „felső állású” magyar idézőjel utáni első karakternél kezdődik, mármint a végrehajtása kezdődik ott. Itt állhat `//` jelpárossal kezdődő megjegyzés, ami megmagyarázza hogy ez miféle függvény, mit csinál, stb, sőt e komment több soron át is tarthat, persze csak úgy, ha minden megjegyzéssor a `//` jellel kezdődik. A `/*...*/` típusú kommentek viszont itt **nem** alkalmazhatóak. (Legalábbis egyelőre).

Mindenesetre, a `"` jel után amennyiben nem kommentek állnak, akkor a hívott függvény input paramétereinek fogadásához szükséges változók kell hogy itt szerepeljenek, természetesen ezek is teljes alakjukban, a `#` casting operátorral jelezve a fajtájukat. E változókat egymástól elválaszthatja vessző, de ez nem kötelező, valamint ezen változók közt is állhatnak `//` formájú kommentek. Sőt, érdemes tudnunk, hogy bár a hívó függvénynél az input paramétereket egymástól, valamint az output paramétereket egymástól csak whitespace vagy vessző választhatja el, a hívott függvény esetében akár az input, akár az output paraméterek listájában is a paraméterek elválasztására a pontosvesszőt is használhatjuk, ha akarjuk!

Vagyis, a hívási helyen nem használhatunk kommenteket, magának a hívott függvénynek a törzsében azonban a paraméterátadási helyeken igencsak szabad kommentelési lehetőségünk van, ami nagyon jó, mert kifejezetten helyes, ha alaposan elmagyarázzuk, melyik változó mire szolgál egy függvényben.

Amint a hívott függvénynél felsoroltuk az input változókat, kezdődik a lényegi kód, a függvény törzse. Ennek kezdetét semmiféle speciális jel nem kell hogy jelölje, tudniillik azt hogy ez hol kezdődik, onnan tudja, hogy elfogytak a hívási helyen az input változók/kifejezések, azaz ha nincs már több paraméter amit átvehetne a hívott függvény, onnan tudja hogy innentől már nem folytatódik az ő paraméterlistája, TEHÁT csakis a lényegi rész következhet, a végrehajtandó kód. Ennek végét pedig vagy egy

**xx**

utasítás jelzi (ügyeljünk rá hogy ezek **kisbetűs** ikszek!), ezesetben nem kell semmiféle visszatérési értéket visszaadnia, vagy egy `\` karakter jelzi. Utóbbi esetben ezután következnek a visszatérési értékek. Mint írtuk, ezeket is elválaszthatja egymástól vessző, pontosvessző vagy `//` stílusú komment, de ez nem kötelező. Minden visszatérési érték tetszőleges aritmetikai vagy stringkifejezés lehet. Nem baj ha ezek alapvetően más típusúak mint amiket a hívó függvény output paraméterlistájában szereplő változók elvárnak, csak az a lényeg, hogy castolhatóak legyenek arra a típusra. Na most amint vége az output paraméterlistának a hívott függvénynél, maga a függvény is végetér, ezekből ilyen esetben nem kötelező az **xx** utasítás sem hogy jelezze a függvény végét. Ennek ellenére, célszerű

kiírni, hogy olvashatóbbá tegye a kódot. Valójában a hívott függvény akkor is végetér, ha szerepel a végén a \ jel, de utána egyetlen output paraméter sem, mert olyat nem is vár a hívó függvény vissza, vagy ha szerepel ugyan valahány output paraméter a hívott függvény listájában, de a hívó valamiért kevesebbet vár el, azaz kevesebbnek tart fenn változót a hívási helyen! Ezesetben nem történik semmi baj, nincs hibajelzés, a felesleges változókat nem adja át (sőt ki se számolja) a hívott függvény. Akkor van csak baj, ha a hívó több visszatérési értéket várna, mint amennyit a hívott szolgáltatni akarna... Ekkor hibajelzést kapunk.

**NAGYON FONTOS TUDNUNK AZT IS ( !!!!!!!!! )**, hogy bár minden függvény végrehajtása mindig, kivétel nélkül a legelején kezdődik minden meghíváskor, ám a függvény által felhasznált belső változók értéke csak az **első** meghíváskor garantáltan nulla! Ugyanis az egyes meghívások során NEM TÖRLŐDIK A KORÁBBAN MEGKAPOTT ÉRTÉKÜK, akármennyi legyen is az!

Meg lehetett volna oldani könnyen, hogy ez ne így legyen, s filóztam is rajta, hogy megcsináljam-e — úgy kb egyetlen rövid programsor beszúrásából állt volna csak az egész! Mégsem tettem így, azaz ez tényleg a legkifejezettebben „nem bug hanem feature”, ahogy mondani szokás. Nagyonis sok esetben rendkívül hasznos ugyanis, ha a függvény képes emlékezni mindenfélére az „előző életéből”! Például ha apránként adogatunk át neki mindenféle értékeket, melyeknek nem lehet előre tudni a darabszámát, ezekkel ő bütyköl mindenfelét, azt hogy nem jön több adat neki egy speciális adatféleség jelzi például nulla vagy negatív érték, s ekkor ő visszaad pár fontos adatot, mondjuk a kapott paraméterek darabszámát, átlagát, összegét, vagy lineáris regressziót számol belőlük, stb. Ha nem emlékezne az előző hívások eredményeire, akkor efféle csak rém macerás kerülőutakon lehetne leprogramozni. Ha ellenben emlékszik az előző értékekre, akkor az ilyesmi nagyon könnyű, ellenben azon esetek megvalósítása se nehéz, amikor nem szabad emlékeznie a korábbi adatokra: azon fontos változókat ugyanis, melyek nem szabad hogy korábban létrejött adatokat tartalmazzanak, egyszerűen explicit módon nullára kell állítani a program legelején, például így:

```
#c@c=0;
```

Arra az esetre ha sok adatot kéne lenullázni a függvény elején, két megoldás is megkönnyíti az életünket. Egyik az, hogy valahova ahova nekünk tetszik, beírjuk ezt a kódba:

```
00;
```

Ez garantáltan minden „single” adatot lenulláz, azaz minden nem tömb változót és nem veremváltozót, ezenfelül nullára állítja az **if** utasítás eredményét tároló flagot és a BRAINFUCK flagot is (utóbbiról egy későbbi fejezetben lesz elmagyarázva, hogy mi a fene is az). Ez egy igen durva, azonban rém gyors módszer a „tisztá lappal induláshoz”. Fontos azonban észben tartani, hogy ez a kapott input paramétereket is elfelejteti a függvénnyel, azaz ha azokra mégis szükség van, előbb verembe vagy tömbbe kell menteni őket! Ezenkívül, a lenullázás csak a numerikus változókra vonatkozik, a stringváltozókra NEM.

Abban az esetben ha mindent le szeretnénk nullázni a függvény indulásakor úgy, ahogy azt fentebb a **00;** utasításnál írtam, de KIVÉVE az input adatokat, azt az input adatok listája előtt kell jelölnünk, EGYETLEN nulla jellel, eképp (zölddel emeltem ki a nulla helyét az alábbi példában):

```

„Hányados és maradék” 0 // Két számnak adja vissza a hányadosát és a maradékát
#i@e; // első paraméter - az osztandó
#i@m // második paraméter - az osztó
// Itt kezdődik a függvénytörzs
#i@h=(@e)/(@m); // hányados
#i@M=(@e)>@m; // maradék
\#i@h; // hányados
#i@M;
xx

```

Ellentétben a korábban bemutatott névterekkel, ami a függvényeket illeti, azok is külön névtérrel rendelkeznek, de ezek olyan névterek, hogy a függvények esetén a CÍMKÉK IS ebben a külön névtérben vannak már, azok se globálisak a függvény számára, azaz egy fájlban minden függvénynek (és a „main” programnak is) lehet ugyanolyan nevű címkéje.

Megjegyzendő, hogy ha az interpreter morog valami hiba miatt, akkor általában igyekszik közölni a hibaüzenetében azt a konkrét, bájtra pontos pozíciót, ahol szerinte a hiba megjelenik a fájlban. Tapasztalataim szerint ha nem is mindig, de az esetek óriási többségében egész jól eltalálja a hiba helyét. Ez azonban csak akkor igazán informatív a programozónak, sajnos, ha a hiba a főprogramban van. Függvények esetén ugyanis a hiba pozíciója a függvény kezdetétől számított, s nem a fájl elejétől számított pozíciót adja meg. Ezen nyilván lehetne segíteni, nem akarom e dolgot featureként feltüntetni, bár lehetne védeni ezt a hibajelzési stílust is, szóval nem tartom bugnak se. Elismerem azonban hogy jobb lenne a fájl elejétől számított pozíciót kijelezni, a dolog azonban nem olyan egyszerű mint azt némelyek gondolnák, és e kérdés alárendelt fontosságú a szememben, szóval előbb fontosabb dolgokat akarok megoldani. Csak jelzem hogy e problémáról tudok. Bár a szememben nem „probléma”, legfeljebb „szépséghiba”.

Minden beolvasott függvénynek van egy sorszáma. Egy függvény mindenképpen van, ez maga a „main”, azaz a főprogram, ezesetben ha nincs más függvény, az ő sorszáma természetesen a nulla. Amint azonban több függvény is van beolvasva, nem okvetlenül lesz igaz hogy a main sorszáma a nulla, tudniillik amint beolvasta a teljes forrásprogramot, a függvényeket névsorba rendezi, a stringeknél bemutatott sorrendiségi szabályok szerint. Ezesetben a „main” nagy valószínűséggel már nem a nullás számot kapja.

Na most, ez nem baj, mert az esetek óriási többségében bennünket abszolút nem érdekel, miféle sorszáma van egy függvénynek, mert úgyis név szerint hivatkozunk rájuk! Ennek hogy mi a sorszáma, csak rendkívül speciális esetekben lehet jelentősége. Ekkor azonban fontos lehet megtudni, melyik függvénynek mi a sorszáma, vagy ellenkezőleg, melyik sorszámhoz épp melyik függvény tartozik, egyáltalán, hány függvényt is olvastunk épp be?! Nos, ezt a legkönnyebben úgy tudhatjuk meg, ha a forrásfájlba valahova beírjuk e kis függvényt, aztán amikor nekünk úgy tetszik, meghívjuk:

```

„Betöltve”
"Betöltve " ?l ?f; " függvény!\n"
{| ?f // A ciklus annyiszor hajtódik végre, ahány betöltött függvény van,
"Az " ?l ?f-?|; ". függvény neve: „" ?s ?#s " „" ?f-?|; ""\n"
|}
xx

```

Szerintem aki eddig eljutott a könyvem olvasásában, az komolyan érdeklődik a mau programnyelv iránt, s eddig bőven felszedett magára annyi ismeretet e nyelvből, hogy a fenti kis függvényből külön szájbarágás nélkül is megértse, miféle szintaktikával tehet szert a betöltött függvények darabszámára, s hogy melyik számhoz melyik függvénynev tartozik.

Íme egy példa ennek használatára, azaz e fenti függvény összeépítve egy olyan „main” programmal, ami kiírja a teljes programfájl kódját, azaz kilistázza önmaga forrását:

```
(„Betöltve”);

"A progí mérete: " ?l ?p; " bájt\n";
"Most kiírom önmagam programkódját:\n"
{| ?p; ? ?#c "P" ?p-?|; |}
/;
XX
„Betöltve”
"Betöltve " ?l ?f; " függvény!\n"
{| ?f // A ciklus annyiszor hajtódik végre, ahány betöltött függvény van,
"Az " ?l ?f-?|; ". függvény neve: „" ?s ?#s „,” ?f-?|; ""\n"
|}
xx
```

Némileg egyszerűbben írva a fenti programot, az így néz ki:

```
(„Betöltve”);

"A progí mérete: " ?l ?p; " bájt\n";
"Most kiírom önmagam programkódját:\n"
{| ?p; ? ?#c "P" {|}; |} /;
XX
„Betöltve”
"Betöltve " ?l ?f; " függvény!\n"
{| ?f // A ciklus annyiszor hajtódik végre, ahány betöltött függvény van,
"Az " ?l {|}; ". függvény neve: „" ?s ?#s „,” {|}; ""\n"
|}
xx
```

Ennek ellenkezője, amikor ismerjük a függvény nevét, s keressük a sorszámát:

```
"A \"Betöltve\" nevű függvény sorszáma: " ?l ?#l "[" "Betöltve"; /;
```

Na most a sorszámokat azért lehet jó ismerni némelykor, mert van lehetőségünk függvényt meghívni nem a nevének, hanem a sorszámának a megadásával! Például, ha igaz az hogy egy függvény sorszáma 1, akkor azt így is meghívhatjuk:

```
([1]); // Meghívtunk egy függvényt paraméterek nélkül. Nincs se input, se output paraméter
```

Természetesen ez csak a legegyszerűbb példa, mert egyrészt az „1” helyén a szögletes zárójelek közt tetszőleges aritmetikai kifejezés állhat ami meghatározza a meghívandó függvény sorszámát, másrészt a szögletes zárójeles kifejezés kizárólag a függvény nevét helyettesíti, azaz a csukó szögletes zárójel után halál-nyugodtan meg lehet adni ugyanúgy az input és/vagy output paraméterlistát, mintha a „**név**” szintaktikával a függvényt a maga neve szerint hívtuk volna meg. Azaz a függvényhívó szintaxis szerint, ha a függvényre igaz az, hogy a neve NÉV és a sorszáma X, akkor a függvényhívás vagy úgy kell kezdődjön hogy

```
(„NÉV”
```

Vagy úgy, hogy

**([X])**

és minden más teljesen azonos. Ja és a "**NÉV**" helyén állhat tetszőleges stringkifejezés, az **X** helyén pedig tetszőleges aritmetikai kifejezés.

És hogy e sorszám szerinti hívogatásra miért van lehetőség... Hát amiatt, mert függvényből nem csak annyi lehet mint változóból, hanem akár jócskán több is! Konkrétan, jelenleg most maximum 10 ezer lehet betöltve egyszerre, de ennek értéke fordítási paraméterként állítható, a vz.h include fájlban van meghatározva ezzel az utasítással:

```
#define PGMTOMBSIZE 10000
```

Na de nem ez a legnagyobb baj, hanem hogy a függvényeknél már semmi esetre se követhettem el olyan illetlenséget, hogy a nevük csak maximum 1 vagy 2 vagy netán 3 karakterből állhasson! Amint ilyesmit előírok a mau programnyelvben, azonnal kinéztek volna minden jobb társaságból... és nem azért, mintha mondjuk a maximum 3 karakteres nevek variációs lehetősége túl kevés lett volna, mert ha csak az angol ABC kis- és nagybetűit veszem alapul, az 52\*52\*52 lehetőség, ami 140608, s ez már aránylag egész szép mennyiség. Hanem ez amiatt nem lett volna mégsem megfelelő, mert ha elkezd sok mindenféle ember netán majd függvény-könyvtárakat írni a mau nyelvhez, ugyan miként is dőljön el, melyiküknek jut ez vagy az a 3 karakteres név... Állandóan keveredés lett volna belőle, melyik "kis" vagy "cut" nevű függvényt hívja meg a programunk...

Szóval, muszáj volt úgy megoldani ezt, hogy itt már tényleg tetszőleges string lehessen a függvények neve. Igenám, de ez meg azzal jár, hogy nem lehet elkerülni egyáltalán semmiképpen sem, hogy bizony a függvénynek minden meghívásánál végignyálazza a belső táblázatot az interpreter, megtudni, létezik-e az a nevű függvény, s ha igen, mi az ő címe!

Na most, ez azért mégsem akkora tragédia mint a változók esetében lenne, mert függvényeket nem hívogatunk olyan gyakran, mint ahogy a változókat használjuk. Mégis, ha nem is túl gyakran, de ELŐADÓDHAAT olyan „élethelyzet”, amikor a program valami olyan sebességkritikus részhez ér, ahol még ez az idővesztés is számottevő lehet. Ilyen esetekben megszabadíthatjuk ezen felesleges keresgéléstől az interpreterünket, ha a sebességkritikus részbe való belépés előtt lekérdezzük az ott sokszor meghívandó függvényünk sorszámát a neve alapján EGYSZER, mondjuk az F változóba betöltve, eképp:

```
#l@F=?#l "[" "Baromisokszor meghívandó függvény neve áll itt"
```

Majd nem a neve hanem e változó alapján hívjuk meg őt a sebességkritikus részben (mondjuk egy ciklus belsejében 120 milliószor...):

```
([#l@F] input paraméterek \ visszatérési értékek );
```

Na most, ez volt az első, s kétségtől gyakoribb variációja a függvényeknek, ez, amikor a hívó programmal egy fájlban vannak. Amikor más fájlban vannak, annak leírása alább következik.

Ezen esetről azt kell tudni, hogy kizárólag azért, mert én olyan rém gonosz vagyok, és mindenféleképpen meg akarom nehezíteni annak a sok szerencsétlennek a sorsát aki a mau programnyelv elsajátítására adja a fejét, emiatt addig



törtem a fejemet, azaz addig lustálkodtam, míg kitaláltam, hogy programnyelvem bonyolultságát növelendő, a más fájlokban megvalósított függvények meghívásának szintaxisa PONTOSAN UGYANAZ, mintha a függvény nem is más, hanem a hívó függvénnyel azonos fájlban lenne!

Ugye milyen rémségesen gonosz vagyok, de tényleg és igazán?!

Ez azonban tényleg csak azt jelenti, hogy ott ahol meghívjuk a függvényt, ugyanúgy kell írunk mindent. Azt semmiképp se kerülhetjük el, hogy a függvény első meghívása előtt valamiképp „betöltsük” őt, azaz beolvastassuk az őt tartalmazó fájlt az interpreterrel. Erre a célra természetesen egy külön utasítás szolgál, aminek szintén „**mau**” a neve. Fontos megjegyezni, hogy ez kisbetűvel irattatik, tudniillik később ismertetek majd egy nagybetűs **MAU** utasítást, egy másik fejezetben, aminek egészen más a feladata.

E kisbetűs **mau** szintaxisa:

**mau path név**

ahol a „path” a függvény fájljának elérési útvonala, a „név” pedig az a név, aminek alapján majd meg óhajtjuk hívni őt a programunkban. Bár ezen utóbbi nevet illik feltüntetni a hívott program forrásfájljának elején valahol, de ez tulajdonképpen nem kötelező, sőt, ott lehet akármilyen más név is, ez a hívó programot baromira nem fogja izgatni, ő azon a néven keresi és hívogatja majd, amit itt a beolvasáskor adunk neki. Természetesen a „path” és a „név” helyén is állhat tetszőleges stringkifejezés.

Van azért lehetőség arra is, hogy a függvény megtudja, őt épp miféle néven azonosítva hívogatják! Mondjuk nem nagyon tudom elképzelni hogy ez miért, milyen esetekben lehet fontos egy függvény vagy program számára, de a lehetőség azért megvan rá hogy ezt lekérdezze, s e lekérdezés eredményét, ami természetesen egy string, például ilyesféleképpen irathatja ki:

**"Az én nevem: " ?s ?#s "I"; /;**

A fenti névlekérdezési lehetőség természetesen adott azon függvények számára is, melyek a hívó programmal azonos fájlban vannak. Ezek azt is megtehetik, hogy miután így megtudták a saját nevüket, ezután lekérdezzék a sorszámukat, mondjuk így:

**"Az én sorszámom: " ?l ?#l "[" ?#s "I"; /;**

Bár ennek se tudom, miféle gyakorlati haszna volna. Ennek ellenére, megteheti a függvény. Azok a függvények azonban, melyek külön fájlban vannak, már ezt az utóbbit NEM tehetik meg!

A nevüket le tudják kérdezni. A sorszámukat azonban nem. Lekérdezhetik, csak hogy ami eredményt kapni fognak ekkor, az NEM LESZ AZ IGAZI sorszámuk! Ennek okát a nyelv megalkotásáról szóló tanulmányban ismertettem, itt nem részletezem.

Ehelyütt kell kitérnem arra, hogy más fájlokból csak EGY program hívható meg ezzel a módszerrel - az, ami ott a főprogram, azaz „legelöl áll”. Természetesen lehet azon fájlban akárhány további másik program is, de azokat a mi progra-

munk, ami ezt a másik programfájlt meghívta, már nem tudja meghívni, elérni, azokat csak az a program hívogathatja, ami abban a fájlban a főprogram! Ő azonban nem képes meghívni azokat a programokat, amik az őt hívó programmal vannak azonos fájlban. Azaz, minden program azon másik programokat tudja csak meghívni, amik vele azonos fájlban szerepelnek, illetve ami egy betöltött külső programfájl legelső programja, tehát ami abban a fájlban a főprogram!

Na most, azt ugye írtuk, hogy ezen programot ami másik fájlban szerepel (s abban a fájlban a főprogram) azt ugyanolyan szintaxissal kell meghívunk mint a többi más programokat. Miként kell azonban megírni ezen más fájlban levő programokat?

Nos, nem nagy a különbség. Egy efféle például így néz ki az elején:

```
#!„mauls” // Mau nyelvű tartalomjegyzék-listázó külső függvény
// Input paraméterek:
#s@A // A listázandó tartalomjegyzék neve
```

Vagy ha le akarjuk nullázni nála a változókat a hívás kezdeténél:

```
#!„mauls” 0 // Mau nyelvű tartalomjegyzék-listázó külső függvény
// Input paraméterek:
#s@A // A listázandó tartalomjegyzék neve
```

Az első sor tulajdonképpen csak egy megjegyzés, nem kötelező. Ez csak egy konvenció, ami arra utal, hogy ez egy mau nyelven írt külső függvény. E tényre a „ ” jelpáros utal, köztük van megadva a függvény neve. Utána állhat akármennyi **//** stílusú komment, majd a korábban már elmagyarázott szabályok szerint a függvény input paraméterei.

A legelső input paraméter előtt azonban okvetlenül kell álljon ha nem is egy efféle névmeghatározás, de legalább egy közönséges **//** típusú komment, és/vagy egy vessző vagy pontosvessző. Azaz semmiképp se kezdődjön a forráskód egyszerűen az input paraméterrel, mert akkor hibát fog jelezni. Ezt könnyen ki lehetne kerülni, de nem teszem, mert helyesnek látom ha ki van kényszerítve hogy ha sehol máshol nem is, de legalább a függvények legislegelején álljon legalább egyetlen nyamvadt megjegyzéssor, ami megmondja, mi a fészkes fenét is csinál ez a függvény.

Mint azt már tudjuk eddigre, a függvények külön névtérrel rendelkeznek, azaz ha van a függvényben mondjuk egy **#c@G** nevű változó, annak semmi köze a függvényt meghívó progi ugyanilyen nevű és típusú változójához. Azoknál a függvényeknél azonban, melyek a hívó függvénnyel EGY FÁJLBAN VANNAK, és CSAK EZEKNÉL (!!!!), lehetősége van a hívott függvénynek kiolvasni a SZÜLŐ NÉVTÉR valamely változójának tartalmát!

**FIGYELMEZTETÉS!** NEM A HÍVÓ VÁLTOZÓJÁT olvashatja ki, hanem annak a „főprogramnak” a változóját, mely a „main” az ő fájljában, tehát azon fájl „legfelső szintű”, azaz a fájl legelején álló programnak a változóját!

Ennek jelölése pedig úgy történik, hogy a változónevet bevezető **@** jel után közvetlenül odairunk egy **^** jelet is.



Azaz például:

**#s@g** a függvény „g” nevű stringváltozóját jelenti.

**#s@^g** a függvény szülőnévterének „g” nevű stringváltozóját jelenti.

És ismétlem, ezek csak input változók lehetnek, azaz csak „jobbértékben” szerepelhetnek. Értéket adni nekik nem lehet. Valamint, ez nem vonatkozik a veremváltozókra. Irtó nagy gondok lehetnének belőle, ha egy hívott függvény elkezdene értékeket kiszedegetni a szülőnévtér vermeiből.

Használható azonban e módszer még néhány más speciális esetben is, ezek a következők amint e példák mutatják őket:

```
<s @^o @S; // Output fájlba írás
```

```
?#l "T#" ^ t @t; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó @t típusú bejegyzéseinek darabszámát kérdezhetjük így le
```

```
?#s "Tn" ^ t, #l@i, @t; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó @t típusú bejegyzései közül a #l@i változó által megadott sorszámúnak a nevét kérdezhetjük így le
```

```
?#s "Tg" ^ t, #l@i, @t; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó @t típusú bejegyzései közül a #l@i változó által megadott sorszámúnak a csoportját kérdezhetjük így le
```

```
?#s "Tt" ^ t, #l@i, @t; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó @t típusú bejegyzései közül a #l@i változó által megadott sorszámúnak a tulajdonosát kérdezhetjük így le
```

```
?#c "EXE" ^ t, #l@i, @t; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó @t típusú bejegyzései közül a #l@i változó által megadott sorszámúnál azt kérdezhetjük le, hogy végrehajtható állomány-e.
```

```
?#c "L?" ^ t, #l@i, @t; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó @t típusú bejegyzései közül a #l@i változó által megadott sorszámúnál azt kérdezhetjük le, hogy link-e, s ha link akkor törött-e.
```

```
?#s "JOG" ^ t, #l@i, @t; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó @t típusú bejegyzései közül a #l@i változó által megadott sorszámúnak a jogosultságait kérdezhetjük így le oktális formában
```

```
?#s "LINK" ^ t, #l@i; // A függvény szülő névterében levő „t” nevű tartalomjegyzék-változó link típusú bejegyzései közül a #l@i változó által megadott sorszámúnál azt a stringet kérdezhetjük így le, amely fájlra vagy könyvtárra a symlink mutat.
```

Igazából, általában véve is kerülendőnek tartom e változóhasználatot. Hatalmas, és rém nehezen kideríthető hibák forrása lehet. Ezen **^** által hívott változók tulajdonképpen globális változóknak minősülnek minden, az adott fájlban szereplő függvény számára, márpedig jól tudjuk már a C nyelvű programozásban szerzett tapasztalataink alapján is, hogy a globális változók használata nagyonis kerülendő! Néha azonban, nagyritkán, jól jöhet e lehetőség, például olyankor, ha egy nagy tömbről van szó, amit sok függvénynek kell használnia, márpedig tömb összes elemét egyenként átadogatni egy függvényhívásnál paraméterekként, hát az izé... se nem szép, se nem hatékony, szóval akkor már legyen az inkább globális változó. De mondom, az ilyesmivel nagyon óvatosan, „ésszel” kell bánni, mert ez veszélyes lehetőség!

Egy példa azért egy programomból, amikor jól jön ilyesmi:

```
{| ?#l "T#" ^ t @t ; // Fix számszor lefutó ciklus, annyiszor fut le ahány fájlbejegyzésünk van.
if #c@^(@t)[?~] E G #l@B // Meghívjuk a tartalomjegyzék-bejegyzés típusától függő szubrutint
|}
```

Nagyon fontos azt is megemlítenünk, hogy e függvényhívási módszer **TELJESEN ALKALMATLAN REKURZÍV HÍVÁSOK KEZELÉSÉRE!** Ennek oka, hogy egyszerűen nem veremtáron keresztül adja át illetve fogadja a hívó függvény a paramétereket. Valamint a programmutató sincs ilyenkor veremtárba mentve. Az egész függvényhívási metódus megvalósítása, sőt, már a függvények de még magának a PGM nevű mau-programosztálynak az adatszerkezete is amire ez az egész épül, olyan, hogy teljességgel alkalmatlan bármiféle rekurzivitás lekezelésére is!

Sokat töprengtem e dolgon, s végül oda jutottam, hogy hiába alkalmatlan rá, MEG TUDNÁM CSINÁLNI. Igen, meg lehetne erőszakolni a struktúráját úgy, hogy erre képes legyen, de csak azon az áron, hogy egyrészt iszonyatosan túlbonnyolódna a kód, másrészt örületes memóriapocséklással járna egy rekurzív hívás lebonyolítása. Nem érné meg. Nem kell azonban elcsüggedni, van mód a mau nyelvben is rekurzív hívások lebonyolítására TERMÉSZETESEN, de annak leírása egy másik fejezet tárgya.

## 19. fejezet - Változó hosszúságú paraméterlista kezelése

Ami a változó hosszúságú paraméterlistával rendelkező függvények kezelését illeti, íme rögvest egy mau nyelvű példa, ezen magyarázok el mindent!

```
"Meghívom a függvényt\n"
```

```
(„pontok” 4, 425,#c64,#c65,#c66,#c67)
(„pontok” 6, 512,#c68,#c69,#cz,#c70,#c71,#cu \ #l@L)
```

```
"Visszatérési érték: " ?l @L; /;
```

```
XX
```

```
„pontok” // változó hosszúságú paraméterlistát váró függvény
// Input paraméterek:
#c@c, // Ez mondja meg hány további paramétert várunk
#i@i,
[#c = #c@c], // Memória foglalás a megfelelő számú unsigned char értékek
... // Minden további paraméter verembe kerül
```

```
? #c@[0]; /;
?i @i; /;
```

```
{| #c@c; ? #c@[?]-1]; |} /;
```

```
\ 65537
xx
```

Látható, 2 alkalommal hívtuk meg a „pontok” nevű függvényt, s tényleg különböző hosszú paraméterlistával! Na most, a hívott függvény az elején felsorol 2 (unsigned char és unsigned short int típusú, de lehetne más típus is) input változót. Ez rendben is van, mindkét alkalommal amikor ő meghívatik, ezekbe belekerül az első 2 átadott paraméter, először 4, 425, azután 6, 512. Ezután azonban egy olyan fura tudákos hogyishívják jön, hogy:

```
[#c = #c@c], // Memória foglalás a megfelelő számú unsigned char értékek
```

Na most, ez a fenti kutyulmány memóriefoglalás. Veremtárnak foglalunk memóriát, a **#c** azt mondja hogy unsigned char értékeknek. Az egyenlőségjel után van megadva, hány darab ilyen értéknek foglalunk vermet. Itt most ezt a darabszámot egy változó határozza meg, az, amit az imént kaptunk a hívó függvénytől! Lehetne a helyén valamilyen konstans szám is, hogy mondjuk 30, vagy a **^** karakter segítségével hivatkozhatnánk itt a szülő névtér valamely változójára is. Az a darabszám amennyinek itt helyet foglalunk, a **maximális** darabszám, tehát nem muszáj hogy mindegyik használtassék is.

Veremtárlefoglaló utasításunk ismerős kell legyen már, de van egy furcsasága: csak a típusa van megadva, de nincs NEVE! S ez nem véletlen. Ezek különleges vermek. Ezzel az utasítással kizárólag a függvény legelején, a paraméterátadás ideje alatt foglalhatunk vermet, s e vermeknek abszolút semmi közük a korábban megismert verem-típusú változóinkhoz! E vermekbe a hívott függvény nem is képes adatot tenni, ezek kizárólag „input” vermek, ezekbe semmi más nem kerülhet, mint a hívó függvény által szolgáltatott input paraméterek! Ezokból tehát e vermek adatait csak „jobbértékként” lehet felhasználni. E vermek minden híváskor újra létrejönnek, nem is okvetlenül ugyanazon mérettel, s a függvény lefutása után megsemmisülnek. Minden adattípusnak külön vermet specifikálhatunk, a megfelelő típusjelölő karakter segítségével. Azaz lehet vermünk a

**#c, #C, #i, #I, #l, #L, #g, #G, #f, #d, #D** és a **#s** típusokra.

Ebben a sorban:

```
? #c[0]; /;
```

látható, miként hivatkozunk a verembe rakott elemre. A hívó függvény által a verembe rakott első elem lesz a nulladik indexű a hívott függvény számára, azaz ahogy fel vannak sorolva a paraméterek a hívó függvény listájában balról jobbra, úgy nő az indexük nullától számítva. A verembe legelőször akkor kerül elem a paraméterátadás során, amikor a hívott függvénytől eléri egy „...” karakterhez. Ez jelzi azt, hogy innentől az összes további esetleges paramétert ami még van, a vermekbe kell pakolja a hívó függvény. Természetesen minden adat a maga megfelelő típusú vermébe kell kerüljön, s ha annak a típusnak nem foglalt le memóriát a hívott függvény, akkor az hibajelzéshez vezet, amint a hívó épp egy olyan típusú adatot óhajtana bepakolni.

Nézzük aztán meg ezt a sort, a függvényhívást:

```
(„pontok” 4, 425, #c64, #c65, #c66, #c67)
```

A paramétereknél a 4 és 425 számoknál nincs semmi baj. A többi azonban mintha kissé furcsán festene. Nos igen, amiatt, mert mindegyik szám előtt ott a casting operátor! Ha nem konstans számokat adnánk meg hanem változókat a teljes nevükkel, nem volna probléma, mert mondjuk a **#d@f** alakból egyértelműen kiderül, hogy ez egy double szám. Ha azonban csak egy szám áll ott, abból nem tudja az interpreter, ezt most épp melyik verembe kell tegye: az unsigned char verembe, vagy a signed long long verembe, vagy szóval most mit is kezdjen vele?! Emiatt a verembe kerülő paramétereknél minden esetben szigorúan ki kell tenni a paraméter elejére a megfelelő casting operátort.

Nézzünk meg egy másik példát, ahol stringet is átadunk:

```
(„pontok3” 6, 2, 4, #c68, #c69, #cz, #i384#c70, #c71, #cu, #i1024, #s"String-2!", #s(#s@s)+"valami" \ #s@v)
```

Látható itt a végén, hogy egy olyan kifejezést adunk át, ami úgy keletkezik, hogy a **@s** típusú stringváltozóhoz hozzáadjuk a "valami" konstans stringet. Minthogy efféle stringkonkatenáció esetén a változókat zárójelbe tesszük, így hogy rögvest tudja az interpreter hogy string következik, muszáj ezelé is odabiggyeszteni a **#s** casting operátort. Ez előtt van egy idézőjelek közt álló stringkonstans is, az elé is kell a casting operátor, láthatjuk. E stringekre a hívott függvényben így hivatkozhatunk, mondjuk egy kiíró utasításnál:

```
?s #s@[0]; /;
?s #s@[1]; /;
```

Bár e paraméterek veremekben vannak, de attól hogy a hívott függvényben valahol eképp használjuk őket, nem kerülnek még ki a veremből. Csak akkor törlődnek, amikor a függvény teljesen végetér, s a vezérlés visszakerül a hívóhoz.

Egy hosszabb példa a változó hosszúságú paraméterlistára:

```
"Meghívom a függvényt\n"

#s@s="Ez a második string!";

(,,"pontok"" 4, 425,#c64,#c65,#c66,#c67)
(,,"pontok"" 6, 512,#c68,#c69,#cz,#c70,#c71,#cu \ #l@L)

"Visszatérési érték: " ?l @L; /;

//(,,"pontok2"" 6, 2, #c68,#c69,#cz,#c70,#c71,#cu,#i384,#i1024)
(,,"pontok2"" 6, 2, #c68,#c69,#cz,#i384#c70,#c71,#cu,,#i1024)
"Stringátadás:\n"
(,,"pontok3"" 6, 2, 4, #c68,#c69,#cz,#i384#c70,#c71,#cu,,#i1024,#s"String-1!",#s@s)
"Stringátadás 2:\n"
(,,"pontok3"" 6, 2, 4, #c68,#c69,#cz,#i384#c70,#c71,#cu,,#i1024,#s"String-2!",#s(#s@s)+"valami" \
#s@v)

"Visszatérési érték: " ?s @v; /;

XX

,,pontok" // változó hosszúságú paraméterlistát váró függvény
// Input paraméterek:
#c@c, // Ez mondja meg hány további paramétert várunk
#i@i,
[#c = #c@c], // Memórafoglalás a megfelelő számú unsigned char értéknek
... // Minden további paraméter verembe kerül

? #c@[0]; /;
?i @i; /;

{| #c@c; ? #c@[?|-1]; |} /;

\ 65537
xx

,,pontok2" // változó hosszúságú paraméterlistát váró függvény
// Input paraméterek:
#c@c, // Ez mondja meg, hány további unsigned char paramétert várunk
#c@i, // Ez mondja meg, hány további unsigned short int paramétert várunk
[#c = #c@c], // Memórafoglalás a megfelelő számú unsigned char értéknek
[#i = #c@i], // Memórafoglalás a megfelelő számú unsigned short int értéknek
... // Minden további paraméter verembe kerül

"Pontok2\n"
```

```

? #c@[0]; /;
?i #i@[0]; /;

{| #c@c; ? #c@[?|-1]; |} /;
{| #c@i; ?i #i@[?|-1]; /; |} /;

xx

„pontok3” // változó hosszúságú paraméterlistát váró függvény
// Input paraméterek:
#c@c, // Ez mondja meg, hány további unsigned char paramétert várunk
#c@i, // Ez mondja meg, hány további unsigned short int paramétert várunk
#c@s, // Ez mondja meg, hány string paramétert várunk
[#c = #c@c], // Memórafoglalás a megfelelő számú unsigned char értéknek
[#i = #c@i], // Memórafoglalás a megfelelő számú unsigned short int értéknek
[#s = #c@s], // Memórafoglalás a megfelelő számú string értéknek
... // Minden további paraméter verembe kerül

"Pontok3\n"

? #c@[0]; /;
?i #i@[0]; /;

{| #c@c; ? #c@[?|-1]; |} /;
{| #c@i; ?i #i@[?|-1]; /; |} /;
"Az átadott stringek:\n"
?s #s@[0]; /;
?s #s@[1]; /;
\ "Ez a string van visszaküldve!"
xx

```

## Eredménye:

```

Meghívom a függvényt
@
425
CBA@
D
512
uGFzED
Visszatérési érték: 65537
Pontok2
D
384
uGFzED
1024
384

Stringátadás:
Pontok3
D
384
uGFzED
1024
384

Az átadott stringek:
String-1!
Ez a második string!
Stringátadás 2:
Pontok3
D
384
uGFzED
1024
384

Az átadott stringek:
String-2!
Ez a második string!valami
Visszatérési érték: Ez a string van visszaküldve!

```

Miután már elég sok mindent jelölünk szögletes zárójelekkel, illendő egy rövid, tömör összefoglalást iderittyenteni e jel szintaktikájáról:

```
#c = #c@c], // Memória foglalás a megfelelő számú unsigned char input paraméter értéknek
#c@k=#c@[0]; // Kiolvasás az unsigned char értékeket tartalmazó input paraméter veremből, 0
indexszel
#c@c=#c@v[]; // A v nevű verem aktuális unsigned char eleme.
#c@v[]=4; // Beteszünk 4-et a v nevű verembe unsigned char számként.
[@v[2000]] // 2000 bájtot lefoglalunk a „v” nevű verem számára
[@v[]]; // Törölöm a v nevű vermet
[#c@t=100]; // Memória foglalás 100 darab unsigned char értéknek a "t" nevű tömbbe
#c@t[#i@i]=4; // Értékadás a t nevű, unsigned char értékeket tartalmazó tömb egy elemének
#c@c=#c@t[#i@i]; // A tömb egy értékének kiolvasása
[#c@t]; // A t nevű unsigned char értékeket tartalmazó tömb törlése (a neki lefoglalt
memóriaterület felszabadítása)
#c@c=#s@s[4]; // A 4-es indexű karakter az s nevű stringből
#s@s=[3,7]#s@s; // Az s nevű string egy darabjának képzése
[@t[[4]]]; // lefoglalok területet 4 string részére egy t nevű stringtömbbe
#s@t[[1]]="macska"; // Értékadás stringtömb egy elemének
#s@t[[1]][3]='@'; // Értékadás egy olyan karakternek, mely egy stringtömb egyik stringjében van.
[@t[[[]]]]; // Felszabadítom a stringtömb memóriaterületét
#c@c=#s@s[[6]][4]; // A 4-es indexű karakter az S nevű stringtömb 6-odik eleméből
```

## 20. fejezet - „Igazi” függvények és rekurzív függvények

Amikor páran megtudták hogy programnyelvet írok, s megnézték az eddigi eredményeimet, egyesek kritizálni kezdték, mondván hogy amit függvénynek nevezek, igazából nem függvény, mert nem építhető be aritmetikai kifejezésekbe! Nos, ez már attól függ, miként definiáljuk a függvény fogalmát. Mint tudjuk jól a C nyelv esetéből is, egy függvény visszatérési értékét nem okvetlenül kell fogadnunk. Sőt, lehet hogy direkt **void** visszatérési értékűre írjuk, s akkor máris nagyon hasonló az eddigi mau függvényekhez! Ettől azonban még csinálhat nekünk valami hasznosat, sőt, még értékeket is adhat vissza, mutatókon vagy referencia típusú paramétereken keresztül.

Ennek ellenére, kritikusaimnak igazuk van abban, hogy hasznos volna ha a mau nyelv rendelkezne valami „hagyományosabb” stílusú függvényhívási lehetőséggel is. Ennek persze az az ára, hogy egy efféle függvénynek tényleg csak 1 visszatérési értéke lehet, ráadásul az se mindegy, annak milyen a típusa! Természetesen e függvények egymásba ágyazhatóak kell legyenek, na és ha már megcsináljuk e huncutságot, akkor ez már legyen olyan, ami alkalmas önmaga meghívására is, azaz rekurzívan is hívható!

Nos, ezt a függvénytypust a következő programon mutatom be. E program kiszámolja az 5 és a 6 faktoriálisát. A faktoriálisszámító függvény önmagát hívja rekurzívan:

```
#l@i=?(#l „faktoriális” #l5); // Itt az 5 szám faktoriálisát számíttatjuk ki
?l @i; // Ki is írjuk az eredményét
#l@i=?(#l „faktoriális” #l6); // Itt meg a 6 szám faktoriálisát számíttatjuk ki
?l @i; // Ezt is kiíratjuk
```

XX // Itt a (főprogram) vége, fuss el véle!

„faktoriális” // Faktoriális-számoló függvény. REKURZÍV!

```

if (#l@^[0])<2 T \ #l1; // A nulla és az 1 faktoriálisa mint tudjuk 1.
// Ha nem 0 vagy 1, az adott számot megszorozza az eggyel kisebb szám faktoriálisával, s ezt
adja vissza:
\ #l@^[0]*?(#l „faktoriális” #l(--@^[0]));
xx // vége a függvénynek

```

Látható, hogy miként kell meghívni az efféle programokat. A **?(** parancs jelzi, hogy függvény következik, rögtön ezután kell álljon a **#** casting operátor, mely az utána álló karakterrel azt mondja meg, milyen típusú függvényhívás következik. Ez azért fontos, mert olyan típusú paramétert ad vissza a hívott függvény a hívónak. Ezután következik a hívott függvény meghatározása. Ez egészen ugyanúgy megy mint az eddigi függvények esetében: azaz a „ és ” karakterek közt megadhatjuk a függvény nevét stringkifejezéssel, vagy használhatjuk a **[sorszám]** alakot is, ahol a sorszám egy aritmetikai kifejezés. Ezután jönnek az input paraméterek, amikből akármennyi lehet, azaz ez változó hosszúságú paraméterlista átadására is alkalmas. Természetesen amiatt, mert itt is veremtárakba kerülnek a paraméterek, s itt is muszáj jelezni mindegyik előtt a casting operátorral a paraméter típusát. Ám jelen esetben ezek a veremtárak amikbe a paraméterek kerülnek, nem azonosak a korábbi függvényhívásnál említett veremtárakkal! Ezek mérete a **vz.h** fájlban van meghatározva, eképp:

```

#define gveremcmax 1000
#define gveremCmax 1000
#define gveremimax 1000
#define gveremImax 1000
#define gveremlmax 1000
#define gveremLmax 1000
#define gveremgmax 1000
#define gveremGmax 1000
#define gveremfmax 1000
#define gveremdmax 1000
#define gveremDmax 1000
#define gveremsmax 1000
#define gverempmax 1000

```

Ezen veremek globálisak, s eképp vannak deklarálva a main.cpp fájl elején:

```

// Globális adatvermek

VEREM gveremc;
VEREM gveremC;
VEREM gveremi;
VEREM gveremI;
VEREM gvereml;
VEREM gveremL;
VEREM gveremg;
VEREM gveremG;
VEREM gveremf;
VEREM gveremd;
VEREM gveremD;
VEREM gverems;
VEREM gveremp;

```

A **?(** utasításhoz érve, az aritmetikai kifejezés kiértékelő függvény beolvassa az utána következő casting operátort, s ennek megfelelően hívja meg valamelyik függvénykiértékelő rutint. Ez a casting operátor a szokott **#** karakter, ami után a már jólismert típusjelölő karakterek valamelyikének kell állnia - vagy ha nem azok egyike áll ott, akkor egy nyitó kerek zárójel kell következzen, s utána egy unsigned char típusú kifejezés amit csukó kerek zárójel zár le, s ami meghatározza a függvény típusát. (Hogy milyen típusú visszatérési értéket ad vissza). Azaz, e függvényeink lehetnek olyanok is, amelyek típusa futásidőben dől el! Efféle függvény meghívása tehát mondjuk így nézhet ki:

```

#(t)@a=?(#(t) „függvény neve”, #i@p);

```



Látható az is, miként hivatkozunk ilyen függvényekben az input paraméterekre amik a típusuknak megfelelő veremtárakban csücsülnek:

```
#l@^[0]
```

Ez nagyon hasonló ahhoz, ahogy az előző fejezetben bemutatott függvényhívásnál láthattuk, s logikus is hogy itt a ^ karaktert használjuk megkülönböztetésül, mert ezen vermek globálisak.

## 21. fejezet - Bencsmarkok

Jó dolog ha le tudjuk mérni, egyes programrészek mennyi ideig futnak, azaz ha bencsmarkolhatjuk ezeket. Ennek érdekében a mau programnyelvnek van 256 speciális időváltozója, melyek nevére ugyanazon szabályok érvényesek, mint a többi változóéra (azaz unsigned char értékek jelzik a nevüket). Ezek típusát a #t casting operátor jelzi. Fontos megjegyezni azonban, hogy a közönséges változóktól eltérően ezek kezdeti értékét semmi módon nem garantálja a mau programnyelv! E változók a processzor aktuális „tick” értékeinek a számát tárolják, ami jelenleg (legalábbis 64 bites rendszerben, Linux oprendszer alatt, x86 architektúrán) egymilliomod másodpercekben van mérve.

