ÁTS LÁSZLÓ

turbo PASCAL

kezdőknek

NOVOTRADE

ÁTS LÁSZLÓ: TURBO PASCAL KEZDŐKNEK

ATS LÁSZLÓ

TURBO PASCAL
kezdőknek

NOVOTRADE KIADÓ, 1990

Lektorálta: dr. Csébfalvi György
Hegedüs Sándor

Második, változatlan kiadás
A kiadványért felel Rényi Gábor,
a NOVOTRADE Rt. vezérigazgatója
Budapest, 1990
ISBN 963 585 082 4

készült a Somogy Megyei Nyomdaipari Vállalat
Kaposvári üzemében (44 A/5 ív)
Felelős vezető: Mike Ferenc igazgató

B E V E Z E T E S

Ezzel a könyvvel abban szeretnénk segíteni az Olvasót, hogy csatlakozzon a Turbo Pascal programozók mintegy 700 000-es táborához. Ehhez semmiféle előismeretet nem igénylünk sem a programozás, sem pedig általában a számítástechnika területéről, csak némi általános problémamegoldási intelligenciát, motivációt és kitartást.

Kitarásra annál is inkább szükség lesz, mert - bár nulláról indulunk - az anyag feldolgozása meglehetősen tempós, így esetenként komoly erőfeszítést igényel. A nehézségeken a könyvben feldolgozott bő példaanyaggal és a fejezetekhez kapcsolódó számos feladattal szeretnénk átsegíteni az Olvasót.

Programozást tanítunk, s ehhez a Turbo Pascal nyelvet használjuk. Ez a Pascal megvalósítás eltér a szabványtól. Az eltérésekkel - s általában a programok hordozhatóságának (más gépen, más környezetben való használhatóságának) kérdésével - e könyvben nem foglalkozunk.

A hangsúlyt a módszeres és jó stílusú programozási gyakorlatra helyeztük. Az elméleti hátteret csak ott mutatjuk meg, ahol ez nem vezet nagyon messzire. Ilyenkor esetenként elemi matematikai fogalmakat is használunk. Ez ne riassza el az Olvasót, mert e részek elolvasását a programozási gyakorlat szempontjából mellőzheti. Erre mégsem biztatjuk, mert meggyőződésünk, hogy igazán jó programozó csak matematikailag képzett ember lehet.

A matematikai ismereteken kívül az angol nyelvismeret is fontos egy leendő informatikus szakember számára. Ezért az MS-DOS és a Turbo Pascal rendszer angol nyelvű üzeneteit - bár értelmezzük - nem fordítjuk magyarra. Bogarássza ki a kifejezések értelmét az Olvasó szótár segítségével! (Ebben segítségére lehet a NOVOTRADE Kiadó gondozásában megjelent Homonnay: Angol-Magyar számítástechnikai szótár 1989-ben megjelent 2. átdolgozott kiadása is.) Bármilyen szoftverrel dolgozik a jövőben, ugyanezt kell tennie. (Ugyanakkor a hibaüzenetek magyar fordítását a Mellékletben megadjuk.)

A fejezetek végén összegyűjtött feladatok között könnyűek és nehezek is előfordulnak. A megoldást minden esetben célszerű önállóan megkísérelni, az illető fejezetben fellelhető példaanyagra támaszkodva. Ha sikerül a megoldás, azért érdemes lehet a Mellékletben közölt programot is megnézni, mert néha itt mutatunk be újabb programozási eszközöket vagy technikákat. Ne keseredjen el az Olvasó, ha a szerző megoldása eltér az övétől.

Ez természetes. Egy feladat nagyon sokféleképpen is megoldható. Bizonyára nem egy esetben az Olvasó megoldása lesz a jobb.

Nagyon fontos, hogy a programokat számítógépen ténylegesen kipróbáljuk. Csak könyvből nem lehet megtanulni programozni, ebből a könyvből sem! Ehhez gép is kell. A könyvben található példaprogramok és feladatmegoldások nagy része alprogram, önmagában nem futtatható. A futtatáshoz szükséges meghajtó programot és további szükséges alprogramokat az Olvasónak kell megírnia.