Ezek jó nagy számok, emiatt nemigen érdemes másképp kezelni őket mint unsigned long long értékeket, ettől függetlenül azonban tettem róla hogy átalakíthatóak legyenek más típusokká is. Használatuk:

```
#t@a; // inicializálunk egy időváltozót
{| 3000000
#i@i=?|;
|}
#t@b; // inicializálunk egy másik időváltozót
#g@e=(#t@b)-(#t@a);
"Eredmény: " ?g @e; /;
```

A fenti programnak persze a világon semmi értelme, hótt' ökörség számos szempontból. Arra azonban jó, hogy lecsekkoljuk, mennyi ideig tart egy 3 milliószor lefutó ciklus nála. Az eredményt milliomod másodpercekben kapjuk meg; az egyes futások eredményei közt - főleg ha sokmagú gépen futtatjuk - lehet akár 3-4% eltérés is. Nálam, T530-as gépen core i7 processzorral ez kb 1.15 másodpercig tartott. A fenti kis programból teljesen világosan kiderül e bencsmark-változók használata.

Ha már a bencsmarkoknál tartunk, ide kívánczik - mert ez is olyan „idővel összefüggő dolog” - két pici utasítás, melyek meghatározott időre felfüggesztik a program végrehajtását:

```
"Várok 5 másodpercet!\n"
WS 5;
"Most várok 500000 mikroszekundumot!\n"
WS 500000;
```

Gondolom e példa is magáért beszél.

## 22. fejezet - A mau interpreter és a shell kapcsolata

A mau programok illik hogy a következő sorral kezdődjenek:

```
#!/mau
vagy valami ilyesmivel:
#!/bin/mau
```

Ez nem kötelező, de ajánlatos, mert ebből tudja a shell, hogy ez egy olyan fájl, amit a „mau” nevű programmal kell végrehajtani. Ezesetben tehát megpróbálja meghívni a mau interpretert, ami normális esetben épp a „mau” néven kell szerepeljen, s jó dolog ha olyan könyvtárban van, ami szerepel a **\$PATH** rendszerváltozónkban. A shell ekkor ezt tehát meghívja, s átadja neki parancssori paraméterként a végrehajtandó mau program fájlját. Meg minden más parancssori paramétert is, amit e fájl után felsoroltunk. E parancssori paraméterek lekérdezésére a mau programból van lehetőség, ezt leírtam korábban e doksiban a „Mau rendszerváltozók és rendszerfüggvények” című fejezetben.

A mau program végetér az utolsó utasításnál. Ha előbb akarjuk befejezni, arra az **XX** parancs szolgál. (Mindkét X karakter NAGYBETŰS kell legyen!) Miután a főprogram a fájl első programja kell legyen, ezért ezen **XX** parancsot okvetlenül muszáj használnunk azesetben, ha a főprogram után a fájlba függvényeket is írunk.

Amennyiben kifejezetten úgy akarjuk befejeztetni a mau programot, hogy a shell tudomására hozzuk hogy e mau program valami hibával ért véget, akkor használjuk az

**X!**  
mau parancsot a kilépéshez. Ez ugyanaz, mintha egy C programból az `exit(EXIT_FAILURE);` paranccsal lépnénk ki.

A shell meghívható a mau programból is, nagyon hasonlóan mint az a C nyelv esetén szokásos a „system” rendszerfüggvénnyel, már amiatt is, mert az alábbi mau parancs éppen pontosan e „system” függvényt használja:

```
"Tartalomjegyzék-lista:\n"
SY "ls -l";
"Tartalomjegyzék-lista vége.\n"
```

Másik példa:

```
"Tartalomjegyzék-lista:\n"
#s@p="ls ";
#s@s="-l";
SY (@p)+(@s);
"Tartalomjegyzék-lista vége.\n"
```

Gondolom, e példák maguktól értetődőek. A megfelelő mau parancsnak persze épp amiatt **SY** a neve, mert ez a „**s**ystem” nevű C függvényre utal.

Azt is megtehetjük stringekkel, hogy a stringbe letárolt „mau” programnyelven írt parancsokat végrehajtjuk! Ez efféleképp működik:

```
#c@h=3;
#s@s="{| 5; ?c @h; /; #c++@h; |} xx"
"Kezdődik a stringparancs végrehajtása!\n"
MAU @s;
"Befejeződött a stringparancs végrehajtása!\n"
"A \"h\" értéke most: " ?c @h; /;
```

A program kimenete:

```
Kezdődik a stringparancs végrehajtása!
3
4
5
6
7
Befejeződött a stringparancs végrehajtása!
A "h" értéke most: 8
```

A string végrehajtását, mint látható, a „**MAU**” nevű parancs végzi. Ez ugyan 3 karakter látszólag, illetve valóságosan is, igazából azonban ezt a „**MA**” nevű parancs végzi a rendszerben, épp csak legelső teendőjeként leellenőrzi, a legislegelső karakter a programkódban őtána az „**U**” betű-e. Ha nem az, akkor syntax error hibaüzenettel kilép.

Nem vagyok a híve a felesleges ellenőrizgetéseknek, de efféle egzotikus dolgot mint egy string végrehajtása, feltehetőleg nem fogunk gyakran végezni, s akkor egyetlen plusz bájt ellenőrzése még talán elfogadható, azért, hogy a forráskódból egyértelműen kiderüljön, mit is csinálunk ott. Azaz tudható legyen ez rögvest a parancs nevéből. Pláne mert azért az ilyesmi rizikós dolog, s esetleg biztonságügyileg is aggályosnak mondható. Jó azonban ha van, csak ésszel kell használni, mert néha igenis aranyat érhet!

Mint látható a példaprogram stringjéből, azt ugyanazzal az **xx** nevű utasítással kell lezárni mint a függvényeket. A dupla nagy X, vagyis az **xx** mint tudjuk a program végét jelzi, e dupla kis ixek pedig a „BREAK” utasítás. Ez direkt arra van hogy visszatérjünk rendben valami efféle huncutságból. Ezzel kell lezárni az efféle paraméter(vagy parancs?)stringeket.

Fontos tudni, hogy efféle, stringben tárolt parancs-sorozat végrehajtása közben **nem működnek a címkéink!** Az értékük kiolvasható ugyan, de hiába próbálunk ugrani rájuk, mert a programmemória mutatója ideiglenesen át van állítva a string kezdetére, azaz nem találja meg a címkék által jelölt pontokat. Lehet azonban ugrálni a stringen belül, tudniillik ha konkrét numerikus értéket adunk az ugróutasításunk paraméteréül, amely a string valahányadik bájtját jelöli. Végeredményben egy ciklus esetén is ugróutasításokról van szó, s látjuk a fenti példán, hogy a ciklus remekül működik!

Igenám, de előre tudható, hogy hajlamosak leszünk elfeledkezni a string végébe beleírni az „**xx**” karaktereket... Nos, nem kell aggódni! A mau interpreter jól tudja, hogy az ember, még a programozó is, „esendő lény”, és csak egy „csekélyértelmű medvebocs”, azaz azzal kezdi az efféle stringek végrehajtását, hogy a végükre odabiggyeszt egy ilyen kiegészítést:

```
" ; xx"
```

Vagyis, nem muszáj lezárni nekünk a stringeket az **xx** paranccsal, ezt az interpreter megteszi helyettünk. Ha azonban megteesszük mi is, akkor az nem hiba.

Na most, a stringben letárolt parancs végrehajtásának lehetőségét amiatt épp ebbe a fejezetbe írtam be, mert ennek leginkább tényleg olyankor lehet jelentősége, amikor valamely általános célú programot írunk, melyet a shellen keresztül óhajtunk vezérelni, neki küldött parancs-stringekkel, hogy épp mit csináljon! Tipikusan ilyen lehet például egy parancssoros kalkulátorprogram... Hogy ne csak a levegőbe beszéljek, itt van mindjárt egy, ami elméletileg úgy kb mindent tud, és mégis alig kell hozzá programsor:

```
#!/mau // kalkulátorprogram
#s@fff[10000]=0; // parancs-string maximális méretének a beállítása

$st; // Kezdődik a kalkulátorprogram
#i@fff:7=0; // A beolvasandó bájtok kezdőindexe
#s@fff[0]=0; // parancs-string lenullázása
Bw; Sr; "mau> "; Bd; Sd; // kalkulátor-prompt kiírása
$ci; // A beolvasóciklus eleje
#c@fff=?#c "("; // beolvasunk 1 bájtot a standard inputról
if(@fff)==10 T »$ki // Ha Entert ütöttünk le, vége a beolvasásnak, ugrás a végrehajtási részhez
#s@fff[#i@fff:7]=@fff; // elementjük a beolvasott bájtot
#i++@fff:7; // Az index inkrementálása
»$ci // ugrás a beolvasóciklus elejére

$ki; // Végrehajtási rész
if(#s@fff[0]) E »$st; // Ha üres a string, nem hajtja végre
if(#s@fff[0]==X) T XX;
#s@fff[#i@fff:7]=''; #i++@fff:7; #s@fff[#i@fff:7]=10;
#i++@fff:7; #s@fff[#i@fff:7]=32;
#i++@fff:7; #s@fff[#i@fff:7]=0; #s!@fff;
MAU @fff; // végrehajtjuk a stringet
»$st // Ugrás a starthoz
```

A progi futásának outputja egy példasorral:

```
vz@Csiszilla /Releases/2014/U/Common/vz/MAU=>./mau calc.mau
mau>
mau> "Ezt írom ki!\n"
Ezt írom ki!
mau> ?g 8*(3+2); /
40
mau> #l@h=100
mau> ?l (@h)+3; /
103
mau> #c@h=3
mau> {| 5; ?c @h; /; #c++@h; |} /
3
4
5
6
7

mau> X
```

Látható a fentiekből, ez tényleg tud mindent amire a mau nyelv képes... még ciklusokat is tud, meg a betáplált változóértékeket megjegyzi, „meg a satöbbi és a minden”. Egyedül arra kell figyelni a használata során, hogy a parancsbetáplálás közben ha változókat használunk, ne épp a **#s@fff** stringváltozót használjuk, és ne a **#i@255:7** unsigned short int változót, meg ne a **#c@255:0** unsigned char változót. Azaz ne a 255-ös indexű változókat úgy általában, mert az kell e

proginak. Ezeknek épp amiatt ez a nehézkes név van adva a kalkulátor-programban, hogy minimalizáljuk az esélyét annak hogy a Felhasználó épp olyan változót választ, amit használ a progí maga is.

Természetesen nyilván lehetne e progít is csicsázni még, mondjuk ha teszem azt az utasításvégrehajtás előtt a progí a maga változóit verembe menti vagy valami ilyesmi... De tojok rá, nekem így is nagyon megfelel, nem fogom agyonbonyolítani, különben is ez csak egy gyors kis példa. Csak annyit még, hogy a **MAU** parancs végrehajtása előtt amiatt van ott az a sok értékadás, hogy némi kényelmet nyújtunk a Felhasználónak, azaz ugye a kiíratások végét le kell zárni egy **/** jellel, hogy új sort is írjon ki, de ez után ne neki kelljen mindig kirakni a pontosvesszőt... Valamint, muszáj nekünk lezárni a stringet a nullabájtjal és „rendbehozni” a hosszát, mert ha megnézzük a kódot, látjuk hogy az elején 10000 hosszúra állítottuk be a stringet. És azután e hossz nem is változik neki, tudniillik mert csak indexeléssel pakolászunk bele bájtokat. Na most a MAU parancs a string végéhez ugyan hozzámásolja amit akar, de az édeskevés, mert a 10 ezredik karakter s az általunk betáplált parancs-string vége közt akármilyen memóriaszemét is lehetséges, vagy az előző utasításaink maradványai...

E témakörhöz tartozónak érzem a szintaxiskiemelő fájlt is, amit az **mc** editorához készítettem. Ezt itt nem közlöm, de a honlapomról letölthető e fájl épp aktuális verziója, a neve: **mau.syntax**

Erről azt kell tudni, hogy másoljuk be valahova ahová jólesik nekünk, és keressük meg a disztrónkban azt az mc-hez tartozó fájlt, aminek az a neve, hogy:

### **Syntax**

Na most e fájlt szerkesszük úgy, hogy az utolsó 2 sora elé beszúrjuk ami kell nekünk a mau nyelvhez. Ezután a fájl legvége így kell kinézzen tehát valahogy:

```
file ..*\.*\.[mM][aA][uU])$ Mau\sprogramming\slanguage ^((#[mM][aA][uU])|(!.*[mM][aA][uU]))
include /home/vz/MAU/mau.syntax
```

```
file .* unknown
include unknown.syntax
```

Természetesen a

```
include /home/vz/MAU/mau.syntax
```

sor csak az én rendszeremre vonatkozik, azaz te az „include” szó után a **mau.syntax** fájl azon elérési útvonalát írd be, ami neked jó, ami ugye attól függ, te hova mentetted el e szintaxisfájlt.

## **Shell parancs eredményének tömbbe olvasása**

E fejezethez tartozónak érzem az itt ismertetendő funkciót is. A rendszerem építése/hackelése közben nagyon sok esetben fordult elő, hogy olyan szkriptet kellett írnom, ami valami a shell által meghívott parancs eredményét kellett hogy „tovább-feldolgozza”. Ez úgy ment, hogy a **SY** mau paranccsal meghívtam a shellt, az utasítást úgy felparaméterezve hogy az eredményt egy ideiglenes fájlba mentse, majd ezt a fájlt beolvastam a mau programban, s végül amikor már nem kellett, töröltem.

Na most a módszer oké, de ezt mindannyiszor megírni, ahányszor erre szükség van... Ki a fenének van ehhez kedve?! Írtam erre egy megfelelő „full-extra-de-luxe” mau utasítást. Ennek szintaxisa a következő:

```
<< S, t, m
```

ahol az „S” egy stringkifejezés, a futtatandó parancsot tartalmazza (az esetleges paraméterekkel együtt), a „t” a stringtömb nevét meghatározó unsigned char típusú aritmetikai kifejezés, az „m” pedig a munkakönyvtár, ahova az átmenetileg létrehozott ideiglenes fájlt menti el. Az „m” is egy stringkifejezés, természetesen. A parancs megadható így is:

```
<< S, t;
```

ezen esetben kötelező a végére a pontosvessző, ekkor az „m” értéke alapértelmezés szerint: `"/tmp/"`. Természetesen ez az alapértelmezés is mint minden a mau nyelvben, állítható, ez - mint az alapértelmezések - fordítási direktívaként, azaz ha ezen könyvtáron változtatni akarunk, akkor a mau interpreter fordítása előtt írjuk át a nekünk tetsző módon a `vz.h` fájl ezen sorát:

```
#define DEFAULTTMPDIR "/tmp/"
```

Megjegyzendő, hogy az „m” string megadásakor teljesen mindegy, hogy az ott specifikált könyvtár végére odabiggyesztjük-e a záró `"/"` jelet vagy sem. Ha ugyanis nincs ott, majd maga az utasítás odapótolja, elég értelmes ehhez...

Ez az utasítás tehát végrehajtja a shell parancsot amit az S stringkifejezés tartalmaz, s az eredményét elmenti egy ideiglenes állományba, amit az „m” string által megadott könyvtárban helyez el. A fájl neve

### **MauTemporaryFileForShellCommand**

de ezt még kiegészíti a futó processz PID stringjével is, nehogy esetleges másik futó mau program példányokkal ütközés legyen. Ezután e fájlt beolvassa abba a stringtömbbe, aminek a nevét a „t” unsigned char (azaz **#c** típusú) aritmetikai kifejezés határozza meg, majd törli az ideiglenes fájlt. Abban az esetben ha a fájl mérete nulla (mert a lefutott shell parancs nem produkált semmi kimenetet) nem változtatja meg a beolvasás számára specifikált stringtömb tartalmát, azaz nem törli annak régi tartalmát. Amennyiben azonban a fájl mérete nem nulla, akkor törli a régi tartalmat (ha volt egyáltalán korábbi tartalom - ha nem volt akkor értelemszerűen nem töröl), majd oda beolvassa az ideiglenes fájlot, sorról-sorra, a stringtömb minden egyes stringjének a fájl egy sora felel majd meg. A sorok természetesen nullától számoznak.

Az utasítás lefutása után az **ifflag** értéke 1, ha az ideiglenes fájl mérete nulla volt, vagy ha azt valamiért nem tudta megnyitni olvasásra. Ellenkező esetben e flag értéke 0. Valamint, a **?n** rendszerváltozó tartalmazza a beolvasott sorok számát, azaz a stringtömbünket **0** és **?n-1** közt indexelhetjük.

Lássunk erre egy gyors példát: A „moc” zenelejátszótól kérdezzük le az épp játszott zeneszám mindenféle adatait parancssorból:

```
#!mau
#s@p="mocc -i 2>/dev/null";
<< @p, t, "/tmp/";
T "A MOC zenelejátszó nem fut!\n" »$ve;
{| ?n;
?s #s@t[?[n-?|]];
|}
```

```
;/  
$ve ;  
XX
```

Egy futtatás eredménye:

```
vz@Csiszilla /Releases/2014/U/Common/vz/MAU=>mau mocim.old  
State: PAUSE  
File: /Mount/Zene/Zene/Klasszikus/Grigoras_Dinicu_-_Vioara_-  
_Electrecord_remastered_(EDC_745)/08_Ciocarla_(Gr._Dinicu).mp3  
Title: 8 Grigoras Dinicu - Ciocarla (Gr. Dinicu) (Remasterd Records)  
Artist: Grigoras Dinicu  
SongTitle: Ciocarla (Gr. Dinicu)  
Album: Remasterd Records  
TotalTime: 02:50  
TimeLeft: 01:45  
TotalSec: 170  
CurrentTime: 01:05  
CurrentSec: 65  
Bitrate: 320kbps  
AvgBitrate: 320kbps  
Rate: 44kHz
```

Ezen fellelkesülve e programot fejlesszük tovább, hogy egy gombnyomásra kiírja a DWM ablakkezelőnk tetejére az épp futó zeneszám címét! Ehhez persze felhasználunk olyan mau utasítást is amit még nem ismertettünk, de sebjaj, később arról is olvashatunk majd. E program:

```
#!mau  
#s@p="mocp -i 2>/dev/null";  
#s@s="A MOC zenelejátszó nem fut!";  
MC f, "-*-fixed-*-*-*-28-*-*-*-*-";  
<< @p, t;  
E #s@s=[6,]@t[[1]];  
#s@v=?#s "1" @s;  
XX
```

Rémségesen hosszú, ugye?! Ja, ahhoz hogy működjön, nevezzük el mondjuk úgy, hogy „mocim.mau”, és vegyük fel a \$HOME/.xbindkeysrc fájlba e sort vagy valami hasonlót:

```
# (JobbShift + m) A moc-cal épp játszott zene címet kiírja a DWM ablakkezelő tetejére  
"mau /_/P/Mau/-/maubin/mocim.mau"  
Mod3 + m
```

(A megadott gyorsbillentyű persze attól függ, neked épp melyik a szimpatikus, meg hogy épp miféle billentyűzetkiosztást használsz. Én természetesen egy abszolút különlegeset, sajátot, amit magam hekkeltem össze magamnak...)

És még azt meri valaki mondani, hogy a mau nyelv NEHÉZ meg BONYOLULT?! Oldjon meg valaki valami efféle feladatot ennél egyszerűbben, rövidebben... Te, aki e sorokat olvasod, gondold csak bele: ez a program meghív egy shell parancsot. Megvizsgálja, van-e egyáltalán bármiféle eredménye a végrehajtott parancsoknak. (ha nincs, akkor nem fut a MOC...) Az eredményt beolvassa, majd kiválasztja egy bizonyos sorát ezen eredménynek, s ezt a stringet a megfelelőképp csonkolja nekünk. Beállítja, miféle fontkészlettel írjon a GRAFIKUS képernyőre, (!!! GRAFIKUSRA! Parancssoros program! !!!) ki is írja oda a stringet, vár egy billentyűlenyomásra, majd eltünteti onnan azt. ÉS MINDEZT MINDÖSSZE CSAK 8 programsorban! És minden programsor csak 1 utasítást tartalmaz, azaz ez



egyszerűen 8 mau utasítás! (És elég lenne 7 utasítás is, ha a default fontkészlettel iratnánk ki az eredményt a grafikus képernyőre...)

Van egy másik lehetőség is arra, hogy shell parancs eredményét tömbbe olvassuk. Ennek szintaxisa és paraméterezése rém hasonló az előzőéhez:

```
<<< S, t;
```

aholis a paraméterek jelentése azonos az előzőével, tehát az S a végrehajtandó stringkifejezés, a „t” pedig a stringtömb neve. Ez se bántja az eredeti stringtömböt ha az S hossza nulla, vagy ha a shell parancs eredménye semmi. Ellenkező esetben beolvassa soronként az eredményt a stringtömbbe, a **?n** rendszerfüggvény értékét beállítja a sorok számára, amik itt is nullától számozódnak, tehát a „t” tömb 0 és **?n-1** közt lesz indexelhető.

Ha az S hossza 0, nem módosítja az ifflag állapotát. Ellenkező esetben, ha a shellparancs visszatérési értéke „semmi”, az ifflag értéke 1, különben 0 lesz.

A 3 db **<** jel egyetlen token, nem lehet köztük whitespace! A különbség ezen utasítás és az előző közt az, hogy itt nem kell megadnunk semmiféle könyvtárat ideiglenes fájl helyéül, mert nem is hoz létre a lemezen ideiglenes fájlt. A memóriában hoz létre magának ilyet... Emiatt gyorsabb is a futása, méréseim szerint az időszükséglete csak kb 85%-a az előzőének. Ami a visszaadott eredménystringeket illeti, itt is a shellparancs eredményének egy-egy sora lesz a „t” stringtömb egy-egy stringjébe (elemébe) téve, de fontos tudnunk, hogy az előző paranccsal ellentétben ezen sorok nem lesznek lezárva egy sorvége-karakterrel (azaz CHR\$(10) karakterrel Linux alatt), hanem ezek helyén rögvést a stringzáró nullabájt szerepel!

A Moc zenelejátszóval kapcsolatos előző szkriptünk e parancsra átírva tehát így fest:

```
#!mau
#s@p="mocp -i 2>/dev/null";
#s@s="A MOC zenelejátszó nem fut!";
MC f, "-*-fixed-*-*--28-*-*--*-*";
<<< @p, t;
E #s@s=[6,]@t[[1]];
#s@v=?#s "1" @s;
XX
```

## 23. fejezet - A BRAINFUCK interpreter, avagy „ez itt a humor helye”...

Egyes trollfajzatok, kik magukból penetráns, dögletes memetikai bűzölgést eregetnek állandóan mert másra nem képesek, és intellektuális csicskásaik, a „mockosz kisz hobbitkák”, minden bizonnyal belekötnének szívesen a csodálatos mau nyelvbe, mondván hogy ez vajon „Turing-teljes-e”, tudom-e ezt BIZONYÍTANI?! Mert ha nem, akkor esetleg és netántán létezhetnek olyan feladatok, melyekre más programnyelvek alkalmasak, de az enyém nem... Nos, e helyütt leszámolunk e gonoszkodókkal!

Amit ugyanis a fenti bekezdés jelent, hogy a programnyelv alkalmas bármi olyan feladatra amit manapság egy programnyelvtől elvárnak, azt szaknyelven azzal a fogalommal fejezik ki, hogy a programnyelv úgynevezett „Turing-teljes”. Ez a „Turing-teljes” fogalom természetesen megmagyarázható lenne ennél sokkal

tudományosabban is, de ezt mellőzöm, mert e leírásban (is...) kerülöm a felesleges „felHOMÁLYosításokat”... Lényeg az, hogy ha valamely programnyelvre a nagyokosok azt mondják hogy ez „Turing-teljes”, akkor ezzel elismerték, hogy az a nyelv alkalmas mindarra, amire bármelyik eddig létrejött programnyelv, bár természetesen elképzelhető, hogy valami feladatot abban a nyelvben sokkal nehezebben lehet leprogramozni, mint egy másik nyelvben. Azaz, bármire is legyen képes egy tetszőleges mai programnyelv, azt bármelyik másik programnyelv is képes megcsinálni, amennyiben az egy Turing-teljes programnyelv.

Hogyan is lehetne garantálni, hogy a mau nyelv Turing-teljes? Nos, a Turing-teljességnek van egy számunkra (azaz a mau nyelv minden igaz szívű és lánglelkű rajongója számára) módfelett előnyös szabálya, ami így hangzik:

**Minden olyan programnyelv Turing-teljes, ami részhalmazként tartalmaz valamely másik Turing-teljes programnyelvet!**

Ez különben logikus. Gondoljunk csak bele: Egy programnyelv lényegében utasítások egy halmaza. Ha van mondjuk 30 utasításunk, s azzal meg tudunk oldani bármely problémát, akkor ha ezen 30 utasításhoz hozzáveszünk még újabb mondjuk 47-et (vagy akárannyit), nyilvánvaló hogy akkor is meg tudjuk oldani az előző problémákat! Tudniillik ha olyan gondba ütközünk amik nem megoldhatóak az újabb utasításainkkal, akkor megoldjuk azokat legfeljebb a régi 30 utasításunk közül néhányal...

Nekünk tehát most nincs más dolgunk, mint hogy megvalósítsuk programnyelvünkben valamelyik olyan már létező másik programnyelv utasításait, melyről már korábban elismerték a „nagyokosok”, hogy az Turing-teljes. Ha ez sikerül nekünk, akkor teljesen mindegy, ezen felül mi mindent építünk még be a nyelvünkbe: az egyszerismindenkorra Turing-teljesnek fog számítani, efelől vita se lehet!

Nos, ezt megtettem. E nyelv amit „részhalmazként” beleépítettem a mau nyelvbe, a talán lehető legegyszerűbb Turing-teljes nyelv, az úgynevezett „brainfuck” nyelv. Ténylegesen ez a neve, ami magyarul talán „agybaszónak” fordítandó, már elnézést a csúnya szóért... És nem véletlenül ez a neve, ugyanis irtózatosan nehéz benne programokat írni! Ennek ellenére azonban Turing-teljes. És bár e nyelven egy programot megírni tényleg nehéz, de a programnyelv interpreterének a megalkotása nagyonis könnyű... Épp emiatt választottam!

A nyelv mindössze csak 8 utasítást tartalmaz ugyanis... A nyelvet *Urban Müller* készítette, azzal a céllal, hogy olyan Turing-nyelvet hozzon létre, amire a lehető legkisebb fordítóprogramot tudja megírni. A mau nyelv azonban ezt interpreterként valósítja meg, természetesen.

A brainfuck nyelvről itt található részletes Wikipédia-szócikk:

<https://hu.wikipedia.org/wiki/Brainfuck>

A *Brainfuck* nyelvnek egy univerzális byte-mutatója van, aminek a neve „pointer”, és ami szabadon mozoghat az adatmemóriában, mely legalább 30000 byte nagyságú tömb illik hogy legyen, és amelynek alapértékei nullák. A pointer a

tömb elején indul. Nem tudom, miért van ez a fétisizmus a brainfuck programozók körében, hogy a memóriatömb értéke éppen 30000 bájt nagyságú legyen - még ha legalább 32768 lenne a mérete, megérteném, mert az „kerek” szám kettes számrendszerben írva. De 30000?!

Na mindegy.

A nyelv nyolc parancsát egy-egy karakter reprezentálja:

- > A pointer növelése eggyel
- < A pointer csökkentése eggyel
- + A pointernél levő byte növelése eggyel
- A pointernél levő byte csökkentése eggyel
- . A pointernél levő byte kiírása
- , Byte bekérése és a pointernél tárolása
- [ Ugrás a következő, megfelelő ] jel utánig, ha a pointer alatti byte nulla.
- ] Ugrás az előző, megfelelő [ jelig.

Ez igazán egyszerűnek tűnik.

A baj csak az, hogy e 8 utasítás 8 karaktere nagyonis olyan, amilyeneket máshol is használ a mau nyelv! De van megoldás. Van a mau nyelv kódjában egy F nevű struktúra, mely mindenféle fontos adatokat tárol. Felvettem belé, e struktúrába a brainfuck flaget:

```
USC BRAINFUCKflag; // Ha 1: A "+ - > < . , [ ]" karaktereket brainfuck módban értelmezi, ha 0: másképp.
```

E flag kezdetben 0 értékű, erre állítja be a konstruktorunk. (Azaz, a mau program indulásakor ez nulla). Kell tehát nekünk megfelelő utasítás e flag 1-re állításához (bekapcsolásához) és nullára állításához, azaz a kikapcsolására. E két utasítás a mau nyelvben a

/+

és a

/-

utasítás. Természetesen a mau nyelv tartalmazza a megfelelő BRAINFUCK utasításokat is a fenti táblázat minden karakterére, de eszerint azokat csak akkor hajtja végre, ha a BRAINFUCK flaget bekapcsoltuk előbb a /+ utasítással.

Ezek után, ha efféle brainfuck progival találkozunk az Internet setét virtuális sikátoraiban, azt a következőképp adaptálhatjuk a mau nyelvünkhöz:

1. Kitörölünk belőle minden karaktert, ami nem a 8 brainfuck utasítás, azaz nem a .,<>+-[] karakterek valamelyike, vagy pedig megjegyzéssé fokozzuk le őket a // karakterpáros eléjük tevésével. (A sortörések, szóközők, tabulátorok maradhatnak).

2. Beszúrjuk az elejére:

[#c@0=30000]

Ez foglalja le a szükséges memóriaterületet, még hozzá a nullás sorszámu tömb unsigned char típusú értékeire. A 30000 helyett megadható más szám is, amennyi elég az aktuális brainfuck proginak, de a **#c@0** nem lehet más, ez „fixen behuzalozott” a brainfuck rutinokba. (Nem tartom e funkciót ugyanis olyan fontosnak, hogy azon görcsöljek, miként érhetek el e téren nagyobb rugalmasságot...).

3. Beszúrjuk az elejére:

```
/*  
Ez kapcsolja be a brainfuck-értelmezést.
```

4. Szükség esetén, ha a program futása után a prompt rossz helyen jelenne meg, a legvégére szúrjunk be egy

```
/*;  
jelet, hogy kiírjon egy üres sort még.
```

Most már boldogok lehetünk, a nyelvünk részhalmazként tartalmazza a **brainfuck** nyelvet, ami Turing-teljes, TEHÁT ezek után a mi mau nyelvünk is Turing-teljes, efelől nem lehet vita, s ez azt jelenti, hogy minden szaktekintély által elismerten alkalmas a nyelvünk BÁRMI feladat megoldására, ami bármi más programnyelvvél is megoldható egyáltalán!

Példákat BRAINFUCK progikra nem közlök itt e leírásban, de a mau forráskód mellé becsomagolok párat. Azokat azonban nem én írtam, hanem máshonnan vadászgattam őket össze az Internetről, csak a fenti leírás szerint „adaptáltam” őket a mau nyelvhez. Akinek tehát van mau interpretere, az ezentúl nem kell kétségbeessen, ha talál valami efféle titokzatos stringet egy aláírásban:

```
++++[>+++++<-]>[<+++++>-]+<+>[>+>+<-]>+>>[<<+>>-]>>>[-]>+>[-]  
+>>>+[-]>+++++>>>]<<<[<+++++>+>>-]+<.<[>----  
<-]<[<<[>>>>]>>>[-]>+++++>+>+>[>-<-]>+++++COMMENT+COMMENT+>>[-<-  
>-]>+<<<]<[>+<-]>]<<-]<<-]
```

Ezt a hup.hu portálról vadásztam le, e link alól:

<http://hup.hu/node/116498>

Ebből ez lett a „mau-képessé” átalakítás után:

```
#!mau  
[#c@0=30000]  
/*  
  
++++[>+++++<-]>[<+++++>-]+<+>[>+>+<-]>+>>[<<+>>-]>>>[-]>+>[-]>+>>+[-]>+++++>>>]  
<<<[<+++++>+>>-]>+<.<[>----<-]<[<<[>>>>]>>>[-]>+++++>+>+>[>-<-]  
+++++  
// COMMENT  
+  
// COMMENT  
+++++[-<->-]>+<<<]<[>+<-]>]<<-]<<-]  
  
/*;
```

Amint látom, a 0-10000 tartományba eső négyzetszámokat írja ki.

## 24. fejezet - Hasznos feladatokat ellátó mau nyelvű példaprogramok

E fejezetben végül olyan mau nyelvű programokat közlök, amiket mind én írtam, mau nyelven, s úgy vélem hasznosak is már valamely konkrét feladatra. Nekem legalábbis hasznosak, de jóeséllyel másoknak is. E csoportba tartozik tulajdonképpen a korábban bemutatott kalkulátorprogram is, szerintem, mert én azt is hasznosnak tartom, de mert azt már bemutattam, itt most nem közlöm újra. Néhány itt bemutatott programban előfordulhatnak olyan funkciók, utasítások, függvények, melyek e doksi eddigi fejezeteiben még nem kerültek bemutatásra. Tehát:

### A „maudir” program

Létezik a linuxos világban egy „vidir” nevű kis program, bár nem mindenki ismeri. Ez annyit csinál, hogy elindítása után a megadott nevű tartalomjegyzéket beolvassa (illetve ha nincs megadva tartalomjegyzék, akkor az „aktuális”-at), s ebből a könyvtár- és fájlneveket kiírja egy ideiglenes fájlba, úgy, hogy mindegyiknek az elejéhez odabiggyeszt egy sorszámot (a sorszám és a név közt természetesen van valamennyi whitespace). Miután e fájlt előállította, meghívja rá a \$EDITOR változóban beállított szövegszerkesztő programot, ha ilyen nincs beállítva akkor a „vi” nevű programot próbálja meghívni. Ezzel aztán szerkeszthetjük e fájlt, amennyiben a benne levő fájl- és könyvtárneveket átírhatjuk úgy, ahogy az nekünk tetszik, csak arra kell vigyáznunk hogy a sorok elején a sorszámokat ne piszkáljuk. Ellenben az szabad, hogy egész sorokat kitöröljünk.

Amikor ezzel megvagyunk s elmentettük a fájlt, a vidir program azt beolvassa, elemzi, s ahol úgy találja hogy az adott sorszámú sorban már nem az a könyvtár- vagy állománynév áll amire ő emlékszik, akkor azt átnevezi a megfelelőképpen, az új névre. Ahol meg nincs is már meg az adott sorszámú sor, azt a könyvtárat vagy fájlt egyszerűen törli.

Ez nagyon kényelmes a fájloknak főleg az átnevezésére, ha egyszerre sok efféle műveletet akarunk végrehajtani, s pláne ha a fájlnev hosszú, s esetleg teli is van mindenféle ékezetes karakterekkel. Az alábbiakban e program mau nyelvű megvalósítását közlöm. Ez azonban nem rendelkezik editor-alapértelmezéssel, ha nincs beállítva a rendszerben a \$EDITOR változó, akkor bizony nem fog szerkeszteni nekünk... Ezt nem lenne nehéz beleraknom, hogy legyen alapértelmezése, de nem érzem fontosnak.

Íme a program:

```
#!/mau // maudir program.
// Készítette: Viola Zoltán, violazoli@gmail.com
// A program a "mau" nevű programozási nyelven íródott, azt is én Viola Zoltán készítettem.
// Mindegyik GPL licenc alatt van.

// Konstansok meghatározása

#c@0=c;
```

```

#s@P="/tmp"; // Annak a könyvtárnak a PATH-ja, ahova az elkészült ideiglenes fájl kerül, az,
// amit a $EDITOR változóban meghatározott texteditorral szerkesztünk majd.
#c@c='-'; // A pidstring üres helyeit kitöltő karakter
#c@p=6; // A pidstring maximális hossza
#s@i="MAUDIR_temporary_file"; // Az ideiglenes file alapneve a pidstring nélkül
#s@k=".tmp"; // Az ideiglenes file kiterjesztése

#s@Z="DRL"; // Ezekkel a típusú tartalomjegyzék-bejegyzésekkel foglalkozunk a maudir programban.
És ebben a sorrendben
// fognak megjelenni a szerkesztendő ideiglenes fájlban felülről lefelé:
// D azaz Directoryk,
// R azaz Reguláris (közönséges) fájlok
// L azaz Linkek.

// Előkészítő műveletek
#s@p=?#s "PID" @p @c; // A pidstring előállítása
#s@n=(@i)+"_PID-"+(@p)+(@k); // A leendő ideiglenes file nevének előállítása

// A leendő ideiglenes fájl teljes útvonalat tartalmazó nevének előállítása:
if((#s@P[(!#s@P)-1])!= '/') T #s@N=(@P)+"/"+ (@n);
E #s@N=(@P)+(@n);
// A fenti két sor azt csinálja, hogy ellenőrizzük, e progi legelején az ideiglenes fájl
helyének megjelölt könyvtár útvonala
// egy "/" jellel végződik-e. Ha ez nincs így, pótoljuk e per-jelet.

if (?a)<3 T #s@A="."; // Ha nem adtunk meg parancssori paramétert, az aktuális
könyvtárra vonatkozik a hívás,
E #s@A=?#s "ARGV" 2; // különben a megadott tartalomjegyzékre

// A tartalomjegyzék beolvasása

#T@t= @A; // A lekérdezendő tartalomjegyzéket jelölő stringet a #s@A tartalmazza. Itt olvassuk
be a directoryt.

// A szerkesztendő ideiglenes fájl megnyitása írásra
#K@o=@N; // Megnyitjuk az output file-ot. Ennek neve természetesen a korábban meghatározott
string lesz,
// ami a PID értékét is tartalmazza. A @N változó e nevet tartalmazza, az elején kiegészítve a
könyvtárát tartalmazó
// elérési útvonallal.

if #K@o E "Az output file nem megnyitható!\n" XX // Hiba esetén kilépünk.
// Figyeljük meg, a fenti if utasítás nem is tartalmaz THEN ágat... Így egyszerűbb, nem
bajlódunk az összehasonlító
// művelet eredményének negálásával.

if ((#s@A[(!#s@A)-1])!= '/') T #s@A=(@A)+"/"; // Itt ellenőrizzük, e progi legelején a beolvasandó
tartalomjegyzék útvonalának
// vége egy "/" jel-e. Ha ez nincs így, pótoljuk e per-jelet. Ha netán nem lett volna ott, az
nem baj a tartalomjegyzék
// beolvasásakor még, mert ezt akkor pótolja az interpreter beolvasórutinja maga is nagy
előzékenyen. De most ezt nekünk is
// meg kell csinálni.

{! #s@Z; if (#l (?#l "T#" t, ?#c "{!}")>0) T [#(@0)@(?#c "{!}")=?#l "T#" t, ?#c "{!}"];
!} // E ciklusban foglalunk le memóriát annyi darab unsigned char értéknek a megfelelő nevű
tömbbe,
// ahány tartalomjegyzéki bejegyzés van abból a típusból. Ez egy fix számszor lefutó ciklus,
annyiszor fut le,
// amennyi a string hossza. A ?#c "{!}" adja vissza a string aktuális karakterét.

{! #s@Z; („Kiíró rutin” ?#c "{!}"); !} // A kiíró függvény meghívása ciklusban.
// Ez egy fix számszor lefutó ciklus, annyiszor fut le amennyi a string hossza.
// A ?#c "{!}" adja vissza a string aktuális karakterét.

[#K@o]; // Lezárom az output file-ot

SY (?#s "ENV" "EDITOR") + " " + @N; // Megnyitjuk a fájlt a $EDITOR környezeti változó által
meghatározott text editorral

```

```

#B@b=@N; // Megnyitjuk a szerkesztett fájlt input fájlként.
if #B@b E "Az ideiglenes fájl nem megnyitható beolvasásra!\n" XX

{( // beolvasóciklus kezdete. Végtelenciklus.
#s@s=?#s "NL", b, 1000; // Beolvassuk a sort, de max. 1000 karaktert
if(!#s@s)==0 TT »$tt; // Ha üres sort olvastunk be, vége a fájlnek, ekkor elugrunk a
törlésellenőrzési részhez
// Itt kezdődik a sor feldolgozása.
#c@t=#s@s[0]; // A file típusa
#s@h=[2,11]@s; // A file sorszáma stringként
#l@h=#s ?#s "SP" @h; // Átalakítjuk a file sorszámát stringből unsigned int számmá. Ehhez előbb
le kell vágni
// a string elejéről a szóközöket. Ezt végzi el az iménti "SP" függvény: levág minden whitespace
karaktert a string elejéről.
#s@a=[16,]@s; // A file neve
//#s@a=(#s@a) - 1; // A hosszából le kell vonnunk 1-et, mert másképp a file nevéhez tartozónak
tartaná a 10-es karakterkódú
// sorvégjelet is.
#s--@a; // A hosszából le kell vonnunk 1-et, mert másképp a file nevéhez tartozónak tartaná a 10-
es karakterkódú
// sorvégjelet is.

?? @t @Z; E. "Ismeretlen bejegyzéstípus az ideiglenes fájlban!\n"
E. "Esetleg kitörölted vagy megváltoztattad a sor első karakterét?\n"
E. "A hibás típusazonosító karakterkódja: " ?c @t; " Maga a karakter: " ? @t; /; XX;

#c@(@t)[#l@h]=1; // Ebben a sorban azt vizsgáljuk, milyen típusú fájlról szól a beolvasott sor.
// Amilyen típusú, az ahhoz tartozó tömb megfelelő sorszámú elemét 1-re állítjuk. Ez amiatt
// fontos, mert így tudjuk követni, nincs-e kitörölve a fájlból az adott sorszámú fájl sora.
// Ha ugyanis azt kitörölte a Felhasználó, majd törölni kell azt a fájlt vagy könyvtárat is!

if (#s (?#s "Tn" t, #l@h, (@t)) != (#s@a)) T RN (@A) + (?#s "Tn" t, #l@h, (@t)), (@A) + (@a);
// Fentebb azt csináltuk, hogy ellenőrizzük, a megfelelő típusú és sorszámú fájl neve a
beolvasott tartalomjegyzék-struktúrában
// azonos-e azzal, amit épp beolvastunk az ideiglenes fájl sorából. Ha NEM azonos vele,
átnevezzük.
}) // Visszaugrás a beolvasóciklus elejére

$tt

[#B@b]; // Bezárjuk az ideiglenes fájlt.
SY "rm " + (@N); // Töröljük az ideiglenes fájlt a shell segítségével

{! #s@Z; („Törlő rutin”” ?#c "{!}"); !} // A törlő függvény meghívása ciklusban

XX // A program vége

// ===== Függvények

„Törlő rutin” // Törli a megfelelő fájlbejegyzést
// Input paraméterek:
#c@t // típus: D: directory, R: reguláris fájl, L: link

if (#l (?#l "T#" ^ t @t)==0) T xx; // Ha nincs a megfelelő típusú fájlból, vége a rutinnak.

if (@t)==D T #l@B=$rd;
E #l@B=$rm;

{| ?#l "T#" ^ t @t ; // Fix számszor lefutó ciklus, annyiszor fut le ahány fájlbejegyzésünk van.
if #c@^(@t)[?-] E G #l@B // Meghívjuk a tartalomjegyzék-bejegyzés típusától függő szubrutint
|}

xx

$rm // fájl-törlő szubrutin
RM (@^A) + (?#s "Tn" ^ t, ?-, @t)
«

$rd // könyvtárat rekurzívan törlő szubrutin

```



```

RD (@^A) + (?#s "Tn" ^ t, ?-, @t)
«

„Kiíró rutin” // Kiírja a megfelelő sort a fájlba
// Input paraméterek:
#c@t // típus: D: directory, R: reguláris fájl, L: link

#s@y=" :"; #s@y[0]=@t;
#l@i=0; // Ciklusszámláló kezdeti értéke
if (#l (?#l "T#" ^ t @t)==0) T xx; // Ha nincs megfelelő típusú fájl, nem ír ki azokról semmi
adatot. Nem hülye ő...
{| #?l "T#" ^ t @t; // Ciklus indul, annyiszor fut le, ahány fájl van az adott típusból.
Legalább egy egészen biztosan akad.
<s @^o (@y)+(#l@i)+ = "+(?#s "Tn" ^ t, #l@i, @t)+"\n"; // A sor kiírása
// A fenti 3 sorban a ^ jel azt mutatja, hogy a t nevű speciális tartalomjegyzék-
objektumot(változót) a szülő-névtérből
// veszi (ott keresi) és nem a függvény saját névtérében található változók közül.
#l++@i; // Ciklusszámláló növelése
|} // A ciklus vége
xx

```

## File sorait névsorba rendező program

Az alábbi program egy parancssori paraméterként megadott fájl sorait névsorba rendezi, majd azt egy új nevű fájlba kiírja. Az eredeti fájlt nem módosítja.

Ez az első, legprimitívebb változat. Ezután megadom a fejlettebb változatot is, bár az már be lett mutatva az output fájlokról szóló fejezetben:

```

#!mau // Egy file sorainak névsorba rendezése
// rendez.mau
if (?a)<3 T "Nem adtad meg a rendezendő állomány nevét!\n" XX;
E #s@A=?#s "ARGV" 2; // A rendezendő file neve

<? @A; // Megszámoljuk a sorait
"A file sorainak száma: " ?l ?n ; /;
"A file leghosszabb sora " ?l ?!; " bájttal hosszú.\n";

#B@b=@A; // Megnyitjuk az input fájlt
if #B@b E "Az input file nem megnyitható!\n" XX

[@t[[?n]]]; // lefoglalok területet annyi string részére, ahány sor van a fájlban, a t nevű
stringtömbbe
{| ?n; // A ciklus annyiszor fut le, ahány sor van a fájlban
#s@t[[?n-?]]=?#s "NL", b, ?!; // Beolvassuk a sort, de max. annyi karaktert, amennyi a
leghosszabb sor a fájlban.
|} // vége a ciklusnak
[#B@b]; // bezárjuk az input fájlt
#t@a; // inicializálunk egy időváltozót
QS t,?n; // névsorba rendezzük a tömböt
#t@b; // inicializálunk egy másik időváltozót
#g@e=(#t@b)-(#t@a);
"A rendezés ennyi milliomod másodpercig tartott: " ?g @e; /;
#K@o=(@A)+".rendezett"; // Megnyitjuk írásra az output fájlt
if #K@o E "Az output file nem megnyitható!\n" XX
{| ?n; // A ciklus annyiszor fut le, ahány sor van a fájlban
<s @o, @t[[?n-?]]]; // Kiírjuk a sort a fájlba
|} // vége a ciklusnak
[#K@o]; // Lezárom az output fájlt

XX // vége a programnak

```

A fejlettebb változat:

```
#!/mau // Egy file sorainak névsorba rendezése
// rendez.mau
if (?a)<3 T "Nem adtad meg a rendezendő állomány nevét!\n" XX;
    E #s@A=?#s "ARGV" 2; // A rendezendő file neve

BE @A, t; // Beolvasom a fájlt a „t” nevű stringtömbbe
T "Az input file nulla méretű!\n" XX
"A file sorainak száma: " ?l ?n ; /;
"A file leghosszabb sora " ?l ?!; " bájttal hosszú.\n";

#t@a; // inicializálunk egy időváltozót
QS t,?n; // névsorba rendezzük a tömböt
#t@b; // inicializálunk egy másik időváltozót
#g@e=(#t@b)-(#t@a);
"A rendezés ennyi milliionod másodpercig tartott: " ?g @e; /;

KI (@A)+".rendezett", t; // Kiírjuk a „t” tömböt a fájlba
XX // vége a programnak
```

## Állománylistázó program, olyasféle mint az "ls -l"

(ráadásul ez színesben listáz...):

```
#!/mau

// Konstansok meghatározása

#c@c:15=0; // Ha ez nem 0, csak akkor listázza az ext2 specifikus flagek attributumstringjét

if (?a)<3 T #s@A="."; // Ha nem adtunk meg parancssori paramétert, az aktuális könyvtárat listázza
E #s@A=?#s "ARGV" 2; // különben a megadott tartalomjegyzéket
if (?a)>=4 T #c@c:14=#s(?#s "ARGV" 3)[0]; if((@c:14)=e) T #c@c:15=1;
// A felső sor jelentése: Ha van megadva parancssori paraméter hogy melyik könyvtárat kell listázni, és
// ezután van megadva még valami másik paraméter is, akkor leellenőrzi, hogy ezen következő
// paraméter első karaktere egy „e” betű-e. Ha igen, beállítja a flaget arra, hogy ki kell listázza
// az ext2 specifikus attributumokat is.

#c@c:0=30 // A neveket ennyi hosszúságú stringbe igazítja balra
#c@c:1=12 // A fájlméretet jobbra igazítva írja ki, ennyi hosszú mezőbe. Ha nem fér ki, csonkol...
#c@c:2=10 // A csoport és tulajdonos nevét ennyi karakter hosszúságú mezőbe balra igazítva írja ki
#s@c=" ==> " // Ez a string utal arra, melyik fájlra mutat a symlink
#s@c=" --> " // Ez a string utal arra, melyik fájlra mutat a symlink, az esetben, ha a link „törött”
#s@K=(?#s "SZIN" c)+": "; // Ez határozza meg a csoportstringet és tulajdonos-stringet elválasztó karaktert
#s@G= ?#s "SZIN" b; // Ez határozza meg a csoportstring és tulajdonos-string színét
#s@o= ?#s "SZIN" g; // Az oktális jogosultságok színe
#s@m=(?#s "SZIN" g)+(?#s "SZIN" v); // állományméret színe
#s@R=(?#s "SZIN" y); // A közönséges állományok színe
#s@x=(#s@R)+(?#s "SZIN" v)+(?#s "SZIN" u); // végrehajtható állományok színe
#s@l=(?#s "SZIN" m); // A symlinkek színe
#s@L=(#s@l)+(?#s "SZIN" v); // Törött symlinkek színe

// Előkészítő műveletek
#c@c:1=21-#c@c:1
#c@c:2=10+#c@c:2 // Ez amiatt kell, mert a karakterszám méretet meg kell növelni 2*5 bájtal, a kettőspont elé és mögé
```

```

// beszűrt színstringek bájtmérete miatt.

"Tartalomjegyzék:\n"
#T@t= @A; // A lekérdezendő tartalomjegyzéket jelölő string. Itt olvassuk be a directoryt.

// A directoryk listázása

#l@i=0;
if (#l (?#l "T#" t D)==0) T »$DV; // Ha nincs altartalomjegyzék, nem listáz
$S; $Bw; // Directory színek fehér háttéren vörös
"Altartalomjegyzékek száma: " ?l ?#l "T#" t D; $Bb; $Sd; $Sw; /;
{| ?#l "T#" t D;
$Sw; // Directory színek fehér
?s ?#s "SB" (?#s "Tn" t, #l@i, D) #c@c:0; // A dir neve
?s @o; " \" ?s ?#s "JOG" t, #l@i, D; "\" " // Az altartalomjegyzék jogai
if (@c:15) T ?s @o; ?s ?#s "A" t, #l@i, D; " " // A directory ext2 specifikus attributumai
?s (#s@G) + ?#s "SB" ((?#s "Tg" t, #l@i, D) + (#s@K) + (#s@G) + (?#s "Tt" t, #l@i, D)) #c@c:2; //
A dir csoportjának és tulajának neve
" "; $Sc; $Sv;
(,,"Dátumstring tömörítés" ?s(?#s "UM" t,#l@i,D) \ #s@d); ?s @d; " ";
//(,,"Dátumstring tömörítés" ?s(?#s "UV" t,#l@i,D) \ #s@d); ?s @d; " ";
(,,"Dátumstring tömörítés" ?s(?#s "UE" t,#l@i,D) \ #s@d); ?s @d;
/;
#l++@i;
|}
$Sd; // Directory szín vége
$SDV // Directoryk vége

// A közönséges fájlok listázása

#l@i=0;
if (#l (?#l "T#" t R)==0) T »$SRV; // Ha nincs közönséges fájl, nem listáz
$Sg; $Bw; // Fehér alapon zöld
"Közönséges fájlok száma: " ?l ?#l "T#" t R; $Bb; $Sd; /;
{| ?#l "T#" t R;
$Sd;
?s @m; // fileméret szín beállítása
//?s [#c@c:1,] ?#g "FM" t,#l@i,R; // A fileméret
?s ?#s "S" ([#c@c:1,] ?#g "FM" t,#l@i,R),4,14,12,11; // A fileméret, különböző színekkel
// kiírva a bájtokat, Kbyteokat, Mbyteokat és gigabájtokat
$Sd; " "
if (?#c "EXE" t, #l@i, R) T ?s @x; // Ha végrehajtható állomány, ez lesz a színe
E ?s @R; // Ha nem végrehajtható, akkor pedig ez
?s ?#s "SB" (?#s "Tn" t, #l@i, R) #c@c:0; // A file neve
$Sd;
?s @o; " \" ?s ?#s "JOG" t, #l@i, R; "\" " // A file jogai
if (@c:15) T ?s @o; ?s ?#s "A" t, #l@i, R; " " // A file ext2 specifikus attributumai

?s (#s@G) + ?#s "SB" ((?#s "Tg" t, #l@i, R) + (#s@K) + (#s@G) + (?#s "Tt" t, #l@i, R)) #c@c:2; //
A file csoportjának
// és tulajának neve
" "; $Sc; $Sv;
(,,"Dátumstring tömörítés" ?s(?#s "UM" t,#l@i,R) \ #s@d); ?s @d; " ";
//(,,"Dátumstring tömörítés" ?s(?#s "UV" t,#l@i,R) \ #s@d); ?s @d; " ";
(,,"Dátumstring tömörítés" ?s(?#s "UE" t,#l@i,R) \ #s@d); ?s @d;
/;
#l++@i;
|}
$SRV // Közönséges fájlok vége

// Szimbolikus linkek listázása

$Sd; #l@i=0;
if (#l (?#l "T#" t L)==0) T »$SLV; // Ha nincs symlink, nem listáz
?s @l; // A symlinkek színe
"Linkek száma: " ?l ?#l "T#" t L; /;
{| ?#l "T#" t L;
$Sd; ?s @l; // A symlinkek színe
?s ?#s "SB" (?#s "Tn" t, #l@i, L) #c@c:0; $Sd;
if ((?#c "L?" t, #l@i, L)==2) T $Sd; ?s @L; ?s @C;
E $Sd; ?s @l; ?s @c;

```

```

?s ?#s "LINK" t, #l@i; /;
#l++@i;
|}
Sd;
$LV // Symlinkek vége

// Összesítés

"Összes bejegyzés száma: " ?l ?#l "T#" t m; /;

XX

„Dátumstring tömörítés” // Csonkolja a dátumstringet
#s@d;
#s@s=[,13]@d; #s@s[10]='_'; #s@s[11]=#s@d[20]; #s@s[12]=' ' ;
if(#s@s[11]==S) T if(#s@d[22]==o) T #s@s[11]=o;
#s@s+=[11,5]@d;
#s@s=[2,]@s // levágjuk az évszám első 2 karakterét
\ #s@s
xx

```

A fenti mau programokon különben minden bizonnyal lehet még csicsázni, úgy is hogy többet tudjanak, úgy is hogy mindenfelét ellenőrizgessenek ha valaki hülye adatot adna meg nekik, vagy más hiba történe, s az sem kizárt, hogy egyszerűbben is meg lehet őket oldani, pár mau parancs hatékonyabb használatával. Nem vitás ugyan, hogy jelenleg én tudok az egész Világmindenségben a legjobban programozni mau nyelven, de azért ennek ellenére az is igaz, hogy még magam is kezdő vagyok e nyelvben, hiszen fél éve sincs, hogy használok...

## Szótárprogi

Készítettem egy primitív kis szótárprogit is. Angol-magyar szótár, jelenleg töltöm fel adatokkal. Egy kifejezés - egy sor. A szintaxisa egyszerű: egy sor = egy kifejezés és annak a jelentése. Itt egy példa rá:

```

a ::5    egy ... (határozatlan névelő)
and::4   és
be ::2    van; létezik
to::7    -ni;<br>azért, hogy ...<br>irányába

```

A szótár sorának szintaxisa: elől van az angol szó vagy kifejezés, utána egy TAB, aztán a magyar jelentés(ek). Az angol szó és a TAB közt opcionálisan állhat két db kettőspont, és utána egy decimális szám, ami meghatározza, az angol szó kb hányadik gyakorisági sorrendben egy átlagosnak tekinthető angol szövegben. Bár minden szónak és jelentésének egyetlen sorban kell lenni, de mert egy angol szóhoz több magyar jelentés is társulhat, ezek kiíratását szabályozhatjuk azzal, hogy sortörést illesztünk be a magyar részbe, a sortörés jele ugyanúgy a **<br>** szimbólum, mint a html állományok esetében.

Na most e szótárállomány használatára írtam korábban egy C nyelvű programot, ami annyit csinál, hogy vár míg beütök egy angol vagy magyar szót, majd kilistázza mindazon sorokat a szótárból, az esetleges sortörésjeleket figyelembe véve, amikben szerepel a betáplált szó. E kilistázott sorokat a „dmenu” nevű proginak továbbítja, ami az általam használt DWM ablakkezelő hasznos kiegészítője, ez megjeleníti e sorokat a képernyőn, s a kurzorral lehet mozogni köztük fel-le, s a kiválasztás az Enter-rel lehetséges. Na most a progim, miután e kiválasztás megvolt, leellenőrzi, a kiválasztott angol szóhoz létezik-e hangállomány egy

speciális könyvtárban, s ha igen, az ogg123 progi segítségével azt le is játssza, azaz ez tulajdonképpen egy „hangos szótár” is.

E C nyelven írt programnak most elkészítettem a mau változatát is. Érdekessé-  
képpen bemutatom mindkettőt:

A C nyelvű program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXBETU 1024
#define SPACE 32
char keresem[MAXBETU];
const char *catsor="cat /_P/Szotar/0/angol/angolmagyar.szotar | grep \\";
const char *dmenusor="\\" | sed -e 's/ *:::/ :/' -e 's/\\t/\\n -> /g' -e 's/<BR>/\\n -> /g' -e 's/<br>/\\n -> /g'
| dmenu -l 36 -sf yellow -nf yellow -sb "\\#d2001e\\" -fn \\"*-fixed-*-*-*-*18-*-*-*-*18-*-*-*-*\\";

const char *sed2=" | sed -e 's/ *:::[0-9]*//' -e 's/ /_g'";
char parancs[1024];
int main(int argc, char **argv) {
    int keresemhossz;
    printf("Írd be a keresett szót: ");
    fgets(keresem,MAXBETU,stdin);
    keresemhossz=strlen(keresem);
    keresem[keresemhossz-1]=0; // A sorvégjel miatt!
    sprintf(parancs,"zeneneve=%s%s%s"; zeneneveogg=\\/_P/Szotar/0/angol/felnottno/\\$zeneneve\\.ogg\\"; "
    "if [ -f $zeneneveogg ] ; then ogg123 -q $zeneneveogg ; fi";, catsor,keresem,dmenusor,sed2);
    system(parancs);
    exit(EXIT_SUCCESS);
}
```

A mau nyelvű program:

```
#!/mau

#i@M=1024; // MAXBETU
#s@k[#i@M]=0; // keresem-stringváltozó max. méretének beállítása.
#s@c="cat /_P/Szotar/0/angol/angolmagyar.szotar | grep \\"; // catsor
#s@d="\\" | sed -e 's/ *:::/ :/' -e 's/\\t/\\n -> /g' -e 's/<BR>/\\n -> /g' -e 's/<br>/\\n -> /g'
-> /g' | dmenu -l 36 -sf yellow -nf yellow -sb "\\#d2001e\\" -fn \\"*-fixed-*-*-*-*18-*-*-*-*18-*-*-*-*\\";
*-**\\";
// fent: dmenusor
#s@e=" | sed -e 's/ *:::[0-9]*//' -e 's/ /_g'"; // sed2
#i@$ff:7=0; // A beolvasandó bájtok kezdőindexe
$be; // A beolvasóciklus eleje
#c@$ff=?#c "("; // beolvasunk 1 bájtot a standard inputról
if(@$ff)==10 T »$ki // Ha Entert ütöttünk le, vége a beolvasásnak, ugrás a végrehajtási részhez
#s@k[#i@$ff:7]=@$ff; // elmentjük a beolvasott bájtot
#i++@$ff:7; // Az index inkrementálása
ha(#i(@$ff:7)>=(@M)) "Túl hosszú a bevitt szó/kifejezés!\\n"; XX;
»$be // ugrás a beolvasóciklus elejére
$ki // vége a beolvasásnak
#s@k[#i@$ff:7]=0; // Lezártuk a string végét 0-val
#s!@k; // rendbehozzuk a string hosszát

#s@p="zeneneve=`"+(@c)+(@k)+(@d)+(@e)+"`;
zeneneveogg=\\/_P/Szotar/0/angol/felnottno/\\$zeneneve\\.ogg\\"; "+
    "if [ -f $zeneneveogg ] ; then ogg123 -q $zeneneveogg ; fi";
SY @p;

XX
```

A fenti proginak a dict.mau nevet adtam. Készítettem azonban belőle egy másik változatot is, aminek a neve dict\_clipboard.mau, s nem véletlenül! Ez ugyanis ugyanazt csinálja mint a fenti progi, épp csak az adatot nem a standard inputról várja, hanem a vágólapról! Azaz: ez olyasféléképp működik, mint a Windows rend-  
szerek alatt ismert, s meglehetősen híres MobiMouse program! Tehát kijelölünk

egy szót a böngészőben vagy a LibreOffice-ban, vagy akár a virtuális terminálban, majd el kell indítani e mau programot, s az megjeleníti (a dmenu segítségével) a szóhoz tartozó találatokat, azok közt mozoghatunk a kurzorral, majd Enter, s ekkor ha van hozzá hangállomány, ki is mondja azt. A program elindítása természetesen egy gomb lenyomásával történik, amit az XBindKeys progi figyel nálam. Ennek érdekében ezt vettem bele a \$HOME/.xbindkeysrc fájlba:

```
"mau /_/P/Mau/-/maubin/dict_clipboard.mau"  
Mod3 + k
```

Ez nálam a jobbshift+k lenyomását jelenti, mert átmappeltem a jobboldali shift gombot egy mod3 nevű módosítóbilleentyűvé.

Maga a mau program pedig így néz ki:

```
#!/mau  
// ----- Pluginkönyvtár betöltése  
#s@L="/_/P/Mau/-/lib/maulib_string.so"; // A mau stringplugin-könyvtár  
#s@f="mau_strstr"; // A függvény neve a pluginkönyvtárból  
#M@U=@L; // objectlib megnyitása  
ha(?d) "A dinamikus könyvtár nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX  
[[ 'j,u,@f ]]; // Betöltjük a kereső rutint a 'j' karakterre  
ha(?d) "A függvény nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX  
// ----- Pluginkönyvtár betöltésének vége  
  
// ----- Konstansok beállítása  
#s@S="/_/P/Szotar/0/angol/angolmagyar.szotar"; // Szótárfile neve  
#s@p=?#s "PID" 10 '_'; // A pidstring előállítás  
  
<? @S; // Meghatározzuk a file sorainak számát, s a leghosszabb sor bájt méretét  
  
#s@k=?#s "C"; // A keresendő szó vagy kifejezés beolvasása a vágólapról  
  
#K@o="/tmp/dict.mau_" + (@p); // Megnyitjuk írásra az output fájlt  
if #K@o E "Az output file nem megnyitható!\n" XX  
  
{-} @S, $su; // Lefuttatjuk a „su” nevű szubrutint a fájl minden sorára  
T "Az input file nem megnyitható!\n" XX  
[#K@o]; // Lezárom az output fájlt  
  
#s@P="zeneneve=`"+cat "+"/tmp/dict.mau_" + (@p) +  
" | dmenu -l 36 -sf yellow -nf yellow -sb \"#d2001e\" -fn \"*-fixed-***-18-***-***-  
*\""+  
" | sed -e 's/ *::[0-9]*//' -e 's/ /_/g'"+  
"; zeneneveogg="/_/P/Szotar/0/angol/felnottno/"$zeneneve".ogg"; "+  
"if [ -f $zeneneveogg ] ; then ogg123 -q $zeneneveogg ; fi;";  
SY @P;  
RM "/tmp/dict.mau_" + (@p); // Töröljük az ideiglenes fájlt  
{[j]}; // Visszatettük a függvény régi címét  
[#M@U]; // objectlib bezárása  
  
XX  
  
$su // E szubrutin annyiszor fut le, ahány sor van a szótárfájlban  
#s@s=?#s "{-}"; // Az aktuális beolvasott sor  
j i,@s,@k; // Most a pozíció a #L@i:0 változóban van  
ha(#L(@i)>=0) // Ha megtalálta a beolvasott sorban a keresendő szót, akkor:  
T #s@s=?#s "<<< @s, " ::", " ::";  
T #s@s=?#s "<<< @s, " ::", " ::";  
T #s@s=?#s "<<< @s, \"\t\", \"\n => ";  
T #s@s=?#s "<<< @s, "<BR>", "<br>";  
T #s@s=?#s "<<< @s, "<br>", \"\n => ";  
  
T <s @o, @s; // Kiírjuk a sort a fájlba  
  
« // vége a szubrutinnak
```

Ugyanez a program külső pluginkönyvtár betöltése nélkül:

```
#!/mau

// ----- Konstansok beállítása
#s@S="/_P/Szotar/0/angol/angolmagyar.szotar"; // Szótárfile neve
#s@p=?#s "PID" 10 '_; // A pidstring előállítása

<? @S; // Meghatározzuk a file sorainak számát, s a leghosszabb sor bájt méretét

#s@k=?#s "C"; // A keresendő szó vagy kifejezés beolvasása a vágólapról

#K@o="/tmp/dict.mau_" + (@p); // Megnyitjuk írásra az output fájlt
if #K@o E "Az output file nem megnyitható!\n" XX

{-} @S, $su; // Lefuttatjuk a „su” nevű szubrutint a fájl minden sorára
T "Az input file nem megnyitható!\n" XX
[#K@o]; // Lezáró az output fájlt

#s@P="zeneneve=`"+cat "+"/tmp/dict.mau_" + (@p)+
" | dmenu -l 36 -sf yellow -nf yellow -sb \"#d2001e\" -fn \"*-fixed-*-*-*-*18-*-*-*-*-*-*
*\""+
" | sed -e 's/ *:::[0-9]*//' -e 's/ /_/g'"+
"; zeneneveogg=\"/_P/Szotar/0/angol/felnottno/\"$zeneneve\".ogg\"; "+
"if [ -f $zeneneveogg ] ; then ogg123 -q $zeneneveogg ; fi;";
SY @P;
RM "/tmp/dict.mau_" + (@p); // Töröljük az ideiglenes fájlt

XX

$su // E szubrutin annyiszor fut le, ahány sor van a szótárfájlbán
#s@s=?#s "{-}"; // Az aktuális beolvasott sor
ha(#L(?#L "POS" @s,@k)>=0) // Ha megtalálta a beolvasott sorban a keresendő szót, akkor:
T #s@s=?#s "<<<\" @s,\" ::\", \"::\";
T #s@s=?#s "<<<\" @s,\" ::\", \" ::\";
T #s@s=?#s "<<<\" @s,\" \t\", \"\n => \";
T #s@s=?#s "<<<\" @s,\"<BR>\", \"<br>\";
T #s@s=?#s "<<<\" @s,\"<br>\", \"\n => \";

T <s @o, @s; // Kiírjuk a sort a fájlba

« // vége a szubrutinnak
```

## Mau plugin készítése a Surf böngészőhöz

A Surf böngésző 0.6 verziójáról lesz szó. Ezt illik [DWM](#) ablakkezelővel együtt használni, amihez feltettük a [dmenu](#) kiegészítőt is.

Na most, mindenekelőtt csináljuk meg, hogy oldalakat bookmarkolni tudjunk, azaz magyarul könyvjelzőzni! Ehhez picit változtatni kell rajta fordítás előtt. Másoljuk át a config.def.h fájlt egy új fájlba de ugyanabban a könyvtárba, config.h néven. Ezután ezt szerkesszük az alábbiak szerint:

Töröljük ki belőle ezt a sort:

```
"prop=\"`xprop -id $2 $0 | cut -d \"'\" -f 2 | xargs -0 printf %b | dmenu`\" &&\" \\"
```

Ellenben e kitörölt sor helyére írjuk be e két sort:

```
"prop=\"`(xprop -id $2 $0 | cut -d \"'\" -f 2 | xargs -0 printf %b && \"\n
cat ~/.surf/bookmarks) | dmenu -l 36 -sf yellow -nf yellow -sb \"#d2001e\" -fn \"*-fixed-*-*-*-*18-*-*-*-*-*-*
*\"`\" &&\" \\"
```



Keressük meg a fileban (nagyon rövid a fájl, ne aggódj!) ezt a sort:

```
#define MODKEY GDK_CONTROL_MASK
```

és közvetlenül e fölé (üres sort hagyatsz ki azért) másold be ezt:

```
#define BM_ADD { .v = (char *){" /bin/sh", "-c", \
    "(echo `xprop -id $0 _SURF_URI | cut -d ' ' -f 2` && \"\
    \"cat ~/.surf/bookmarks | awk '!seen[$0]++' > ~/.surf/bookmarks_new && \"\
    \"mv ~/.surf/bookmarks_new ~/.surf/bookmarks\", \"\
    winid, NULL } }
```

Kicsit alább a fájlban keresd meg e sort:

```
{ MODKEY, GDK_u, scroll_h, { .i = -1 } },
```

és ez alá másold be ezt:

```
{ MODKEY, GDK_b, spawn, BM_ADD },
```

Ezek után, amikor a Surfben megnyomod egy url begépeléséhez a Ctrl-g gombot, a begépelésre szolgáló parancssor alatt megjelenik majd neked minden addig könyvjelzőzött weboldal linkje, amik közt válogathatsz a kurzorbillentyűkkel (választás: Enter). De ettől még természetesen gépelhetsz be új -t is. Ja, és az épp mutatott oldalt a Ctrl-b billentyűkombinációval könyvjelzőzheted. (ugye, b=„bookmark”).

Ezek után jön, ami a mau pluginnel kapcsolatos. Ez pedig a keresés a google-ban, wikikben, egyéb keresőkben

Természetesen ezt is úgy oldjuk meg, mint az illik egy modern böngészőnél, azaz a címsorba integrálva. Ehhez a **surf.c** fájlba kell módosítanunk a következőképpen:

Keressük meg benne ezt a részt (úgy a 600. sor környékén lesz):

```
    u = g_strdup_printf("file://%s", rp);
    free(rp);
} else {
    u = g_strrstr(uri, "://") ? g_strdup(uri)
Ezt kell átírnunk így:
    u = g_strdup_printf("file://%s", rp);
    free(rp);
} else if (*uri == ' ') {

char mauparancs[1024];char file neve[1024];pid_t pid; char pidstring[60];
char *mauparancseleje="mau /_/P/Mau/-/maubin/surflink.mau ";
char *ideiglenesfile neve=" /tmp/Surf_temporary_file_";
char *ideiglenesfileextension=".txt";
    if ((pid = getpid()) < 0) {printf("Nem tudom beolvasni a pid-et!\n"); exit(EXIT_FAILURE); }
    sprintf(pidstring, "%d", pid); // stringgé konvertáljuk a pid értéket
    sprintf(file neve, "%s%s", ideiglenesfile neve, pidstring, ideiglenesfileextension);
    sprintf(mauparancs, "%s%s %s \n", mauparancseleje, file neve, uri+1);
    system(mauparancs);
    FILE *maufile=fopen(file neve, "rb");
    if(!maufile) {printf("Az ideiglenes maufile nem megnyitható!\n"); exit(EXIT_FAILURE);}
    {register unsigned int i;for(i=0;i<1023;i++) mauparancs[i]=0;}
    fgets(mauparancs, 1000, maufile);
    fclose(maufile);
    remove(file neve);
    u = g_strdup_printf("%s", mauparancs);
    } else {
        u = g_strrstr(uri, "://") ? g_strdup(uri)
```

Na most, a fenti C nyelvű sorok közt van ez:

```
char *mauparancseleje="mau /_/P/Mau/-/maubin/surflink.mau ";
```

Itt az idézőjelek közt a „mau” szó után azt a linket látod, ahova egy bizonyos surflink.mau nevű progi elérési útvonala van megadva. E program bárhol lehet neked, csak ne feledd el e linket ennek megfelelően beírni. Maga a program meg természetesen mau nyelven van írva, s így néz ki:

```

#!mau

#s@0=?#s "ARGV" 2; // Az output file neve
#s@s=?#s "ARGV" 3; // A Surf böngészőbe begépelte link vagy parancs első tagja

ha (?a)<5 #s@e="https://www.google.com/?q="+(@s); »$ir;

#s@g=?#s "ARGV" 4; // A Surf böngészőbe begépelte link vagy parancs második tagja

?! (#s@s[0])
...v #s@e="http://violazoli.info/poliverzumwiki/doku.php?do=search&id="+(@g);
...p #s@e="http://parancssor.info/dokuwiki/doku.php?do=search&id="+(@g);
...g #s@e="https://www.google.com/?q="+(@g);
...h #s@e="https://hu.wikipedia.org/wiki/"+(@g);
...e #s@e="https://en.wikipedia.org/wiki/"+(@g);
..... #s@e="https://www.google.com/?q="+(@g);
.....
$ir
#K@0=@0; // Megnyitjuk az output file-ot.
<s @0 @e;
[#K@0]; // Lezárom az output file-ot

XX

```

Ezek után (ha újrafordítottad a Surf-öt) a keresés nálad úgy megy, hogy megnyomod a Ctrl-g gombot, mintha url-t akarnál beírni. De nem azt teszed, hanem nyomsz egy szóközt. Na most ha a szóköz után beírsz egy szót, majd Entert nyomsz, azt megpróbálja megkeresni az alapértelmezett keresőszolgáltatással, azzal, ami a fenti mau nyelvű szkriptben e sorban szerepel:

```
a (?a)<5 #s@e="https://www.google.com/?q="+(@s); »$ir;
```

Ha azonban a szóköz után egy (nem ékezetes!) betűt írsz, majd megint szóközt, majd azután a keresett szót, akkor azokat a keresőszolgáltatásokat használja, amik a szkriptben a megfelelő betű után vannak beírva (a betűk a „...” (hárompont) karakter után következnek, minden sorban 1, a v,p,g,h,e... nézd csak meg!).

Ahhoz hogy kibővítsd valami neked tetsző másik keresőszolgáltatással, nem kell tudnod feltétlenül programozni mau nyelven, elég ha valamelyik ilyen sorról csinálsz egy másolatot a szkriptbe, mondjuk a

```
...e #s@e="https://en.wikipedia.org/wiki/"+(@g);
```

sorról, majd óvatosan átírod: az elején a hárompont után az „e” betűt kicseréled arra ami nálad a keresés során jelzi majd hogy ezt a keresőszolgáltatást akarod használni, s kicseréled az idézőjelek közti linket is arra, amire óhajtod. A végén a + jelet s ami utána van, azt azért hagyd meg...

Ha pedig olyan karaktert ütöttél a címsorba az első szóköz után ami nem szerepel a mau szkript e listájában, akkor azt a szolgáltatást próbálja keresésre használni, ami e sorba van írva:

```
..... #s@e="https://www.google.com/?q="+(@g);
```

Ez is a google nálam, mint látható. Persze te ezt is átírhatod.

## Mau nyelvű statusbarkezelő program a DWM ablakkezelőhöz

Az alábbi program kiírja a DWM ablakkezelő statusbarjára folyamatosan az aktuális internet le- és feltöltési sebességét, a szabad memória százalékát, a gyökérfájlrendszeren található szabad lemezterület százalékát, valamint az aktuális dátumot és időt (az évszámnak csak az utolsó 2 számjegyét). A frissítési időköze 1 másodperc. A programot leghelyesebb ha a \$HOME/.xinitrc fájlba írjuk be indítani, így:

```
statusbar.mau &
```

Előbb természetesen adjunk rá futtatási jogot. A program:

```
#!/bin/mau
// Statusbar program a DWM ablakkezelőhöz
#c@w=1; // frissítési időköz másodpercben
#s@i="eth0"; // net interfész neve

Xi; // Képernyőnyitás

{( // ÖRÖKCIKLUS!

#c@d=?#c "s", "/"; // Szabad terület százaléka a gyökérfájlrendszerben
#l@s=?s; // A szabad memória százaléka

#s@d=?#s "ST"; // Az aktuális dátum és idő stringje
#s@s= [,13]@d; #s@s[10]='_'; #s@s[11]=#s@d[20]; #s@s[12]=' ' ;
if(#s@s[11]==S) T if(#s@d[22]==o) T #s@s[11]=o;
#s@s+= [11,5]@d;
#s@s=[2,]@s // levágjuk az évszám első 2 karakterét

#s@k=(?#s "n" (@i) "D: " " ^")+ " "+([2,](#s#c#l@s))+ "% " +([2,](#s#c@d))+ "% " +(@s);

Xs @k; // Kiírás a statusbarra
WS 1;

)} // ÖRÖKCIKLUS vége!

XX
```

Látható ebből, mennyire „tömör” a mau nyelv — rém komplex feladatokat lehet benne rövidke, nyúlfarknyi programokkal megoldani! Gondoljuk el, a fenti program milyen hosszú és bonyolult lenne mondjuk C nyelven...

## Parancssoros GMAIL watcher program mau nyelven

A „Gmail watcher” firefox extensiont nem fejleszti tovább a készítője. Ki kellett találnom helyette valamit. Olyat, ami hangjelzéssel figyelmeztet, mert én azt szeretem. És természetesen olyan megoldás kell, ami parancssoros, egyrészt mert „Igaz Kocka” csak azt szereti, másrészt mert az nem függ attól, miféle böngészőt használunk.

Mindenekelőtt telepíteni kell a fetchmail nevű programot.

Ezután készítenünk kell egy **\$HOME/.fetchmailrc** fájlt, aminek ez a tartalma:

```
poll imap.gmail.com port 993 protocol IMAP username "ezanevem@gmail.com" password
"IdeKerulASzupertitkosJelszavam" keep ssl
```

Ezután már csak a mau nyelvű szkript megírása van hátra, ami így néz ki:

```
#!/bin/mau
// parancssoros gmailwatcher program.
// Készítette: Viola Zoltán, violazoli@gmail.com
// A program a "mau" nevű programozási nyelven íródott, azt is én Viola Zoltán készítettem.
// Mindegyik GPL licenc alatt van.

// Konstansok meghatározása
#s@P="/tmp"; // Annak a könyvtárnak a PATH-ja, ahova a fetchmail üzenetét tároló ideiglenes fájl
kerül.
#c@c='_'; // A pidstring üres helyeit kitöltő karakter
#c@p=6; // A pidstring maximális hossza
#s@i="fetchmailmessage_mau_temporary_file"; // Az ideiglenes file alapneve a pidstring nélkül
#s@k=".tmp"; // Az ideiglenes file kiterjesztése
#s@F="/home/vz/.fetchmailrc"; // A fetchmailrc fájl elérési útvonala
#l@W=60; // Ennyi másodpercig várakozik 2 lekérdezés közt

// Előkészítő műveletek
#s@p=?#s "PID" @p @c; // A pidstring előállítás
#s@n=(@i)+"_PID-"+(@p)+(@k); // A leendő ideiglenes file nevének előállítása
// A leendő ideiglenes fájl teljes útvonalat tartalmazó nevének előállítása:
ha((#s@P[(!#s@P)-1])!='/') #s@N=(@P)+"/"+ (@n);
E #s@N=(@P)+(@n);
// A fenti két sor azt csinálja, hogy ellenőrizzük, e progi legelején az ideiglenes fájl
helyének megjelölt könyvtár útvonala
// egy "/" jellel végződik-e. Ha ez nincs így, pótoljuk e per-jelet.

#s@c="2>/dev/null fetchmail -f "+(@F)+" --check >"+(@N); // A shellnek küldendő parancs
előállítása

{( // ÖRÖKCIKLUS!
SY @c; // A parancs futtatása
BE @N, t; // Beolvasom az ideiglenes fájlt a „t” nevű stringtömbbe
RM @N; // Törölöm az ideiglenes fájlt
T »$ki; // Ha nulla méretű volt a fájl, nincs mit tenni

// A következő ciklus annyiszor hajtódik végre, ahány sor van a beolvasott ideiglenes fájlban
{| ?n
#L@L=?#L "POS" #s@t[[? -]], "messages";
ha(#L((@L)==(-1))) »$ci;
#L@L=?#L "POS" #s@t[[? -]], "seen";
ha(#L((@L)==(-1))) »$ci;
#s@r=#s@t[[? -]]; ^^ »$va;
$ci;
|}
»ki;
$va;
#l@0=#s@r; // Összes levelek száma
//?l @0; /;
#L@L=?#L "POS" @r,"(";
#L++@L;
#l@v=#s[#L@L,]#s@r; // Olvasott levelek száma
#l@k=(@0)-(@v); // Különbség
//?l @v; /;
ha(#l(@k)>0) SY "ogg123 -q _/P/Szkriptjeim/audio/ApuciLeveledJott_hangosabb.ogg";

$ki;

WS #l@W; // várakozás

)} // ÖRÖKCIKLUS vége!

XX
```

Természetesen ha a lejátszandó hangállomány nálad nem a /Programs/Szkriptjeim/audio könyvtárban van, írd át az útvonalat, s a hangállomány nevét is. Az **ogg123** programhoz a **vorbis-tools** csomagot kell telepítened. De lejátszathatsz itt mp3 hangállományt is, ha ehelyett az **mpg123** programot használod. A **-q** a „csendes üzemmódot” írja elő, hogy ne irogasson mindenféle infókat a terminálra.

## Menü a DWM ablakkezelőbe

Sosem titkoltam, hogy jóideje a DWM a kedvenc ablakkezelőm. Egyetlen hiányossága van: nincs benne menü...

No ha nincs, majd írunk bele! Írtam is korábban egyet, C nyelven, na de milyen már az hogy immár van saját programnyelvem, s nem abban van a menü... E bosszantó és szégyenletes dolgon most túllépünk!

Tehát, mindenekelőtt: csináljunk egy \$HOME/.menu KÖNYVTÁRAT! Ezután a menüfájlokat kell létrehoznod. E fájlok neve mind vagy NAGYbetűvel kell kezdődjék. A formátumuk rém egyszerű: egy sor = egy program megnevezése, utána esetleg írhatunk mindenféle paramétereket is szóközzel elválasztva. A programok neve mindig kisbetűvel kell kezdődjék! Egy menüfájlba azonban felvehetünk nemcsak programokat, de más menüket is, melyek neve már természetesen NAGYbetűvel kezdődik. Okvetlenül kell legyen egy

### Fomenu

nevű fájl is, mindig ezt hívja meg a programunk elsőként. Példaként az én pár rövid menüfájlom:

```
vz@Csiszilla ~/.menu $ cat Fomenu
```

```
Internet  
Grafika  
Video  
Terminalemulatorok  
Audio
```

```
vz@Csiszilla ~/.menu $ cat Internet
```

```
Fomenu  
transmission-cli  
gftp  
firefox
```

```
vz@Csiszilla ~/.menu $ cat Grafika
```

```
Fomenu  
pinta  
tuxpaint  
gimp
```

```
vz@Csiszilla ~/.menu $ cat Video
```

```
Fomenu  
smplayer
```

```
vz@Csiszilla ~/.menu $ cat Terminalemulatorok
```

```
Fomenu  
urxvt -fg green -bg black  
xterm  
terminology
```

```
vz@Csiszilla ~/.menu $ cat Audio
```

```
Fomenu  
audacity
```

Amint látjuk, minden almenü legelején ott szerepel a sor, hogy

## Fomenu

Ez amiatt fontos, mert a programunk olyan, hogy ha almenü-bejegyzést észlel, akkor egyszerűen azt a fájlt írja ki, s nem jegyzi meg az előzményeket. Azaz, számára teljesen egyforma, egyenrangú minden menü, halvány fogalma sincs a szerencsétlennek arról, hogy azt a menüt amiben épp bóklászunk, miféle másik menüből hívtuk meg. Netán épp az a főmenü... Azaz, ha lehetőséget akarunk rá adni, hogy visszatérjünk a menü előző („felső”) szintjére, akkor bizony amaz előző menü nevét bele kell írunk valahová az almenübe, elvileg mindegy hova, de gyakorlatilag a legelejére illik, mert akkor mutat szépen. Könyvtárlistázáskor is ugyebár a legelső, az a szülőkönyvtárra mutató link...

A választás az Enter gombbal történik, a választás nélküli kilépés pedig az Esc-vel. A mozgás a menüben pedig a kurzorgombokkal. Ha változtatsz valamit a menü-fájlokon, nem kell miatta újraindítanod nemhogy a rendszert de a grafikus felületet sem, ugyanis minden meghívásnál újraolvassa a menüfájlokat. Ha meggondoltad magad s mégse akarsz választani a menüből, akkor Esc. A menüfájlt bármikor szerkesztheted, utána nem kell újraindítani a rendszert vagy a grafikus felületet.

Na és hogy ezt miféle mau program csinálja? Hát ez amit itt alább bemutatok... Legyen a neve mondjuk dwmmenu.mau, és kreálj rá egy gyorsbillentyűt az XBindKeys progiban, s máris lesz szép menőd a DWM-hez... Szóval a program:

```
#!/bin/mau
// Menüprogram a DWM-hez

// Konstansok meghatározása
#s@p=".menu"; #s@F="Fomenu";
#s@h=?#s "ENV" "HOME";

MC f, "--fixed-***--28-***--***";
MC i, "yellow"; // normál írásszín
MC h, "gray"; // normál háttérszín
MC I, "red"; // kiválasztott tétel írásszíne
MC H, "yellow"; // Kiválasztott tétel háttérszíne

$be;
#s@S=(@h)+"/"+(@p)+"/"+(@F);

BE @S, t; // Beolvasom a menüfájlt a „t” nevű stringtömbbe

#s@s=?#s "D" t; // Megjelenítem a tömböt a dmenu-vel. A kiválasztott sort a @s fogja
tartalmazni.
if(!#s@s) E XX;
ha(#c #s@s[0]==32) XX;
#u@u=#c#s@s[0];
ha(|>|@u) #s@F=@s; »$be; // Ha a menüpont NAGYbetűvel kezdődik, almenüről van szó
SY @s;

XX
```

## 25. fejezet - Stringkonstansok a programmemóriában

Nagyon gyakori programozási feladat, hogy valahova egy stringkonstanst kell kiírni, vagy ha nem is kiírni, de felhasználni azt valamire, mint afféle konstans input adatot. Például hozzáfűzni egy másik string végéhez. Na most ha ezt ugyanazzal a stringgel kell eljátszani sokszor, a program különböző pontjain, az meglehetősen helypazarló módszer, ha ekkor azt a stringet mindannyiszor leírjuk.

Ráadásul ez esetben ha azt valamiért megváltoztatjuk, írhatjuk át mindegyik helyen. Egyszerűbb, ha készítünk egyetlen stringpéldányt valahol ebből, s ezentúl állandóan arra hivatkozunk.

Na most ezt az eddigi eszközeinkkel is megtehetjük a mau nyelvben, de ezzel az a baj, hogy lefoglalna nekünk annyi stringváltozót, ahány stringről szó van, pedig stringváltozónk úgyszólván kevés van! Stringtömböt kreálni e célra pedig macerás, és annak különben is memóriát kellene külön allokálni. És minek is, amikor mi nem akarjuk e stringeket soha megváltoztatni!

E problémára kínál elegáns megoldást a mau nyelven a **#a** típusosztály, ahol az „a” karaktert magyar embernek könnyű megjegyeznie abból, hogy „adat”. Ebből a fajta változóból is 256 fajta lehetséges természetesen, csak hogy ezek mindegyike egy speciális tömb, mely tömbbe akárhány string „rakható”, ám csak olyan stringek, amelyek a program szövegébe vannak beleírva. Ezen stringek kizárólag egy-egy sor legelején kezdődhetnek, kivéve ezen tömbök nulladik elemét. Mindezt a legegyszerűbb bemutatni egy példával:

```
#!mau
#a@a=$da; #a@d=$d;
"Fordított sorrendben:\n" {| 3 ?s #a@a[?~]; /; |}
"Rendes sorrendben:\n" #i@i=0; {| 3 ?s #a@a[#i@i]; /; #i++@i; |}
"Rendes sorrendben a d adatok:\n" #i@i=0; {| 3 ?s #a@d[#i@i]; /; #i++@i; |}
"Kiíratom a stringeket úgy, hogy a t legyen a záróbájt:\n"
#a@a=,t;
#i@i=0; {| 3 ?s #a@a[#i@i]; /; #i++@i; |}
"Beállítom a változót a megfelelő címre, s úgy, hogy az r legyen a záróbájt:\n"
#a@a=$da, r;
#i@i=0; {| 3 ?s #a@a[#i@i]; /; #i++@i; |}

XX

$da; //nulladik adat
első string
második string
3. string
$d; // A d nulladik adata
d 1 adat
d 2 adat
```

Eredménye:

```
Fordított sorrendben:
második string
első string
; //nulladik adat
Rendes sorrendben:
; //nulladik adat
első string
második string
Rendes sorrendben a d adatok:
// A d nulladik adata
d 1 adat
d 2 adat
Kiíratom a stringeket úgy, hogy a t legyen a záróbájt:
; //nulladik ada
első s
második s
Beállítom a változót a megfelelő címre, s úgy, hogy az r legyen a záróbájt:
; //nulladik adat
első st
első st
második st
```



Látható a fenti programból, hogy a **#a** típusú változóknak egy címkét adunk értékül. Amúgy bármi más is lehetne ott a címke helyett, ami tetszőleges unsigned int-ként értelmezhető aritmetikai kifejezés. Ezután a kiíratás során ezen változókra hasonlóan hivatkozunk mint a tömbökre. E változók egy-egy stringet adnak vissza, úgy, hogy a nulladik elemüknek tekintett string az, ami ott kezdődik ahova a programkódban mutat a nekik értékül adott cím, azután pedig mindegyik másik index lényegében egy ehhez viszonyított sorszámot jelent: az annyiadik sor elejénél kezdődő stringet adja vissza! Látható az is, függetlenül attól hogy a címke amit értékül adunk, „látszatra” egy vagy 2 karakteres, a nulladik string mindig a paragrafusjeltől számított második bájton kezdődik.

A programkódba elhelyezett efféle stringek mint látható nem kell hogy idézőjelek közé zárattassanak, ugyanakkor viszont escape szekvenciákat se tartalmazhatnak. Ami bájt ott van a kódban, az a string része és kész. Azt hogy meddig tart a string, onnan tudja az interpreter, hogy meg kell adni minden **#a** változónak, hogy ő melyik bájtot tekintse a string végét lezáró jelnek. Alapértelmezés szerint ez megegyezik a sorvégjellel, ami Linux alatt a 10 kódú karakter, azaz a teljes sort a stringhez tartozónak tekinti. Megadható azonban más is, amint arra a fenti kódban példa is van. Ez a sor:

```
#a@a=,t;
```

mutatja, hogy miként kell sorvégjelet megadni. Itt kötelező a vessző! A vessző előtt ugyanis a címkét adhatnánk meg a **#a** változónak, amint arra is van példa fentebb, ez:

```
#a@a=$da, r;
```

Ha nem adunk meg címkét csak határolókaraktert, akkor a címke régi értékét nem változtatja meg.

Az is látható a futási eredményből, mi lesz, ha az adott sorban nem találja meg a megfelelő határolókaraktert, mint ebben az esetben - ugyanis a

```
;; // Nulladik adat
```

szövegben nincs „r” karakter. Ezesetben halálnyugodtan tovább keres a következő sorokban is, így megadható neki többsoros szöveg is, a záróbájt gondos megválasztásával.

E pdata stringek alkalmazására remek példa az a kis mau progí, amit azért csináltam, hogy a mau interpreter fejlesztése közben a fájlokat átmásoljam vele valami másik könyvtárba. Ugyanis már elég sok mindenféle fájl kell hozzá, és macerás ezeket kiválogatni „kézzel” a könyvtárból egyenként, a mindenféle ideiglenes fájlok meg próbafájlok, példaprogramok közül. Írtam hát rá egy szkriptet, természetesen mau nyelven, aholis az átmásolandó fájlok nevét pdata stringeként tároltam el. Azt hogy mikor nincs már több filenév megadva, egyszerűen onnan tudja, hogy egy üres pdata stringet olvas be...

Íme a program:

```
#!mau
```

```
ha (?a)<3 "Nem adtad meg a könyvtárat, ahova másolnom kell a mau interpreter fájljait!\n"; XX;  
E #s@k=?#s "ARGV" 2;
```

```
#s@S="";  
#a@a=$fi;  
#l@l=1;
```

```

$ci ;
#s@s=#a@a[#l@l];
if((#s@s)==(#s@s)) T XX
SY "cp "+(s)+" "+(k);
#l++@l;
»$ci;

XX

$fi // Az átmásolandó filenevek pdata tömbje
adat.cpp
arrays.cpp
clipboard.cpp
clipboard.h
datum.cpp
draw.h
egyebek.cpp
ido.cpp
iofiles.cpp
iofiles.h
kif.cpp
logol.cpp
main.cpp
Makefile
mappa.cpp
mau.h
maulib_string.cpp
maumainfunctions.h
mauststring.cpp
mauststring.h
mau.syntax
muvelet.cpp
muvelet.h
mydmenu.cpp
mydmenu.h
mydraw.cpp
objektum.cpp
objektum.h
prepare.cpp
vz.h

```

## 26. fejezet - Álfüggvények

A mau programnyelv előző kiadása után azonnal megkaptam a kritikát pár „jó-akarómtól”, hogy e nyelv használhatatlan, mert... és jött a sok trollkodás. Az egyik érv természetesen az volt, hogy mire is lenne jó, amikor oly kevés benne a változónevekre a lehetőség!

Nos, nekem a véleményem az, hogy aki ezt mondja, annak fogalma sincs a moduláris programozásról. Régen rossz, ha az egész program egyetlen óriási rutinból áll, s abba akarunk minden funkciót belegyömöszölni, pláne mindent single változókkal megoldva. A helyes megközelítés ezzel szemben az, hogy a programot apró, jól elkülöníthető feladatrészekre bontjuk, s mindet egy-egy függvénnyel oldjuk meg. Minthogy a mau nyelvben a függvények külön névterekkel rendelkeznek, emiatt így cseppet se kevés a 256 lehetőség a változónevekre.

E verzióban azonban van már lehetőség több változónevet is használni mint 256, egy függvényen belül, s ez a lehetőség az „álfüggvények” révén érhető el!

Ez a következőt jelenti: Létrehozunk a mau programban valahol (de természetesen a főprogram után ami az első muszáj legyen) egy függvényt, ami semmi másból nem áll, mint a nevéből, s az őt lezáró

xx

utasításból.

S ezután bármely más, vele egy fájlban levő függvényből hivatkozhatunk ennek változóira úgy, hogy megadjuk e függvény nevét is a változónév megadásakor! Természetesen így nem csak efféle álfüggvény változóira hivatkozhatunk (amit azért nevezünk álfüggvénynek mert nem tartalmaz végrehajtandó kódot) hanem „igazi” függvények változóira is, bár azoknál e módszer erősen kerülendő.

Efféle dolog így néz ki szintaktikailag, amint itt bemutatom e példán:

#!mau

```
#t@a; // inicializálunk egy időváltozót
{| 3000000
#l@a=@w;
#l@c=@g;
|}
#t@b; // inicializálunk egy másik időváltozót
"Első ciklus vége.\n"
#t@c; // inicializálunk egy időváltozót
{| 3000000
#l@a=@, "a" w;
#l@, "a" c=@g;
|}
#t@d; // inicializálunk egy másik időváltozót
#g@e=(#t@b)-(#t@a);
#g@f=(#t@d)-(#t@c);
"A főprogram változóinál az időszükséglet: " ?g @e; " milliomod másodperc.\n";
"Az „a” álfüggvényt is használva az időszükséglet: " ?g @f; " milliomod másodperc.\n";
"A különbség: " ?g ((@f)-(@e)); " milliomod másodperc.\n"
XX
```

„a” // Álfüggvény

xx

Természetesen az álfüggvény neve nem muszáj hogy egyetlen betűből álljon csak, lehet tetszőlegesen bonyolult karaktersorozat, mint az igazi függvények esetén. Viszont minél több bájtból áll a név, nyilván annál lassúbb lesz a végrehajtás. S e lassúság fokozódni fog, ha van betöltve a tárba még sok másik függvény is, melyek a névsorban megelőzik a mi függvényünket, azaz van mondjuk olyan függvény is hogy aa, meg ab, ac, ghij, kiir, stb, s a mi álfüggvényünk neve mondjuk „z”, s így ő az utolsó a sorban, őt találja meg a rendszer a névkeresés során utoljára.

A fenti példa tehát úgy kb a legegyszerűbb eset, a leggyorsabb variáció. A futási eredménye nálam:

Első ciklus vége.

A főprogram változóinál az időszükséglet: 2132748 milliomod másodperc.

Az „a” álfüggvényt is használva az időszükséglet: 4840824 milliomod másodperc.

A különbség: 2708076 milliomod másodperc.

Na most a fenti adatok szerint a két ciklus idejének aránya úgy néz ki, hogy  $4840824/2132748=2.26976$

Azaz, bizony ha álfüggvény változóit használjuk, akkor számíthatunk rá, hogy még legislegjobb esetben is ez két és egynegyedszer lassabb művelet lesz, mint ha

megmaradunk az aktuális függvény „rendes” változóinak használata mellett. Mindazonáltal, a lehetőség erre mint látjuk, természetesen adott.

Megjegyzendő, hogy az álfüggvény változóinak használatához szükséges időszükséglet bár függ attól hogy a függvény neve milyen hosszú, s hogy a névsor elején áll-e vagy a végefelé a betöltött függvények neveinek sorrendjében, de attól már szerencsére NEM függ, konkrétan a forráskód fájljában hol helyezkedik el, az mindegy neki. Ott csak az a lényeg, hogy a főprogram, a „main” kell legelől legyen, azután mindegy, mi milyen sorrendben van.

## 27. fejezet - Közös kódú függvények

Gyakori eset lehet, hogy valamely kódrészlet teljesen azonos két vagy több függvényben, és pazarlás lenne leírni többször. Természetesen megtehetjük ilyenkor, hogy ezt „kiszervezzük” egy másik függvénybe, de ez nem mindig célravezető és hatékony, mert mindenekelőtt: egy új függvény készítésekor a programfájl beolvasásakor létrejön neki egy új adatterület is, ami memóriapazarlás. Aztán meg ennél is nagyobb gond lehet, hogy e kódrészlet esetleg rendkívül sok változóval dolgozik, amit nehézkes és lassú átadogatni paraméterlistán keresztül, továbbá, előfordulhat hogy ez már amiatt is lehetetlen, mert nagy és összetett adat-típusokat kéne átadni, például tömböket vagy vermeteket, fájlváltozókat... Erre a gondra kínál megoldást a mau nyelv azon lehetősége, hogy közös függvényeket írjunk! A „közös függvény” olyasmi, amit a „globális változók”-hoz hasonlóan képzelhetünk el, csak míg utóbbiak esetén az adatterület „közös” azaz „mindenki” által elérhető, addig a közös függvények esetén ez a végrehajtandó kód szempontjából van így.

A „közös függvény” tulajdonképpen nem egy teljes függvény, hanem egy „igazi” függvénynek egy része, amit címkével jelölünk meg. Az az „igazi” függvény amiben e megjelölt rész van, teljesen ugyanolyan mindenben mint bármi „normális” függvény, nyugodtan meg is hívhatjuk a szokásos módszerekkel ha óhajtjuk. Azonban a lényeg az, hogy a dupla kettőspont utasítás segítségével úgy hívhatunk meg egy másik függvényben elhelyezett, címkével megjelölt kódrészletet, hogy azon kód végrehajtása közben az utasítások a hívó adatterületére, azaz változóira vonatkoznak! E kódrészletet a szokásos

xx

utasítás kell lezárja.

Példa:

```
#!mau
$ki
"Főprogram 1. üzenet\n"
#c@c=7;
:,:, "a" $ki; // Meghívjuk az „a” függvény „ki” rutinját
"Főprogram 2. üzenet\n"
"A main c= " ?c @c; /;
XX
„a” //
#c@c=2;
$ki
"Ezt az „a” írja ki!\n"
"Az a c= " ?c @c; /;
xx
```

Eredménye:

```
Főprogram 1. üzenet
Ezt az „a” írja ki!
Az a c= 7
Főprogram 2. üzenet
A main c= 7
```

Látható e példából, hogy tényleg az „a” függvény „ki” címkéjére vonatkozott a hívás, nem a főprogram ugyanezen címkéjére.

Másik példa:

```
#!mau
$ki
"Főprogram 1. üzenet\n"
#c@c=7;
G $id; // szubrutinhívás
"Főprogram 2. üzenet\n"
"A main c= " ?c @c; /;
XX
$:::,"a"$ki; // Meghívjuk az „a” függvény „ki” rutinját
«
// Itt a vége a főprogramnak

„a” //
#c@c=2;
$ki
"Ezt az „a” írja ki!\n"
"Az a c= " ?c @c; /;
xx
```

Az eredménye ugyanaz mint az előzőnek. Látható viszont, milyen elegánsan megoldható, hogy akár más néven is meghívjuk a másik függvénybe elrejtett függvényt: a mi függvényünk vagy a főprogramunk megfelelő címkéje után egyszerűen odabiggyesztjük a közös rutin függvényének nevét és a címkéjét:

```
$:::,"a"$ki; // Meghívjuk az „a” függvény „ki” rutinját
```

Ezen duplakettőspont utasítás általános alakja, azaz szintaxisa:

```
:::„név”$cimke
```

vagy

```
:::[sorszám]$cimke
```

ahol a név egy stringkifejezés lehet, a sorszám egy unsigned int kifejezés, de a **\$cimke** helyén NEM állhat tetszőleges kifejezés, hanem CSAK és KIZÁRÓLAG **\$**-sal meghatározott címke!

A :: után állhatnak whitespace karakterek.

## 28. fejezet - A mau programozás csapdái

Avagy: mik azok a tipikus hibák, amiket könnyű elkövetni mau nyelven programozás közben. Nem szégyellem bevallani, ezek java részére a magam kárából jöttem rá...

1. A pontosvessző C stílusban használata. Azaz: a pontosvessző után nem hagysz szóközt vagy más whitespace karaktert. Ezesetben ugyanis a következő utasítást átugorja. Például:

`"kiírja e szöveget";/;`

A fenti esetben bizony nem fog neked újsort kiírni, mert a csukó idézőjel utáni pontosvessző és a `/` jel közt nincs whitespace hagyva.

2. A programkód egy részét megjelölő címke `$` jele nem a sor legelső karaktere.

3. Amikor a `»` utasítással el akarsz ugrani egy címkére, mondjuk a "ki" nevűre, akkor így írod:

`»ki`

És nem így:

`»$ki`

Az első esetben ugyanis az interpreter veszi a „k” karakter ASCII kódját, ami valami 128-nál kisebb szám, s halálnyugodtan elugrik neked a programkód annyiadik bájttjára, valahova az elejére. És még örülhetsz mint majom a farkának, ha ott valami olyasmi van ami miatt syntax errorral leáll neked. Mert ennél ezer-szer rosszabb, ha ott értelmes utasítás kezdődik, s onnan folytatja neked a progit, aztán csak nézel, miféle hülye eredményeket produkál neked. De ha leáll syntax errorral, akkor is irtó nehéz kiderítened, hogy a francba került oda... Erre tehát **nagyon** ügyeljünk!

4. Azt hiszed, az `if` itt is úgy működik mint a C nyelvben... Azaz: Írsz mondjuk egy ciklust, amiben van egy feltételvizsgálat. A ciklus rém rövid, simán belefér egy sorba. És te tényleg egy sorba is írod... Valami efféle módon:

```
{| 30 if(valami) T »$ki; |}
```

A fenti példában a 30 helyett persze bármi szám állhat, s teljesen mindegy az is, miféle feltétel van a „valami” helyén. A lényeg, hogy ha a feltétel igaz, a `T` végrehajtja a sor további részét. DE CSAK AKKOR... Azaz ha a feltétel NEM igaz, elugrik a következő sorra. De mert nálad a ciklust lezáró `|}` utasítás is a `T` mögött szerepel, emiatt azt soha az életben nem fogja végrehajtani. Még akkor se ha a feltétel igaz, mert akkor előbb jön a `»` ugróutasítás...

Különben rajtad az se segítene ha a `T` után nem egy `»` állna. Mert akkor meg az van hogy CSAK a feltétel igaz volta esetén jön ciklusvég, különben soha. Na és neked nyilván az se jó, mert nem azt akartad írni. Efféle esetben tehát hiába férne ki a teljes ciklus egy sorba, így kell írnod:

```
{| 30 if(valami) T »$ki;  
|}
```

Ügyeljünk erre, ez a hibafajta egy alkalommal még engem is jócskán megszivatott, pedig én magam alkottam a mau nyelvet... És persze hogy az interpreter forrás-kódjában kerestem a hibát kétségbeesetten, holott OTT, ott minden jó volt... Mert a hibát a mau nyelvű programban vétettem!

5. A függvény kezdetét jelző `„` karakter nem a sor legelső karaktere. Aztán meg csodálkozol amikor futás közben a progi méltatlankodik, hogy olyan függvényt akarsz végrehajtani vele, ami be sincs töltve...

6. Változóhoz úgy akarsz értéket adni vagy belőle kivonni, hogy nem teszed a változót zárójelek közé. Azaz ezt írod például:

`@a+1`

És nem ezt:

`(@a)+1`

Erről már írtam e könyv elején, de olyan fontos hogy itt újra megemlítem.

7. Azt hiszed, a mau nyelvben is magasabb precedenciájú a szorzás mint az összeadás, pedig nem...

Azaz:

$3*7+2$  eredménye itt 27 lesz, és nem 23...

Ezt is írtam már korábban, de ezt is újra megemlítem, mert nagyon fontos.

8. Egy string egy karakterének az index operátorral értéket adsz, a nulladik karakternek a CHR\$(0) értéket. Az eredeti stringed mondjuk az "abcd" volt, ami tehát 4 karakter hosszú, s te kis naiv azt hiszed, ezek után a stringed 0 hosszú, hiszen ott a nullabájt a nulladik pozíción... Holott nagyon nem! A stringedet a mau interpreter továbbra is 4 hosszúnak tekinti, mert a hosszat ő külön letárolja... Az egészen más kérdés, mit ír ki, ha kiíratod a stringet ezek után... Vagyis, ha már mindenáron efféleképp akarsz haxorkodni, akkor használd a 0 érték stringbe rakása után azt az utasítást is, ami „rendbehozza” neked a string hosszát, azaz egy mondjuk #s@s nevű string esetén ezt:

```
#s!@s
```

vagy stringtömb esetén, mondjuk a t nevű stringtömb 2. értékénél:

```
#s!@t[[2]];
```

9. Készítesz egy remek programot, ami valami általad írt mau nyelvű függvényt is tartalmaz, s ez a program a szülő névtér valamely változójára hivatkozik a ^ jel segítségével, például így: @^M;

Aztán később valamiért átírod a programot, s eközben a függvényből mindössze egy § jellel jelölt szubrutin lesz. Aztán csodálkozol, hogy az addig jól működő programod hirtelen „elszáll” szegmentálási hibával, s érdekes módon mindig akkor, amikor belép e „függvénybe”... Holott logikus: a függvényeknek van saját névtérük. Onnan tehát „ki lehet hivatkozni” a szülő névtérbe. Amint azonban te „lefokozod” (vagy inkább felemeled?) a korábbi függvényt egy szubrutin szintjére, annak máris nem lesz külön névtére, ugyanaz lesz tehát a névtére mint annak az akárminek ami őt meghívja, tehát nem létezik olyan másik névtér, ami „magába foglalná” az ő névtérét, eképp nem tud abból „kihivatkozni” holmi felsőbb szintű névtérbe, mert nincs névtér már rajta kívül... Azaz jól jegyezzük meg: saját névtére a FÜGGVÉNYEKNEK van, annak, ami így kezdődik:

```
„Ez egy függvény neve”
```

A szubrutinoknak viszont nincs külön névtérük, annak tehát ami efféleképp kezdődik:

```
§ru;
```

## 29. fejezet - Önmódosító programok készítése

Az önmódosító program egyszerűen egy olyan program, ami átírja a saját kódját. Efféle programok készítése a legtöbb esetben természetesen „ön-tökönlövés”, mert mi az ördögnek is csináljunk effélét... Sőt, az ilyesmi kétségkívül kockázatos biztonsági szempontokból is. Nem véletlen az sem, hogy a számítástechnikában kitalálták, hogy a programok kód- és adatterülete elkülönüljön egymástól.

Mégis, akadnak olyan esetek, amikor hasznos lehet adatként értelmezni a programterületen megtalálható bájtokat, s erre már ismertettünk is módszert e dokumentációban, azaz erre képes a mau nyelv. Mégritkábban ennek a fordítottja



is előfordulhat, hogy adatokat szeretnénk „betáplálni” a kódterületbe, amiket aztán utasításokként értelmez az interpreterünk. Ennek például akkor lehet jelentősége, ha olyan extravagáns programot írunk, ami valamiképp a mesterséges intelligenciával kapcsolatos, vagy ha holmi „genetikus algoritmust” próbálunk megvalósítani, afféle „kód-evolúciót”, aholis egy függvény vagy szubrutin lefutásának eredményét megvizsgáljuk, majd attól függően változtatunk valamit a kódján, megint lefuttatjuk, ekkor megint változtatunk a kódon ugyanott vagy másutt így vagy úgy, esetleg e változtatás ráadásul véletlenszerű, mint a mutációk a genetikában, s így találgatással akarjuk megtalálni valamihez a legmegfelelőbb képletet, mondjuk 1 milliárdszor lefuttatva a szubrutint...

Na erre ad nekünk eszközt a mau nyelv 2 egyszerű utasítás segítségével. Ezek:

**Pc cím, adat**

ahol a cím egy mau\_l azaz „unsigned int” típusú aritmetikai kifejezés, az adat pedig egy mau\_c vagyis „unsigned char” típusú aritmetikai kifejezés.

és

**Ps cím, adat**

ahol a cím egy mau\_l azaz „unsigned int” típusú aritmetikai kifejezés, az adat pedig egy MAUSTRING típusú stringkifejezés.

Példa a használatára:

**#!mau**

**#a@a=\$on;**  
**#s@s="NEM";**  
**?s #a@a[1]; /;**

**Pc \$on+44, W;**  
**?s #a@a[1]; /;**  
**Ps \$on+44, @s;**  
**?s #a@a[1]; /;**

**XX**

**\$on // önmódosításra kijelölt terület**  
**Ez itt az, amit kiírok!**

Eredménye:

**Ez itt az, amit kiírok!**  
**Ez Wtt az, amit kiírok!**  
**Ez NEM az, amit kiírok!**

A fenti programnak persze semmi értelme, azon kívül hogy szemlélteti az utasítások használatát. E dokumentum előző fejezeteinek ismeretében a működése már magától értetődő kell legyen az Olvasó számára. Az egyedüli amire ügyelni kell, hogy a

**\$on+44**

képletnél a „44” helyére pontosan az a szám kerüljön ami meghatározza, hogy a címke végétől kezdve hányadik bájton kezdődik az a terület, amire a program-memóriában írni akarunk. Ezt nem szabad eltévesztenünk semmiféleképpen sem...

## 30. fejezet - Pluginek készítése a mau interpreterhez

Mindenekelőtt pár hasznos függvény, ami kell majd e rész későbbi információinak megértéséhez:

Ez a függvény:

```
?_ k
```

Visszaad egy unsigned int számot, ami azonban csak 0 vagy 1 lehet. Akkor 1, ha az épp használt mau interpreterben létezik olyan 1 karakteres utasítás, aminek az ASCII kódja a „k”, (mely tetszőleges unsigned char típusú aritmetikai kifejezés lehet).

Íme erre egy példaprogram:

```
#c@c='#;  
if(?_ #c@c) T "Van ilyen függvény!\n" XX  
E "Nincs ilyen függvény!\n"  
XX
```

A fenti program nyilván azt írja majd ki hogy van ilyen függvény, tudniillik mert a „#” egy valóban létező utasítás a mau nyelvben, ezzel kezdődnek az értékadó utasítások.

Ez a függvény pedig:

```
?#g "_" k
```

visszaadja unsigned long long számként azt a konkrét memóriacímet, ami az adott „k” karakterrel kezdődő nevű mau utasítás függvényének kezdőcíme az operatív tárban. A „k”, mint az előző fenti esetben is, itt is lehet tetszőleges unsigned char típusú aritmetikai kifejezés. Ügyeljünk rá, hogy mind a lehetséges 256 esetben kapunk valamiféle eredményt ezen hívás során, akkor is ha nincs hivatalosan utasítás rendelve az adott karakterhez, tudniillik minden utasítás-belépési pont létezik akkor is, ha az az utasítás valójában nem létezik, ekkor ugyanis azon a helyen a „syntax error” hibaüzenetet megvalósító függvény belépési pontja áll!

Erre is egy példaprogram:

```
#!mau  
{| 20  
? ((#g20-?|)+32); " = " ?g ?#g "_" ((#g20-?|)+32); /;  
|}  
XX
```

Eredménye:

```
= 4286028  
! = 4391134  
" = 4342604  
# = 4388092  
$ = 4282448  
% = 4282448  
& = 4282448  
' = 4282448  
( = 4397669  
) = 4395221  
* = 4342234
```

```
+ = 4339119
, = 4339174
- = 4339229
. = 4339284
/ = 4338659
0 = 4282583
1 = 4282448
2 = 4282448
3 = 4282448
```

Egy másik hasonló függvény:

**?u S**

E fenti függvéynél az S egy PONTOSAN 2 bájt hosszú stringet eredményező stringkifejezés kell legyen. A függvény visszaad 1-et ha létezik a stringet alkotó 2 bájtal kezdődő nevű mau utasítás, és 0 lesz a visszaadott érték, ha ilyen nem létezik.

Legegyszerűbb ennek kipróbálásához elindítani a calc.mau nevű programot, s beírni efféle sorokat mint e példán látható:

```
./mau calc.mau
mau> ?c ?u "jk"; /
0
mau> ?c ?u "#!"; /
1
mau> ?c ?u "#d"; /
1
mau> ?c ?u "SY"; /
1
mau> ?c ?u "?"; /
LOG:> 2014.05.15 00:04:02 : A string hossza itt csak pontosan 2 lehet! Hely a programban:9
vz@Csiszilla /Releases/2014/U/Common/vz/MAU=>./mau calc.mau
mau> ?c ?u "#d@"; /
LOG:> 2014.05.15 00:04:23 : A string hossza itt csak pontosan 2 lehet! Hely a programban:11
```

Na most a fenti függvények azért fontosak nekünk - de csak a „haladó”, „expert” mau-programozóknak! - mert a mau nyelv rendelkezik azzal a rendkívüli képességgel, hogy plugineket írhatasz hozzá, olyan funkciókat ráadásul, melyeket megfeleltethetsz konkrét mau utasítástokeneknek, és éppúgy használhatod őket a programokban, mint bármi eleve beépített utasítást! Ezek ráadásul „menet közben” ki- betölthetőek, azaz azt is megcsinálhatod hogy a program elején egy utasítás neked egészen mást jelentsen mint a közepén vagy a végén.

A plugin, az azonban nem egy mau nyelven megírt program. Hogy mau nyelvű programokat/librarykat miként futtathatunk mau programból, azt korábban leírtam már egy fejezetben. Itt arról van szó, hogy magát a mau interpretert „patkolhatod meg” új funkciókkal! Épp emiatt ezeket neked bizony ugyanúgy C/C++ nyelven kell megírnod, mint a mau interpreter eddigi részét írtam én. Leírom ide, miként alkothatsz plugineket, azaz saját utasításokat a mau interpreterhez.

Nulladik lépésként el kell döntsöd, melyik ASCII kódú karakter lesz a te utasításod tokenje. Célszerű (bár nem kötelező) olyat választanod, ami még nem foglalt, hogy ez melyik, azt a fentebb bemutatott **?\_** mau függvénnyel tesztelheted. Ehhez a teszteléshez még csak azzal se kell fárasztanod magadat, hogy külön programot írj: Egyszerűen indítsd el a példaprogramok közt levő kis mau kalkulátor-programot és add ki neki a következő utasításokat mint e példán látható:

```

vz@Csiszilla /Releases/2014/U/Common/vz/MAU=>./mau calc.mau
mau> if(?_ '#' ) T "Van!" /;
Van!
mau> if(?_ 'Y') T "Van!" /;
mau> if(?_ 'Y') T "Van!" /;
mau> if(?_ '#' ) T "Van!" /;
Van!
mau> XX

```

Amint látod e fenti példán, a **#** karakterre azt írja ki, hogy Van!, s tényleg, ilyen utasításunk van. Az **Y** karakter esetén viszont nem ír ki semmit. Mert olyan betűvel kezdődő utasításunk nincs.

Ez volt tehát a nulladik lépés. Első lépésként meg kell írnod magát a plugint, mint egy C (vagy C++) programot. Legyen a neve mondjuk **teszt.cpp** a programodnak. Egy efféle mondjuk így nézhet ki:

```

#include "vz.h"
#include "mau.h"

#include <stdlib.h>
#include <stdio.h>

mau_L pikiir(F &f) {
    (*f.L)("Ez egy olyan LOG üzenet, amit a pikiir rutin küld neked!");
    printf("Pi=3.14\n");
    return 0;
}

mau_L elsokarakterem(F &f) {
    printf("E függvény nevének első karaktere: %c\n",f.a);
    printf("Karakter=%c\n",(*f.MAU_mau_c)(f));
    return 0;
}

```

E fenti program jelentését, hogy mi miért van benne, mindjárt elmagyarázom, előbb azonban haladjunk tovább a teendők ismertetésében! A lényeg az, hogy ebben 2 függvény van, amiket te majd bele akarsz építeni a mau interpreterbe, ezeknek a neve az, hogy „**pikiir**”, és hogy „**elsokarakterem**”. Na most, e fenti programot, ha a forráskód fájljának a „teszt.cpp” nevet adtad, ezzel az utasítással kell lefordítanod:

```
g++ -c -fPIC teszt.cpp
```

Ezután libraryt kell készítened a létrejött **teszt.o** fájlból, ezzel a paranccsal:

```
g++ -shared -fPIC -o teszt.so teszt.o
```

Miután így elkészült neked a **teszt.so** nevű „dinamikus library”, meg kell tudnod, a függvényeid milyen néven szerepelnek benne. Tudniillik, a „**geci-C**”, akarom-mondani a gcc fordítóprogram (de a g++ is) sajnos mindenféle ingyombingyom kiegészítéseket fűz hozzá az általad adott nevek elejéhez és végéhez, amik arra utalnak, hogy azon függvénynek milyen típusú a visszatérési értéke, valamint hogy hány és milyen típusú paramétert vár! Sőt, szerintem még valami sor-számszerűséget is, de bevallom ez utóbbiban már nem vagyok biztos. Szerencsére azonban, nem is kell hogy ezt tudjam én, vagy az aki plugint készít. Elég annyi, hogy ki kell adni e parancsot:

```
nm teszt.so
```

S erre valami efféle listát ír ki neked:

```

0000000000200b28 B __bss_start
0000000000200b28 b completed.6360
                                w __cxa_finalize@@GLIBC_2.2.5
0000000000000600 t deregister_tm_clones
0000000000000670 t __do_global_dtors_aux
00000000002008c0 t __do_global_dtors_aux_fini_array_entry
0000000000200b20 d __dso_handle
00000000002008d0 d _DYNAMIC
0000000000200b28 D _edata
0000000000200b30 B _end
000000000000077c T _fini
00000000000006b0 t frame_dummy
00000000002008b8 t __frame_dummy_init_array_entry
00000000000008b0 r __FRAME_END__
0000000000200ae8 d _GLOBAL_OFFSET_TABLE_
                                w __gmon_start__
0000000000000590 T _init
                                w _ITM_deregisterTMCloneTable
                                w _ITM_registerTMCloneTable
00000000002008c8 d __JCR_END__
00000000002008c8 d __JCR_LIST__
                                w _Jv_RegisterClasses
                                U printf@@GLIBC_2.2.5
                                U puts@@GLIBC_2.2.5
0000000000000630 t register_tm_clones
0000000000200b28 d __TMC_END__
0000000000000720 T _Z14elsokarakteremR1F
00000000000006e8 T _Z6pikiirR1F

```

Ebben a fenti listában kell megkeresned azokat a sorokat, amik tartalmazzák a szívednek oly kedves (mert általad írt) függvények nevét. Jelen esetben ez a két utolsó sor. Persze ha jó nagy libraryt írsz, akkor kellemetlen egy hosszú efféle listát végigbogarászni, de ez elkerülhető, hogy ha tudod a függvényed nevét, s ezért így adod ki a parancsot, mondjuk a „pikiir” függvényre keresve:

```
nm teszt.so | grep "pikiir"
```

Ekkor csak ezt a sort listázza:

```
00000000000006e8 T _Z6pikiirR1F
```

Ezen eredményből téged a jobb szélső mező érdekel, az, hogy

```
_Z6pikiirR1F
```

Ez az a név, amin a függvényed a libraryban szerepel.

Na most, ezt egy mau programban imígyen használhatod fel:

```

#!mau
#s@L="/Releases/2014/U/Common/vz/MAU/teszt.so";
#s@f="_Z6pikiirR1F";
#s@g="_Z14elsokarakteremR1F";

#g@g=?#g "_" '3; // Elementjük a függvény régi címét
#M@U=@L; // objectlib megnyitása
if(?d) T "A dinamikus könyvtár nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX

? " 3 = " ?g ?#g "_" '3; /;

[['3,U,@f]]; // Betöltjük a kiíró rutint a '3' karakterre
if(?d) T "A függvény nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX

[['j,U,@g]]; // Betöltjük a másik rutint a 'j' karakterre
if(?d) T "A függvény nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX
// végrehajtás
? " 3 = " ?g ?#g "_" '3; /;
"Végrehajtom:\n"
3;
j $61;

```

```
[[ '3=@g]]; // Visszatettük a függvény régi címét
[#M@U]; // objectlib bezárása
"Vége a proginak!\n"
3; // Erre most syntax error-t kell dobni
```

XX

Megegyeszer közlöm a fenti programot, a megfelelő magyarázatokat beszúrva a kódba:

```
#!mau
#s@L="/Releases/2014/U/Common/vz/MAU/teszt.so";
```

Itt fent eltároljuk a betöltendő dinamikus library elérési útvonalát egy stringváltozóba.

```
#s@f="_Z6pikiirR1F";
#s@g="_Z14elsokarakteremR1F";
```

Itt fentebb meg a meghívandó függvényeink nevét tároljuk el stringváltozókba.

```
#g@g=?#g "_" '3; // Elmentjük a függvény régi címét
```

Ugyebár, el kell döntenünk, melyik karakterre kötjük rá a parancsunkat, azaz mi legyen a függvényünk „tokenje”, ami által hivatkozunk majd rá a program későbbi részében. Én itt e példában a „3” karaktert választottam most, ami persze nem jelenti azt, hogy ezentúl a 3-as számot ne lehetne számjegyként használni aritmetikai kifejezésekben. De más dolog az, és megint más az hogy utasításként használhatjuk-e a „3” karaktert! Na most, illendő először is elmenteni e karakter régi függvénycímét, mert amikor már nem kell nekünk a mi magunk új függvénye, melegen ajánlott a régi címet visszapakolászni az interpreter megfelelő helyére. Mert ugye hátha olyan utasítástokenről van szó, amit később is használna a progi a régi értelmezése szerint, szóval hagyjunk magunk után rendet... Mint látható, a függvényeink/utasításaink belépési pontjainak címét unsigned long long típusú számként adja vissza nekünk e **?#g "\_"** függvény.

```
#M@U=@L; // objectlib megnyitása
```

A mau interpreter külön speciális változótípusként kezeli a függvénykönyvtárakat, ezek casting operátora a **#M** karakterpáros. Ezekből is 256 darab van természetesen, ezek közül az „U” nevűnek itt fentebb értékül adtam a **@L** stringváltozót. Állhatna a **@L** helyett ott tetszőleges stringkifejezés is. Ennek hatására ő rögvest megpróbálja beolvasni a megadott libraryt a string által jelzett útvonallról.

```
if(?d) T "A dinamikus könyvtár nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX
```

A fenti sorban leellenőrizzük, hogy sikeres volt-e a library beolvasása. A **?d** rendszerváltozó tartalma sikeres betöltés esetén nulla, hiba esetén 1. Ha hiba következett be, a hibaüzenetet a **?#s "d"** rendszerfüggvény adja vissza stringként, ezt iratjuk ki fentebb (ha volt egyáltalán hiba).

```
? " 3 = " ?g ?#g "_" '3; /;
```

A fenti sor nem feltétlenül szükséges - itt mindösze kiiratjuk a „3” karakterhez tartozó régi függvény címét. Mivel e karakterhez eddig nem rendeltünk hozzá semmiféle „értelmes” tevékenységet, így ezen a helyen a megfelelő rendszermátrixban annak a rutinnak a címe van, ami a „syntax error” hibaüzenetet generálja.

```
[[ '3,U,@f]]; // Betöltjük a kiíró rutint a '3' karakterre
```

A fenti sor magáért beszél - így kell betölteni egy a megnyitott libraryban levő függvényt a megadott utasítástokenhez. Az egész cumó mint látszik dupla sz(e)gletes zárójelek közt van. Az első paraméter egy unsigned char típusú aritmetikai kifejezés, azt mondja meg, melyik ASCII kódú tokenhez rendeljük

hozzá az új utasításunkat. Itt a 3 karakter előtt amiatt áll aposztróf, hogy karakterként értelmezze, különben számként olvasná be az értéket. A második paraméter is egy unsigned char típusú aritmetikai kifejezés, azt mondja meg, melyik library-változóhoz hozzárendelt libraryból töltjük be a függvényt. Ugye, fentebb az „U” nevűvel nyitottuk meg, ezért szerepel itt is U. A harmadik paraméter pedig egy stringkifejezés lehet, itt egyszerűen megadtuk a program elején meghatározott stringváltozót.

```
if(?d) T "A függvény nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX
```

Itt a hibaellenőrzés. Mint látható, abszolút mindenben ugyanúgy kell eljárni, mint a library megnyitása utáni tesztelésnél, legfeljebb az általunk kiíratott szöveg más.

```
[[ 'j,U,@g]]; // Betöltjük a másik rutint a 'j' karakterre  
if(?d) T "A függvény nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX
```

Itt a másik függvényünket rendeljük hozzá a „j” karakterhez, és végrehajtjuk az ellenőrzést.

```
// végrehajtás  
? " 3 = " ?g ?#g "_" '3; /;
```

Kiíratjuk érdekességképpen a „3” karakterhez rendelt függvény belépési pontját. Most más címet kell kiírjon, mint először.

```
"Végrehajtom:\n"  
3;
```

Itt hajtjuk végre a függvényt, amit a „3” karakterhez rendeltünk. Ekkor kell kiírnia a pi értékét, sőt, előtte egy log üzenetet is.

```
j $61;
```

Itt meg a „j” karakterhez rendelt rutinunkat hívjuk meg, s van utána megadva egy paraméter is, ami egy hexadecimális számkonstans.

```
[[ '3=@g]]; // Visszatettük a függvény régi címét
```

A „3” karakterhez visszaállítjuk annak eredeti függvénycímét. Ez ugye az ami a „syntax error” hibaüzenetet generálja. Ezt korábban elmentettük egy változóba, na most abból töltjük vissza.

```
[#M@U]; // objectlib bezárása
```

Itt fentebb „bezárjuk” a korábban megnyitott objectlibraryt. Ezután nem tudjuk elérni az abban levő függvényeket!

```
"Vége a proginak!\n"
```

```
3; // Erre most syntax errorrt kell dobnia
```

A fenti sor azt jelenti, hogy mert a „3” karakter újra a „régimódon” viselkedik, azaz sehogyan se mert nincs hozzárendelve utasítás, így most e sornál a programunk el kell halálozzék egy syntax error hibaüzenetttel, ami KIVÉTELESEN azt jelenti, hogy minden a legnagyobb rendben van...

```
XX
```

E fenti sor jelenti ugyebár a program végét.

Na és most akkor nézzük át részletesen, hogy miként is kell megírni azt a programot, amiből a betöltendő libraryt készítjük el! Megint idemásolom, ezúttal zöld színnel, s beleírom soronként a magyarázatot.

```
#include "vz.h"  
#include "mau.h"
```

E két fenti sor mutatja, miket kell include állományként betölteni a mau programnyelv forráskódját tartalmazó csomagból.

```
#include <stdlib.h>  
#include <stdio.h>
```

Itt fent hogy miféle egyéb include állományok szerepelnek, az attól függ, mi kell a te megálmodott, leprogramozandó funkcióidhoz.

```
mau_L pikir(F &f) {
```



Itt fent kezdődik e sorral a(z egyik) általad óhajtott Nagyon Okos Dolgokat Művelő függvényed. A neve természetesen nem muszáj hogy „pikiir” legyen, lehet akármi más is. Ellenben nagyon fontos, hogy a típusa okvetlenül **mau\_int** legyen! Ezenfelül az is kötelező, hogy csak és kizárólag egyetlen paramétere legyen, mégpedig ez:

```
(F &f)
```

Viszont ha neked az úgy jobban tetszik, írhatod ezzel a szintaktikával is mert ugyanazt jelenti:

```
(F& f)
```

Mindenesetre, ez egy „F” típusú struktúra referenciáját jelenti, ennek a struktúrának a leírását pedig a programod elején beinclude-olt „**mau.h**” headerfájlban találod meg (sok más dologgal együtt). Ebben van letárolva az égvilágon mindenféle izémizé, ami csak kellhet az interpreter bármely funkciójához. E struktúra minden mau utasításnál átadódik mint paraméter (illetve nem ő, hanem csak a címe, de mert referenciaként adjuk át, ugyanúgy hivatkozhatunk rá, mintha ténylegesen átadásra került volna). Minthogy a te függvényed teljesen ugyanúgy kell viselkedjen mint a „hivatalos” mau függvények, emiatt neki is pontosan ugyanannyi darab paramétere kell legyen, s ugyanolyan típusú is. Emiatt egyáltalán semmiféle más akármit se írhatsz a függvényed input paraméterének.

```
(*f.L)("Ez egy olyan LOG üzenet, amit a pikiir rutin küld neked!");  
printf("Pi=3.14\n");
```

A fenti sorok csak példák, e fenti 2 sor helyett kell álljon mindenféle más sok-sok roppant okosság, amit te a függvényedben meg óhajtasz valósítani, azaz a LÉNYEG, amiért a függvényt egyáltalán megírtad.

```
return 0;
```

Kötelezően ez kell legyen a függvényed visszatérési értéke, ha minden rendben van. Egyéb lehetőségek:

```
return 1;
```

A program azonnal befejeződik „sikeresen”, azaz az **EXIT\_SUCCESS** C-beli makro-nak megfelelően.

```
return -1;
```

A program azonnal befejeződik „sikertelenül”, azaz az **EXIT\_FAILURE** C-beli makro-nak megfelelően.

```
return 3;
```

A program azonnal befejeződik „sikertelenül”, de előbb még megajándékozza a felhasználót külön is egy „syntax error” hibaüzenettel a megfelelő log állományba.

Akadnak más lehetőségek is, de azokat jobb ha nem birizgáljuk pluginból.

```
}
```

A fenti sor természetesen a függvényed végét jelenti.