Hacsak meg nem vásárolja a könyvhöz tartozó mágneslemezes mellékletet, amely a könyv összes programját és a feladatok megoldásait tartalmazza, de ezen kívül még sok minden mást. A feladatmegoldásokat - a terjedelmi korlátoktól némiképp megszabadulva - magyarázó kommentárokkal láttuk el. Ezen kívül mindegyik alprogramhoz megteremtettük azt a - meghajtó programból és esetleg további szükséges alprogramból álló - környezetet, amelyben az adott alprogram már futtatható. Ezzel a teljes programanyagot közvetlenül kipróbálhatóvá tettük.

Röviden a könyv tartalmáról:

Az 1. fejezetben az algoritmusokról, a programozás alapjairól nyújtunk ismereteket a teljesen kezdők számára. Itt egy magyar alapszavakból felépített pszeudo programnyelven mutatjuk be a programok legfontosabb szerkezeti elemeit.
A 2. fejezet célja teljesen gyakorlati. A számítógép, az MS-DOS és a Turbo Pascal 3.0-ás rendszer használatába nyújtunk itt bevezetést. Itt mindenféle elméleti alapvetés nélkül adjuk meg azokat a tudnivalókat, amelyek a további munkához nélkülözhetetlenek.
A 3. és a 4. fejezet a nyelv elemi eszköztárát és ennek használatát mutatja be.
Az 5. fejezet témája a nagyobb programok felépítése szempontjából nélkülözhetetlen alprogramok definiálása és használata. Kitérünk a rekurzív alprogramok használatára is - a rekurzió, mint problémamegoldási technnika a könyv további fejezeteiben is fontos szerephez jut.
A 6. fejezetben betekintést nyújtunk a programozás technológiájába.
A 7., 8., 9., 10. és 11. fejezet a Pascal nyelv adatszerkezeteit tárgyalja, szoros összefüggésben a feldolgozó algoritmusok szerkezetével és a nyelv programozási eszközeivel.
A 12. fejezetben betekintést nyújtunk a dinamikus adatszerkezetek - listák és fák - használatába.
A mellékletekkel a gyakorlati programozási munkát kívánjuk támogatni.

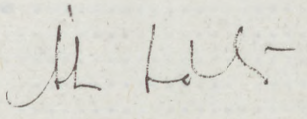
Kevés kivételtől eltekintve a Turbo Pascal 3.0-ás változatának eszközeit használjuk ebben a könyvben. Néhol - s itt elsősorban a 10.4-es alfejezetben tárgyalt "unit"-okra gondolunk - célszerűnek látszott kitekinteni az újabban megjelenő verziókra is. Mivel a 4.0-ás és az 5.0-ás változatok bővítései a 3.0-ásnak, úgy véljük,

nem vész kárba a könyv tanulmányozására fordított idő akkor sem, ha az olvasó a nyelv újabb változatát használja. A közölt programok elején azonban általában deklarálni kell a CRT unitot.

Szeretném hálámat kifejezni a könyv két lektorának, dr. Csébfalvi György kandidátus, egyetemi docensnek és Hegedűs Sándor villamosmérnöknek a kézirat és a programok gondos átnézéséért, a számos hasznos tanácsért és jobbító szándékú észrevételért. Minden ami a könyvben jó, az ő érdemük, a hibákért a szerző vállalja a felelősséget. Hálás vagyok a PMMF műszaki informatikus hallgatóinak, akik a könyv megírására inspiráltak. Külön köszönettel tartozom Csala György elsőéves hallgatónak, aki a 8. fejezet több feladatának megoldásában volt segítségemre.
Végül - de nem utolsósorban - köszönettel tartozom a NOVOTRADE Kiadónak a gondos szerkesztői munkáért és a könyv megjelentetéséért.

Az Olvasónak pedig jó munkát és eredményes tanulást kívánok.

Pécs, 1989. ápr. 26.



"Minden dolgokhoz
több utak vezetnek ..."

(Wesselényi)

T A R T A L O M J E G Y Z E K

P R O G R A M O K

A könyvben közölt programokat és a feladatok megoldását lemezen is megvásárolhatja az Olvasó. Az alprogramokat meghajtó programmal együtt rögzítettük, így ezek is futtathatók.

KÉRJE A KÖNYVESBOLTBAN!

1. TEVEKENYSEGEK LEIRASA

1.1 Algoritmusok

Legelőször az algoritmus fogalmával kell megismerkednünk.
A szó eredetével kapcsolatban el szokták mondani, hogy valószínűleg Al Khvarizmi arab matematikus nevéből származik, de - mint a matamatikában oly gyakran - maga a fogalom a hellenisztikus kor görög matematikájában bukkan fel először. Euklidész leírt egy módszert két természetes szám legnagyobb közös osztójának meghatározására, ezt a módszert euklidészi algoritmusnak nevezzük.
Algoritmus alatt valamely egyértelműen előírt módon és sorrendben végrehajtandó tevékenységet fogunk érteni. Ezt a mondatot nem definíciónak szántuk, az algoritmus fogalmát csak körülírni kívánjuk, nem definiálni.
Ha leírjuk egy másodfokú egyenlet megoldásának tevékenységét, akkor algoritmust adunk meg. Ehhez általában elegendő a következő:

"Az

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}$$

és az

$$x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

képletekben helyettesítsük

"a" helyébe a másodfokú,
"b" helyébe az elsőfokú tag együtthatóját,
"c" helyébe a konstans tagot!

Ha a négyzetgyök alatti kifejezés értéke nem negatív, akkor elvégezve a számolást, megkapjuk az egyenlet gyökeit. Egyébként az egyenletnek nincs valós gyöke."

Vegyük észre, hogy a tevékenységet egyszerűbb tevékenységekkel írtuk le. Ilyen tevékenységeket használtunk:

"helyettesíts "a" helyébe egy számot"

"számítsd ki egy kifejezés értékét".

Ezeket a tevékenységeket csak megfelelő előképzettségű ember képes minden további magyarázat nélkül megérteni és végrehajtani. Képzeljük el, ha végrehajtó egy kisgyerek, aki csak a négy alapműveletet ismeri! Hogyan írhatnánk le a számára a másodfokú egyenlet megoldását? Milyen résztevékenységekre bontanánk?
Másrészt azok számára, akik ismerik a másodfokú egyenlet megoldóképletét, a következő utasítás is elegendő:

"oldd meg a következő egyenletet:

$$x^2 - 4x + 3 = 0!"$$

Az eddigiekből két tanulságot szűrhetünk le:

1) ha egy tevékenységet a végrehajtó nem ért és nem tud közvetlenül végrehajtani, akkor olyan tevékenységekből kell felépíteni, amelyek számára is érthetők;

2) az algoritmus leírásához használható tevékenységek bonyolultsága a végrehajtó előzetes ismereteitől, felkészültségi szintjétől függ.

Célszerű bevezetni az elemi tevékenység fogalmát:

> Elemi tevékenységnek nevezzük az olyan legbonyolultabb tevékenységet, ami a végrehajtó számára közvetlenül megérthető és egyértelműen végrehajtható anélkül, hogy további résztevékenységekre kéne bontani.
>
> Azokat a tevékenységeket, amelyek elemi tevékenységekből állnak, összetett tevékenységeknek mondjuk.

Algoritmus alatt a továbbiakban egyértelmű és véges összetett tevékenységet fogunk érteni.
Az egyértelműség követelménye azt jelenti, hogy bármely elemi tevékenység végrehajtása után a következő elemi tevékenység egyértelműen meghatározott és korlátozás nélkül végrehajtható.
Egy összetett tevékenységet akkor nevezünk végesnek, ha véges sok elemi tevékenység elvégzése után befejeződik.
Ha az algoritmust számítógépes végrehajtásra szánjuk, programról beszélünk.