```
mau_int elsokarakterem(F &f) {  
printf("E függvény nevének első karaktere: %c\n",f.a);  
printf("Karakter=%c\n",(*f.MAU_USC)(f));  
return 0;  
}
```

A fenti sorok természetesen egyszerűen egy másik függvényt mutatnak be példaként.

Na most mint látható, a fenti program művel pár titokzatos dolgot: hivatkozik mindenfélére az F struktúrából, adatokra is és függvényekre is. Természetesen aki átnézi a **mau.h** fájlban az F struktúra leírását, abból sok érdekeset tanulhat, mert az a mau interpreter „lelke”, de ha valaki nem akar nagyon „extra” dolgokat varázsolni, annak azért ideírom a legfontosabbakat, hogy ne nagyon kelljen keresgélnie:

Az **f.p** pointer egy unsigned char típusú tömb elejére mutat, ebbe van betöltve a teljes mau forráskód ami épp végrehajtódik. Az **f.P** unsigned int típusú változó az „utasításszámláló”, azaz az **f.p[f.P]** mindig az aktuális végrehajtandó karakterre mutat. Amikor a vezérlés bekerül a te függvényedbe, akkor ez az **f.p[f.P]** pontosan a te utasításod tokenje utáni első (unsigned char típusú) karaktert adja vissza, függetlenül attól, hogy ez whitespace karakter-e vagy sem.

Amennyiben tehát valami olyan utasítást írunk ami „ugrással” jár, akkor ezen **f.P** mutató értékét kell módosítanunk. Többnyire persze szerintem ez szükségtelen lesz.

Fontos lehet azonban az, hogy beolvassunk mindenféle adatokat, paramétereket. Nos, a mau nyelvben többféle aritmetikai kifejezés kiértékelő rutin létezik, attól függően, miféle típusú számot kívánunk beolvasni, sőt van külön stringkifejezés kiértékelő rutin is. Ezek mindegyike input paraméterként egy F struktúra referenciáját várja el, pontosan azért, amit a te plugined is megkap inputként előzőeken az interpretertől. Azaz a programkódban soron következő aritmetikai kifejezés kiértékelése unsigned char számként e függvénnyel történik:

```
(*f.MAU_mau_c)(f)
```

Vagyis, ha neked van egy ilyen változód hogy:

```
unsigned char c;
```

Akkor ennek így adhatsz a pluginedben értéket a mau programból:

```
c=(*f.MAU_mau_c)(f);
```

Ez amiatt irandó ilyen bonyolultan, mert ugye a plugin dinamikusan töltődik be, nincs fixen hozzálinkelve az interpreterhez, így a plugin nem tudhatja meg más-ként az interpreter bizonyos függvényeinek helyét, csak úgy, hogy ezek le vannak tárolva maguk is az F struktúrában. Egyéb típusú aritmetikai kifejezés kiértékelő rutinok :

```
kif (*MAU_mau_c)(F& f);
kif (*MAU_mau_C)(F& f);
kif (*MAU_mau_i)(F& f);
kif (*MAU_mau_I)(F& f);
kif (*MAU_mau_L)(F& f);
kif (*MAU_mau_L)(F& f);
kif (*MAU_mau_g)(F& f);
kif (*MAU_mau_G)(F& f);
kif (*MAU_mau_f)(F& f);
kif (*MAU_mau_d)(F& f);
kif (*MAU_mau_D)(F& f);
```

A stringkiértékelő rutin:

```
MAUSTRING (*MAU_mau_s)(F& f);
```

Vannak más hasznos rutinok is amikről jó ha tudunk. Ez például:

```
mau_int (*MAU_WSPC)(unsigned char a);
```

visszatér egy 1-es értékkel, ha a karakter amit megadtunk neki, a mau interpreter szerint whitespace, különben nullával tér vissza.

Ez pedig:

```
void (*MAU_NEMSPACE)(F& f);
```

Előrelépteti a programszámlálót (tehát az **f.P** változót) úgy, hogy az a következő nem whitespace karakterre mutasson a forráskódban.

Ez pedig a logoló rutin:

```
void (*L)(const char * format, ...);
```

Annyit csinál, hogy mint a printf utasítás, kiírja a megadott üzenetet a log fájlba (ez alapesetben az stdout, azaz a képernyő). Teljesen ugyanúgy működik valóban, mint a printf, épp csak a log fájlba ír, elfogad ő is változó hosszúságú paraméter-

listát, egyetlen különbség, hogy a kiírás után automatikusan kiír még egy sorvég karaktert is, azzal nem kell törődnünk, valamint a kiírást azzal kezdi, hogy a sor elejére odaírja ezt:

LOG:> 2014.04.08 22:02:53 :

Azaz hogy ez log, és a dátumot meg az időt.

Ezek után már tényleg csak a fantázia szab határt bárkinek is abban, mivel bővíti a mau programnyelvet, vagy miként alakítja azt át... Például, ha valakinek nem tetszik az a névsorbarendezerési szisztéma, amit a **gs** utasítás művel, írhat sajátot. És akár épp a **g** karakterre is rákötheti, ami azzal kezd hogy leellenőri, utána az **S** karakter következik-e, ha nem, akkor syntax error, ha igen, akkor megrendezi a tömböt...

Hogy ez a képesség a mau programnyelvben mennyire hasznos, az mindjárt be is mutattatik egy konkrét példával! Ugyebár, más programnyelveket megvizsgálva, láthatjuk, hogy azokban akadnak mindenféle libraryk, köztük a stringek kezelésére is. És ezekben eszméletlenül sok mindenféle függvény van a stringek kezelésére, s ezek többségét eddig bizony nem tudta a mau nyelv! Nem véletlenül: ezek azért mégiscsak olyasmik, amik nem kellenek okvetlenül minden kis programhoz. Jó lenne azonban, ha adott esetben rendelkezésünkre állnának... Nos, írjunk akkor egy dinamikus könyvtárat a stringek hiányzó funkcióira! Legyen ennek neve mondjuk

`maulib_string.cpp`

amiből a lefordítás után a

`maulib_string.o`

objectfile képződik, majd ebből létrehozzuk a

`maulib_string.so`

shared libraryt!

E sorok írásakor e programkönyvtár még csak egyetlen függvényt tartalmaz, azt, ami egy stringben megkeresi egy másik string első előfordulását, és visszaadja annak pozícióját. Íme a program:

`vz@Csiszilla /Releases/2014/U/Common/vz/MAU=>cat maulib_string.cpp`

`// Shared string library a mau programnyelvhez`

```
#include "vz.h"
#include "mau.h"

#include <string.h>

mau_L mau_strstr(F &f) {
// Szintaxis:
// STRSTR x, #s@s, #s@g
// Visszaadja a #L@x:0 változóba azt az indexet, ahol a #s@s stringben megtalálta a #s@g stringet.
// Az x mau_c kifejezés lehet, a @s és @g pedig stringkifejezés.
// Ha nem talált egyezést, a változó tartalma -1 lesz.
mau_c x;MAUSTRING s;MAUSTRING g;
mau_L index;
// Beolvassuk a paramétereket:
x=(*f.MAU_mau_c)(f);
s=(*f.MAU_mau_s)(f);
g=(*f.MAU_mau_g)(f);
// Kiszámoljuk az indexet:
mau_c *pointer=(mau_c *)strstr((const char *)s.s,(const char *)g.s);
if(pointer==NULL) {index=-1;} else {index=(mau_L)(pointer - s.s);}
f.v[x].L[0]=index;
return 0;
}
```

A szintaxis leírásakor az „STRSTR” nem azt jelenti hogy e paranccsal hívható, csak utal a funkcióra. Nyilván olyan utasításokkal lesz meghívva, amire ráköjtük...

Na most ezt le kell fordítsuk. Igenám, de ha ezt a teszt-hez hasonló korábbi példa szerint fordítjuk, futáskor symbol lookup error hibaüzenetet kapunk, mert nem találja majd meg a kis aranyos librarynk a MAUSTRING típus konstruktorát (meg semmi mást se hozzá). A megoldás az, hogy le kell fordítani mellé a MAUSTRING osztály függvényeit is, és összeszerkeszteni vele. Ez azt jelenti, hogy e librarynkat e két paranccsal hozhatjuk létre:

```
g++ -c -fPIC maulib_string.cpp mausttring.cpp
g++ -shared -fPIC -o maulib_string.so maulib_string.o mausttring.o
```

Természetesen a mausttringosztály használatához kell a **mausttring.h** headerfájl is, de ez nem gond, mert a **mau.h** include állomány úgyis betölti magának.

Ezek után pedig így használhatjuk:

```
#!/mau
#s@L="/Releases/2014/U/Common/vz/MAU/maulib_string.so";
#s@f="_Z10mau_strstrR1F";
#s@g="Ebben a stringben keresek!";
#s@h="ing";

#g@g=?#g " " 'j; // Elmentjük a függvény régi címét
#M@U=@L; // objectlib megnyitása
if(?d) T "A dinamikus könyvtár nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX

[['j,U,@f]]; // Betöltjük a kereső rutint a 'j' karakterre
if(?d) T "A függvény nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX
// végrehajtás
"Végrehajtom:\n"
j i,@g,@h;
"Első string, amiben keresek: " ?s @g; /;
"Ezt keresem benne: " ?s @h; /;
if(#L(@i)<0) T "Nincs benne a stringben!\n"
E "A stringbeli pozíció: " ?L @i; /;
[['j=@g]]; // Visszatettük a függvény régi címét
[#M@U]; // objectlib bezárása
"Vége a proginak!\n"

XX
```

```
Eredménye:
Végrehajtom:
Első string, amiben keresek: Ebben a stringben keresek!
Ezt keresem benne: ing
A stringbeli pozíció: 11
Vége a proginak!
```

Na most, ez mind gyönyörű, de macerás mindig kideríteni, a függvényünknek a dinamikus könyvtárban mi lesz a neve, pláne mert élünk a gyanúperrel, hogy az egyes platformokon működő C++ fordítók eltérő kiegészítéseket raknak a függvényeink nevéhez, sőt az se kizárt hogy ugyanazon platformon is megváltozhat ez egyik pillanatról a másikra, amikor verziószámot ugrik a gcc vagy a g++. (Láttam én már ennél jóval nagyobb disznóságokat és inkompatibilitásokat is...). Na de szerencsére VAN MEGOLDÁS! Ez pedig egyszerűen csak annyi, hogy a dinamikus könyvtárba szánt függvényünk nevének elejéhez biggyesszük hozzá ezt:

```
extern "C"
```

Vagyis, a fenti példánkban így néz ki majd a függvényünk „eleje”, azaz a neve a forráskódban:

```
extern "C" mau_L mau_strstr(F &f) {
```

Ez igazán nem nagy ördögösség. Ezek után pedig hivatkozhatunk is rá a mau programban mindenféle más kiegészítés nélkül, az ő saját nevén. Ennek örömeire íme itt az iménti mau program, ezen név használatával újra:

```
#!mau
#s@L="/_P/Mau/-/lib/maulib_string.so";
#s@f="mau_strstr";
#s@g="Ebben a stringben keresek!";
#s@h="ing";

#g@g=?#g "_" 'j; // Elementjük a függvény régi címét
#M@U=@L; // objectlib megnyitása
if(?d) T "A dinamikus könyvtár nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX

[['j,U,@f]]; // Betöltjük a kereső rutint a 'j' karakterre
if(?d) T "A függvény nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX
// végrehajtás
"Végrehajtom:\n"
j i,@g,@h;
"Első string, amiben keresek: " ?s @g; /;
"Ezt keresem benne: " ?s @h; /;
if(#L(@i)<0) T "Nincs benne a stringben!\n"
E "A stringbeli pozíció: " ?L @i; /;
[['j=@g]]; // Visszatettük a függvény régi címét
[#M@U]; // objectlib bezárása
"Vége a proginak!\n"

XX
```

Mindez szép és jó, de macerás dolog azzal tökölni, hogy elmentsük a függvény eredeti belépési pontját! Szerencsére erre nincs is szükség. A mau interpreter rendelkezik az ő összes, mind a 256 tokenjéhez tartozó belépési pontjának a másolatával, amit nincs is módunkban piszkálni a magunk forrásprogramjából. Emiatt nagyon egyszerűen megmondhatjuk neki, hogy csinálja vissza valamelyik token jelentését az alapértelmezés szerintire! Erre szolgál ez az utasítás:

{[x]}

ahol is az „x” egy unsigned char típusú aritmetikai kifejezés, mely természetesen a token ASCII kódját határozza meg. Így a fenti program ennek megfelelően:

```
#!mau
#s@L="/_P/Mau/-/lib/maulib_string.so";
//#s@L="./maulib_string.so";
#s@f="mau_strstr";
#s@g="Ebben a stringben keresek!";
#s@h="ing";

#M@U=@L; // objectlib megnyitása
if(?d) T "A dinamikus könyvtár nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX

[['j,U,@f]]; // Betöltjük a kereső rutint a 'j' karakterre
if(?d) T "A függvény nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX
// végrehajtás
"Végrehajtom:\n"
j i,@g,@h;
"Első string, amiben keresek: " ?s @g; /;
"Ezt keresem benne: " ?s @h; /;
ha(#L(@i)<0) "Nincs benne a stringben!\n"
E "A stringbeli pozíció: " ?L @i; /;
{[j]}; // Visszatettük a függvény régi címét
[#M@U]; // objectlib bezárása
"Vége a proginak!\n"
j i,@g,@h; // syntax error-t kell dobnia itt

XX
```

Persze, azért azt jó tudni, hogy ez az „alapértelmezés visszaállító” utasítás is egy utasítás, s emiatt csak addig működik, amíg magát a „{” utasítást át nem definiáltuk valamelyik pluginunkban... Igaz viszont, hogy mert a ciklusok is mind a { karakterrel kezdődnek, ezért az már nagyon perverz fickó aki épp azt írja át. Nem hiszem hogy az ilyesmi szükséges lenne vagy pláne gyakori. De persze, attól még SZABAD...

A mau interpreter azonban (a 11. verziótól kezdve) úgy működik, hogy előbb megnézi, van-e utasítás definiálva neki a forráskód következő 2 karakterére, s ha nincs, akkor nézi csak meg, van-e utasítás legalább a következő egyetlen karakterre. (Ez bámulatos mértékben gyorsított a futásán...) Emiatt aztán logikus volt beletenni olyan lehetőséget is, hogy bármely 2 karakteres utasítástokenhez is rendelhessünk pluginben megírt utasítást (vagy valamely meglévőt átdefiniálhassunk). Ennek fényében nézzük az előző programot, 2 karakteres utasítástoken-átdefiniálásra átírva:

```
#!mau
#s@L="/Releases/2014/U/Common/vz/MAU/maulib_string.so";
#s@f="mau_strstr";
#s@g="Ebben a stringben keresek!";
#s@h="ing";

#g@g=?#g "2" "ha"; // Elmentjük a függvény régi címét
#M@U=@L; // objectlib megnyitása
if(?d) T "A dinamikus könyvtár nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX

[[["ha",U,@f]]; // Betöltjük a kereső rutint a '#' utasításra
if(?d) T "A függvény nem betölthető! A hiba oka:\n" ?s ?#s "d"; /; XX
// végrehajtás
"Végrehajtom:\n"
ha i,@g,@h;
"Első string, amiben keresek: " ?s @g; /;
"Ezt keresem benne: " ?s @h; /;
if(#L(@i)<0) T "Nincs benne a stringben!\n"
E "A stringbeli pozíció: " ?L @i; /;
[[["ha"=@g]]; // Visszatettük a függvény régi címét
[#M@U]; // objectlib bezárása
ha 4<3 "Ezt nem szabad kiírnia!\n"
ha 3<4 "Ezt ki kell írnia!\n"
"Vége a proginak!\n"

XX
```

Amint látható, itt a „**ha**” utasítást definiáltuk át ideiglenes jelleggel. Az átdefiniálás pedig a 3 db szegletes zárójellel történik, amint a függvény eredeti címének visszaállítása is. Itt nincs lehetőség a régi függvénycím automatikus visszaállítására, ugyanis ahhoz az kéne hogy minden lehetséges esetnek tárolja a címét, ami (64 bites gépek esetén) 65536\*8 bájt memóriát igényelne, ami teljes 1 megabájt. Ennek a pocsklásnak semmi értelmét nem láttam. Könnyen meg lehetne oldani, de MINEK?! A mi új extra, pluginben megírt függvényeinket feltehetőleg úgyis valami nem használt karakterpároshoz fogjuk hozzárendelni.

A felhasznált utasítások konkrét szintaxisa:

```
?#g "2" X;
```

A fenti függvény visszaadja egy #g típusú számként azt az értéket, ami azon utasítás címe, melynek 2 karakterét az X stringkifejezés tartalmazza.

```
[[[X,M,S]]]
```

ahol M egy mau\_c kifejezés, az S pedig egy MAUSTRING kifejezés. Az S a függvény neve az M-edik dinamikus könyvtárban. Az M természetesen a **#M** típusú mau változók megfelelő elemét jelenti. Az X pedig egy pontosan 2 karakterből álló stringkifejezés, melynek karakterei az utasítás tokenjét határozzák meg.

[[[X=g]]]

Itt az X jelentése ugyanaz mint fentebb, a „g” pedig egy #g típusú aritmetikai kifejezés. (célszerűen az, amibe elmentettük az utasítás eredeti címét).

## 31. fejezet - A logolás egy mau programban

A mau interpreter rendkívül sokféle logolási szintet támogat - pontosan 256 darabot, pontosabban csak 255-öt, mert a 0 azt jelenti hogy semmit se logol. Persze ha valami okból ki kell lépnie rendkívüli módon hibajelzéssel, akkor azt azért ez esetben is kiírja.

A logolási szintek azt jelentik, hogy van egy speciális tömb, az a neve hogy **logflags**, unsigned char értékekből, e tömb mérete 256, s ha a megfelelő sorszámú bájt értéke nem 0, akkor az ahhoz a szinthez tartozó üzeneteket kiírja egy **stdlog** nevű fájlba. Ez az stdlog fájl szokásosan megegyezik az stdout-tal, de magában a mau programban átirányíthatjuk máshová. A mau interpreter indulásakor az összes logflag értéke 0, kivéve a nullásat. De mert az összes többi 0, emiatt a nullás hiába nem 0, semmit se fog logolni. A nullás értéke csak amiatt nem 0, hogy ne kelljen külön ezt is állítgatni ha valamelyik logolási szintet be akarjuk kapcsolni.

A logflagok jelentése:

```
0: Ha ennek értéke 0, egyáltalán semmit se logol, hiába van bekapcsolva valami más logflag.
1: logol minden címkézett sort
2: logolja az alprogramok indítását és befejezését
3: logolja a főprogram indulását és befejezését
4: logolja a kiírási műveleteket
5: logolja a shell-parancsokat
6: logolja az ugróutasításokat
7: logolja az üres utasításokat
8: logolja a BREAK utasításokat ( „xx” )
9: logolja a logfájl átirányításait
```

Legalábbis ez a szép elmélet. Gyakorlatilag nem kizárt hogy valahova ahova a fentiek szerint kellett volna tennem logolást, elfelejtettem beírni a programba... Ha így van bocs és sorry!

Amint látható, a lehetséges 255 logolási szintnek csak elenyésző töredéke van kihasználva, azaz aki fejleszteni akarja a mau nyelvet, annak bőséges lehetősége van a saját log üzenetei számára kiválasztani valamely még szabad logolási szintet. Egy efféle log üzenet betétele a forráskódba nagyon egyszerű, csak ennyiből áll:

```
if(logflags[x]) L("ide kerül a log üzenet szövege");
```

És a fenti programsorban az „x” az a szám, ami a logolási szintet jelzi.



A logolás be- és kikapcsolására nem áll rendelkezésre parancssori opció, ellenben ezek dinamikusan kapcsolgathatóak a mau programon belül ki és be, a következő utasításokkal:

#### **LOG x,y**

A fenti utasítás az x-edik logflag értékét beállítja y értékűre. Az x és y egyaránt egy-egy unsigned char típusú aritmetikai kifejezés lehet.

#### **LOF S**

Az stdlog fájlt (amibe a log üzenetek kerülnek) átirányítja abba a fájlba, aminek a nevét az S stringkifejezés tartalmazza. Előbb természetesen megpróbálja megnyitni írásra az adott fájlt. Ha ez nem sikerül, a korábbi logfájlba beír erről egy méltatlankodó üzenetet, s a program futása elhalálozik. Amennyiben sikerül a fájl megnyitása, akkor - ha az előző logfájl nem az stdout vagy az stderr volt - azt bezárja, s az stdlog ezentúl ez az új fájl lesz.

#### **L00**

Az stdlog ezentúl az stdout lesz. Ha a korábbi stdlog nem az stdout vagy az stderr volt, azt előbb bezárja.

#### **LOE**

Az stdlog ezentúl az stderr lesz. Ha a korábbi stdlog nem az stdout vagy az stderr volt, azt előbb bezárja.

## **32. fejezet - Saját billentyűzetkiosztás készítése**

Két hobbym van: regények írása és a programozás. Sajnos, mindkettőhöz teljesen más billentyűkiosztás a jó. Ráadásul szeretem a wireless billentyűzetet, ami pedig kicsi, nincs rajta numerikus billentyűrész (keypad), ugyanis azon a részen egy tapipad van. Aminek nagyon örülök. Viszont így túl kevés a gomb. Már a szükséges karaktereknek is kevés, holott jó lenne pár extra módosítóbillentyű is, amik segítségével és az xbindkeys progival gyorsbillentyűkre köthetnék mindenféle gyakran használt progikat és funkciókat.

Tovább súlyosbítja a helyzetet, hogy amikor programozok, akkor nem elég az ha átváltok az angol billentyűkiosztásra, mert nagyon gyakori, hogy a programírás közben be kell gépelnem valami magyar szöveget, akár kommentbe, akár valami üzenet részeként, amit a program ír majd ki. Márpedig angol kiosztáson nincsenek magyar ékezetes betűk. Emellett az olyan gyakori karakterek is mint az írásjelek vagy műveleti jelek is teljesen másutt vannak a két kiosztásban.

Sokat töprengtem, hogyan oldjam meg a dolgot. A billentyűzetem olyan, hogy alapból angol feliratú. Ez a regényírás közben nem zavar annyiból, hogy rég megszoktam, hol vannak rajta a magyar ékezetes betűk. De zavar programozás közben, mert ott gyakran épp a mindenféle speciális karakterek kellenek (amik az írás közben szinte soha, például a []{}#\$ jelek), s ezek teljesen máshol vannak a magyar kiosztás szerint, mint ahova fel vannak festve.

Szerintem hasonló gondokba mindenki belefutott, aki gyakran használja a számítógépet gyökeresen eltérő funkciókra, illetve különböző nyelvi környezetekben.

A megoldás:

Saját billentyűkiosztást kell tervezni. Ez két szempontból nehéz. Egyrészt, az X billentyűzet-értelmezése bizánci bonyolultságú! Cseppet se könnyű megérteni, mit csinál és miért. Többször nekifutottam, míg valamennyire megvilágosodtam benne. Gyakorlatilag most is van bőven mindenféle rejtélyes baromsága előttem. Lényegében egy zavaros, áttekinthetetlen katyvasz. Az egész másról se szól, mint-hogy a billentyűzet küld a gépnek minden billentyű esetén egy kódot, s ezt aztán az X össze-vissza indexelgeti többszörösen, mindenféle táblázatok alapján. Másrészt, az se könnyű hogy megtervezzük, melyik karakter hova kerüljön.

Leírom lépésről-lépésre, én mit alkottam és miért.

A legfontosabb amivel kezdenünk kell, a módosítóbillentyűk meghatározása. Az interneten erről rengeteg cikk van, DE GYAKORLATILAG MIND NEMHOGY HIÁNYOS, DE MÉG TÉVES IS!

A dolog ugyanis úgy áll, hogy a módosítóbillentyűknek a nevét millióféle-képpen variálhatjuk, és szinte bármelyik billentyűt kinevezhetjük módosítóbillentyűnek, sőt ugyanazon névvel több billentyűt is elláthatunk — ámde van sajnos pár furcsa név, ami csak látszatra ugyanolyan módosítóbillentyű mint a többi, valójában kitüntetett szerepe van! Ugyanis azalatt hogy „módosítóbillentyű”, sajnos 2 teljesen külön fogalmat szoktak összemosni.

**1.** Azon módosítóbillentyűk, melyek meghatározzák, hogy a billentyűzet által küldött billentyűkódból az X milyen karakterkódot állítson elő.

**2.** Azon módosítóbillentyűk, melyeket azért nevezünk így, mert önálló lenyomásuk nem eredményez külön karaktert, sőt más billentyűvel együtt lenyomva se módosítja azon billentyű karakter-jelentését, de a programok ezen módosítóbillentyű állapotát lekérdezhetik, s ettől függően csinálnak valamit.

Ha a fenti magyarázat netán zavaros volt, elárulom hogy a valóság ennél is rosszabb, mert az X a maga xmodmap táblázata alapján dönti el azt is, melyik billentyű melyik módosítóbillentyűként értelmeztetik...

Mindez abszolút érthetetlen konkrét példa nélkül, úgyhogy vágjunk is bele.

Mindenekelőtt ki kell választani, melyik már létező kiosztásból induljunk ki. Kétféle az ami szokásosan szóbajöhet, a magyar vagy az angol. Én a magyarból indultam ki, beállítottam rá a disztrómat a

`setxkbmap hu`

paranccsal, majd e kiosztást kimentettem egy fájlba, eképp:

`xmodmap -pke > modmap`

A feladatunk ezután a következő lesz:

1. Szerkeszteni kell e modmap fájlt megfelelően
2. Létre kell hozni egy parancsfájlt a módosítóbillentyűk számára, ami az X minden indulásakor lefut
3. A fenti 2. pont érdekében, meg azért hogy tudomást vegyen az új modmap fájlról, szerkeszteni kell a \$HOME/.xinitrc fájlt.

Mint mondtam, mindenekelőtt tisztában kell lennünk a módosítóbillentyűk szerepével. Nos, ha belenézünk a modmap fájlba, ott efféle sorokat látunk:

```
keycode 17 = 8 parenleft multiply multiply asterisk asterisk dead_abovedot
```

Ha nálad nem pont ezek a fura szavak vannak a „keycode 17 =” kezdetű sorban, cseppet se aggódj! A lényeg, hogy mindegyik sor úgy kezdődik hogy **keycode**, s utána van egy szám, na ez az amit a billentyűzet küld a gépnek amikor lenyomjuk a gombot. Mondjuk még ez is csak elnagyolt fogalmazás, mert vannak trükkös progik amikkel még azt is külön lehet vizsgálni, a billentyű lenyomva van-e vagy már fel is engedték, stb... de ebbe most ne menjünk bele.

Na most látható, hogy minden keycodehoz pontosan 1 darab sor tartozik, amikben azonban változó számú paraméter szerepel. A paramétereket egymástól mindig szigorúan szóköz választja el, azaz olyan paraméter nincs, ami szóközt is tartalmazhatna. A paraméter vagy egy általunk jólismert karakter, vagy egyetlen számjegy, vagy valami fura név.

Egy sorban az egyenlőségjel után legfeljebb 8 különböző paraméter szerepelhet, többnyire azonban ennél jóval kevesebb van. Gyakori, hogy ugyanazon nevű paraméter több helyen is szerepel, akár egy soron belül, akár több sorban is itt-ott.

Amint mondtam, maximum 8 mező lehetséges egy sorban, azaz 8 paraméter. Mindegyik mező azt jelenti, ahhoz a billentyűzetkódhoz aminek a soráról épp szó van, milyen konkrét karakterkód rendeltetik, amennyiben a megfelelő billentyűt olyan módosítóbillentyűvel nyomták le, amely módosítóbillentyűt e mező reprezentál. Ezen mezők, pontosabban paraméter-pozíciók a következőket jelentik:

**1. mező** (ez természetesen a BALOLDALI első mezőt jelenti, azt ami a baloldali egyenlőségjel után közvetlenül van jobboldalt, azaz a mezők balról jobbra számozódnak!): Ide azon karakterkód kerül, mely akkor generáltatik, ha a megfelelő billentyűt bármiféle módosítóbillentyű nélkül, önmagában nyomjuk le.

**2. mező:** Ide az a kód kerül, ami a SHIFT-tel való lenyomáskor keletkezik.

**3. mező:** Ide az a kód kerül, ami akkor keletkezik, ha a billentyűt azzal a módosítóbillentyűvel együtt nyomjuk le, aminek a neve az, hogy **Mode\_switch**. Nem hallottál még róla?! Ecsém, ez tök ciki! Nem ismerni a Mode\_switchet, hí! Ember, hogy élhetsz ekkora szegénnyel... Na sebjaj, majd kicsit később felhomályosítalak felőle...!

**4. mező:** Ide az a kód kerül, ami a gombnak a Mode\_switch és a SHIFT együttes megnyomásával keletkezik.

**5. mező:** Az itteni kód akkor keletkezik, ha a gombot az úgynevezett **ISO\_Level3\_Shift** nevű módosítóbillentyűvel együtt nyomjuk meg. (De szép neve van, egyem a zuzáját annak a perverznek aki ezt elnevezte...)

**6. mező:** Az itteni kódot az ISO\_Level3\_Shift és a SHIFT együttes megnyomásával csíholhatjuk ki a billcsinkból.

Sajnos, a hetedik és nyolcadik mező kódjára nem tudtam rájönni, hogy az hogyan keletkezik. Gyötörtem a gépet órákon át, és eljutottam oda, hogy bizonyos roppant speciális beállításokkal el tudtam érni, hogy kiadja a NYOLCADIK mező kódját. A hetedikét azonban nem. És a nyolcadikat is csak úgy, hogy emiatt megváltozik az összes korábbi mező elérhetősége, azokat már más módosítóbillentyűkkel lehet ezután elérni. Illetve egy másfajta beállításnál elérhető az első, a második, a hetedik és nyolcadik mező kódja, de a közbeeső 3,4,5,6 mezőké nem. Fasz...

Szóval, úgy döntöttem, megelégszem az első 6 mezővel, ez is jó sok variáció. Megjegyezném, e nem teljes eredmény nem okvetlenül azt jelenti hogy én vagyok az idióta, ugyanis ha valamit nem értettem nagyon félre, hivatalosan csak AZ ELSŐ 4 MEZŐ kódjának elérhetősége garantált minden hardveren illetve implementációban! Általában azért biztosított az ötödik és hatodik mező is, ezért is van az, hogy ha belenézel valami „gyári” xmodmap fájlba, akkor abban a második 2 mező gyakorlatilag az első két mező ismétlése, mert hátha az ökör felhasználó összecseréli a shiftet a másik módosítóbillentyűvel. A ritkábban használt többi karaktert meg az ötödik és hatodik oszlopba rakják, mert szinte biztos hogy az is létezik majd az adott hardware- és szoftverkörnyezetben. Ez azonban sajnos tényleg csak *majdnem* mindig van így, és nagy szívás lehet belőle, hogy egyes esetekben nem tudjuk kicsiholni a gépből az oda eldugott hosszú í betűt mondjuk... (amit szokásosan az AltGr-J alá rejtene).

A hetedik és nyolcadik oszlop azonban már tényleg abszolút semennyire sem garantált. Arra jobb ha nem számítunk, kb olyan mintha nem is lenne.

Szóval, elégedjünk meg az első hattal. Látjuk, hogy itt három titokzatos billentyűt kell meghatároznunk, ezek a SHIFT, az ISO\_Level3\_Shift és a Mode\_shift.

Na most ezek meghatározása az utóbbi kettő esetén úgy megy, hogy e fura neveket beírjuk a táblázatba valahová, olyan keycode sorba, amihez hozzá szeretnénk rendelni. Egyszerre több helyre is beírhatjuk őket, ezesetben többféle-képp is elérhetjük az adott funkciót. A SHIFT esetében ez kissé bonyolultabb, annyiból, hogy abból nincs olyan hogy SHIFT, ezt csak én nevezem így, tehát nem azt kell beírni hogy SHIFT. Hanem ebből elvileg lehet kettőfajta is, melyek a Shift\_L és Shift\_R nevekre hallgatnak. Nem véletlenül, mert ezek szokásosan a baloldali illetve jobboldali Shift feliratú gombokhoz vannak rendelve. De te természetesen változtathatsz ezen, ha akarod.

Fontos tisztázni, hogy miként derítheted ki, billentyűzeted melyik gombjához miféle keycode tartozik! Ehhez neked az xbindkeys nevű progira van szükséged. Ezt el kell indítsd parancssorból, így:

**xbindkeys -k**

erre megjelenik egy fehér ablakocska, ekkor nyomd meg a téged érdeklő gombot, mire efféle üzenetet ír ki neked:

```
Press combination of keys or/and click under the window.
You can use one of the two lines after "NoCommand"
in $HOME/.xbindkeysrc to bind a key.
"NoCommand"
  m:0x2000 + c:108
  Mode_switch
```

A fenti üzenetet a jobboldali Alt gomb, más néven AltGr megnyomása után kaptam. Ebből téged a **c:108** rész kell érdekeljen, ez jelenti azt, hogy e gombhoz tartozó mezőket a

keycode 108 =

kezdetű sorban kell meghatározni.

E gombból úgy lesz egy **Mode\_switch** nevű módosítóbillentyű, hogy a megfelelő keycode sorba beírjuk azt, hogy Mode\_switch, legalábbis a legelső mezőbe okvetlenül. Oda muszáj. Őszinténszólva nem vagyok biztos benne, van-e bármi jelentősége annak, ha a többi mezőbe is beírjuk; fene se tudja mit csinál a rendszer, hogyan értelmezi azt, ha egy módosítóbillentyűt más módosítóbillentyűvel együtt nyomunk le! A legtisztább ha vagy csak a legelső mezőbe írjuk bele, vagy mindegyikbe. Fontos tudni, hogy nem kötelező kitölteni a sor mind a nyolc mezőjét, a vége felé levőkből bárhányat el lehet hagyni. E téren hogy kifejezetten épp egy módosítóbillentyűt beírjunk-e egynél több mezőbe, bevallom nem merem magamat nagyon okosnak és az isteni bölcsesség tudorának feltüntetni, mert napok óta ezzel a billentyűkérdéssel kísérletezem, és olyan észvesztő, meglepő, hülye logikátlanságokat tapasztaltam e téren a drága Xorg részéről, hogy bármiféle aljasságot is habozás nélkül elhiszek róla ezek után, az első szóra. Azaz, tartok tőle hogy e kérdés meglehetősen hardware- és implementációfüggő. (Erre való utalásokkal bőven találkoztam mindenféle leírásokban is). Ezalatt azt értem, hogy már maga az a tény is számít, egyrészt miféle gyártótól van a billentyűzeted, aztán az, melyik verziójú xorgot használsz, sőt az is, hogy a linuxod, vagy legalábbis az X indulásakor annak xorg.conf fájljában – már ha van neki olyan egyáltalán az adott disztróban – mit mondasz neki, milyen típusú a billentyűzeted eredetileg, pld pc104, vagy ha épp hu akkor hány gombos, meg miféle „variant”, stb. Eszméletlen kavargások lehetségesek e dolgokból. Én tehát leírom hogy magam hogyan csináltam hogy nekem jó legyen, közreadom a megfelelő fájlokat is, de ne végy mérget rá hogy ez nálad istenbizony tutira jó lesz, lehet hogy kell majd vele kísérletezned.

Miután ezzel a **Mode\_switch**-cel megvagyunk, hasonlóképp meg kell határoznunk, melyik gomb legyen az **ISO\_Level3\_Shift** módosítóbillentyű, valamint a **Shift\_L** és **Shift\_R**. Utóbbi kettőből bármelyiket el is hagyhatjuk, ha úgy döntünk, nekünk elég egyetlen shift gomb is.

Illik hogy meghatározzuk azt a két gombot is, melyek neve **Control\_L** és **Control\_R**.

Na most, a táblázatban megadott mindenféle nevekből csoportokat képezhetünk, azaz egyes szimbólumoknak adhatunk közös neveket! Ez úgy megy, hogy ha én parancssorban kiadom az

xmodmap

nevű parancsot, akkor ezt írja ki nekem:

xmodmap: up to 3 keys per modifier, (keycodes in parentheses):

```
shift      Shift_L (0x32)
lock
control    Control_L (0x25)
mod1       Alt_L (0x40),  Alt_L (0xcc)
```

```

mod2      Hyper_L (0x4d), Hyper_L (0xb4), Hyper_L (0xcf)
mod3      Super_R (0x3e), Super_R (0x86)
mod4      Super_L (0x85), Super_L (0xce)
mod5      ISO_Level3_Shift (0x42), ISO_Level3_Shift (0x5c)

```

E fenti kis lista első oszlopa felsorolja a rendszeremben használatos azon szimbólumokat, melyek úgynevezett „módosítóbillentyűk”-nek vannak tekintve, s utánuk fel van tüntetve, e szimbólumok mely konkrét, a keycode-sorokat tartalmazó táblázatban szereplő karakterkódokat jelentik. Például látható, hogy nálam van olyan módosítóbillentyű, melynek az a neve hogy „control”, s ehhez tartozik egy olyan kód melynek neve a keycode-táblázatban Control\_L, és ott van zárójelben hogy ennek értéke (0x25). Ez egy hexadecimális szám, értéke decimálisan  $2 \cdot 16 + 5$ , azaz  $32 + 5$ , azaz 37, tehát e Control\_L billentyű a táblázat 37-es sorában lett meghatározva, az a gomb fogja ezt a controlkodást művelni, aminek a billentyűkódja 37.

Itt most csak ezt az egy billentyűt foglaltam bele abba a csoportba, aminek a neve az hogy „control”. Megtehettem volna hogy más billentyűt is kinevezek controlnak, de később ismertetendő okokból nekem elég egyetlen. Vannak más csoportok is, például olyan nevekkkel hogy **shift**, **lock**, **mod1**, **mod2**, **mod3**, **mod4**, **mod5**. Ennél több sajnos nem lehetséges. És e csoportnevek is rögzítettek. Próbáltam trükközni, és definiálni mod6 nevű csoportot, de a rendszer hevesen tiltakozott azon nyomban, hogy mit képzelek én, nem úgy van ám az!

Na most ha már megvan a magunk xmodmapos táblázata, rendesen kitöltve a keycode kezdetű sorokat mindazzal a sok roppant okossággal amit beleálmodtunk, akkor jöhet e csoportok meghatározása, erre kell nekünk egy kis szkript. Legyen a neve mondjuk stílszerűen „modkeys”, és a tartalma az esetemben ez:

```
#!/bin/bash
```

```

xmodmap -e 'clear shift'
xmodmap -e 'clear Lock'
xmodmap -e 'clear control'
xmodmap -e 'clear mod1'
xmodmap -e 'clear mod2'
xmodmap -e 'clear mod3'
xmodmap -e 'clear mod4'
xmodmap -e 'clear mod5'
xmodmap -e 'add shift = Shift_L'
xmodmap -e 'add control = Control_L'
xmodmap -e 'add mod1 = Alt_L'
xmodmap -e 'add mod2 = Hyper_L'
xmodmap -e 'add mod3 = Super_R'
xmodmap -e 'add mod4 = Super_L'
xmodmap -e 'add mod5 = ISO_Level3_Shift'

```

Ez a következőt csinálja:

A „clear” szöveget tartalmazó sorok törlik a billentyűcsoportok esetlegesen meglevő korábbi definícióit. Majd utána az „add”-ot tartalmazó sorokban szépen sorra meghatározzuk, hogy mely csoportba mely nevű billentyűk tartozzanak. Minthogy a keycode sorokba a táblázatban ugyanazt a nevet többször is beírhatuk, ezért e fenti szkriptben egy név beírása – mondjuk a Super\_L-é – egyszerre több konkrét gombot is belerakhat a megfelelő, jelen esetben a mod4 nevű csoportba.

Hagyományból egyes programok feltételezik, hogy a rendszerben létezik **shift** és **control** nevű módosítóbillentyűcsoport, sőt még olyan is hogy **mod1**, és ezen utóbbi mod1, ez okvetlenül azonos az **Alt** billentyűvel.

Szemfüles Olvasóim biztos észrevették eddigre, hogy hiszen eszerint nekem csak egyetlen Shift és egyetlen Ctrl billentyűm van illetve lesz ami működik, miért jó nekem az hogy a többiről lemondok?

Azért, mert kevés a billcsimen a gomb. Nem engedhetem meg a luxust, hogy több fizikailag különböző gomb is ugyanazt a szoftveres funkciót lássa el, ez értelmetlen pazarlás. És tök felesleges is, mert jobbkezes vagyok, emiatt mindig úgy volt nálam hogy örökké csak a bal shiftet és bal ctrl-t használtam. Azaz a jobboldali efféle gomboknak simán definiálhatok más feladatot.

Most tehát ez úgy van nálam, hogy a rendszeremben a következő módosítóbillentyűk élnek:

**shift** = ez nálam a baloldali shift. Funkciója a szokásos.

**control** = ez is a baloldali Ctrl gomb, ennek is a megszokott a funkciója.

**mod1** = ez a baloldali Alt gomb, ennek is a megszokott a funkciója.

A fenti 3 csoportot ugyanis jobb nem piszkálni, mert a mindenféle programok erősen számítanak rájuk. Itt tehát ezeknél csak annyit tettem, hogy mindegyik csoportból kiszedtem a jobboldali párjukat.

Most jönnek a nyalánkságok! Vannak ugyanis még nálam a következő csoportok:

**mod2** = ez a keycode táblázatban a Hyper\_L néven szerepel, és e nevet ahhoz a keycodehoz írtam be, melyet a wireless billentyűzetem tetején levő Home nevű gomb produkál, e gombra egy házacskva van rajzolva. Ez természetesen nem azonos a megszokott „Home” gombbal, ami van minden billentyűzetten, semmi köze hozzá.

**mod3** = Super\_R — ez a jobboldali Shift feliratú gombra van mappelve nálam.

**mod4** = Super\_L — Na ez nálam a bal oldali Windows® gomb hogy egyem a lelkét... utálok ránézni is, de még nem találtam pingvines matricát rá való méretben...

**mod5** = ISO\_Level3\_Shift. Na ez van nálam a CapsLock gombhoz hozzárendelve...

**Mode\_switch** = Jobboldali Alt gomb, más néven AltGr.

Bár elvileg létezik olyan nevű csoport is hogy **lock**, de ahhoz nem rendeltem semmit, utálok ha megnyomok valamit aztán onnantól örökké nagybetűvel ír.



Kérlek gondolkozz most el e kiosztás nagyszerűségén! Tele vagyok jól használható, szabadon bűtykölhető módosítóbillentyűkkel. A progik csak a shift, ctrl és alt gombokra számítanak, esetleg nagy ritkán a windowsra. A többit a magam kedve szerint használhatom. De az igazi nagy zsenialitás szerintem nem is ez e kiosztásban, hanem az, ami pedig első pillantásra talán nem is olyan szembeötlő! Ez pedig az, hogy nálam teljesen külön billentyűre került a **Mode\_switch** és az **ISO\_Level3\_Shift** !

Ugyanis a legtöbbször az úgy van, hogy amint azt láthatod is szinte bármelyik eredeti xmodmapos fájlban, a harmadik és negyedik mező az elsőnek és másodiknak a megismétlése. Az ezutáni, ötödik és hatodik oszlopban vannak azok a karakterek, amiket szokásosan az AltGR, vagyis a jobboldali Alt lenyomásával érhetisz el. Ez amiatt érdekes és szomorú, mert mint e doksi legelején írtam, ezen ötödik és hatodik oszlop elvileg arra van, hogy ezeket az ISO\_Level3\_Shift lenyomásával érjük el! Vagyis ez azt jelenti, hogy szokásosan a billentyűkiosztások készítői ezen ISO\_Level3\_Shift szimbólumot mintegy egybemossák a Mode\_switch szimbólummal, gyakorlatilag a Mode\_switch látja el az ISO\_Level3\_Shift feladatkörét. (vagy a franc se tudja, de nem izgatom magamat emiatt. Nyilván számtalan módon elérhetik ennek megrövidítését). A lényeg, hogy ők csak látszatra használnak maguk is 6 oszlopot a táblázatból, gyakorlatilag csak 4 az amit ténylegesen használnak, hiszen az első kettővel azonos a harmadik és negyedik, és hát ez szerintem pocséklás.

Ahhoz hogy ez megváltozzék, kénytelen voltam e két szimbólumot, a Mode\_switch-et és az ISO\_Level3\_Shift-et szigorúan különválasztani. Ennek nyilvánvalóan az a veszélye, hogy ha valami fura hardveren mégsem érhető el az ötödik és hatodik oszlop, akkor ráfaragtam. Ennek esélye azonban csekély, s akkor legfeljebb visszatérek a „szabványos” kiosztáshoz.

Persze hogy örömem ne legyen teljes, a drága X azért megköti a kezemet. Mert mi is az első logikus ötlet: ugye az, hogy a mod névkezdetű módosítóbillentyűket hagyjuk meg mindenféle extra funkcióra, s legyen egy külön billentyű ISO\_Level3\_Shift néven, ami majd előállítja nekünk a megfelelő karaktereket! Így eggyel több lehet a módosítóbillentyűnk!

Hát nem. Akármivel is próbálkoztam ennek érdekében, annak vége az lett, hogy a rendszerem a legváratlanabb hülyeségeket produkálta. Még olyat is akár, hogy az xmodmap esetén kiírt táblázatban nem is szerepelt módosítóbillentyűként egy billentyű, de attól még halálnyugodtan úgy működött. (this is maybe a bug?) Vagy hogy azt gondolta a CapsLock, hogy ő ezentúl nem csak egy ISO\_Level3\_Shift gomb, de ellátja még a control gombok feladatát is, mert miért is ne. Hiába nem mondom neki ilyesmit, de ő jobban tudja, okosabb nálam. Szerinte az úgy a jó. Szóval ez a fura ISO\_Level3\_Shift micsoda úgy tűnik csak akkor műxi magát korrektül, ha egyben ki van nevezve valami mod-os módosítóbillentyűnek is.

Na most hogy melyik karaktert hova pakoljuk, mert ugye innen indult ki az egész. Hát kéremalássan, ugye nem véletlen indultam ki a magyar kiosztásból. Többnyire azt használom, de nekem belőle főként csak az ékezetes karakterek kellenek. Szóval az eredeti hu kiosztás az alap, ezt változtatjuk meg. Mégpedig a következőképp:

Marad minden betű a helyén ami az első és második mezőben szerepel a keycode sorokban. Ez azt jelenti hogy amit simán vagy shiftelve nyomunk le, nem változik, így ugyanúgy érhetjük el az összes kis- és nagybetűt. A J betű sorába, ami a **keycode 44** sor, oda a harmadik mezőbe írjuk be hogy iacute, a negyedikbe meg hogy Iacute, különben nem lesz hosszú í és Í betűnk, ahol megszoktuk. Ne feledjünk ugyanis hogy az AltGr most nem az ötödik és hatodik, hanem a harmadik és negyedik oszlopokért felelős! Célszerű ezen iacute és Iacute szavakat beírni az I betű harmadik és negyedik mezejébe is. A számjegyeket is ugyanúgy érhetjük el, és azon karaktereket is, melyek a magyar kiosztás szerint a számjegyek shiftelt megnyomására jelennek meg, s amik persze tök mások mint amik egy angol billcsi esetén e helyre fel vannak rajzolva. De ezt célszerű így hagyni, mert hátha megszoktuk ezek helyét már az eredeti magyar kiosztáson.

Na most jön a nagy okosság! Az ötödik és hatodik oszlopot ugye az ISO\_Level3\_Shift segítségével érhetjük el, az ötödiket e gomb, a hatodikat meg e gomb plusz a shift lenyomásával. Ezen ISO\_Level3\_Shift billentyű pedig jelen esetben nem más mint a CapsLock. Abszolút jól kézreáll, pláne egy jobbkezes embernek, aki a bal kezét használja a módosítóbillentyűk számára, s a jobbal gépel főleg. Én tehát azt tettem, hogy a megfelelő gombok soránál az ötödik mezőbe felvettem sorra azokat a karaktereket amik az angol billentyűzeten azon a gombon szerepelnek (a betűket nem, tehát az a-z karaktereket nem, mert minek), a hatodik oszlopba meg azokat, amik az angol billentyűzeten szintén e gombon szerepelnek de felül, amit ott tehát shifttel lehet elérni! Ez azt jelenti, hogy ha látom a billentyűzetemen (ami angol) hogy valami speciális karakter hova van rajzolva, azt simán el tudom érni, úgy, hogy ha alulra van rajzolva akkor a CapsLockkal nyomom meg, ha pedig felülre van rajzolva, akkor a Shift+CapsLockkal, s e kombináció könnyen lenyomható, akár még egyetlen ujjal is, mert közvetlenül egymás alatt van a CapsLock és a baloldali Shift. Legalábbis nálam a LenovoThinkPad T530-as laptop esetében, meg a Logitech K400-as wireless billentyűzeten. Ez messze sokkal könnyebb, mint mondjuk programírás közben váltogatni a billentyűzet-kiosztást, vagy ha valami levelet írunk magyarul, akkor közben emlékezni kell rá hogy merre is van a kérdőjel vagy a felkiáltójel, vagy más akármi, mert csak abban az egyben lehetünk biztosak, hogy tutira nem ott van ahova fel van rajzolva (ha nem magyar a billentyűzetünk). Így most garantáltan minden speciális jel ott van, ahova fel van rajzolva, épp csak ezeket a CapsLock vagy a Shift + CapsLock segítségével érhetjük el.

Megjegyzendő, hogy legalább alul a pont és vessző karakterek ugyanott vannak az angol és magyar kiosztáson is egyaránt, így azokkal nincs gond.

Minthogy a számjegyek fölötti speckó jelek is „felülre” vannak rajzolva ugyebár, ezért ezeket is logikusan a Shift+CapsLockkal érhetjük el, be is vannak téve e helyre a táblázatba mert tiszteljük a logikát, de ugyanakkor belevettem ugyan-ezeket az ötödik oszlopba is, hogy shift nélkül, sima CapsLockkal is elérhetőek legyenek. Nem sok értelmét láttam ugyanis annak, hogy a sima CapsLockos nyomkodás ezen gombok esetében a számjegyeket eredményezze, mert azok oly gyakoriak, hogy azokat minden épeszű ember úgymint mindenféle módosítóbillentyű lenyomása nélkül akarja majd beütni, s így a számok fölötti jelekhez nem kell a

shiftet is nyomkodni, elég egyszerűen magát a CapsLockot. E szimbólumok úgys gyakoriak nagyon a programozásban.

Na most az eképp elkészített táblázatot mentsük el a \$HOME könyvtárunkba .myxmodmap\_hu néven, s készítsünk egy megfelelő .xinitrc fájlt, amiben szerepelnek a következő sorok:

```
setxkbmap hu
xmodmap $HOME/.myxmodmap_hu
modkeys
Keymap --
```

A **setxkbmap hu** sor amiatt kell, mert ha ezt nem adjuk ki, akkor általam ismeretlen okoknál fogva (valószínűleg valami olyasmi amit a legelején taglaltam, hogy az X nem tudja miféle hardwerre számítson, de ez csak tipp a részemről) a kiosztásunk hiányosan fog működni, azaz nem funkcionál majd jól a CapsLock mint mod5.

Az xmodmap-os sor tölti be a táblázatunkat, a modkeys pedig az a szkript amit korábban már fentebb közöltem, ami beállítja a megfelelő módosítóbillentyűket.

Hátravan még e titokzatos sor:

```
Keymap -
```

Nos ez egy kis szkript a részemről, aminek tartalma a következő:

```
#!/bin/bash

KEY="hu"

if [[ $1 == "--" ]] ; then
cp /_P/Szkriptjeim/-/mysettings/aktkeymap.txt /Mount/RAM
exit
fi

if [[ $1 != "" ]] ; then
KEY=$1
fi

echo $KEY > /Mount/RAM/aktkeymap.txt

setxkbmap hu
xmodmap $HOME/.myxmodmap_$KEY
modkeys

exit
```

Mit is csinál ez? Elárulom, a benne megemlített aktkeymap.txt fájl tartalma mindössze ennyi:

hu

Nos e szkript ha a -- paraméterrel hívjuk meg, e fájlt átmásolja egy speckó könyvtárba, ami történetesen a ramdiszken található, de ez úgy különben nem igazán fontos. A lényeg hogy ha nem e -- paraméterrel hívjuk meg, akkor a para-

méterét is beleírja e fájlba, és beállítja a \$HOME könyvtárunk azon .myxmodmap fájljára a billentyűzetkiosztást, aminek a neve végén a kapott paraméter szerepel! Ez amiatt lényeges, mert eddig elsunnyogtam azt a döntő, lényegi kérdést, mire is használok a jobboldali Ctrl billentyűmet. Nos, ezt kivettem a módosítóbillentyűk köréből, s elneveztem úgy, hogy **F22**. Ha ugyanis nem tudnád, azok az általad kívánt héber, arab, vietnámi, orosz vagy akármilyen más karakterek amiket be szeretnél tenni a billentyűzetkiosztásodba, azok milyen névre hallgatnak, mit írnál bele a magad xmodmap fájljába rájuk hivatkozandó, akkor keresd meg a rendszeredben a **keysymdef.h** fájlt, abban megtalálod. (Az Xproto vagy valami hasonló nevű progicsomag része, ami pedig az X szerver része). Épp csak ott mindegyiknek a neve úgy kezdődik, hogy **XX\_**, na neked a név ez után következő része kell.

Na és hát látom a keysymdef.h fájlban hogy elvileg van egy rakás további F-es funkcióbillentyű-szimbólum, nemcsak 12-ig mennek azok! Felhasználtam az F22 nevűt, s ezt tettem a jobboldali Ctrl helyére. Nekem eredetileg úgyis csak F12-ig mennek a funkcióbillentyűk, s nem is találkoztam olyan billentyűzettel, amin ennél több lett volna. Most van F22 is már nekem. Eképp e billentyűre hivatkozhatok az xbindkeys progival, s meg is tettem ezt. A magam \$HOME/.xbindkeysrc fájljában szerepel e rész:

```
# saját hu keymap
"Keymap hu"
F22
```

```
# saját eo keymap
"Keymap eo"
Control + F22
```

Ez azt jelenti, hogy ha simán megnyomom ezen F22-nek kinevezett jobboldali Ctrl gombot, akkor átvált a \$HOME/.myxmodmap\_hu kiosztásra. Ha a (baloldali)Ctrl-lel együtt nyomom meg, akkor a \$HOME/.myxmodmap\_eo nevű kiosztásra vált át. Hasonlóképp még sok más kiosztást is definiálhatnék. Lehetne olyan amit rádefiniálok a következő kombinációkra:

```
mod1 + F22
mod2 + F22
mod3 + F22
```

stb, sőt ha telepítjük az **xchainkeys** progit is, na akkor aztán a lehetőségeink végtelenek...

Eképp tehát lehet több saját billentyűkiosztásunk is.

Az eo nevűt arra használok, hogy benne szerepelnek az eszperantó nyelv úgynevezett „kalapos” betűi is, melyeket az AltGr illetve Shift+AltGr segítségével érhetek el, a következőképpen:

Az eszperantó betű	A betű neve	Billentyűzetkombináció
ĉ	ccircumflex	AltGr C
Ĉ	Ccircumflex	Shift+AltGr C
ĝ	gcircumflex	AltGr G
Ĝ	Gcircumflex	Shift+AltGr G

ĥ	hcircumflex	AltGr H
Ĥ	Hcircumflex	Shift+AltGr H
ĵ	jcircumflex	AltGr J
Ĵ	Jcircumflex	Shift+AltGr J
ŝ	scircumflex	AltGr S
Ŝ	Scircumflex	Shift+AltGr S
ŭ	ubreve	AltGr U
Ŭ	Ubreve	Shift+AltGr U

Mindez valójában csak amiatt történt hogy legyen mivel kipróbálnom, működik-e az elképzelésem a saját billentyűkiosztások közötti váltásról, sok értelme nincs, mert ugyanezen betűk simán elférnének ugyanezen helyeken a magyar (azaz hunak nevezett) kiosztásban is. Azonban annak aki sokat használ egyszerre több olyan nyelvet is, melyekben nagyon sok a magyarétól eltérő karakter szerepel, pld orosz, görög, héber, koreai, japán, arab, stb, annak számára e lehetőség kincset ér.

Még elárulom az is, hova tettem pár a magyaroknak fontos különleges karaktert:

<b>A karakter magyar neve</b>	<b>A karakter</b>	<b>A karakter kódneve</b>	<b>Billentyűkombináció amivel elérhetjük</b>
Nyitóidézőjel	„	doublelowquotemark	AltGr + 9
Záró idézőjel	”	rightdoublequotemark	AltGr + 0
Három pont	...	ellipsis	AltGr + pont
Gondolatjel (félkvirtmínusz)	–	endash	Shift + AltGr + ü
Hosszú kötőjel (kvirtmínusz)	—	emdash	AltGr + ü
Szorásjel	×	multiply	AltGr + 8
Paragrafusjel	§	section	Shift + 0
duplajobbranyíl	»	guillemotleft	Shift+AltGr+pont
duplabalranyíl	«	guillemotright	Shift+AltGr+vessző
pluszmínusz	±		AltGr + ó
	÷		AltGr + /
	ı		AltGr + 1
	ı		Shift+AltGr+ /
	ı		Shift+AltGr+1
	£		CapsLock+f
	¢		CapsLock+c
	€		CapsLock+u

Na és amikor a Keymap szkript átvált a megfelelő xmodmap fájlra, eltárolja a beállított keymap nevének kódját. Ez arra jó, hogy valami okossággal ezt ki lehet jelezni folyamatosan a képernyő statusbarjára, hogy tudjuk mivel dolgozunk épp, ne kelljen találgatnunk. De ez egy másik cikk témája lesz majd.

Befejezéskeppen egy emlékeztető, hogyan módosíthatjuk a táblázatokban a megfelelő keymap kezdetű sorokat, melyik mezőbe írt karaktert melyik módosítóbillentyűvel érhetjük el ezentúl:

Egy példa:

keycode szöveg	billentyű kódja	=	Önmagában	Bal Shift	AltGr	Shift + AltGr	CapsLock	Shift + CapsLock
keycode	44	=	j	J	jcircumflex	Jcircumflex	iacute	Iacute

A vonatkozó szkriptek és táblázatok letöltési linkjei:

(a myxmodmap\_hu és myxmodmap\_eo fájlok természetesen letöltés után átnevezendők úgy, hogy .myxmodmap\_hu és .myxmodmap\_eo).

[http://parancssor.info/kiss/myxmodmap\\_eo](http://parancssor.info/kiss/myxmodmap_eo)

[http://parancssor.info/kiss/myxmodmap\\_hu](http://parancssor.info/kiss/myxmodmap_hu)

<http://parancssor.info/kiss/modkeys>

<http://parancssor.info/kiss/Keymap>

<http://parancssor.info/kiss/aktkeymap.txt>

Ezek után azt hiszem semmiféle akadály nem lehet annak, hogy mindenki megcsinálja a maga számára az ő szívének kedves billentyűzetkiosztást, amiben szerepel az összes, a mau nyelven programozáshoz szükséges karakter is!

## 33. fejezet - Esettanulmány: egy mau program felgyorsítása

Szeretett mau programnyelvünk ugyebár lehetőséget nyújt bencsmarkok készítésére, azaz „megkukkolhatjuk”, egy-egy program vagy programrészlet mennyi idő alatt fut le. Ezt összevethetjük azzal, ugyanazon feladatot elvégző más programnyelven megírt programok mennyi ideig futnak.

Nos, az efféle összevetésre jó példa lehet, ha programot írunk arra, hogy kiszámolja a pi értékét valamilyen pontossággal a Leibniz-formula szerint!

Mindenekelőtt készítsük el e program „legeredetibb”, C nyelvű változatát! A C ugyebár compiler típusú programnyelv, azaz lefordul binárisra, s így majdnem olyan gyors lesz, mintha „gépi kódban” írtuk volna meg azt. Lemérjük így mennyi ideig tart, s ez lesz az „etalon” amihez viszonyítunk majd azontúl! A sebességteszt egy Lenovo ThinkPad T530-as laptopon lett mérve, Core i7 processzorral, 16 giga RAM mellett, 64 bites Linux operációs rendszeren, a GCC 4.8.1-es fordítóprogramja lett használva a fordításhoz. (A mau interpreter esetén is).

A C nyelvű program:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
```

```

int main(void) {
double p; // összegváltozó
int n; // nevező
int s; // számláló
unsigned long long i;
clock_t CLOCK1;
clock_t CLOCK2;

p=0;
n=1;s=1;
CLOCK1=clock();
for(i=0;i<3000000; i++) {
p=p+(double)s/(double)n;
n=n+2;
s=(-1*s);
}
CLOCK2=clock();
p=p*4;
printf("idő: %g\n", (double)CLOCK2 - (double)CLOCK1);
printf("Pi közelítés: %g\n",p);

return 0;
}

```

6 futtatás után a kijelzett időszükséglet átlaga:

27012

(Ez és minden további időadat e fejezetben ha mást nem írok mellé, milliomod másodpercet jelent).

Ha a fenti programot a -O2 optimalizációs flaggal fordítom, akkor 6 futás átlaga **24925** lesz. Ezt vesszük alapnak, mert a mau interpreter is -O2 -vel van fordítva. Ez mintegy 8 százalékos javulást jelent e fenti program esetén.

Készítsük el a fenti feladatot ellátó program első változatát mau nyelven:

```

#!mau
// kiszámoljuk a pi-t a leibniz formulával.
#t@a; // inicializálunk egy időváltozót
#d@p=0; // Az összegváltozó
#L@n=1; // A nevező
#L@s=1; // A számláló
{| 3000000
#d@p=(@p)+(#L@s)/(#L@n);
#L@n=(@n)+2;
#L@s=-@s;
|}
#t@b; // inicializálunk egy másik időváltozót
#g@x=(#t@b)-(#t@a);
"Idő: " ?g @x; /;
#d@p = 4*#d@p;
"Pi/4 közelítés: " ?d @p; /;

```

A futási idők átlaga: **1002710**. Az arány az etalonhoz képest:

$1002710/24925 = 40.2291$

Azaz, a mau program 40-szer lassabb. Hm... Fel kéne gyorsítani! Az első ötletünk az, hogy minek a 3 milliószor lefutó ciklus belsejébe minden sor után pontosvessző?! Nyilván az is idő amíg azt a plusz karaktert feldolgozza a program... Ki vele! Azokról a pontosvesszőkről beszélek, amiket a fenti programlistában kékeszöld színnel emeltem ki. Ezek után a mau program (átlagban) **970468** ideig fut. Ez az előző változathoz képest 3.3% javulás! Nem azt mondom hogy világrengető siker, de már valami. Végeredményben e 3.5 százalékot csak 3 pontosvessző megspórolásával értük el, azaz durván 1% javulást jelent minden felesleges pontosvessző eliminálása! Megfontolandó tapasztalat...

Ezután az jut eszünkbe, hogy a nevezőt kettővel megnövelő programsort cseréljük ki 2 db inkrementáló utasításra! Ezesetben az ezek előtti sor végére vissza kell tenni a pontosvesszőt. A program ekkor így néz ki:



```

#!mau
// kiszámoljuk a pi-t a leibniz formulával.
#t@a; // inicializálunk egy időváltozót
#d@p=0; // Az összegváltozó
#L@n=1; // A nevező
#L@s=1; // A számláló
{| 3000000
#d@p=(@p)+(#L@s)/(#L@n);
++Ln
++Ln
#L@s=-@s
|}
#t@b; // inicializálunk egy másik időváltozót
#g@x=(#t@b)-(#t@a);
"Idő: " ?g @x; /;
#d@p = 4*#d@p;
"Pi/4 közelítés: " ?d @p; /;

```

A futásidők átlaga **766118**. Ez az előzőnek csak 79%-a! Azért ez már nem semmi nyereség...

Ezután az ötlik az eszünkbe, hogy a két különálló inkrementáló utasítást cseréljük ki egyetlenre, ami rögvest kétszer inkrementál! Erről az utasításról van szó:

```
+2Ln
```

A futásidők átlaga ekkor: **745515**, ami az előzőnek a 97.3%-a. Jó, hát nem világrengető időnyereség, de sok kicsi sokra megy...

Ezután kicseréljük az előjelváltást egy „profi módszerre”, az előjelváltó gyorsutastásra! Ekkor a programunk így néz ki:

```

#!mau
// kiszámoljuk a pi-t a leibniz formulával.
#t@a; // inicializálunk egy időváltozót
#d@p=0; // Az összegváltozó
#L@n=1; // A nevező
#L@s=1; // A számláló
{| 3000000
#d@p=(@p)+(#L@s)/(#L@n);
+2Ln
±Ls
|}
#t@b; // inicializálunk egy másik időváltozót
#g@x=(#t@b)-(#t@a);
"Idő: " ?g @x; /;
#d@p = 4*#d@p;
"Pi/4 közelítés: " ?d @p; /;

```

Ennek a futásideje **444595**, és hát ez az előzőnek csak 59.6%-a! Hm...! Vaskos, igen jelentős időnyereség...

No és hát ezt ha összevetjük azzal az állapottal amiből kiindultunk, azt kapjuk hogy annak a 44.3%-ánál járunk! Ezért érdemes volt dolgozni, optimalizálni, hiszen az időszükségletet lecsökkentettük kevesebb mint a felére!

Az „etalonnal” összevetve, most **17.83**-szor vagyunk csak lassabbak, mint a „gépi kód”... És még ezen is gyorsíthatunk! Mert ezt a sort:

```
#d@p=(@p)+(#L@s)/(#L@n);
```

cseréljük ki erre:

```
#d@p+=(#L@s)/(#L@n);
```

Így az időszükséglet: **364682**. Ez az előzőnek a 82%-a csak... És most már csak 14.6-szor vagyunk lassabbak, mint a C nyelvű változat, az etalon... Én amondó

vagyok, hogy ez már elfogadható! Ez „egy nagyságrendnyi” sebességkülönbség, ami teljesen érthető, hiszen mi nem compilerek vagyunk: a mau az egy INTER-PRATER. Nyilvánvaló, ha megfeszülünk se tudjuk utolérni sebességben a compilereket.

Következő ötletünk az, hogy mi a csudának használunk L típusú változókat, amikor ezek értékét úgyis át kell konvertálni double-vá?! Legyen minden double. Ehhez persze újra kell írni a C nyelvű programot is, hogy legyen új etalonunk. Annak futásideje most **14644** lett, azaz az is sokkal gyorsabbá vált. A mi mau programunk első változata:

```
#!mau
// kiszámoljuk a pit a leibniz formulával.
#t@a; // inicializálunk egy időváltozót
#d@p=0; // Az összegvaltozo
#d@n=1; // A nevező
#d@s=1; // A szamlalo
{| 3000000
#d@p=(@p)+(@s)/(@n);
#d@n=(@n)+2;
#d@s=-@s;
|}
#t@b; // inicializálunk egy másik időváltozót
#g@x=(#t@b)-(#t@a);
"Idő: " ?g @x; /;
#d@p = 4*#d@p;
"Pi/4 közelítés: " ?d @p; /;
```

Ennek futásideje: **1146300**, s ez az etalonnak a 78-szorosa. Durva... Nem baj azért, lesz ám ez kisebb is...

Végezzük el rajta az összes módosítást, amit az előző, nem csak double típust használó mau progin elvégeztünk! Ekkor a programunk így fog kinézni:

```
#!mau
// kiszámoljuk a pit a leibniz formulával.
#t@a; // inicializálunk egy időváltozót
#d@p=0; // Az összegvaltozo
#d@n=1; // A nevező
#d@s=1; // A szamlalo
{| 3000000
#d@p=(@p)+(@s)/(@n);
+2dn
±ds
|}
#t@b; // inicializálunk egy másik időváltozót
#g@x=(#t@b)-(#t@a);
"Idő: " ?g @x; /;
#d@p = 4*#d@p;
"Pi/4 közelítés: " ?d @p; /;
```

Ennek az időszükséglete pedig már csupán **465408**. Az előző variáció 40.6%-a...

És ezt még tovább csökkenthetjük! Használjunk a cikluson belül gyorsfüggvényt! Ekkor a progi így néz ki:

```
#!mau
// kiszámoljuk a pit a leibniz formulával.
#t@a; // inicializálunk egy időváltozót
#d@p=0; // Az összegvaltozo
#d@n=1; // A nevező
#d@s=1; // A szamlalo
{| 3000000
#d@p=(@p)+#d@n;
+2dn
±ds
|}
#t@b; // inicializálunk egy másik időváltozót
#g@x=(#t@b)-(#t@a);
```

```
"Idő: " ?g @x; /;
#d@p = 4*#d@p;
"Pi/4 közelítés: " ?d @p; /;
```

Ekkor az időszükséglet: **386190**. Ez az előző 83%-a.

És most a függvényen belül a gyorsfüggvényes sort írjuk összevont utasításként!  
A programunk most eképp fest:

```
#!mau
// kiszámoljuk a pi-t a leibniz formulával.
#t@a; // inicializálunk egy időváltozót
#d@p=0; // Az összegváltozó
#d@n=1; // A nevező
#d@s=1; // A számláló
{| 3000000
#d@p+=#d/sn;
+2dn
±ds
|}
#t@b; // inicializálunk egy másik időváltozót
#g@x=(#t@b)-(#t@a);
"Idő: " ?g @x; /;
#d@p = 4*#d@p;
"Pi/4 közelítés: " ?d @p; /;
```

Ennek időátlaga pedig **331629**.

Pillanatnyilag nincs jobb ötletem, hogyan lehetne tovább gyorsítani... Minden-  
esetre ez az előző variációhoz képest is újabb 14% időspórolást jelent!

E végső eredmény a kiindulállapothoz képest (ami csak double típust használt)  
annak 28.9%-a, azaz KEVESEBB MINT EGYHARMADA! Az etalonhoz képest meg  
annak 22.6-szerese. Ez rosszabb mint a kevert típusokat használó változat  
etalonnal összevetett aránya, de csak mert úgy tűnik, a C nyelvű változat esetén  
nagyobb időnyereséget jelent ha nem kell a típusok közt konvertálgatni, mint ami  
előnyt ez a mau programban jelent. De így is megérte nekünk is kerülni a  
típuskonverziókat, mert ezen egytípusú utóbbi program végső változata a kevert  
típusú változat végső variánsával összevetve, mintegy 10%-kal gyorsabban fut,  
csak amiatt, mert nem kell a típusok közt konvertálgatni!

## 34. fejezet - Hasznos X parancsok

### A DMENU integráció

A mau interpreterbe bele van építve pár grafikus lehetőség, ami ugyan meg-  
lehetősen szerény, de legalább nem erőforrás-zabáló, s jó arra, hogy könnyen  
írjunk programokat amik valamit megjelenítenek nekünk a grafikus képernyőn -  
üzenetet vagy információt, ráadásul azt is megtehetjük hogy több lehetőség közül  
választatunk valamit ki a Felhasználóval, s visszakapjuk a kiválasztott stringet!  
Ehhez még csak külön libraryt se kell használnunk, ez bele van építve az  
interpreterbe - ha X11 integrációval fordítjuk. Ez alpból be van kapcsolva nála.  
E függvények melyek mindezt lehetővé teszik nekünk, részletesen ismertetésre  
kerültek a rendszerfüggvényekről szóló fejezetben.

Itt most ennek a háttéréről szólnék egy keveset. Az a szoftveregyüttes ami ezt lehetővé teszi, a mau interpreter többi részével ellentétben javarészt nem az én alkotásom, hanem egyszerűen „elcsórtam” egy ezt lehetővé tevő programot, azaz azt megfelelően módosítva beépítettem az interpreterembe. Ez az, aminek a fileneve a forráskódok közt „mydmenu.cpp”. Ennek eredeti változata az, ami a dmenu-4.5 nevet (verziószámot) viseli, s itt a honlapja:

<http://tools.suckless.org/dmenu/>

Mint látható, megadtam az eredeti forrást, nem akarok „idegen tollakkal ékeskedni”. A fájl végére bemásoltam az eredeti licencfeltételeket is.

Na most, az eredeti programon kellett azért módosítanom ezt-azt, például mert az parancssorból paraméterezhető volt. A mau programnyelvben a lehetséges megjelenítési paraméterek globális változókon keresztül változtathatóak, amik érvényesek maradnak mindaddig, minden megjelenítésre, míg újra meg nem változtatjuk őket. E beállításokra különböző mau utasítások szolgálnak, melyek a következők:

```
MC x, S
vagy
MC x, y
vagy
MC x
```

Itt az S egy stringkifejezés, az y egy **mau\_l** kifejezés, az x pedig egy **mau\_c** kifejezés. Ez utóbbitől függ, az S illetve az y mire vonatkozik. A jelentések:

```
F vagy f: A megjelenítés ezzel az S fontkészlettel történik.
h:      normal background color, vagyis a megjelenítés háttérszíne S.
H:      selected background color, vagyis az épp kiválasztott menütétel háttérszíne S.
i:      normal foreground color, vagyis a megjelenítés írásszíne S.
I:      selected foreground color, vagyis az épp kiválasztott menütétel írásszíne S.
t:      A menü a képernyő tetején helyezkedik el. (alapértelmezés).
b:      A menü a képernyő alján helyezkedik el.
L:      A megjelenített sorok száma a menüben y. Alapértelmezés: 27.
```

Egy példa a használatra:

```
MC f, "-*-fixed-*-*-*-*28-*-*-*-*-*";
MC I, "white";
MC H, "gray";
```

Hogy a fontkészletben mit jelent az a fura string? Bevallom, én se vagyok tisztában a jelentésével. Az X szerver ilyen idióta módon várja a font paramétereinek a megadását. A középén a „28”, az mindenesetre a fontkészlet mérete.

Az alapbeállítások a mau interpreter indulásakor:

```
Fontkészlet: "-*-fixed-*-*-*-*18-*-*-*-*-*";
normal background color ="black";
selected background color ="#d2001e";
normal foreground color ="yellow";
selected foreground color ="yellow";
```

Ezek az alapbeállítások különben fordítási direktívaként megváltoztathatók, a vz.h fájlban vannak letárolva, eképp:

```
#define dmenuDMENUFONT "-*-fixed-*-*-*-*18-*-*-*-*-*"
#define dmenuDMENUSf "yellow"
```

```
#define dmenuDMENUf "yellow"
#define dmenuDMENUb "#d2001e"
#define dmenuDMENUc "black"
#define dmenuDMENUtopbar 1
#define dmenuDMENUSOROKSZAMA 27
```

A **dmenuDMENUtopbar** 1-es értéke jelenti azt hogy alapértelmezés szerint a menü a képernyő tetején helyezkedik el, ha lent akarjuk látni állítsuk ezt nullára.

## Kiiratás a statuszbarra

Amennyiben a mau interpretert az X11 integrációval fordítjuk, az esetben lehetőségünk nyílik rá, hogy valamely stringet elküldjünk az X szervernek, beállítani ezzel az ő úgymond „státuszát”. Hogy ez miért „státusz”, azt nem tudom, mert láthatóan semmit nem kezd vele az X, mindenesetre ez egy olyan string, amit egynémely ablakkezelők, mint például a DWM amit használok (s amit kifejezetten ajánlok mint ablakkezelőt a mau interpreter mellé!) megjelenítenek a grafikus képernyő valamely kitüntetett pontján. E pont neve az, hogy „statusbar”, vagy „állapotsor”. Ez a DWM esetén alapértelmezés szerint jobbra fent van.

Ide egy stringet küldeni a következő utasítással lehet:

**Xs string**

Ehhez azonban valahol a program elején egy „képernyőt” kell igényelnünk az X szervertől, erre a paraméterek nélküli

**Xi**

utasítás szolgál.

Egy példa a használatára:

```
Xi;
#s@s="Mau státusz 1!";
#s@g="Ez is mau státusz!";
Xs @s;
WS 2;
Xs @g;
XX
```

## 35. fejezet - A mau programnyelv fejlesztését segítő eszközök

E fejezetben olyasmikről lesz szó, melyek teljesen feleslegesek azoknak, akik csak meg szeretnének írni valamely mau nyelvű programot a maguk céljaira. Itt ugyanis azok az utasítások, függvények, stb vannak megemlítve/elmagyarázva, melyek elsősorban NEKEM MAGAMNAK fontosak, vagy - ha lesz netán efféle illető - általában véve azoknak, akik magát a mau programnyelvet óhajtják kibővíteni és/vagy megváltoztatni, azaz fejleszteni, forkolni, a maguk kedvére igazítani.

Mindjárt legelsőnek azt fontos bemutatni, miként deríthető ki könnyen, melyik 2 karakteres utasítástokenek szabadok még, valami újabb utasítás számára! (például mert pluginben akarjuk azt megvalósítani, vagy akár mert kibővítenénk

magát az interpretert vele). Nos, e célra használhatjuk a **to** utasítást, mely természetesen a „token” szó rövidítése. Legegyszerűbb ha a **calc.mau** programot indítjuk a futtatásához. Ezzel is mutatjuk be a használatát:

```
vz@Csiszilla /Releases/2014/U/Common/vz/MAU=>./mau calc.mau
mau> to "to"
TOKEN HEXA      DEC      NAME
to   $74,$6f 116,111  tokenek
mau> to "XX"
TOKEN HEXA      DEC      NAME
XX   $58,$58 88,88   fuggveny_XX
mau> to "xx"
TOKEN HEXA      DEC      NAME
xx   $78,$78 120,120 fuggveny_xx
mau> to "if"
TOKEN HEXA      DEC      NAME
if   $69,$66 105,102 fuggveny_if
mau> to "ha"
TOKEN HEXA      DEC      NAME
ha   $68,$61 104,97  fuggveny_ha
mau> to "kl"
mau> to "uj"
mau> to "kuz"
LOG:> 2014.05.20 16:32:33 : Itt csak pontosan 2 bájt hosszú string adható meg! Hely: 8
```

Látható a fentiekből, miként működik: Az utána következő stringkifejezést olvassa be, ami pontosan 2 bájtól kell álljon, s az ezekben meghatározott mau tokenhez tartozó utasítás nevét írja ki, valamint e név karaktereinek ASCII kódját hexában is és decimálisan is. Sőt, emellé nagy előzékenyen kiírja nekünk azt a nevet is, ami az ezen utasítás működését megvalósító C nyelvű függvény neve a mau interpreter forráskódjában, azaz aki haxorkodni akar, s átírni valamit, vagy csak kíváncsi arra melyik funkció miféle trükkökkel lett leprogramozva, az most már könnyedén rákereshet a megfelelő névre a forráskódban!

Az is látható, hogy amennyiben olyan név lett megadva neki amihez nem tartozik utasítás (itt a példában a „kl” és az „uj”) akkor egyszerűen semmit se ír ki.

Továbbá, megadható ezen utasítás így is:

**to;**

Ekkor KÖTELEZŐ a pontosvessző használata utána, ráadásul kivételesen semmiféle whitespace se állhat a **to** és a pontosvessző között! Ezesetben MINDEN érvényes utasítástokenet kilistáz.

Na most, a lista kissé „töredékesen” néz majd ki, azaz ocsmányul külalakra. Ennek az az oka, hogy e „2 karakteres” utasítástokenek közt bizony jó sok olyan is akad, amelyek tulajdonképpen igazából csak 1 karakteresek, de be vannak ide táplálva (a megfelelő tömbbe) az utánuk következő space-vel, pontosvesszővel vagy újsor-karakterrel együtt, mert így gyorsabban megtalálja őt az interpreter, kevesebb bájtot kell végignyá laznia a mau forráskódban. Azaz így nő a sebesség. És hát ha a második karakter egy újsor karakter, akkor itt a kiírt listában is sort fog dobni nekünk azon a helyen, ha meg ott tabulátorkarakter van akkor azt írja ki!

Egy példa a futtatására (ne feledjük ez csak az e pillanatban érvényes lista, a későbbiek során ez hőtzticher hogy bővülni fog!):

```

mau> to;
TOKEN  HEXA      DEC      NAME
      $20,$20  32,32   semmi
      $20,$9   32,9    semmi
;      $20,$3b 32,59   pontosvesszo

      $20,$a   32,10   semmi
;      $3b,$20 59,32   semmi
;;     $3b,$3b 59,59   pontosvesszo
;
      $3b,$a   59,10   semmi
;
      $3b,$a   59,10   semmi
;      $3b,$9   59,9    semmi

      $a,$20   10,32   semmi
;      $a,$3b   10,59   pontosvesszo

      $a,$9    10,9    semmi

      $9,$a    9,10    semmi
      $9,$20   9,32    semmi
      $9,$9    9,9     semmi
;      $9,$3b   9,59   pontosvesszo
//     $2f,$2f 47,47   perper
/*     $2f,$2a 47,42   percsillag
/      $2f,$20 47,32   sorkiir
/
      $2f,$a   47,10   sorkiir
/      $2f,$9   47,9    sorkiir
/;     $2f,$3b 47,59   sorkiir
/+     $2f,$2b 47,43   perplusz
/-     $2f,$2d 47,45   perminusz
/0     $2f,$30 47,48   per0
/1     $2f,$31 47,49   per1
/F     $2f,$46 47,70   perF
<(     $3c,$28 60,40   kisebbjel_mindentipus
<C     $3c,$43 60,67   kisebbjel_mindentipus
<D     $3c,$44 60,68   kisebbjel_mindentipus
<G     $3c,$47 60,71   kisebbjel_mindentipus
<I     $3c,$49 60,73   kisebbjel_mindentipus
<L     $3c,$4c 60,76   kisebbjel_mindentipus
<c     $3c,$63 60,99   kisebbjel_mindentipus
<d     $3c,$64 60,100  kisebbjel_mindentipus
<f     $3c,$66 60,102  kisebbjel_mindentipus
<g     $3c,$67 60,103  kisebbjel_mindentipus
<i     $3c,$69 60,105  kisebbjel_mindentipus
<l     $3c,$6c 60,108  kisebbjel_mindentipus
<s     $3c,$73 60,115  kisebbjel_mindentipus
<?     $3c,$3f 60,63   kisebbjel_kerdojel
[#     $5b,$23 91,35   nyitoszegleteskereszt
[@     $5b,$40 91,64   nyitoszegleteskukac
[[     $5b,$5b 91,91   nyitoszegletesnyitoszegletes
?"     $3f,$22 63,34   textkiir
?(     $3f,$28 63,40   kerdojel
?;     $3f,$3b 63,59   sorkiir
??     $3f,$3f 63,63   kerdojelkerdojel
?C     $3f,$43 63,67   kerdojel
?D     $3f,$44 63,68   kerdojel
?G     $3f,$47 63,71   kerdojel
?I     $3f,$49 63,73   kerdojel
?L     $3f,$4c 63,76   kerdojel
?c     $3f,$63 63,99   kerdojel
?d     $3f,$64 63,100  kerdojel
?f     $3f,$66 63,102  kerdojel
?g     $3f,$67 63,103  kerdojel
?i     $3f,$69 63,105  kerdojel
?k     $3f,$6b 63,107  kerdojel
?l     $3f,$6c 63,108  kerdojel

```



```

?s      $3f,$73 63,115 kerdojel
!}      $21,$7d 33,125 felkialtojel_csukokapcsos
{       $7b,$20 123,32 nyitokapcsoszarojel
{;      $7b,$3b 123,59 nyitokapcsoszarojel
{
    $7b,$a 123,10 nyitokapcsoszarojel
{
    $7b,$9 123,9 nyitokapcsoszarojel
{!      $7b,$21 123,33 nyitokapcsoszarojelfelkialtojel
{#      $7b,$23 123,35 nyitokapcsoszarojelkereszt
{(      $7b,$28 123,40 nyitokapcsosnyitorendeszarojel
{-      $7b,$2d 123,45 nyitokapcsosminuszcsukokapcsoszarojel
{[      $7b,$5b 123,91 nyitkapcsosszegletes
{{      $7b,$7b 123,123 duplanyitokapcsoszarojel
{|      $7b,$7c 123,124 nyitokapcsoszarojelvonal
{|}     $7b,$7d 123,125 nyitokapcsoscsukokapcsoszarojel
}       $7d,$20 125,32 csukokapcsoszarojel
};      $7d,$3b 125,59 csukokapcsoszarojel
}
    $7d,$a 125,10 csukokapcsoszarojel
}
}       $7d,$9 125,9 csukokapcsoszarojel
}}      $7d,$7d 125,125 duplacsukokapcsoszarojel
)}      $29,$7d 41,125 csukosimacsukokapcsoszarojel
|}      $7c,$7d 124,125 vonalkapcsoscsukozarojel
^^      $5e,$5e 94,94 fuggveny_felfel
::      $3a,$3a 58,58 fuggveny_duplakettospont
++      $2b,$2b 43,43 pluszplusz
+1      $2b,$31 43,49 pluszplusz
+2      $2b,$32 43,50 plusz2
--      $2d,$2d 45,45 minuszminusz
-1      $2d,$31 45,49 minuszminusz
#c      $23,$63 35,99 ertekadas
#C      $23,$43 35,67 ertekadas
#i      $23,$69 35,105 ertekadas
#I      $23,$49 35,73 ertekadas
#l      $23,$6c 35,108 ertekadas
#L      $23,$4c 35,76 ertekadas
#g      $23,$67 35,103 ertekadas
#G      $23,$47 35,71 ertekadas
#f      $23,$66 35,102 ertekadas
#d      $23,$64 35,100 ertekadas
#D      $23,$44 35,68 ertekadas
#T      $23,$54 35,84 ertekadas
#B      $23,$42 35,66 ertekadas
#K      $23,$4b 35,75 ertekadas
#a      $23,$61 35,97 ertekadas
#t      $23,$74 35,116 ertekadas
#M      $23,$4d 35,77 ertekadas
#s      $23,$73 35,115 ertekadas
#!      $23,$21 35,33 perper
B(      $42,$28 66,40 background
BB      $42,$42 66,66 background
Bb      $42,$62 66,98 background
BC      $42,$43 66,67 background
Bc      $42,$63 66,99 background
Bd      $42,$64 66,100 background
BE      $42,$45 66,69 fuggveny_BE
Bf      $42,$66 66,102 background
BG      $42,$47 66,71 background
Bg      $42,$67 66,103 background
BL      $42,$4c 66,76 fuggveny_BE
Bl      $42,$6c 66,108 background
BM      $42,$4d 66,77 background
Bm      $42,$6d 66,109 background
BN      $42,$4e 66,78 fuggveny_BE
BR      $42,$52 66,82 background
Br      $42,$72 66,114 background
Bw      $42,$77 66,119 background
BY      $42,$59 66,89 background
By      $42,$79 66,121 background
E;      $45,$3b 69,59 fuggveny_else

```

E			
	\$45,\$a	69,10	fuggveny_else
E		\$45,\$9 69,9	fuggveny_else
E	\$45,\$20	69,32	fuggveny_else
E.	\$45,\$2e	69,46	fuggveny_E_pont
EE	\$45,\$45	69,69	fuggveny_EE
EO	\$45,\$4f	69,79	fuggveny_EOF
G,	\$47,\$2c	71,44	fuggveny_G
G			
	\$47,\$a	71,10	fuggveny_G
G		\$47,\$9 71,9	fuggveny_G
G	\$47,\$20	71,32	fuggveny_G
ha	\$68,\$61	104,97	fuggveny_ha
if	\$69,\$66	105,102	fuggveny_if
ii	\$69,\$69	105,105	ii
KI	\$4b,\$49	75,73	fuggveny_KI
ki	\$6b,\$69	107,105	fuggveny_ki
LO	\$4c,\$4f	76,79	fuggveny_LOG
M@	\$4d,\$40	77,64	mkukacu
MA	\$4d,\$41	77,65	fuggvenyMAU
MC	\$4d,\$43	77,67	dmenucolors
m@	\$6d,\$40	109,64	mkukacu
ma	\$6d,\$61	109,97	mau
NB	\$4e,\$42	78,66	mau_cbreak
NC	\$4e,\$43	78,67	mau_initscr
NE	\$4e,\$45	78,69	mau_echo
NK	\$4e,\$4b	78,75	mau_keypad
NR	\$4e,\$52	78,82	mau_refresh
Ns	\$4e,\$73	78,115	mau_N
NX	\$4e,\$58	78,88	mau_endwin
Pc	\$50,\$63	80,99	fuggveny_Pc
Ps	\$50,\$73	80,115	fuggveny_Ps
QS	\$51,\$53	81,83	fuggveny_QS
RA	\$52,\$41	82,65	mau_RAW
RD	\$52,\$44	82,68	fuggveny_REMOVEDIRECTORY
RM	\$52,\$4d	82,77	fuggveny_REMOVE
RN	\$52,\$4e	82,78	fuggveny_RENAME
S(	\$53,\$28	83,40	szinek
Sb	\$53,\$62	83,98	szinek
Sc	\$53,\$63	83,99	szinek
Sd	\$53,\$64	83,100	szinek
Sf	\$53,\$66	83,102	szinek
Sg	\$53,\$67	83,103	szinek
Sm	\$53,\$6d	83,109	szinek
Sr	\$53,\$72	83,114	szinek
Su	\$53,\$75	83,117	szinek
Sv	\$53,\$76	83,118	szinek
Sw	\$53,\$77	83,119	szinek
SY	\$53,\$59	83,89	SYstem
Sy	\$53,\$79	83,121	szinek
T;	\$54,\$3b	84,59	fuggveny_then
T			
	\$54,\$a	84,10	fuggveny_then
T		\$54,\$9 84,9	fuggveny_then
T	\$54,\$20	84,32	fuggveny_then
T.	\$54,\$2e	84,46	fuggveny_T_pont
TT	\$54,\$54	84,84	fuggveny_TT
to	\$74,\$6f	116,111	tokenek
WS	\$57,\$53	87,83	fuggveny_WS
Ws	\$57,\$73	87,115	fuggveny_Ws
X!	\$58,\$21	88,33	fuggveny_Xfelkialtojel
Xi	\$58,\$69	88,105	xi
Xs	\$58,\$73	88,115	Xs
XX	\$58,\$58	88,88	fuggveny_XX
xx	\$78,\$78	120,120	fuggveny_xx
00	\$30,\$30	48,48	fuggveny_nulla

## 36. fejezet - Mau megszakításrendszer

Ez a fogalom azt jelenti, hogy beállíthatjuk egy mau függvény automatikus, időzített végrehajtását. Azt tehát, hogy az a függvény végrehajtódjon minden X időtartam letelte után, aholis az X milliomodmásodpercekben értendő. Továbbá, ez az időtartam „közelítőleg” értendő.

Azért „megközelítőleg” csak, mert aritmetikai vagy stringkifejezés kiértékelését nem hagyja félbe emiatt a „rencer”. Hanem minden egyes új mau utasítás végrehajtása előtt vizsgálja meg, van-e teendője ilyen téren. Ezokból természetesen az időzítés nem lesz abszolút pontos, de szerintem amire egy interpreteres nyelvnek kell, ami nem valósidejű folyamatvezérlésre van kitalálva, arra a célra teljesen jó.

Sajnos azonban, ennek ára van! A dolog ugyebár a gép belső órájának lekérdezésén alapszik, amit a `clock()` függvénnyel kérdezek le. És ezt biza lekérdezi ökelme minden egyes utasításvégrehajtáskor! És ilyenkor elvégez egy kivonást is és egy összehasonlítást. Ezen extra teendők miatt a programvégrehajtás ideje tapasztalataim (=méréseim) szerint az ÖTSZÖRÖSÉRE nőtt (!!!!!!!), sőt, előfordulhat akár a tízszeres lassulás is egyes feladatoknál. Szerencsére tényleg csak azon esetekben amikor ez az interrupt dolog be van kapcsolva nála. (Nyilvánvaló ugyanis hogy ez ki-be kapcsolgatható egy utasítással).

Szóval, ésszel kell használni mert nagyon lelassítja a futást.

A szintaxis:

**ii 0;**

vagy

**ii;**

E két fenti utasítás KIKAPCSOLJA a megszakításvégrehajtást. Az **ii;** variációban a pontosvessző előtt nem állhat whitespace!

**ii „S”;**

vagy

**ii [sorszám];**

A fenti szintaxissal adhatjuk meg, melyik függvényt nevezzük ki interrupt-rutinnak. A függvény maga teljesen ugyanúgy nézhet ki, mint bármely közönséges mau függvény, semmi extra dolgot nem kell beleírni ahhoz hogy interruptként használhassuk, egyetlen kikötés, hogy nem várhat el input paramétereket, és nem adhat vissza eredményeket. Azaz, e függvény vezérlését globális változókon át kell megoldanunk, vagy az „álfüggvények”-ről szóló részben leírtak szerint kell birizgálnunk a változóit.

A fenti szintaxisnál az S egy stringkifejezés lehet ami az interruptfüggvény nevét határozza meg, illetve a sorszám egy olyan aritmetikai kifejezés mely a függvény sorszámát mondja meg. Mindenben olyan ez is tehát mint amikor egy függvényt meghívánk. Fontos tudni, hogy hiába adjuk meg név szerint a függvényt, ezt a drága átkonvertálja sorszámmá és a sorszámot tárolja el, emiatt rém rossz ötlet egy bekapcsolt interrupt során betölteni újabb mau függvényeket, mert mint tudjuk olyankor azok névsorba lesznek rendezve, s így előfordulhat hogy megváltozik egy függvény sorszáma...

Az interrupt elindítása:

```
ii x
```

Ahol az x egy **mau\_g** azaz **#g** típusú aritmetikai kifejezés lehet, ez határozza meg, hogy hány milliomed másodpercenként hajtódjon végre. Ha ezen érték 0, akkor az KIKAPCSOLJA az interruptot (de ettől még a beállított interruptfüggvény NEVÉT, pontosabban a SORSZÁMÁT nem felejtí el, azt nem kell újra beállítani egy újabb bekapcsolás előtt).

Mindenféleképpen előbb az interruptfüggvény nevét vagy sorszámát adjuk meg a legelső bekapcsolás előtt, különben automatikusan a nullás sorszámú függvényt óhajtja meghívni, ami legtöbbször a főprogram maga. De ha mégsem az, akkor is véletlenszerű hogy mi, ha nem állítottuk be...

Továbbá, az interruptfüggvény nem fogja magát meghívni interruptként, azaz, amíg az interruptfüggvényt hajtja végre, addig kikapcsolja az interruptot ideiglenesen... Nem fordulhat elő tehát olyan helyzet, hogy mert az interruptfüggvény végrehajtása túl soká tart, azt félbehagyja és újrakezdi. Ezenfelül garantált, hogy az interruptfüggvény lefutása után mindenféleképpen végrehajt legalább egyetlen olyan utasítást, ami nem az interruptfüggvényhez tartozik.

Természetesen az interruptfüggvény is emlékszik a korábbi futásai során kikalkulált eredményeire, változóinak értékeire.

Íme egy példa a használatára:

```
#!mau
```

```
#c@,,"Interrupt""c=a;
ii „Interrupt"";
ii 2000000;

// kiszámoljuk a pi-t a leibniz formulával.
#t@a; // inicializálunk egy időváltozót
#d@p=0; // Az összegváltozó
#L@n=1; // A nevező
#L@s=1; // A számláló
{| 3000000
    #d@p= (@p) + (#L@s)/(#L@n);
    #L@n = (@n) + 2;
    #L@s = -1*(@s);
|}
#t@b; // inicializálunk egy másik időváltozót
#g@x=(#t@b)-(#t@a);
"Idő: " ?g @x; /;
#d@p = 4*#d@p;
"Pi/4 közelítés: " ?d @p; /;
"Vége!\n"
ii;
XX

„Interrupt” // Interruptrutin
?k #c@c; /;
#c++@c;
xx
```

Megjegyzem, szándékosan nem a leghatékonyabb pi-számító rész van e fenti példába beletéve, hogy tovább tartson. Az eredménye:

a  
b  
c  
d  
e  
f  
g  
h  
i  
j  
k  
l  
m  
n  
o  
p  
q  
r  
s  
t  
u  
v  
w  
x  
y  
z

Idő: 5372021  
Pi/4 közelítés: 3.14159  
Vége!

Azaz mint látható, miközben számolja ki a pí közelítését a főprogramban, időnként meghívja az „Interrupt” nevű függvényt, ami olyankor kiír egy újabb karaktert a képernyőre. Amennyiben e programot kipróbálsz a saját gépeden, simán lehetséges hogy a kiírt betűk száma több vagy kevesebb lesz, mert ez attól függ, milyen gyors a számítógéped.

Általában efféle interruptokra „emberileg észlelhető” időtartamokon belül van szükségünk, például egy szövegkurzor villogtatásához. Ez úgy nagyjából tizedmásodperces időtartam. Ennyi idő alatt a számítógép eszméletlenül sok mau utasítást képes lenne feldolgozni, és teljesen felesleges hogy mindegyik alkalommal lekérdezze a gép aktuális idejét, ez csak feleslegesen lassítaná a programfutást. Ezért van rá lehetőségünk, hogy megadjuk neki, hány mau utasítás végrehajtása után ellenőrizgesse csak, hogy letelt-e a beállított időtartam! Ez az **II** utasítással történik. Szintaxisa:

**II x**

Ahol az x egy unsigned int típusú kifejezés. Ezzel kiegészítve az előbbi program így néz ki:

**#!mau**

```
#c@,, "Interrupt""c=a;  
ii „Interrupt”;  
ii 200000;  
II 1000;  
  
// kiszámoljuk a pit a leibniz formulával.  
#t@a; // inicializálunk egy időváltozót  
#d@p=0; // Az összegváltozó  
#L@n=1; // A nevező  
#L@s=1; // A számláló  
{| 3000000  
  #d@p= (@p) + (#L@s)/(#L@n);  
  #L@n = (@n) + 2;  
  #L@s = -1*(@s);  
|}
```

```

#t@b; // inicializálunk egy másik időváltozót
#g@x=(#t@b)-(#t@a);
"Idő: " ?g @x; /;
#d@p = 4*#d@p;
"Pi/4 közelítés: " ?d @p; /;
"Vége!\n"
ii;
XX

„Interrupt” // Interruptrutin
?k #c@c; /;
#c++@c;
xx

```

Az eredménye:

```

a
b
c
d
e
Idő: 1058851
Pi/4 közelítés: 3.14159
Vége!