1.2 Elemi tevékenységek és a számítógép

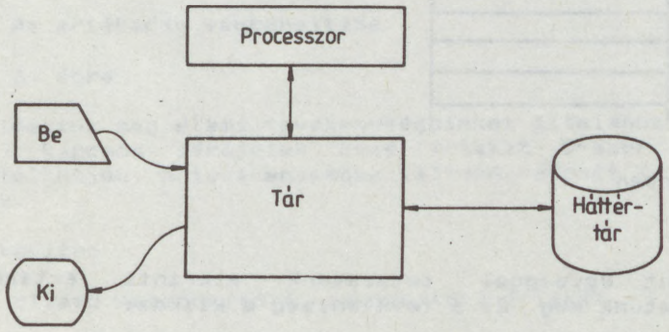
A bonyolultabb elemi tevékenységeket magasabb szintűeknek nevezzük. Minél magasabb szintű elemi tevékenységekkel dolgozhatunk, annál egyszerűbben fogalmazhatjuk meg az algoritmusokat. Láttuk, hogy a használható elemi tevékenységek szintjét a végrehajtó előzetes ismeretei, a végrehajtó

"felkészítettsége" határozza meg. Ha arra gondolunk, hogy egy számítógép "ismeretei" egy 8-10 éves gyerekét sem éri el, nem vagyunk boldogok a programozástól...
Ne csüggedjünk! A számítógépek a számolás terén és elsősorban a műveletvégzés sebességében bárkit felülmúlnak, másrészt "felkészítettségük" is növelhető. Egy magasszintű programozási nyelv - amilyen a Pascal is - nagyon kényelmes és elég magas szintű elemi tevékenységek használatát teszi lehetővé.
Valóban vannak olyan tevékenységek, amelyek elemiek egy gyerek, de nagyon nehezek a számítógépek számára. Ilyen pl. a következő:

> "Alkoss értelmes mondatokat a következő szavak felhasználásával:
> a, az, ég, fenyő, kék, zöld!"

Az ilyen problémák megoldásával a mesterséges intelligencia kutatások foglalkoznak (nem is eredménytelenül).

A továbbiakban azt fogjuk vizsgálni, milyen elemi tevékenységekre van szükségünk egy egyszerű számítógépi algoritmushoz, mint pl. a másodfokú egyenlet megoldása. Ehhez egy nagyon keveset tudni kell a számítógépekről is.



A számítógép felépítése

1. ábra

Nagyon leegyszerűsítve mutatjuk be a számítógép felépítését az 1. ábrán. A számítás adatai a bemeneti (input) egységen át jutnak a tárba, az eredményeket a kimeneti (output) egység teszi számunkra láthatóvá. Személyi számítógépeken a legelterjedtebb input egység a billentyűzet, output egység a monitor képernyője. A tárban lévő adatokkal a processzor aritmetikai műveleteket végez, a műveletek eredménye a tárba írható. Nincs akadálya bonyolultabb műveletek elvégzésének sem (gyökvonás, függvényértékek kiszámítása).
A tárban lévő adatokat a nevükkel azonosíthatjuk. A név a tár egy adott tárolóhelyét - rekeszét - jelöli, adatainkat a megfelelő

nevű rekeszben tartjuk (2. ábra).
Nézzük ezután milyen elemi tevékenységekre van szükség az

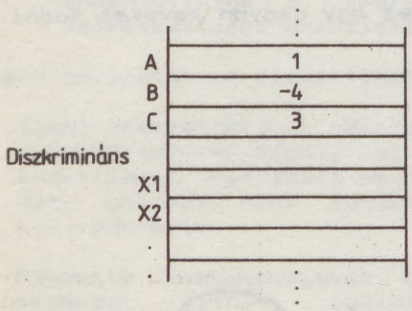
$$x^2 - 4x + 3 = 0$$

egyenlet megoldásához.

1) Az input egységgel a tár megfelelő rekeszébe kell juttatni az adatokat. Ez a tevékenység az adatbevitel:

   olvasd A, B, C értékét

   E tevékenység végrehajtásánál a gép egyszerűen megvárja amíg három számot a billentyűzeten beírunk. Az adatokat a tár megfelelő rekeszeibe írja, a rekeszek automatikusan kapják meg a megfelelő nevet. A rekeszek nevét nagybetűkkel fogjuk jelölni.
   A tár adatbevitel utáni állapotát a 2. ábrán láthatjuk.



   Adatok a tárban

   2. ábra

2) Az output egységgel tetszésünk szerinti értékeket jeleníthetünk meg. Ez a tevékenység a kiírás:

   írd ki X , X
   1 2

   Ilyenkor a megadott rekeszekben tárolt számok íródnak a képernyőre.
   Nemcsak változók értékét jeleníti meg ez a tevékenység, hanem szövegeket, számokat is. Pl.:

   írd ki 'a pi közelítő értéke '

   írd ki 3.14

   A kiírt értékek a képernyőn egymás után folyamatosan jelennek meg. Ha új sort akarunk kezdeni, akkor az

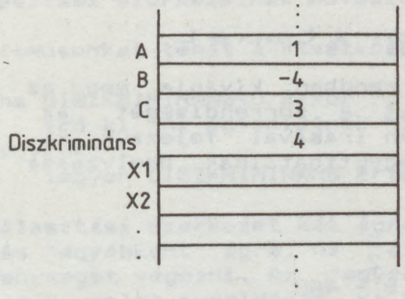
új sor

tevékenységet használjuk.

3) Aritmetikai műveleteket kell végrehajtani, függvényértékeket kell meghatározni, s az így adódó új értékeket is tárolni kell. Az értékadási tevékenységet használjuk erre a célra:

legyen a DISZKRIMINÁNS értéke $B^2 - 4AC$

A kifejezés értékét a gép kiszámítja, az eredményt a tár megfelelő rekeszébe írja (3. ábra).



Az értékadás végrehajtása

3. ábra

Fogalmazzuk meg elemi tevékenységeinket általános alakban is. A {, } kapcsos zárójelek közé írtakat 0-szor vagy többször ismételhetjük. A tevékenységek változó részeit <, > jelek közé írjuk.

Adatbevitel

olvasd <azonosító>{,<azonosító>} értékét

<azonosító> alatt az érintett tárrekesz nevét értjük. Jelölésünk értelme az, hogy az "olvasd" és az "értékét" szavak között legalább egy (vagy vesszővel elválasztva több) azonosítót feltüntetünk.

Értékadás

legyen <azonosító> értéke <érték>

ahol <érték> lehet szám, azonosító (vagyis tárrekesz neve) vagy kifejezés. A kifejezés alatt a szokásos módon írt matematikai formulákat értjük.

Kiírás

írd ki <érték>

vagy

írd ki 'szöveg'

vagy

új sor.

Ezekután megkíséreljük összeállítani a másodfokú egyenlet megoldásának algoritmusát.

1.3 Építsünk algoritmust

Az algoritmus meghatározott sorrendben kívánja meg az elemi tevékenységek végrehajtását. Ezt a sorrendiséget az elemi tevékenységek egyszerű egymás után írásával fejezzük ki. Ilyen kapcsolatban van az egyenlet együtthatóinak beolvasása és a diszkrimináns kiszámítása:

olvasd A, B, C értékét

legyen DISZKRIMINÁNS értéke $B^2 - 4AC$

E két elemi tevékenység egymás utáni végrehajtásával egy összetett tevékenység jön létre. Az ilyen összetett tevékenységet sorozatnak (tevékenység sorozat) nevezzük.

A következő elemi tevékenységnek már tartalmaznia kell a négyzetgyökvonást. Mivel a további számoláshoz a diszkrimináns értékére nem lesz szükségünk, csak a négyzetgyökére, ezért a gyökvonás eredményét ugyanabban a rekeszben tárolhatjuk:

legyen DISZKRIMINÁNS értéke négyzetgyök(DISZKRIMINÁNS)

ahol négyzetgyök(DISZKRIMINÁNS) természetesen DISZKRIMINÁNS

helyett áll.
Vegyük észre, hogy nem tudjuk tevékenység sorozatként folytatni az algoritmust. Ez utóbbi elemi tevékenységet nem írhatjuk az első kettő után, ugyanis nem hajtható végre minden korlátozás nélkül. Ha ugyanis - az adatoktól függően - a diszkrimináns értéke negatív, akkor nem végezhető el a gyökvonás. Ezt az esetet a következő szöveges információ kiírásával szeretnénk jelezni:

"Az egyenletnek nincs valós gyöke."

Ehhez pedig inkább az

```
    írd ki 'Az egyenletnek nincs valós gyöke.'
```

elemi tevékenységet kell végrehajtani.