```

Szépen felgyorsult az ötszörösére, attól az

```

II 1000

```

sortól...

## 37. fejezet - A BETŰ és a JELSOR, vagyis az UTF-8 kódolású karakterek és stringek

Mint azt már e doksi elejétájékán is említettem, a mau interpreter alapból azt feltételezi, hogy egy string (az, aminek a casting operátora a **#s**), az UTF-8 kódolású. Ez rendben is volna, mert még az összehasonlító műveletek is eszerint értelmeztetnek. A baj csak 1, de az NAAAGY: A **#s** stringjeink index operátora, az, amivel hivatkozunk e stringek egy-egy karakterére, (a szegletes/szögletes zárójelek) az valójában nekünk nem KARAKTERRE indexel, hanem csak BÁJTRA! Holott egy UTF-8 kódolású karakter igen gyakran több bájtot is elfoglal!

Meg lehetne csinálni nem túl nehezen, hogy ezen indexelés a karakterekre vonatkozzék, de ez több okból mégis problémát jelentene. Egyrészt, lassú volna, mert mindegyik alkalommal végig kéne bogarásznia a stringet az elejétől, hogy figyelje az UTF-8 bájlhatárokat. Aztán ha olyan karaktert szűrnánk be a stringbe ami több bájból áll mint ami már ott van, újra memóriát kéne allokálni a stringnek, és átmásolni oda a nem változó részeket... Iszonyú lassú volna valóban! Ennél is zűrösebb dolog, hogy sok esetben pedig igenis mi nem UTF-8 karakterek sorozataként akarjuk a stringet kezelni, hanem tényleg bájtonként, például ha mondjuk mau nyelven egy hexa editort írunk!

Szóval, hagytam a **#s** típusú stringeket így ahogy vannak, „hagyományos” stringekként. Ellenben beépítettem egy másik stringfajtát is a mau nyelvbe, ennek - az előzőtől való megkülönböztetés végett - a JELSOR nevet adtam. Ez BETŰKből áll. (Az itt BETŰKnek nevezett típus osztályneve a programkódban az UTF, a JELSOR neve pedig a JELEK, s mindkettő az **utf.h** illetve **utf.cpp** fájlokban van lekódolva, ha valakit érdekelne).

A BETŰK casting operátora a **#u**, a JELSOR-é pedig a **#U**. Egy JELSOR, az tulajdonképpen egy BETŰkből álló tömb.

Érdemes tudnunk a BETŰkről, hogy kivétel nélkül mindegyik pontosan 8 bájtot foglal el, se többet, se kevesebbet! (Technikailag egy unionban vannak, ami értelmeztetik egy 8 bájtos stringtömbként egyrészt, másrészt meg egy unsigned long long számként.) Azért ennyi bájton vannak ábrázolva, mert a 8 bájt egy gépi szó méret, s emiatt a feldolgozása viszonylag gyors, s ez van a legközelebb a 6 bájtos mérethez, márpedig egy UTF-8 kódolású karakter lehet akár 6 bájt is!

Tisztában vagyok vele, hogy később a nemtudomkicsoda főmuftik úgy döntöttek hogy mégsem lesznek 4 bájtosnál hosszabb UTF-8 szekvenciák, de bevallom, egyszerűen NEM HISZEK NEKIK. Az az alig több mint 1 millió lehetőség ami maradt így, számomra nem tűnik meggyőzően soknak. Elvileg bőven elég kéne legyen, sajnos azonban az Unicode-ben dolgozó nemtudomkicsodák úgy tűnik nem rendelkeznek kellő önmérséklettel, s minden szemetet bele akarnak gyömöszölni az Unicode-ba! Legutóbb például (forrás:

<http://prog.hu/hirek/3402/Furcsa+irasjelek+tomegeit+fogja+tartalmazni+a+Unicode+7+0.html> ) egy lebegő öltönyös figurát is beleraktak, csak amiatt, mert a Microsoft Webdings betűkészletében szerepel egy ilyen. Hát csesszék meg, el se tudom képzelni, mi a francra lehet majd használni egy efféle „betűt”... ha így folytatják a kettő a 64-en variáció is kevés lesz nekik... Szóval, megcsináltam a BETŰ típust inkább mind az eredeti 6 UTF-8 bájtra, mert nem akarom hogy ha mégis használatba veszik a többi bájtot, újra kelljen írnom a mau interpretert!

A **#c** típusú unsigned char karakterek és a BETŰK átalakíthatóak egymásba, természetesen csak az ASCII karakterek esetén. Mindez azonban azt jelenti, hogy egy BETŰ, az bizony 8-szor annyi helyet foglal el a tárban, mint egy közönséges, 1 bájton tárolt karakter! ASCII karakterek tárolására használni tehát nagy pocsklás. A BETŰ, az tulajdonképpen egy olyan mau típus, ami önmagában szinte felesleges, kizárólag amiatt létezik, mert a JELSORok BETŰkből állnak. Természetesen egy JELSOR is sokkal több helyet foglal el, mint egy közönséges **#s** string, nem is arra van kitalálva, hogy alaphól ebben tároljunk minden stringet. Ezt a legkifejezettebben csak akkor ajánlott használni, ha nemcsak hogy olyan munkát végzünk, ami ékezetes karakterek feldolgozásával is jár, de ráadásul nagyon gyakran kell efféle stringeket kifejezetten úgy manipulálni, hogy sokszor kell beszúrni beléjük ide-oda karaktereket, vagy kiolvasni belőlük ilyeneket, azaz gyakran kell INDEXELGETNI ilyen stringeket! Lényegében minden más feladatot kiválóan el tudunk végezni a régi stringjeinkkel is könnyen.

## A BETŰk

A BETŰk kezelésének néhány vonását bemutató picike program, példaként:

```
#!mau  
#s@s="álmos";  
#u@u=#s@s; ?u @u; /; ?c !'@u; /;  
#u@u=#c g; ?u @u; /; ?c !'@u; /;  
#u@u="„ez”"; ?u @u; /; ?c !'@u; /;
```



```

>nulladik: " ?c #u@u[0]; /;
"első:    " ?c #u@u[1]; /;
"második : " ?c #u@u[2]; /;
"indexelés nélkül : " ?c #u@u; /;
"nyolcadik: " ?c #u@u[8]; /;
"kilencedik: " ?c #u@u[9]; /;

#u@u[2]=159; ?u @u; /;
>nulladik: " ?c #u@u[0]; /;
"első:    " ?c #u@u[1]; /;
"második : " ?c #u@u[2]; /;
XX

```

Eredménye:

```

á
2
9
1
”
3
nulladik: 226
első:     128
második : 158
indexelés nélkül :226
nyolcadik: 226
kilencedik: 128
"
nulladik: 226
első:     128
második : 159

```

Látható, hogy egy BETŰ változónak (ezekből is 256 különféle lehetséges) értékül adhatunk mindenekelőtt unsigned char, azaz **#c** típusú értéket. Értékül adhatunk neki aztán **#s** típusú stringkifejezést is, ezesetben a string legelső (UTF-8 kódolás szerint értelmezett) karaktere lesz a BETŰ értéke. Ez a lehetőség amiatt fontos, mert csak így adhatunk konstans értéket egy BETŰnek, másképp nem, hiszen az UTF-8 karakterek több-bájtosak, azok nem férnének el másba csak stringbe. Tehát, eképp inicializálhatunk mondjuk egy BETŰváltozót:

```
#u@b="É";
```

Természetesen minden névtér létrejöttékor rendelkezésünkre áll mind a 256 BETŰváltozó, és mindegyik a **CHR\$(0)** karaktert tartalmazza. Azaz, mind a 8 bájt ami egy BETŰváltozóhoz tartozik, nullát tartalmaz.

Az is látható a fenti példaprogramból, hogy egy BETŰt a sz(e/ö)gletes zárójelekkel indexelhetünk, mint egy miniatűr **#c** tömböt: természetesen az indexek csak a 0-7 tartományban értelmezettek. Elméletileg. Azaz, ennek van értelme. Ha azonban nagyobb indexet alkalmazunk, nem történik hibajelzés: a mau interpreter egyszerűen az indexnek csupán az alsó 3 bitjét veszi figyelembe mindig... Amennyiben unsigned char értékkel kell konvertálni egy BETŰt, s nem adunk meg indexet a sz(e/ö)gletes zárójelekkel, akkor egyszerűen a nulladik bájt tartalmát adja vissza nekünk. Ennek az az értelme, hogy ez teljesen megfelel egy **#c** karakternek, amennyiben az egy ASCII karakter. A leggyakoribb eset pedig biztos hogy ez lesz.

Minthogy egy UTF-8 karakter bájtsorozata legfeljebb 6 bájt hosszú lehet, emiatt azok a 0-5 indexekkel érhetőek el, s a 6-odik és 7-edik indexű bájt minden BETŰ esetén mindig üres, azaz „not used”, nem használt, vagyis szabad a számunkra mindenféle járulékos információk esetleges tárolására. Egy BETŰkből álló stringben, azaz egy JELSOR-ban így minden BETŰ esetén tárolhatjuk e bájtokon például mondjuk a BETŰ színinformációját. Vagy amit akarunk.

Egy BETŰ esetén használhatjuk a **!** (Felkiáltójel-Aposztróf) unáris operátort arra, hogy visszaadja (**#c** számként) azt, hogy az adott BETŰ éppen hány UTF-8 bájtot igényel tárolásra. Látjuk erre is a példákat a fenti programban.

Ilyesmit is lehet csinálni:

```
#u++@u[2];
```

Vagy:

```
#u--@u[2];
```

Azaz, inkrementálhatjuk és dekrementálhatjuk a BETŰ bármelyik bájtyát.

**#g** értéket is értékül adhatunk a BETŰnek:

```
#g@r=(158*256*256)+(128*256)+226;  
#u@h=#g@r; ?u @h; /;  
"nulladik: " ?c #u@h[0]; /;  
"első: " ?c #u@h[1]; /;  
"második : " ?c #u@h[2]; /;
```

Eredménye:

```
”  
nulladik: 226  
első: 128  
második : 158
```

Ilyet is művelhetünk:

```
#u@u[2]-=1; ?u @u; /;  
"nulladik: " ?c #u@u[0]; /;  
"első: " ?c #u@u[1]; /;  
"második : " ?c #u@u[2]; /;
```

Azaz használhatóak az összevont utasítások, általában azok, amik egy **#c** tömbnél is használhatóak, vagyis a

```
+=  
-=  
*=  
/=
```

><=  
<<=  
>>=  
~~=  
műveleti jelek.

Természetesen BETŰt átcastolhatunk **#g** értéké is, például:

```
?g #u@u;
```

Ilyen esetben a BETŰ számára fenntartott mind a 8 bájt figyelembe vétetik, egyszerűen ezen bájtokat összességében úgy tekinti, mint egyetlen **#g** számhoz tartozó tárterületet.

BETŰket össze is hasonlíthatunk az **if** vagy a **ha** utasítással, mint bármi más mau típust is. Arra hogy egy BETŰ mikor nagyobb vagy kisebb vagy egyenlő mint egy másik BETŰ, ugyanazok a szabályok vonatkoznak, mint amelyek szerint a mau interpreter a **#s** típusú stringeket is névsorba rendezi. Itt egy kis példa-program a szintaxisra, bár az már ismerős kell legyen:

```
#!mau  
#u@A="Á";  
#u@B="á";  
"A= " ?u @A; " B= " ?u @B; /;  
ha #u (@A)==(@B) "Egyenlőek!\n"  
E "Nem egyenlőek!\n";
```

```

ha #u (@A)!=(@B) "Nem egyenlőek!\n"
E "Egyenlőek!\n";
ha #u (@A)<(>(@B) "Nem egyenlőek!\n"
E "Egyenlőek!\n";
ha #u (@A)<=(>(@B) "A <= B\n"
E "A nem <= B\n";
ha #u (@A)>=(>(@B) "A >= B\n"
E "A nem >= B\n";
ha #u (@A)<(>(@B) "A < B\n"
E "A nem < B\n";
ha #u (@A)>(>(@B) "A > B\n"
E "A nem > B\n";
XX

```

A BETŰKre létezik néhány úgynevezett „**unáris postfix operátor**”. Ezek olyanok, hogy az adott BETŰ aritmetikai kifejezése UTÁN kell őket megadni, s módosítják annak jelentését. Ilyenek azok, melyek az adott BETŰt átalakítják kisbetűssé vagy nagybetűssé. Erre egy példaprogram:

```

#!mau
#u@A="Á";
#u@B="á";
"A= " ?u @A; " B= " ?u @B; /;
ha #u (@A)==(@B) "Egyenlőek!\n"
E "Nem egyenlőek!\n";
ha #u (@A)!=(@B) "Nem egyenlőek!\n"
E "Egyenlőek!\n";
ha #u (@A)<(>(@B) "Nem egyenlőek!\n"
E "Egyenlőek!\n";
ha #u (@A)<=(>(@B) "A <= B\n"
E "A nem <= B\n";
ha #u (@A)>=(>(@B) "A >= B\n"
E "A nem >= B\n";
ha #u (@A)<(>(@B) "A < B\n"
E "A nem < B\n";
ha #u (@A)>(>(@B) "A > B\n"
E "A nem > B\n";
"=== Átalakítom a B-t NAGYbetűssé: ==\n"
#u@B=@B|>;
"A= " ?u @A; " B= " ?u @B; /;
ha #u (@A)==(@B) "Egyenlőek!\n"
E "Nem egyenlőek!\n";
ha #u (@A)!=(@B) "Nem egyenlőek!\n"
E "Egyenlőek!\n";
ha #u (@A)<(>(@B) "Nem egyenlőek!\n"
E "Egyenlőek!\n";
ha #u (@A)<=(>(@B) "A <= B\n"
E "A nem <= B\n";
ha #u (@A)>=(>(@B) "A >= B\n"
E "A nem >= B\n";
ha #u (@A)<(>(@B) "A < B\n"
E "A nem < B\n";
ha #u (@A)>(>(@B) "A > B\n"
E "A nem > B\n";
"=== Átalakítom az A-t kisbetűssé: ==\n"
#u@A=@A|<;
"A= " ?u @A; " B= " ?u @B; /;
ha #u (@A)==(@B) "Egyenlőek!\n"
E "Nem egyenlőek!\n";
ha #u (@A)!=(@B) "Nem egyenlőek!\n"
E "Egyenlőek!\n";
ha #u (@A)<(>(@B) "Nem egyenlőek!\n"
E "Egyenlőek!\n";
ha #u (@A)<=(>(@B) "A <= B\n"
E "A nem <= B\n";
ha #u (@A)>=(>(@B) "A >= B\n"
E "A nem >= B\n";
ha #u (@A)<(>(@B) "A < B\n"

```

```

E "A nem < B\n";
ha #u (@A)>(@B) "A > B\n"
E "A nem > B\n";
"===== \n"
XX

```

Eredménye:

```

A= Á B= á
Nem egyenlőek!
Nem egyenlőek!
Nem egyenlőek!
A <= B
A nem >= B
A < B
A nem > B
=== Átalakítom a B-t NAGYbetűssé: ==
A= Á B= Á
Egyenlőek!
Egyenlőek!
Egyenlőek!
A <= B
A >= B
A nem < B
A nem > B
=== Átalakítom az A-t kisbetűssé: ==
A= á B= Á
Nem egyenlőek!
Nem egyenlőek!
Nem egyenlőek!
A nem <= B
A >= B
A nem < B
A > B
=====

```

Tehát: A nagybetűssé átalakítás unáris postfix operátora a

```

|>|
a kisbetűssé alakításé pedig a
|<|

```

Persze, miért is ne használhatnánk a mau nyelvben egy postfix operátort prefix operátorként is... Azaz, a fenti programocskát így is írhatjuk:

```

#!mau
#u@A="Á";
#u@B="á";
"A= " ?u @A; " B= " ?u @B; /;
ha #u (@A)==(@B) "Egyenlőek!\n"
E "Nem egyenlőek!\n";
ha #u (@A)!=(@B) "Nem egyenlőek!\n"
E "Egyenlőek!\n";
ha #u (@A)<(>@B) "Nem egyenlőek!\n"
E "Egyenlőek!\n";
ha #u (@A)<=(>= @B) "A <= B\n"
E "A nem <= B\n";
ha #u (@A)>=(< @B) "A >= B\n"
E "A nem >= B\n";
ha #u (@A)<(>@B) "A < B\n"
E "A nem < B\n";
ha #u (@A)>(<@B) "A > B\n"
E "A nem > B\n";
"=== Átalakítom a B-t NAGYbetűssé: ==\n"
#u|>|@B;
"A= " ?u @A; " B= " ?u @B; /;
ha #u (@A)==(@B) "Egyenlőek!\n"
E "Nem egyenlőek!\n";
ha #u (@A)!=(@B) "Nem egyenlőek!\n"
E "Egyenlőek!\n";
ha #u (@A)<(>@B) "Nem egyenlőek!\n"

```

```

E "Egyenlőek!\n";
ha #u (@A)<=(@B) "A <= B\n"
E "A nem <= B\n";
ha #u (@A)>=(@B) "A >= B\n"
E "A nem >= B\n";
ha #u (@A)<(@B) "A < B\n"
E "A nem < B\n";
ha #u (@A)>(@B) "A > B\n"
E "A nem > B\n";
"=== Átalakítom az A-t kisbetűssé: ==\n"
#u|<|@A;
"A= " ?u @A; " B= " ?u @B; /;
ha #u (@A)==(@B) "Egyenlőek!\n"
E "Nem egyenlőek!\n";
ha #u (@A)!=(@B) "Nem egyenlőek!\n"
E "Egyenlőek!\n";
ha #u (@A)<>(@B) "Nem egyenlőek!\n"
E "Egyenlőek!\n";
ha #u (@A)<=(@B) "A <= B\n"
E "A nem <= B\n";
ha #u (@A)>=(@B) "A >= B\n"
E "A nem >= B\n";
ha #u (@A)<(@B) "A < B\n"
E "A nem < B\n";
ha #u (@A)>(@B) "A > B\n"
E "A nem > B\n";
"===== \n"

```

XX

Az eredménye természetesen ugyanaz. Azaz, mintha csak inkrementáló vagy dekrementáló operátor lenne, úgy is használhatjuk a nagy- vagy kisbetűssé átalakító operátorunkat:

```

#u|<|@A;
#u|>|@A;

```

Magától értetődően viszont ez a fajta alkalmazás már csak kifejezetten változókra, s nem egész kifejezésekre használható!

Természetesen abban az esetben ha a BETŰ olyan betűt tartalmaz ami igazából nem is betű csak valami más jel, például zárójel vagy a dollárjel, vagy betű ugyan de olyan ABC eleme mely nem különbözteti meg a kisbetűket a nagybetűktől, akkor a **|<|** és **|>|** operátorok nem változtatják meg a BETŰ értékét.

Az hogy miféle kisbetű-nagybetű párokat ismer a mau interpreter, az **utf.cpp** fájlban van leszabályozva. Ennek elejétáján található ugyanis egy tömb, ami ilyesféléképpen kezdődik:

```

kisnagyutf kisnagy[] = {
{"a","A"},
{"á","Á"},
{"ä","Ä"},
{"b","B"},
{"c","C"},

```

Ebbe a tömbbe (lustaságból, meg mert nekem másra nincs szükségem...) egyelőre csak az angol ABC betűpárjai vannak felvéve, meg a magyar ékezetes betűk, és a (német) umlaut (ä-Ä), meg az eszperantó ékezetes karakterek. Amennyiben tehát valaki azt óhajtja hogy a mau interpretere ismerjen más ABC-ket is annyira, hogy tudja, ott melyik kis- és nagybetű párosok vannak, tehát tudja ezt mondjuk a cirill (orosz), görög, francia, cseh stb abc-k esetében is, akkor e tömböt kell értelemszerűen kibővítenie (és újrarendezni a mau interpretert). Nem kell törődnie azzal, hogy valahol átírja hogy ezentúl hány elemből áll a tömb, ezt az interpreter megszámolja magának fordítás közben! Valamint jó hír az is, hogy teljesen

mindegy hova szúrja be az új párosokat, nem lényeges a sorrend, csak arra vigyázzon, hogy egy betűpárnál mindig a kisbetűt írja elsőnek, s utána a nagybetűt.

Unáris postfix operátorunk használata esetén is indexelhetjük a BETŰket hogy megkapjuk valamely bájtjukat:

```
#!mau
#u@u="á";
?u @u; /; ?c !'@u; /;
>nulladik: " ?c #u@u[0]; /;
"első: " ?c #u@u[1]; /;
"második : " ?c #u@u[2]; /;
"nagybetűsen: " ?u @u|>; /;
>nulladik: " ?c #u@u|>|[0]; /;
"első: " ?c #u@u|>|[1]; /;
"második : " ?c #u@u|>|[2]; /;
XX
```

Eredménye:

```
á
2
nulladik: 195
első: 161
második : 0
nagybetűsen: Á
>nulladik: 195
első: 129
második : 0
```

Felmerülhet a mau nyelv elsajátításával kacérkodókban az a gondolat, hogy minek bohóckodok holmi „unáris postfix operátorok” megalkotásával, minek ez a váratlan stílus-szaltó, amikor eddig remekül megvolt a mau nyelv effélék nélkül, minek elbonyolítani ezekkel a szintaxist! Nos, nem hiszem hogy ezek megjegyzése nehezebb lenne, mint annak memorizálása, hogy a rengeteg **?#c**, **?#s** stb függvény közül mikor melyiket kell meghívunk valami cél érdekében. Emiatt egyes fontosabb(nak tűnő...) funkciókat, melyekhez nem kell külön paraméter, előnyösnek tűnik „kiszervezni” efféle operátorokba. Ráadásul, ha a kis/nagybetűs átalakításra megcsináljuk e postfix operátort, akkor ezt felhasználhatjuk rendes „prefix” unáris operátorként is, picit más jelentésben: Úgy ugyanis egy **#c** értéket ad vissza nekünk, attól függően, hogy az utána következő betű miféle típusú! Konkrétan, nézzük csak e példaprogramot:

```
#!mau
#u@u="á"; "u = " ?u @u; /; if |<|@u T "kisbetűs!\n"
E "Nem kisbetűs!\n"
if |.|@u T "Nem betű!\n"
#u|>|@u; "u = " ?u @u; /; if |<|@u T "kisbetűs!\n"
E "Nem kisbetűs!\n"
if |.|@u T "Nem betű!\n"
#u@u="É"; "u = " ?u @u; /; if |>|@u T "NAGYbetűs!\n"
E "Nem NAGYbetűs!\n"
if |.|@u T "Nem betű!\n"
#u|<|@u; "u = " ?u @u; /; if |>|@u T "NAGYbetűs!\n"
E "Nem NAGYbetűs!\n"
if |.|@u T "Nem betű!\n"
#u@u="š"; "u = " ?u @u; /; if |>|@u T "NAGYbetűs!\n"
E "Nem NAGYbetűs!\n"
if |.|@u T "Nem betű!\n"
XX
```

Jól látható tehát, hogy a **|<|**, a **|>|** és a **|.|** unáris operátorok prefix helyzetben mit csinálnak:

|<| visszaad 1-et ha az utána következő BETŰ kisbetű, különben nullát.  
|>| visszaad 1-et ha az utána következő BETŰ NAGYbetű, különben nullát.  
|.| visszaad 1-et ha az utána következő BETŰ nem kisbetű vagy nagybetű, különben nullát.

Természetesen egyszerre is használhatjuk ezen operátorokat prefix és postfix helyzetben, bár ennek tulajdonképpen nem sok értelme van. De a szintaxis megengedi:

```
#!mau
#u@u="á"; "u = " ?u @u; /;
if |<|@u T "kisbetű!\n"
E "Nem kisbetűs!\n"
if |.|@u T "Nem betű!\n"
"Na most figyelj:\n"
if |<|@u|>| T "kisbetűs!\n"
E "Nem kisbetűs!\n"
if |.|@u|>| T "Nem betű!\n"
if |>|@u|>| T "NAGYbetűs!\n"
E "Nem NAGYbetűs!\n"
if |.|@u|>| T "Nem betű!\n"
XX
```

Eredménye:

```
u = á
kisbetűs!
Na most figyelj:
Nem kisbetűs!
NAGYbetűs!
```

## A JELSORok

Mint már említettem, egy JELSOR, az lényegében nem más mint egy BETŰkből álló egydimenziós tömb. Itt is 0-tól kezdődnek az indexek. A JELSOR casting operátora a

#U

és JELSORból is 256 darab áll rendelkezésre minden névtérben, a nevükre a szokásos szabályok érvényesek. Mind a 256 JELSOR létezik a névtér megszületésekor máris, és garantált, hogy ezek pontosan 1 BETŰt tartalmaznak, a nulladik indexűt, s ezen BETŰ értéke a CHR\$(0) karakter. Ez természetesen nem számít bele a JELSOR hosszába amit lekérdezhetünk a JELSORt illetően, azaz kezdetben minden JELSOR hossza 0. És természetesen a JELSOR hosszát külön tárolja az interpreter, azaz ennek érdekében nem kell átbogarásznia mindig a teljes JELSOR minden BETŰjét, ami lassú volna. Ez azonban csak az UTF-8 kódolású BETŰkben értendő hosszra vonatkozik! Ha a JELSOR byte-okban, pontosabban **#c** karakterekben mért hosszára vagyunk kíváncsiak, azt bizony kénytelen minden alkalommal újra kiszámolni a teljes JELSOR bájtónkénti elemzésével...

A JESOR esetén is van lehetőségünk tehát lekérdezni a BETŰkben (azaz UTF-8 karakterekben) illetve a bájtokban mért hosszakat. E célra a

!!!

és a

!|'

unáris prefix operátorok szolgálnak.

Előbbemenve a kritikáknak, rögvest elárulom, e célra miért nem a **#s** stringeknél megszokott **!** illetve **!!** jeleket használom. Azért nem, mert mindkét stringfajtánál



egyaránt **mau\_1** típusú (azaz a **#1** casting operátor illetékességi körébe eső) számérték lehet a string hossza (akár BETŰkben akár bájtokban mérjük), na most ugye minden numerikus típus kiértékelésére külön kifejezéskiértékelő függvénnyel rendelkezik a mau interpreter, s így ugyanaz a kiértékelő kell kiértékelje mindegyik stringfajta mindkét típusú hosszát, mert mindegyik hossz **#1** típusú. Valahonnan tudnia kell, épp melyik típusú string hosszát kérdi le, s e string melyik fajta hosszát. Amennyiben ragaszkodtam volna hozzá hogy a JELSOR esetén is a felkiáltójelek jelentsék a hosszakat, akkor muszáj lettem volna úgy megírni a kifejezéskiértékelő függvényt, hogy az a hossza utaló **!** vagy **!!** unáris operátor után kötelezően elvárja egy **#s** vagy **#U** casting operátor szerepeltetését is, hogy tudja, miféle stringről van szó a továbbiakban. Előre borítékolható volt azonban, hogy abban az esetben ezek kitételéről rendszeresen megfeledezünk, vagy ha kitesszük, gyakran eltévesztjük, melyiket kell kitenni. És azt a szintaxist ugyanúgy feleslegesen bonyolultnak tartották volna egyesek. Szóval egyszerűen inkább úgy döntöttem, hogy külön operátort vezetek be e célokra. A szintaxis annyiból logikus is, hogy a JELSOR stringeknek is közük vana BETŰkhöz mert azokból áll, s így hasonlóképp két **|** jel közt vannak ezen unáris prefix operátorok, valamint a **#s** stringeknél az UTF-8 karakterekben mért hosszát a két felkiáltójel adja vissza, s itt is a **|** jelek közt két felkiáltójel jelzi ezt. Továbbá, BETŰk esetén a bájtban mért hosszát a **!** adja vissza, s itt is ez használtatik a **|** jelek közt e célra.

Példaprogram:

```
#!mau
#U@U="Álmos";
?U @U; /;
"Karakterhossz = " ?l |!|@U; /;
"Bytehossz      = " ?l |!'|@U; /;
#U--@U;
?U @U; /;
"Karakterhossz = " ?l |!|@U; /;
"Bytehossz      = " ?l |!'|@U; /;
#U@U="Álmosé";
?U @U; /;
"Karakterhossz = " ?l |!|@U; /;
"Bytehossz      = " ?l |!'|@U; /;
#U--@U;
?U @U; /;
"Karakterhossz = " ?l |!|@U; /;
"Bytehossz      = " ?l |!'|@U; /;
#U@U="„Álmosé”";
?U @U; /;
"Karakterhossz = " ?l |!|@U; /;
"Bytehossz      = " ?l |!'|@U; /;
#U--@U;
?U @U; /;
"Karakterhossz = " ?l |!|@U; /;
"Bytehossz      = " ?l |!'|@U; /;
XX
```

Eredmény:

```
Álmos
Karakterhossz = 5
Bytehossz      = 6
Álmo
Karakterhossz = 4
Bytehossz      = 5
Álmosé
Karakterhossz = 6
Bytehossz      = 8
Álmos
Karakterhossz = 5
```

```

Bytehossz      = 6
„Álmosé”
Karakterhossz = 8
Bytehossz      = 14
„Álmosé
Karakterhossz = 7
Bytehossz      = 11

```

Megjegyzendő, hogy e hosszlekérdezés csak az előjel nélküli numerikus típusok esetén lehetséges, azaz olyan helyeken ahol amúgy egy **#c**, **#i**, **#l** vagy **#g** értéket óhajt az interpreter beolvasni. Ugyanez vonatkozik a **#u** típusok **!** prefix unáris operátorral történő bájtyszám-lekérdezésére is.

JELSORt természetesen a **#s** stringekhez hasonlóan a sz(e/ö)gletes zárójelekkel indexelve kaphatjuk meg a benne szereplő BETŰket:

```

#!mau
#U@U="„Álmosé”";
?U @U; /;
"Karakterhossz = " ?l |!|@U; /;
"Bytehossz      = " ?l |!|@U; /;
{| |!|@U;
#u@u= #U@U[?~];
?u @u;
|} /;
"Második kiírás: \n"
{| |!|@U;
?u #U@U[?~];
|} /;
XX

```

Eredménye:

```

„Álmosé”
Karakterhossz = 8
Bytehossz      = 14
”ésomlÁ,,
Második kiírás:
”ésomlÁ,,

```

Fontos tudni a JELSORokról, hogy ellentétben a **#s** stringekkel, ennek végét nem tesztelhetjük le holmi „záró nullabájt” vizsgálatával! Azt a lehetőséget a **#s** stringeknél is csak amiatt hagytam meg, hogy könnyebben lehessen „rendszer-manipuláló” programokat írni, C stílusban. Ellenben a JELSORok kifejezetten szövegmanipulációs célokat szolgálnak, emiatt ezek úgy működnek, hogy ha nagyobb elemszámra indexelünk bennük mint ami a ténylegesen bennük meglevő „értékes” karakterek száma, akkor kibővítik a hosszat - ez eddig megegyezik a **#s** stringek viselkedésével - **UGYANAKKOR A LÉTREJÖTT ÚJ KARAKTERHELYEKET FELTÖLTIK SZÓKÖZ KARAKTEREKSEL IS!** Ez teljesen normálisan elvárható szolgáltatás szerintem egy értelmes programnyelvtől, mert SZÖVEGEK esetén úgyis az a logikus nézet, hogy ahol még semmi sincs ott szóköz legyen, nem tetszőleges memóriaszemét. Ezekből azonban ha arra az indexre hivatkozunk ahol elvileg a „nullabájt” kéne legyen (ott is van különben „mélyen a rendszerben”, csak nekünk semmi szükségünk rá) akkor is kibővíti a stringhosszat, és arra a helyre is betesz nekünk egy szóközt! Egy példa:

```

#!mau
#U@U="„Álmosé”";
?U @U; /;
"Karakterhossz = " ?l |!|@U; /;
"Bytehossz      = " ?l |!|@U; /;
{| |!|@U;
#u@u= #U@U[(|!|@U)-?|];

```

```
?u @u; " :: "
{| !'@u
?c #u@u[(!'@u)-?|]; " ";
|} /;
|} /;
"Záróbájt: \n"
{| !'#u@u[!!!|@u]
?c #u@u[!!!|@u][(!'#u@u[!!!|@u])-?|]; " ";
|} /;
?c #u@u[8][0]; /;
XX
```

Eredménye:

```
„Álmosé”
Karakterhossz = 8
Bytehossz     = 14
„ :: 226 128 158
Á :: 195 129
l :: 108
m :: 109
o :: 111
s :: 115
é :: 195 169
” :: 226 128 157
```

```
Záróbájt:
32
32
```

Mint látható, e példában 2 féleképp is kiirattam a záró nullabájtot. Ne ijedjünk meg, azért a mau nyelvben általában nem kell olyan szörnyű „brainfuck” stílusban programozni, mint ahogy elszörnyesztésképpen itt adtam meg:

```
?c #u@u[!!!|@u][(!'#u@u[!!!|@u])-?|]; " ";
```

Aki ugyanis tisztában van a mau nyelv lehetőségeinek roppant tárházával, az azonnal átlátja, hogy bár a fenti sor teljesen logikus és kézenfekvő, úgy érte hogy minden józan embernek ez jut az eszébe e problémára megoldásként már rögvést az első pillanatban (ne vedd komolyan, ez csak vicc...) de ezt így is lehet írni:

```
?c #u@u[!!!|@u][{|}]; " ";
```

Hát ez azért már jóval egyszerűbb... Kis jóindulattal erre már akár azt is mondhatjuk, hogy ez már-már mintha talán és esetleg közelítene az „áttekinthető” fogalomköréhez...

E trükköt különben e program többi helyén is alkalmazhatjuk:

```
#!mau
#U@U="„Álmosé”";
?U @U; /;
"Karakterhossz = " ?l |!!!|@U; /;
"Bytehossz     = " ?l |!'|@U; /;
{| |!!!|@U;
#u@u= #U@U[{|}];
?u @u; " :: "
{| !'@u
?c #u@u[{|}]; " ";
|} /;
|} /;
"Záróbájt: \n"
{| !'#u@u[!!!|@u]
?c #u@u[!!!|@u][{|}]; " ";
|} /;
?c #u@u[8][0]; /;
XX
```

A JELSORok esetén is van lehetőségünk rá, hogy egy JELSOR változóban tárolt stringet csupa NAGYbetűssé vagy épp csupa kisbetűssé alakítsunk. Íme erre a példaprogram:

```
#!mau
#U@U="Álmoséi";
```

```
?U @U; /;
"Karakterhossz = " ?l |!|@U; /; "Bytehossz      = " ?l |!|@U; /;
#U|<|@U; ?U @U; /;
"Karakterhossz = " ?l |!|@U; /; "Bytehossz      = " ?l |!|@U; /;
#U|>|@U; ?U @U; /;
"Karakterhossz = " ?l |!|@U; /; "Bytehossz      = " ?l |!|@U; /;
XX
```

Eredménye:

```
Álmoséi
Karakterhossz = 7
Bytehossz     = 9
álmoséi
Karakterhossz = 7
Bytehossz     = 9
ÁLMOSÉI
Karakterhossz = 7
Bytehossz     = 9
```

E kisbetűsítő/nagybetűsítő operátort természetesen tetszőleges JELSOR kifejezésre is alkalmazhatjuk, ez esetben postfix operátorként, miként azt a BETŰk esetében is megszoktuk:

```
#!mau
#U@U="Álmoséi";
"Eredeti string: " ?U @U; /;
"Kisbetűsített : " ?U @U|<|; /;
"NAGYbetűsített: " ?U @U|>|; /;
"Az eredeti string nem változott: " ?U @U; /;
XX
```

Eredménye:

```
Eredeti string: Álmoséi
Kisbetűsített : álmoséi
NAGYbetűsített: ÁLMOSÉI
Az eredeti string nem változott: Álmoséi
```

Létezik még a fentiekhez hasonlóan a **|^|** operátor is, prefix és postfix változatban is, ami a teljes **#U** stringet (azaz bocsánat, JELSORt) kisbetűsíti, de kivéve a legelső BETŰjét, mert azt meg NAGYbetűssé alakítja. Azaz így használhatjuk:

```
#U|^|@U;
```

Illetve mondjuk így:

```
?U @U|^|;
```

JELSORokat természetesen ugyanúgy összehasonlíthatunk mint a **#s** stringeket, s a szabályok is ugyanazok arra, melyiket tekinti kisebbnek vagy nagyobbaknak:

```
#!mau
#U@U="Álmosé";
#U@P="Álmoséi";
"1. string: " ?U @U; /;
"2. string: " ?U @P; /;
"==\n"
if #U (@U)==(@P) T "Egyenlőek!\n"
E "Nem egyenlőek!\n"
"!=\n"
if #U (@U)!=(@P) T "Nem egyenlőek!\n"
E "Egyenlőek!\n"
"<>\n"
if #U (@U)<>(@P) T "Nem egyenlőek!\n"
E "Egyenlőek!\n"
"<=\n"
if #U (@U)<=(@P) T "Kisebb vagy egyenlő!\n"
E "Nem kisebb vagy egyenlő!\n"
">=\n"
if #U (@U)>=(@P) T "Nagyobb vagy egyenlő!\n"
```

```

E "Nem nagyobb vagy egyenlő!\n"
"<\n"
if #U (@U)<(@P) T "Kisebb!\n"
E "Nem kisebb!\n"
">\n"
if #U (@U)>(@P) T "Nagyobb!\n"
E "Nem nagyobb!\n"
XX

```

Eredménye:

```

1. string: Álmosé
2. string: Álmoséi
==
Nem egyenlők!
!=
Nem egyenlők!
<>
Nem egyenlők!
<=
Kisebb vagy egyenlő!
>=
Nem nagyobb vagy egyenlő!
<
Kisebb!
>
Nem nagyobb!

```

Természetesen JELSORokat is összeadhatunk a stringeknél szokásos módon - ám VIGYÁZZUNK akkor, amikor efféle összeadások közé konstansokat is be óhajtottunk illeszteni, mert itt azért leselkedik ránk némi alattomos csapda... Nézzük csak ezt a programocskát például:

```

#!mau
#U@U="A mau nyelv ";
#U@P="szép";
?U @U; /;
?U @P; /;
?U (@U)+"nagyon "+(@P)+". "; /;
?U (@U)+("nagyon ")+(@P)+(" "); /;
#U@U+=@P;
?U @U; /;
XX

```

Ennek eredménye:

```

A mau nyelv
szép
A mau nyelv nagyon .
A mau nyelv nagyon szép.
A mau nyelv szép

```

Hm... Mi a fene ez a sor, hogy:

```

A mau nyelv nagyon .
????????????!!!!!!!!!!!!!!

```

Ezen fura eredményt e programsor produkálja:

```

?U (@U)+"nagyon "+(@P)+". "; /;

```

Jól látszik, hogy elvileg ki kéne írnia a „szép” szót is, de NEM ÍRJA KI! Vajh' miért is nem...

Nos a megfejtés az, hogy ha egy idézőjelek közé rakott micsodába fut bele a mau interpreter, akkor ő azt **#s** típusú stringként próbálja értelmezni. Ez nagyon rendben is van, és nincs ezzel semmi gond tényleg, még akkor se ha nem **#s** hanem **#U** stringről (JELSORról) van szó, mert a **#U** kiértékelő rutin bőségesen van olyan értelmes hogy tudja, miként kell egy **#s** típusú stringből **#U** JELSORt gyártani. Amint tehát egy idézőjelbe fut bele ökelme, nem is gondolkodik tovább,

hanem meghívja a **#s** típus kiértékelő függvényét, s „aszonygya nekije”, hogy „hé komám, itt egy **#s** típusú micsoda gyűn, csekkold meg, oszt' passzold vissza nekem az eredményt, osztán majd én azzal tovább micsodálom ami a dógom!”. Na most a **#s** kiértékelő elkezd kiértékelni a hátralevő dolgokat mint egy rendes **#s** kifejezést. Ki is olvassa az idézőjelek közti részt, azzal nincs gond, kreál belőle egy stringet. Igenám, de ezután jön egy plusz jel! Az biza egy érvényes operátor a **#s** kifejezésekben (is), tehát beolvassa a plusz jel utáni dolgokat is, megpróbálva értelmezni azokat mint egy **#s** kifejezés további részeit. Ott a **(@P)** részhez ér. Ez egy teljesen korrekt változónév, amit ő azonban a **#s** stringváltozók közt keres... Igenám, de azok közt a **P** nevűnek még nem adtunk értéket, annak tartalma tehát üres string. Ehhez hozzáadja a kifejezés további részében szereplő **."** részt mint **#s** stringet, mindezt hozzáadja a legelső stringhez amit idézőjelek közt talált, majd az eredményt visszapasszolja a **#U** kiértékelőnek...

Azaz, ilyesmire NAGYON ügyeljünk! Ha JELSORt más JELSORok összegeként óhajtunk előállítani, akkor az esetlegesen közéjük ékelt stringkonstansokkal bánjunk el úgy, mint e programban látható helyütt:

```
?U (@U)+("nagyon ")+(@P)+("."); /;
```

Azaz, azon idézőjelek közti konstansokat ne felejtjük el **ZÁRÓJELEK KÖZÉ is RAKNI...**

JELSORokat is multiplikálhatunk **#1** számokkal jobboldalról, amint azt a **#s** stringeknél is megszoktuk:

```
#!mau
#U@U="mau";
"Eredeti string: " ?U @U; /;
#U@m=(@U)*3;
?U @m; /;
#U@p="Programozás ";
?U @p; /;
#U@p*=4;
?U @p; /;
XX
```

Eredménye:

```
Eredeti string: mau
maumaumu
Programozás
Programozás Programozás Programozás Programozás
```

Efféle multiplikációról annyit érdemes tudni, hogy 1-el megszorozva természetesen nem változik a tartalma — nullával való szorzás azonban üres stringet csinál belőle, azaz „lenullázza”! Íme:

```
#!mau
#U@U="mau";
"Eredeti string: " ?U @U; /;
"Egyel megszorozva:\n"
#U@m=(@U)*1; ?U @m; /;
"Nullával megszorozva:\n"
#U@m=(@U)*0; ?U @m; /;
XX
```

Eredménye:

```
Eredeti string: mau
Egyel megszorozva:
mau
Nullával megszorozva:
```

Balértékként használva is hivatkozhatunk egy JELSOR egy BETŰjére:

```
#!mau
#U@U="A_mau_nyelv";
?U @U; /;
#U@U[1]=",";
#U@U[5]=",";
?U @U; /;
#U@U[15]="<";
?U @U; /;
XX
```

Eredménye:

```
A_mau_nyelv
A,,mau"nyelv
A,,mau"nyelv    <
```

Jól látható a fentiekből, hogy ha nagyobb indexre hivatkozunk mint a JELSOR hossza, akkor sincs semmi baj, mert kibővíti a JELSORt a megfelelő hosszúságúra, s a közbeeső helyekre szóközöket szúr be. Azaz szakszavakkal megfogalmazva, egy JELSOR tulajdonképpen egy UTF-8 kódolású karakterekből álló, dinamikusan „nyújtózkodó” egydimenziós tömb.

Használható eképp a **|<|** és **|>|** operátor is:

```
#!mau
#U@U="A_MAU_nyelv";
?U @U; /;
#U|<|@U[2];
#U|>|@U[7];
?U @U; /;
XX
```

Eredménye:

```
A_MAU_nyelv
A_mAU_nYelv
```

JELSOR egy BETŰjét is tovább indexelhetjük, hogy megkapjuk valamely bájtját. Természetesen, a JELSOR egy BETŰjének az indexeléséhez **#1**, a BETŰ egy bájtjának indexeléséhez azonban **#c** típusú aritmetikai kifejezés használható! Példa:

```
#!mau
#U@P="„Álmosé”";
(„Bájtok” #U@P);
#U@P[0][2]=157;
(„Bájtok” #U@P);
"Vége\n"
XX

„Bájtok” // Kiírja a JELSOR BETŰinek bájtértékeit
#U@U;
?U @U; " :: A string bájtjai:\n"
{| |!|@U;
?u #U@U[{|}]; " = ";
#l@l={|};
{| !'#U@U[{|}]
?c #U@U[(@l)][{|}]; " ";
|} /;
|}
xx
```

Eredménye:

```
„Álmosé” :: A string bájtjai:
„ = 226 128 158
Á = 195 129
```



```

l = 108
m = 109
o = 111
s = 115
é = 195 169
" = 226 128 157
"Álmosé" :: A string bájtjai:
" = 226 128 157
Á = 195 129
l = 108
m = 109
o = 111
s = 115
é = 195 169
" = 226 128 157
Vége

```

Természetesen e BETŰk bájtjaira nemcsak az értékadó = operátor használható, de a

```

+=
-=
*=
/=
<<=
>>=
>=<
~~=
&&=
||=

```

operátorok is.

Természetesen JELSORok esetén is használhatjuk a #s stringeknél megismert rész-string képző operátorainkat:

```

#!mau
#U@U="A mau programnyelv igenis csodálatosan könnyű nyelv!";
"Eredeti string: " ?U @U; /;
#U@R=[6,39]@U;
?U @R; /;
XX

```

Eredménye:

```

Eredeti string: A mau programnyelv igenis csodálatosan könnyű nyelv!
programnyelv igenis csodálatosan könnyű

```

Ezen rész-string operátorokat tényleg mindenféle variációban használhatjuk:

```

#!mau
#U@U="A mau programnyelv igenis csodálatosan könnyű nyelv!";
"Eredeti string: " ?U @U; /;
#U@R=("ez ")+[19,]@U|>;
?U @R; /;
#U@R=(("ez ")+[,(!!!@U)-7]@U|^|+("!"));
?U @R; /;
XX

```

Eredménye:

```

Eredeti string: A mau programnyelv igenis csodálatosan könnyű nyelv!
ez IGENIS CSODÁLATOSAN KÖNNYŰ NYELV!
Ez a mau programnyelv igenis csodálatosan könnyű!