összefoglalva tehát a következőket tehetjük:

Ha a diszkrimináns nem negatív,
   akkor kiszámítjuk a négyzetgyökét, majd folytatjuk
         az egyenlet megoldását,

   egyébként kiíratjuk a szöveges információt.

A diszkrimináns értékétől függően két tevékenység közül kell kiválasztani a soronkövetkezőt. Az ilyen tevékenység szerkezetet kiválasztási szerkezetnek nevezzük.

Algoritmusunkat tehát a következő módon folytathatjuk:

```
    ha DISZKRIMINÁNS<0 akkor
       írd ki 'Az egyenletnek nincs valós gyöke.'
    egyébként
       legyen DISZKRIMINÁNS értéke négyzetgyök(DISZKRIMINÁNS)
```

A kiválasztási szerkezet két ágra bontja az algoritmust: "akkor" ágra és "egyébként" ágra. Az "akkor" ágban nem kell már más tevékenységet végezni. Az "egyébként" ágban viszont folytatni kell az egyenlet megoldását. Ki kell számítani a gyököket, majd az eredményt meg kell jeleníteni a képernyőn (figyelem! a DISZKRIMINÁNS már a gyökvonás utáni értéket tartalmazza):

```
    legyen X  értéke (-B+DISZKRIMINÁNS)/(2A)
            1

    legyen X  értéke (-B-DISZKRIMINÁNS)/(2A)
            2

    írd ki 'Az egyenlet gyökei:'
    új sor
    írd ki 'x ='
             1
    írd ki x
            1
    új sor
    írd ki 'x ='
             2
    írd ki x
            2
```

Ezen a ponton a két ág egyesül, de egyben véget ér az algoritmus. A két ág egyesülését az egyértelműség miatt jelölni kell, erre a

```
    ha vege
```

szavakat használjuk. Az algoritmus névvel látjuk el, elejét a

"kezdet", végét a "vége" szavakkal jelöljük meg. Lássuk ezután munkánk eredményét:

```
MÁSODFOKÚ EGYENLET

kezdet

  olvasd A, B, C értékét
                                2
  legyen DISZKRIMINÁNS értéke B - 4*A*C

  ha DISZKRIMINÁNS<0 akkor
     írd ki 'Az egyenletnek nincs valós gyöke.'

  egyébként
     legyen DISZKRIMINÁNS értéke négyzetgyök(DISZKRIMINÁNS)
     legyen X  értéke (-B+DISZKRIMINÁNS)/(2*A)
             1
     legyen X  értéke (-B-DISZKRIMINÁNS)/(2*A)
             2
     írd ki 'Az egyenlet gyökei:'
     új sor
     írd ki 'x ='
              1
     írd ki x
              1
     új sor
     írd ki 'x ='
              2
     írd ki x
              2
     ha vége

  vége
```

Ezuttal a programozási nyelvekben szokásos módon a szorzást *-gal jelöltük a puszta egymás mellé írás helyett.

Ez az algoritmus természetesen ebben a formában nem hajtható végre számítógéppel. Ehhez valamilyen programozási nyelven - pl. Turbo Pascalban - kell megírni. Ha végrehajtható volna, a következőket írná a képernyőre:

```
Az egyenlet gyökei:
x = 3
 1
x = 1
 2
```

1.4 Tevékenység szerkezetek

Foglaljuk össze az eddig megismert tevékenység szerkezeteket! A tevékenydség sorozatot egymás után végrehajtott tevékenységek alkotják:

```
tevékenység-1
tevékenység-2
.............
.............
tevékenység-n
```

Egy tevékenység sorozatot egyetlen (összetett) tevékenységnek tekintünk.

A kiválasztási szerkezettel két tevékenység közül határozzuk meg, melyiket kell végrehajtani:

```
ha <feltétel> akkor
   tevékenység-1
egyébként
   tevékenység-2
ha vége
```

Itt tevékenység-1 és tevékenység-2 egyaránt lehet elemi vagy összetett tevékenység. A <feltétel> igaz vagy hamis logikai értéket képvisel. Ha igaz, akkor a tevékenység-1, ha hamis, akkor a tevékenység-2 hajtódik végre.