```

Ezen sor helyett:

```

#U@R=(("ez ")+[,(!!!@U)-7]@U|^|+("!"));
írhattuk volna ezt is:
#U@R=(("ez ")+[0,45]@U|^|+("!"));

```

Természetesen JELSORból is levonhatunk **#l** típusú számot, ami azt jelenti itt is mint a **#s** stringeknél, hogy ennyivel (ennyi BETŰvel) csökkentjük jobbról a hosszát:

```
#!mau
#U@U="A_MAU_nyelvűt"; ?U @U; /; ?U (@U)-3; /;
XX
```

Eredménye:

```
A_MAU_nyelvűt
A_MAU_nyel
```

Természetesen a JELSORok esetén is rendelkezésünkre áll a **#s** stringek esetén már megismert **!\$!** operátor, csak míg azon stringeknél a **#s** string utolsó **#c** típusú karakterét adja vissza, addig itt a JELSOR utolsó BETŰjét, azaz **#u** típusú értékét:

```
#!mau
#U@P="Álmosé";
?U @P; /;
#u@u=|$|@P;
?u @u; /;
"Vége\r"
XX
```

Eredménye:

```
Álmosé
é
Vége
```

## A JELSOR-tömbök

A BETŰkből nincs lehetőség a mau nyelvben tömbök készítésére, ennek ugyanis semmi értelme nem volna, tekintve hogy egy JELSOR éppen pontosan nem más mint egy BETŰkből álló tömb! Amennyiben tehát BETŰk tömbjére van szükségünk, egyszerűen JELSORT kell használnunk helyette.

A JELSORokból azonban vannak tömbjeink a mau nyelvben, természetesen névterenként 256 darab, a jólismert névadási szabályokkal. Minden éppen pontosan olyan, mint a **#s** stringek tömbjeinél, azaz itt is az van hogy egy JELSOR tömbben levő JELSOR-ra a **#U** casting operátorral kell hivatkoznunk mint a single JELSORok esetén is, s az ő tömb mivoltát a kettős sz(e/ö)gletes zárójelek jelzik. Természetesen memóriát foglalni (és azt felszabadítani) is ugyanúgy kell JELSOR tömbnek, mint a **#s** stringtömböknek. Egy példa:

```
#!mau
#U@U="nyávog"; #U@o="ó"; #U@m="macska";
[#U@t=6];
#U@t[[0]]=#U@U;
#U@t[[1]]=#U@t[[0]]+(#U@o)+(" ")+(@m);
#U@t[[2]]=#U@U;
#U@t[[2]]+=((#U@o)+(" circa"));
?U #U@t[[0]]; /; ?U #U@t[[1]]; /; ?U #U@t[[2]]; /;
#U@t[[2]]-=2; ?U #U@t[[2]]; /;
#U@t[[0]]*=3; ?U #U@t[[0]]; /;
#U@t[[3]]=#U@t[[2]]*2; ?U #U@t[[3]]; /;
#u@u=#U@t[[3]][2];
?u @u; /;
?c #U@t[[3]][2][0]; /;
?c #U@t[[3]][2][1]; /;
#c@c=#U@t[[3]][4][0];
?c @c; /;
[#U@t];
XX
```

Eredménye:

```
nyávog
nyávogó macska
nyávogó cica
nyávogó ci
nyávognyávognyávog
nyávogó cinyávogó ci
á
195
161
111
```

Az is látható a fenti példából, hogy a JELSOR egy BETŰjét is továbbindexelhetjük hogy megkapjuk annak **#c** típusú bájtösszetevőit.

Mindez igaz egyaránt a JELSOR-tömbök stringjeinek, ezen stringek BETŰinek és ezen BETŰk bájtjainak bal- és jobbértékben való szerepeltetésénél egyaránt:

```
#!mau
#U@U="nYávog";
[#U@t=6];
#U@t[[0]]=#U@U;
?U #U@t[[0]]; /;
#U@t[[0]][2]=@u;
?U #U@t[[0]]; /;
#U|<|@t[[0]][1];
?U #U@t[[0]]; /;
#U|>|@t[[0]][4];
?U #U@t[[0]]; /;
#U@t[[1]]=„mau” nyelv”;
?U #U@t[[1]]; /;
#U--@t[[1]][0][2];
?U #U@t[[1]]; /;
#U++@t[[1]][4][2];
?U #U@t[[1]]; /;
#U@t[[1]][0][2]=159;
?U #U@t[[1]]; /;
[#U@t];
XX
```

Eredménye:

```
nYávog
nYvog
nyvog
nyv0g
„mau” nyelv
”mau” nyelv
”mau,, nyelv
"mau,, nyelv
```

JELSORhoz hozzáadhatunk nemcsak másik JELSORt vagy **#s** stringet, de BETŰt vagy **#c** karaktert is:

```
#!mau
#U@P=„álmosé”;
(„Bájtok” #U@P);
#s@s=#U@P;
"s stringként: " ?s @s; /;
#u@u="i";
#U@P+=#u@u;
(„Bájtok” #U@P);
#c@c=t;
#U@U=(@P)+#c@c;
(„Bájtok” #U@U);
XX
```

```
„Bájtok” // Kiírja a JELSOR BETŰinek bájtértékeit
#U@U;
```

```

?U @U; " :: A string bájtjai:\n"
{| |!!|@U;
?u #U@U[{}]; " = ";
#l@l={|};
{| !'#U@U[{}]]
?c #U@U[(@l)][{}]; " ";
|} /;
|}
xx

```

Eredménye:

```

„álmósé” :: A string bájtjai:
„ = 226 128 158
á = 195 161
l = 108
m = 109
o = 111
s = 115
é = 195 169
” = 226 128 157
#s stringként: „álmósé”
„álmósé”i :: A string bájtjai:
„ = 226 128 158
á = 195 161
l = 108
m = 109
o = 111
s = 115
é = 195 169
” = 226 128 157
i = 105
„álmósé”it :: A string bájtjai:
„ = 226 128 158
á = 195 161
l = 108
m = 109
o = 111
s = 115
é = 195 169
” = 226 128 157
i = 105
t = 116

```

Persze, tisztességes mau programozó e programot inkább így írja meg:

```

#!mau
#U@P="„álmósé”";
(„Bájtok” #U@P);
#s@s=#U@P;
"#s stringként: " ?s @s; /;
#u@u="i";
#U@P+=#u@u;
(„Bájtok” #U@P);
#c@c=t;
#U@U=(@P)+#c@c;
(„Bájtok” #U@U);
XX

„Bájtok” // Kiírja a JELSOR BETŰinek bájtértékeit
#U@U;
?U @U; " :: A string bájtjai:\n"
{| |!!|@U;
?u #U@U[{}]; " = ";
{| !'#U@U[{}]]
?c #U@U[{}][{}]; " ";
|} /;
|}
xx

```

## 38. fejezet - Bitmezők kezelése

A mau nyelv képes arra, hogy kényelmesen kezelhessünk benne biteket. Természetesen az eddig megismert lehetőségeinkkel is tudtunk már biteket kezelni, ugyebár például az **&&** és **||** operátorokkal, de hát az ilyesmi nagyon nehézkes, nem „úriembereknek való”... A bitmezők, az azt jelenti, hogy egy numerikus érték egyes bitjeit kiolvashatjuk mintha ott egy külön szám lenne tárolva, illetve ezen bitekre beírhatunk valami számértéket úgy, hogy a többi bit tartalmát ne változtassuk meg.

Erre a célra a mau nyelv külön postfix operátorral rendelkezik, s ez a **%[]**. Természetesen a szegletes zárójelek közt kell hogy álljon valami... Mindjárt be is mutatom a szintaxisát egy példán:

```
#!mau
{|16;
?c {|}; " = %" {|8 ?c {1}%[? -]; |} /;
|}
XX
```

Eredménye:

```
0 = %00000000
1 = %00000001
2 = %00000010
3 = %00000011
4 = %00000100
5 = %00000101
6 = %00000110
7 = %00000111
8 = %00001000
9 = %00001001
10 = %00001010
11 = %00001011
12 = %00001100
13 = %00001101
14 = %00001110
15 = %00001111
```

Szépen látható, mit csinál: A 0-15 közti számokat kiírja decimálisan ÉS binárisan is! A lényeg e kis programból természetesen ez a rész:

```
{1}%[? -]
```

A **{1}** természetesen a külső ciklus számlálójának aktuális értékére utal itt - ez megy 0-tól 15-ig. A **?-** is világos: ez a belső, pontosabban az „aktuális” ciklus változója aktuális értékének eggyel csökkentett értéke, azaz ez megy rendre 7-től nulláig. Na most miután ez az érték egy **%[** és egy **]** jel közt van, ez azt mondja az interpreternek, hogy a korábban beolvasott **#c** típusú aritmetikai kifejezésnek csak az ennyiedik bitjét olvassa ki, s adja vissza mint **#c** számot!

Tehát, ez az operátor úgy működik, hogy ha egy **%[** jelhez ér az interpreter, beolvassa az ott talált **#c** aritmetikai kifejezést (ha nem **#c** típusú, arra castolja...), természetesen azt is leellenőrzi hogy le van-e zárva ez egy csukó **]** jellel, s amit beolvasott a **%[** jel előtt, abból csak az ennyiedik bit értékét adja vissza!

Ügyeljünk a szintaxisra itt is! Ez az egész postfix operátor ugyanis NEM VÁLTOZÓKRA, hanem ARITMETIKAI KIFEJEZÉSEKRE vonatkozik! Nagyon nem mindegy... Mert tegyük fel, te a **#c@c** változód 4-edik bitjét akarnád kiolvasni (a

bitek természetesen 0-tól 7-ig számozódnak, és a 0 sorszámú a legalacsonyabb helyiértékű), és így írod:

```
#c@c%[4]
```

Na most, ez szintaktikailag teljesen helyes kifejezés. Egyedül annyi a baja, semmi több, hogy nem azt jelenti amit szeretnél volna... Amint ugyanis a @ jel után az interpreter be óhajtja olvasni a változó nevét, kiértékeli a @ utáni aritmetikai kifejezést. Megtalálja a "c" karaktert, ennek képi az ASCII kódját. Továbbmegy, látja hogy ott bitmezőre hivatkozás van. Beolvassa a szegletes zárójelek közti értéket, az 4. Erre kiszedi a "c" ASCII kódjának értékéből a 4-edik bitet, ami történetesen 0, mert a "c" ASCII kódja 99, annak bináris formátuma pedig ez:

```
%01100011
```

És erre visszaadja neked a #c@0 változó tartalmát...

Azaz amit akartál, azt így kell írni:

```
(#c@c)%[4]
```

Ismétlem, nagyon nem mindegy...

Egy másik jópofa példa, hasonló mint az előző, ami kiírja a teljes ASCII karakterkészletet binárisan (is), természetesen csak a kiíratható karaktereket:

```
#!mau
{|127;
if({|}>31) T ?k {|}; " :: " ?c {|}; " = %" {|8 ?c {1}%[?~]; |} /;
|}
XX
```

Eredménye:

```
:: 32 = %00100000
! :: 33 = %00100001
" :: 34 = %00100010
# :: 35 = %00100011
$ :: 36 = %00100100
% :: 37 = %00100101
& :: 38 = %00100110
' :: 39 = %00100111
( :: 40 = %00101000
) :: 41 = %00101001
* :: 42 = %00101010
+ :: 43 = %00101011
, :: 44 = %00101100
- :: 45 = %00101101
. :: 46 = %00101110
/ :: 47 = %00101111
0 :: 48 = %00110000
1 :: 49 = %00110001
2 :: 50 = %00110010
3 :: 51 = %00110011
4 :: 52 = %00110100
5 :: 53 = %00110101
6 :: 54 = %00110110
7 :: 55 = %00110111
8 :: 56 = %00111000
9 :: 57 = %00111001
: :: 58 = %00111010
; :: 59 = %00111011
< :: 60 = %00111100
= :: 61 = %00111101
> :: 62 = %00111110
? :: 63 = %00111111
@ :: 64 = %01000000
A :: 65 = %01000001
B :: 66 = %01000010
C :: 67 = %01000011
D :: 68 = %01000100
E :: 69 = %01000101
F :: 70 = %01000110
G :: 71 = %01000111
```

```

H :: 72 = %01001000
I :: 73 = %01001001
J :: 74 = %01001010
K :: 75 = %01001011
L :: 76 = %01001100
M :: 77 = %01001101
N :: 78 = %01001110
O :: 79 = %01001111
P :: 80 = %01010000
Q :: 81 = %01010001
R :: 82 = %01010010
S :: 83 = %01010011
T :: 84 = %01010100
U :: 85 = %01010101
V :: 86 = %01010110
W :: 87 = %01010111
X :: 88 = %01011000
Y :: 89 = %01011001
Z :: 90 = %01011010
[ :: 91 = %01011011
\ :: 92 = %01011100
] :: 93 = %01011101
^ :: 94 = %01011110
_ :: 95 = %01011111
` :: 96 = %01100000
a :: 97 = %01100001
b :: 98 = %01100010
c :: 99 = %01100011
d :: 100 = %01100100
e :: 101 = %01100101
f :: 102 = %01100110
g :: 103 = %01100111
h :: 104 = %01101000
i :: 105 = %01101001
j :: 106 = %01101010
k :: 107 = %01101011
l :: 108 = %01101100
m :: 109 = %01101101
n :: 110 = %01101110
o :: 111 = %01101111
p :: 112 = %01110000
q :: 113 = %01110001
r :: 114 = %01110010
s :: 115 = %01110011
t :: 116 = %01110100
u :: 117 = %01110101
v :: 118 = %01110110
w :: 119 = %01110111
x :: 120 = %01111000
y :: 121 = %01111001
z :: 122 = %01111010
{ :: 123 = %01111011
| :: 124 = %01111100
} :: 125 = %01111101
~ :: 126 = %01111110

```

Érdemes tudni, hogy a **%[** operátor és az előző aritmetikai kifejezés vége közt lehet akármennyi whitespace.

Ez azonban nem minden, amit ezen postfix operátor tud! Megadhatjuk ugyanis nem csak 1 bit vizsgálatát, de akármennyit is (illetve maximum 8-at mert annál több nem fér egy bájtbba...). Ennek formátuma a következő:

**%[a,b]**

Ahol az "a" és a "b" egyaránt egy-egy **#c** típusú aritmetikai kifejezés lehet, az "a" mondja meg, melyik a kiolvasandó bit-tartomány első bitje, a "b" pedig hogy melyik az utolsó kiolvasandó bit. Kivéve amikor nem így van, tudniillik meg-



adhatjuk fordított sorrendben is, abból se lesz semmi baj... értelemszerűen a legnagyobb megadható bitérték itt a 7, mert ugye a bitek 0-tól 7-ig számozódnak, de semmi hibajelzést nem fogunk kapni e tartomány átlépésekor, mert amint beolvasta a számokat az interpreter, azzal kezdi hogy végrehajt rajtuk egy **&&7** műveletet, vagyis csak az alsó 3 bit értékét nézi...

Lássunk erre is egy példát:

```
#!mau
{|16;
?k 96+{|}; " :: " ?c 96+{|}; " = %" {|8 ?c (96+{|})%[? -]; |}
" 1-3 bitek értéke: " ?c (96+{|})%[1,3]; /;
|}
XX
```

Eredménye:

```
` :: 96 = %01100000 1-3 bitek értéke: 0
a :: 97 = %01100001 1-3 bitek értéke: 0
b :: 98 = %01100010 1-3 bitek értéke: 1
c :: 99 = %01100011 1-3 bitek értéke: 1
d :: 100 = %01100100 1-3 bitek értéke: 2
e :: 101 = %01100101 1-3 bitek értéke: 2
f :: 102 = %01100110 1-3 bitek értéke: 3
g :: 103 = %01100111 1-3 bitek értéke: 3
h :: 104 = %01101000 1-3 bitek értéke: 4
i :: 105 = %01101001 1-3 bitek értéke: 4
j :: 106 = %01101010 1-3 bitek értéke: 5
k :: 107 = %01101011 1-3 bitek értéke: 5
l :: 108 = %01101100 1-3 bitek értéke: 6
m :: 109 = %01101101 1-3 bitek értéke: 6
n :: 110 = %01101110 1-3 bitek értéke: 7
o :: 111 = %01101111 1-3 bitek értéke: 7
```

Az értékadás egy bitmezőnek, azaz a bitmezők használata balértékként kicsit trükkös szintaktikájú. Nézzük csak meg e programot:

```
#!mau
(,,"binkiír" #c@c);
#c@c,%[1]=1; (,,"binkiír" #c@c);
#c@c=255; (,,"binkiír" #c@c);
#c@c,%[1]=0; (,,"binkiír" #c@c);
#c@c=255; (,,"binkiír" #c@c);
#c@c,%[1,6]=10; (,,"binkiír" #c@c);

XX

,,"binkiír" // Kiírja binárisan a számot
#c@c;
?c @c; " = %" {|8 ?c (@c)%[? -]; |} /;
xx
```

Eredménye:

```
0 = %00000000
2 = %00000010
255 = %11111111
253 = %11111101
255 = %11111111
149 = %10010101
```

Na most itt az a kérdés, miért is szerepel az értékadásban egy vessző, például itt:

```
#c@c,%[1]=1;
```

Hát azért, mert sajnos muszáj, ugyanis ha nem lenne ott, akkor a korábban elmondottak miatt a **c%[1]** a "c" karakter ASCII kódjának 1-es bitjének értékét jelentené, ami épp 1, s így az 1-es változó nevét jelentené az egész, tehát ekkor a fenti kifejezés vessző nélkül, tehát ez:

```
#c@c%[1]=1;
```

ekvivalens lenne ezzel:

```
#c@1=1;
```

Ami ugye nem jó nekünk. Sajnos zárójelzéssel se oldhatjuk fel ezt a problémát, mert a

```
#c@(c)%[1]=1;
```

jelentése ugyanez volna. Azaz, kellett valami szeparátor a változónév után, ami „megakasztja” a kifejezés további kiértékelését ideiglenesen, amíg az értékadó függvény beolvassa a változó nevét. Elnézést kérek ezért, igazából nekem se tetszik hogy így van. Na de most mit lehet tenni... Elkerülhető volna ez, ha jobbérték-szerepben lekorlátoznám az operátor szerepkörét arra, hogy kizárólag változókból olvassa be a bitmezőket, és ne tetszőleges aritmetikai kifejezésekből. Na de erre viszont nem vagyok hajlandó, ez túl nagy áldozat(=funkcionalitás-csökkenés) lenne e kis szépséghiba feloldásáért cserébe.

A bitmezők kizárólag a **#c** típusokra alkalmazhatóak! Semmi értelmét nem láttam ugyanis annak, hogy megcsináljam a többi típusra. Ez arra való, hogy könnyen tudjunk 1 bájt nál kisebb értéktartományba is beleférő adatokat 1 bájtba tömöríteni. Ezekből balérték-szerepben kizárólag az "=" operátor vonatkoztatható rájuk, vagyis maga az értékadás, semmi más, se a +=, se a -=, semmi. Nincs értelme ilyesmivel agyonbonyolítani a kódot.

**#c** tömbök esetén ilyen a szintaxis:

```
#!mau
[#c@t=10]; // Memóriefoglalás 10 darab unsigned char értéknek a "t" nevű tömbbe
#c@t[3]=255; („binkíír” #c@t[3]);
#c@t[3],[1]=0; („binkíír” #c@t[3]);
XX

„binkíír” // Kiírja binárisan a számot
#c@c;
?c @c; " = %" {|8 ?c (@c)%[? -]; |} /;
xx
```

Stringeknél:

```
#!mau
#s@s="0123456789";
#s@s[3],[1]=0; ?s @s; /; („binkíír” #s@s[3]);
#s@s[3],[2,6]=10; ?s @s; /; („binkíír” #s@s[3]);
XX

„binkíír” // Kiírja binárisan a számot
#c@c;
?c @c; " = %" {|8 ?c (@c)%[? -]; |} /;
xx
```

Stringtömböknél:

```
[@t[[4]]]; // lefoglalok területet 4 string részére egy t nevű tömbbe
// Feltöltöm értékekkel a tömböt
#s@t[[0]]="kutya";
#s@t[[1]]="0123456";
#s@t[[2]]="vadbarom";
#s@t[[3]]="cica";
"Kiíratom a 4 elemű stringtömb mindegyik értékét:\n"
{|4 ?c {|}; ". = " ?s @t[{|}]; /; |}
"Az 1 indexű string 3 indexű karakterét kicserélem:\n"
#s@t[[1]][3],[1]=0; („binkíír” #s@t[[1]][3]);
?s @t[[1]]; /;

"Felszabadítom a stringtömb memóriaterületét.\n"
```

```

[!t[[]]];
XX

„binkir” // Kiírja binárisan a számot
#c@c;
?c @c; " = %" {|8 ?c (@c)%[? -]; |} /;
xx

```

Eredménye:

```

Kiíratom a 4 elemű stringtömb mindegyik értékét:
0. = kutya
1. = 0123456
2. = vadbarom
3. = cica
Az 1 indexű string 3 indexű karakterét kicserélem:
49 = %00110001
0121456
Felszabadítom a stringtömb memóriaterületét.

```

BETŰknél:

```

#!mau
#u@u="a";
(„binkir” #u@u);
#u@u[0],%[1]=1; („binkir” #u@u);
#u@u[0]=255; („binkir” #u@u);
#u@u[0],%[1]=0; („binkir” #u@u);
#u@u[0]=255; („binkir” #u@u);
#u@u[0],%[1,6]=10; („binkir” #u@u);
XX

„binkir” // Kiírja binárisan a számot
#u@u;
?u @u; " = %" {|8 ?c (#u@u[0])%[? -]; |} /;
xx

```

Eredménye:

```

a = %01100001
c = %01100011
ÿ = %11111111
ý = %11111101
ÿ = %11111111
= %10010101

```

JELSOR egy BETŰjének egy bájtjánál:

```

#!mau
#U@u="alsó";
(„binkir” #U@u);
#U@u[0][0],%[1]=1; („binkir” #U@u);
#U@u[0][0]=255; („binkir” #U@u);
#U@u[0][0],%[1]=0; („binkir” #U@u);
#U@u[0][0]=255; („binkir” #U@u);
#U@u[0][0],%[1,6]=10; („binkir” #U@u);
XX

„binkir” // Kiírja binárisan a számot
#U@u;
?U @u; " = %" {|8 ?c (#U@u[0][0])%[? -]; |} /;
xx

```

Eredménye:

```

alsó = %01100001
clsó = %01100011
ÿlsó = %11111111
ýlsó = %11111101
ÿlsó = %11111111
lsó = %10010101

```

JELSORtömbben levő JELSOR egy BETŰjének egy bájtjánál:

```
#!mau
[#U@u=5];
#U@u[[3]]="alsó";
(,,"binkíír" #U@u[[3]]);
#U@u[[3]][0][0],[1]=1; (,,"binkíír" #U@u[[3]]);
#U@u[[3]][0][0]=255; (,,"binkíír" #U@u[[3]]);
#U@u[[3]][0][0],[1]=0; (,,"binkíír" #U@u[[3]]);
#U@u[[3]][0][0]=255; (,,"binkíír" #U@u[[3]]);
#U@u[[3]][0][0],[1,6]=10; (,,"binkíír" #U@u[[3]]);
XX

,,binkíír" // Kiírja binárisan a számot
#U@u;
?U @u; " = %" { |8 ?c (#U@u[0][0])%[-]; |} /;
xx
```

Eredménye ugyanaz mint az előzőnél.

Azaz látható, a bitmezők minden olyan helyen alkalmazhatóak, ahol **#c** típusú bájtokról, tehát „unsigned char” értékekről van szó.

## 39. fejezet - Egzisztenciafüggvények

E titokzatos névvel hogy „egzisztenciafüggvény”, olyan „micsodákat” illetek, amik valaminek a valamilyen állapotát vizsgálják meg, például azt, hogy az az izé amit vizsgálunk, létezik-e egyáltalán, vagy miféle tulajdonságú. Ezek közös tulajdonsága, hogy mind **#c** értékkel térnek vissza, ráadásul ezen érték kizárólag 0 vagy 1 lehet, azaz tulajdonképpen egy logikai érték, s eképp kiválóan felhasználható az **if** vagy a **ha** utasításban. Ezen egzisztenciafüggvények közül igen sok tulajdonképpen a bash shell „test” funkciójának egyes részeit valósítják meg, mint majd látni fogjuk, sőt még a szintaxis is igencsak hasonló.

A szintaxis tehát:

```
[|x] valami
```

vagy

```
[|!x] valami
```

ahol az „x” egy **#c** típusú aritmetikai kifejezés, ami azt határozza meg, hogy épp pontosan melyik egzisztenciafüggvényt hívjuk meg, a „valami” pedig az a valami, ami ezen egzisztenciafüggvénynek a paramétere. Ennek a valamilyen állapotát vizsgálja az egzisztenciafüggvény.

A fent megadott két szintaxisforma kizárólag abban különbözik egymástól, hogy a felkiáltójeles változat NEGÁLJA az egzisztenciafüggvény által egyébként visszaadandó értéket, azaz ha az nullát adna vissza amúgy akkor 1 lesz belőle és fordítva.

Van továbbá ilyen egzisztenciafüggvény is:

```
[=x] valami
```

vagy

```
[=!x] valami
```

Itt ugyanazok a szabályok érvényesek mint az előző szintaktikájúnál, de ezek természetesen másfajta egzisztenciafüggvények, mást csinálnak. Nagy általánosságban azt mondhatjuk, hogy a fájlokkal összefüggő függvények a **[!x]** alakúak, a karakterosztályozóak a **[=x]** alakúak.

Például nézzük e programot:

```
#!/mau
#c@c=[|f]"main.o";
ha(@c) "Van ilyen file!\n"
E "Nincs ilyen file!\n"
ha [|f]"nincsilyenfile.txt" "Van ilyen file!\n"
E "Nincs ilyen file!\n"
ha [|f]"szotar" "Van ilyen file!\n"
E "Nincs ilyen file!\n"
XX
```

Szerintem könnyen kitalálható, mit csinál e függvény: A **[!f]** egzisztenciafüggvény megvizsgálja, az utána megadott stringkifejezéssel meghatározott nevű állomány létezik-e. Pontosabban, hogy korrektül fogalmaznak, azt vizsgálja meg, hogy létezik-e olyan nevű „közönséges”, azaz „reguláris” állomány. Mert hiába létezik olyan néven mondjuk egy könyvtár, akkor is nullát ad eredményül, mert ugye a könyvtár, az nem fájl.

Mindenféle más tulajdonságokat is vizsgálathatunk így állományokról, a megfelelő egzisztenciafüggvény egykarakteres neve megegyezik a bash „test” függvényénél ilyenkor megadandó karakterrel, amint az látható e kis programból:

```
#!/mau
#s@s="nullafile.txt"; („Egzisztencia” #s@s);
#s@s="szotar"; („Egzisztencia” #s@s);
#s@s="tesztelek.mau"; („Egzisztencia” #s@s);
#s@s="IlyenFileNemletezik"; („Egzisztencia” #s@s);
```

XX

```
„Egzisztencia” //
#s@s;
"E file tulajdonságai: " ?s @s; /;
ha [|f] @s "Van ilyen szabályos file!\n"
E "Nincs ilyen szabályos file!\n"
```

```
ha [|e] @s "Létezik ilyen file!\n"
E "Nem létezik ilyen file!\n"
```

```
ha [|d] @s "Létezik ilyen directory!\n"
E "Nem létezik ilyen directory!\n"
```

```
ha [|s] @s "A file mérete nem nulla!\n"
E "A file mérete nulla!\n"
```

xx

A jelenleg megvalósított egzisztenciafüggvények jelentése:

**[!b] #s@s**

Értéke akkor 1, ha létezik a stringkifejezéssel megadott nevű olyan állomány, mely blokkeszköz.

**[!c] #s@s**

Értéke akkor 1, ha létezik a stringkifejezéssel megadott nevű olyan állomány, mely karakteres eszköz.

**[!d] #s@s**

Értéke akkor 1, ha létezik a stringkifejezéssel megadott nevű olyan állomány, mely egy directory (azaz tartalomjegyzék).

[|e] #s@s

Értéke akkor 1, ha létezik egy a stringkifejezéssel megadott nevű bármiféle állomány.

[|f] #s@s

Értéke akkor 1, ha létezik a stringkifejezéssel megadott nevű olyan állomány, mely „szabályos” (reguláris) fájl.

[|g] #s@s

Bevallom nem vagyok teljesen biztos benne ez mi a csuda: mindenesetre ez is az amit a bash „test” parancsánál a „g” csinál, erről az angol leírás ezt állítja: True if *file* has its set-group-id bit set. Sajnos a magyar man parancs is csak annyit ír erről, hogy az esetben 1, ha a fájl létezik, és „set-group-id”-s fájl.

[|k] #s@s

Értéke akkor 1, ha be van állítva a fájl "sticky" bitje.

[|L] #s@s

Értéke akkor 1, ha létezik a stringkifejezéssel megadott nevű olyan állomány, mely egy link.

[|p] #s@s

Értéke akkor 1, ha létezik a stringkifejezéssel megadott nevű olyan állomány, mely egy „FIFO” fájl (pipe).

[|r] #s@s

Értéke akkor 1, ha létezik a stringkifejezéssel megadott nevű olyan állomány, mely olvasható.

[|S] #s@s

Értéke akkor 1, ha létezik a stringkifejezéssel megadott nevű olyan állomány, mely egy „Socket” fájl.

[|s] #s@s

Értéke akkor 1, ha létezik a stringkifejezéssel megadott nevű olyan állomány, melynek a mérete nem nulla.

[|u] #s@s

Bevallom nem vagyok teljesen biztos benne ez mi a csuda: mindenesetre ez is az amit a bash „test” parancsánál a „u” csinál, erről az angol leírás ezt állítja: True if *file* has its set-user-id bit set. Sajnos a magyar man parancs is csak annyit ír erről, hogy „Igaz ha a file létezik és a set-user-id bitje be van állítva”.

[|w] #s@s

Értéke akkor 1, ha létezik a stringkifejezéssel megadott nevű olyan állomány, mely írható.

[|x] #s@s

Értéke akkor 1, ha létezik a stringkifejezéssel megadott nevű olyan állomány, mely végrehajtható.

Példa karakterosztályozó egzisztenciafüggvényekre:

```
#!mau
#c@c=a; („Egzisztencia” #c@c);
#c@c=V; („Egzisztencia” #c@c);
#c@c='/'; („Egzisztencia” #c@c);
#c@c='8; („Egzisztencia” #c@c);
```

XX

```
„Egzisztencia” //
#c@c;
"E karakter tulajdonságai: " ?k @c; " = " ?c @c; /;
ha [= '@] @c "Ez egy betű!\n"
E "Ez nem betű!\n"
ha [= a] @c "Ez egy kisbetű!\n"
E "Ez nem kisbetű!\n"
```

```
ha [=A] @c "Ez egy NAGYbetű!\n"
E "Ez nem NAGYbetű!\n"
ha [=2] @c "Ez egy alfanumerikus karakter!\n"
E "Ez nem alfanumerikus karakter!\n"
ha [=9] @c "Ez egy decimális számjegy!\n"
E "Ez nem decimális számjegy!\n"
xx
```

Azt hiszem a program kimenete könnyen kitalálható.

Az eddig megvalósított karakterosztályozók:

[='@]

Az angol ABC valamely kis- vagy nagybetűje.

[='a]

Az angol ABC kisbetűje.

[='A]

Az angol ABC NAGYbetűje.

[='2]

Alfanumerikus karakter, azaz az angol ABC kis- vagy NAGY betűje, vagy számjegy.

[='3]

Alfanumerikus karakter vagy aláhúzásjel.

[='9]

Decimális számjegy.

## 40. fejezet - Streamkezelés

Képzeljük el a következő szituációt: A programunknak adatokat kell beolvasnia valahonnan, de nem tudjuk előre, hogy mennyi adat fog érkezni! Rengetegszer lehet ilyen helyzet. És az még a jobbik eset, ha legalább annyit megsaccolhatunk, hogy mennyi a MAXIMÁLIS száma az adatoknak, amennyinél több tehát biztosan nem lesz. Ezesetben ugyanis megtehetjük hogy annyinak allokálunk memóriát (a mai nyelvben mondjuk annyinak foglalunk le tömböt), bár ez kétségkívül rém memóriapazarló, s emiatt nem hatékony és nem szép megoldás. De ez még mindig a jobbik eset, tényleg, mert sokszor még sejtésünk se lehet az adatok maximális mennyiségéről! (se).

Például: be kell olvasnunk egy fájlt soronként, de e sorokból csak bizonyosak kellenek nekünk, melyek eleget tesznek egy feltételnek. Mondjuk hogy azok kellenek csak amelyek tartalmazznak egy bizonyos szót. E sorokat kell elmentenünk valahova, további feldolgozás céljából. Azt tudjuk, mekkora a fájl BÁJTMÉRETE, de arról fogalmunk sincs, hány sor van benne, mert a sorok különböző hosszúságúak. Arról se lehet fogalmunk se, hány olyan sor van benne ami nekünk kell. Mit tehetünk ekkor?!

A megoldás a STREAMkezelés. Ez azt jelenti, hogy szépen beolvassuk a fájlt, az egészet természetesen, soronként ahogy nekünk kell, e sorokat leteszteľjük hogy kellenek-e nekünk, s amelyek kell, azt kiírjuk... Eddig oké. De hová is írjuk ki? Nos, egy fájlba! Logikus, hiszen egy output fájl akármekkora is lehet. Igenám, de e fájl nem a lemezen nyitjuk meg, hanem a MEMÓRIÁBAN! Ez messze sokkal gyorsabb, előkelőbb, hiszen úgyis csak ideiglenesen kell... Aztán e fájlba beleírunk mindent a különben is jólismert, megszokott kiíró utasításainkkal, mindent amit akarunk, majd lezárjuk. S ezután a létrejött memóriaterületet mely tartal-



mazza az értékes adatokat, kezeljük valahogyan... Nos, erre nyújt lehetőséget a mau nyelv streamkezelő szolgáltatása! Mindjárt rögvest íme egy példaprogram, majd a magyarázat. A példaprogram a korábban bemutatott szótárprogi, átírva a streamkezelésnek megfelelő előkelőbb verzióra. Ez az a változat amikor a keresendő kifejezést a vágólapról olvassa be („mobi-mouse stílus”):

```
#!/mau
// ----- Konstansok beállítása
#s@N="/_P/Szotar/0/angol/felnottno/";
#s@S="/_P/Szotar/0/angol/angolmagyar.szotar"; // Szótárfile neve
#s@k=?#s "C"; // A keresendő szó vagy kifejezés beolvasása a vágólapról
ha [|f] (@N)+(@k)+".ogg" SY "ogg123 -q "+(@N)+(@k)+".ogg";
#K@o=|; // Megnyitjuk írásra az output fájlt streamra a memóriába
if #K@o E "Az output file nem megnyitható!\n" XX
{-} @S, $su; // Lefuttatjuk a „su” nevű szubrutint a fájl minden sorára
T "Az input file nem megnyitható!\n" XX
[#K@o]; // Lezárom az output fájlt
#s@s=#K@o; // beolvasom a streamból a stringet
[#K@o]; // Törölöm a streamot is
||| @s, t, 10; // szétdarabolom sorokra a stringet, a „t” stringtömbbe
MC f, "-*-fixed-*-*-*-*28-*-*-*-*-*";
#s@v=?#s "D" t; // Kiválasztjuk a dmenu-funkcióval a tömbből a kívánt tételt
#s@P="zeneneve=~"+echo \"\"+(@v)+\"\"+\" | sed -e 's/ *::[0-9]*///' -e 's/ /_/'g'"+
\"; zeneneveogg=\"\"+(@N)+\"$zeneneve\".ogg\"; "+
"if [ -f $zeneneveogg ] ; then ogg123 -q $zeneneveogg ; fi;";
SY @P;

XX

$su // E szubrutin annyiszor fut le, ahány sor van a szótárfájlban
#s@s=?#s "{-}"; // Az aktuális beolvasott sor
if(#L(?#L "POS" @s,@k)>=0); // Ha megtalálta a beolvasott sorban a keresendő szót, akkor:
-> #s@s=?#s "<<<" @s, " ::", " ::";
#s@s=?#s "<<<" @s, " ::", " ::";
#s@s=?#s "<<<" @s, "t", "n => ";
#s@s=?#s "<<<" @s, "<BR>", "<br>";
#s@s=?#s "<<<" @s, "<br>", "n => ";
<s @o, @s; // Kiírjuk a sort a fájlba
<-
« // vége a szubrutinnak
```

Most újra idemácsolom a fenti programot, s jön a részletes magyarázat a sorok közé írva:

```
#!/mau
// ----- Konstansok beállítása
#s@N="/_P/Szotar/0/angol/felnottno/";
```

A fenti sor tartalmazza azon könyvtár nevét, ahol a hangállományok vannak ogg formátumban. Azért „felnottno” a neve, mert egy felnőtt amerikai nő mondja ki a szavakat.

```
#s@S="/_P/Szotar/0/angol/angolmagyar.szotar"; // Szótárfile neve
```

E fenti sor a szótárfájl neve ami a szöveget (sorokat) tartalmazza.

```
#s@k=?#s "C"; // A keresendő szó vagy kifejezés beolvasása a vágólapról
ha [|f] (@N)+(@k)+".ogg" SY "ogg123 -q "+(@N)+(@k)+".ogg";
```

A fenti sor leteszteli, létezik-e hangállomány a keresendő kifejezéshez, és ha igen, lejátssza a megfelelő progival, a „system” parancon keresztül.

```
#K@o=|; // Megnyitjuk írásra az output fájlt streamra a memóriába
```

Na itt fent kezdődik a lényeg: A **#K** típus már ismerős nekünk: output fájlt jelent. Egy effélet szokásosan úgy nyitunk meg, hogy stringkifejezést adunk neki értékül. Most azonban ott valami más van: a string helyén egy „pipe” jel áll... Ez azt mondja a mau interpreternek, hogy a memóriában nyisson meg egy streamot, azaz ott allokáljon egy memóriaterületet, még hozzá dinamikusan, ami tehát mindig akkora, amekkora épp kell a belerakandó akármiféle adatoknak. S e

memóriaterülethez társítsa a **#K@o** outputfile fájl-leíróját (mutatóját). Ennyi az egész, a továbbiakban teljesen ugyanúgy kezelhetjük a **#K@o** outputfile-változónkat, mintha egy „rendes”, „szokványos” fájl nyitottunk volna meg neki a lemezen. Természetesen e stream-módon is 256 különböző output fájl kezelhetünk egyszerre, azaz semmi se változott.

```
if #K@o E "Az output file nem megnyitható!\n" XX
{-} @S, $su; // Lefuttatjuk a „su” nevű szubrutint az input fájl minden sorára
T "Az input file nem megnyitható!\n" XX
[#K@o]; // Lezárom az output fájlt
```

Na itt fent van ugye megint egy „lényeg”. Ha már nem irkálunk többet a streamba, le kell zárni az output fájlt, épp úgy, ahogy azt egy „normális” fájlal is tennénk. Ez teljesen logikus. Az a kérdés azonban nyilván felmerül az Olvasóban, hogy akkor nem tűnnek-e el a korábban beleírt Fontos Adataink?! Hiszen ennek az izének még csak neve sincs amivel megkereshetnénk valami directoryban!

Hogyne lenne neve... Hiszen az a neve, hogy **#K@o**... A fájl lezárása csak annyit jelent, hogy végrehajtja az **fflush** és **fclose** C nyelvű parancsokat a **FILE\*** pointerre. Ellenben nem törli azt a **char\*** mutatót, ami a streamnek lefoglalt memóriaterületre mutat... Azt - hacsak direkt nem utasítjuk erre - kizárólag a „destruktor” törli, azaz akkor törlődik „magától”, ha megszűnik az egész névtér...

```
#s@s=#K@o; // beolvasom a streamból a stringet
```

Na itt történik a fenti sorban a „nagy trükk”... Amint a fenti bekezdésben említettem, a lefoglalt memóriaterületen épen megmaradt a sok Roppant Okosság amit az adataink tartalmaznak... Ezeket nemes egyszerűséggel egyetlen stringként olvassuk be. Ez a beolvasás sem törli különben azt a memóriaterületet, azaz akárhányszor akár mennyi stringbe is beolvashatnánk még. Természetesen ezek után akármit művelünk is a stringgel (e példában a **#s@s** jelűvel) az nem befolyásolja a stream memóriaterületének eredeti adatait, azok változatlanok maradnak, mert a stringbe beolvasáskor nem a pointer kerül átadásra, hanem az egész mindenség lemásolódik, duplikálódik!

Ebből persze máris tudható az a korlátozás, hogy efféle módon maximum akkora streamokat kezelhetünk, melyek mérete nem haladja meg az egy stringbe beférő maximális bájt számot. Jelenleg a **#s** stringek maximális mérete egy **mau\_1** azaz **#1** típusú változóval van megadva, azaz egy effélének a mérete legfeljebb 2 a 32-ediken darab bájt lehet. Ez ha jól számolom, 4 gigabájt. Hát ez azért talán elég egyetlen streamnak...

```
[#K@o]; // Törölöm a streamot is
```

Na itt fent történik az, hogy direkt, explicit módon utasítjuk arra az interpretert, hogy a korábban a streambe beleírt adatokat felejtse el, szabadítsa fel a nekik lefoglalt memóriaterületet. Ezek után többé nem tudjuk azokat újabb stringekbe beolvasni, de természetesen azon stringek melyekbe korábban olvastuk be ezen adatokat, azok tartalma nem vész el.

```
||| @s, t, 10; // szét darabolom sorokra a stringet, a „t” stringtömbbe
```

Ezen utasítás tulajdonképpen nem stream-függő, de logikusan következik a stream lényegéből: nekünk az adatokat szét kell szedni részekre! A streamba az adatok általában úgy kerülnek, hogy sorvég-jelek zárják le őket. Na itt fent ezek mentén daraboljuk szét a stringben tárolt adatokat.

A többi utasítást már nem magyarázom itt el, mert „offtopik” lenne a téma szempontjából.

## 41. fejezet - Az ncurses integráció

A mau interpreter „kulcsra készen” szállítja nekünk a lehetőséget parancssoros, ám mégis „ablakos” programok írásához. Ehhez szüksége van az ncurses librarykra (és fejlécállományokra). Amennyiben e lehetőséget nem igényeljük, s meg óhajtunk szabaulni e függőségétől, úgy annyi a teendőnk csak, hogy fordítás előtt a **vz.h** állományban kitöröljük vagy kikommentezzük e sort:

```
#define NCURSESINTEGRATION
```

valamint a Makefile fájlban e sor végéről:

```
LIBS = -ldl -lX11 -lXmu -lncurses
```

töröljük le a **-lncurses** részt.

Amúgy az ncursessel összefüggő összes utasítás és függvény elvileg az **nc.cpp** fájlban kell szerepeljen (aminek a headerfájlja az **nc.h**), de persze nem kizárt hogy elcsesztem valamit s máshova is került belőle ez-az. A jó szándék mindenesetre megvolt bennem ehhez, de hát én csak egy csekélyértelmű medvebocs vagyok, szóval bocs...

Az ncursessel összefüggő majdnem mindegyik mau utasítás az **N** betűvel kezdődik, így aránylag könnyű őket észben tartani.

Tehát a megfelelő mindenféle függvények, utasítások:

### Ncurses-sel összefüggő mau utasítások

**NC;**

Ez valósítja meg az „initscr” funkciót, ezzel kell kezdődjön az ncurses üzemmódba lépés.

**NX;**

Ez az „endwin” funkció, ezzel lépünk ki az ncurses üzemmódból.

**NR;**

Ez a „refresh”, ezzel történik a tényleges kiiratás a valódi (nem virtuális) képernyőre.

**Ns S;**

ahol az S egy stringkifejezés. Ez az utasítás ír ki egy stringet az ncurses virtuális képernyőjére, az aktuális kurzorpozíciótól kezdve.

**RAW x;**

ahol az x egy unsigned char kifejezés. Ha  $x > 0$ , akkor a RAW engedélyezett, ha 0 akkor nem engedélyezett. A „RAW” az úgynevezett „nyers beviteli mód”.

**NB x;**

ahol az x egy unsigned char kifejezés. Ha  $x > 0$ , akkor a cbreak engedélyezett, ha 0 akkor nem engedélyezett.

A **RAW** és a **cbreak** letiltja a soronkénti bufferelést. A különbség a két megoldás között a vezérlőkarakterek feldolgozásában rejlik. Ezen vezérlőkarakterek a **Ctrl-Z** és a **CTRL-C**. A **RAW** módban az alkalmazás közvetlenül elkapja ezeket a karaktereket, és nem generál szignált. Ellenben **cbreak** módban a vezérlőkaraktereket értelmezi a terminál-kezelő, de a szerkesztőkaraktereket (pld a **Backspace**) már az alkalmazásnak kell feldolgoznia.

**NE x;**

ahol az x egy unsigned char kifejezés. Ha  $x > 0$ , akkor az „echo” engedélyezett, ha 0 akkor nem engedélyezett. Az „echo” azt jelenti, hogy a leütött karakterek „visszhangozzanak-e” a terminálon, azaz kiírásra kerüljenek-e.

**NK x;**

ahol az x egy unsigned char kifejezés. Ha x nem 0, akkor engedélyezett, hogy beolvassuk a funkcióbillentyűket (**F1**, **F2**, stb), a kurzormozgató gombokat, és egyéb speciális karaktereket. Ha az x nulla, akkor ez nem engedélyezett. Ez tulajdonképpen az ncurses „keypad” funkciója/függvénye.

## **Ncurses-sel összefüggő függvények**

**?#L "N" x**

ahol az x egy unsigned char kifejezés, a várakozás idejét határozza meg tizedmásodpercekben. E függvény várakozik valahány tizedmásodpercig egy karakter bevitelére, ha nincs leütés, továbblép. Ezesetben a visszatérési érték az ERR. (Ez az ERR egy ncurses-es konstans vagy makró vagy mi a szósz, mindenesetre nálunk az értéke **-1**, azaz mínusz 1).

(Ennyi, bocs de mint ahogy írtam is e tanulmány elején, ezen szolgáltatás a mai nyelvben befejezetlenül implementált).