Nem ritkán fordul elő, hogy a kiválasztási szerkezetben csak az egyik tevékenység szerepel. Pl. egy szám abszolút értékének meghatározására használhatjuk a következő algoritmust:

```
ABSZOLÚT ÉRTÉK
kezdet
   olvasd X értékét
   ha X<0 akkor
      legyen X értéke -X
   ha vége
   írd ki X
vége
```

Megkülönböztetésül egyágó és kétágó kiválasztási szerkezetről beszélünk. Az egyágó kiválasztási szerkezet:

```
ha <feltétel> akkor
   tevékenység
ha vége
```

Ha a feltétel igaz, akkor a tevékenység végrehajtódik, ha hamis, akkor nem.

Arra is gyakran szükség van, hogy egy tevékenységet ismételten többször végrehajtsunk. Pl. egy főiskola hallgatóinak az

ösztöndíját kiszámító algoritmus a következő szerkezetű lehet:

```
amíg van hallgató ismételd
   olvasd a következő hallgató adatait
   számítsd az ösztöndíját
   írd ki az eredményt
ismétlés vége
```

A leírt tevékenység nem algoritmus - legalább is a definíciónk értelmében nem az -, hiszen nem elemi tevékenységekből áll. Ezeket a résztevékenységeket viszont (nem kis munkával) felépíthetnénk elemi tevékenységekből. Az ilyen nem teljesen kidolgozott algoritmusokat a jövőben algoritmus vázlatoknak nevezzük.

A bemutatott tevékenység szerkezetet ismétlési szerkezetnek mondjuk. Általános alakja:

```
amíg <feltétel> ismételd
   tevékenység
ismétlés vége
```

Egy ilyen felépítésű algoritmus ismételten többször is végrehajtja az ismétlendő tevékenységet mindaddig, amíg a feltétel igaz. Akkor hagyja abba, ha a feltétel hamissá válik. Ha netán eleve hamis a feltétel, akkor a tevékenység egyszer sem hajtódik végre.
Az ismétlési szerkezetet ciklusnak is nevezik. A szereplő feltételt kilépési feltételnek, az ismétlendő tevékenységet ciklustörzsnek mondjuk. A ciklustörzs összetett tevékenység is lehet (példánkban is ez a helyzet, sőt, az ott szereplő tevékenységeket nem is bontottuk elemi tevékenységekre). A ciklustörzs végét mutatják az "ismétlés vége" szavak.
A megismert ismétlési szerkezet elől tesztelő ciklus, mert az ismétlési feltétel vizsgálata megelőzi a ciklustörzset.

Az egyes programozási nyelvekben más ismétlési szerkezetek is használatosak. Ilyen a következő hátul tesztelő ciklus:

```
ismételd
   tevékenység
mígnem <feltétel>
```

Az előző algoritmus vázlatot fogalmazzuk meg hátul tesztelő ismétlési szerkezettel:

```
ismételd
   olvasd a következő hallgató adatait
   számítsd ki az ösztöndíját
   írd ki az eredményt
mígnem nincs több hallgató
```

Hasonlítsuk össze a megismert ciklustípusokat!

"amíg" ciklus	"mígnem" ciklus
elől tesztelő	hátul tesztelő
a ciklustörzset végrehajtja, ha a kilépési feltétel	
igaz	hamis
az ismétlés abbamarad, ha a kilépési feltétel	
hamis	igaz
lehet, hogy a ciklustörzs egyszer sem hajtódik végre	a ciklustörzs egyszer mindenképpen végrehajtódik

1.5 Pszeudokód és folyamatábra

Kialakítottunk egy eszközrendszert algoritmusok leírására, amely elemi tevékenységekből és összetett tevékenységek képzésére való tevékenység szerkezetekből áll.

Elemi tevékenységek

Adatbevitel:

olvasd <azonosító>{,<azonosító>} értékét

Értékadás:

legyen <azonosító> értéke <érték>

Kiírás:

```
írd ki <érték>
írd ki 'szöveg'
új sor
```

Tevékenység szerkezetek

Tevékenység sorozat:

```
tevékenység-1
tevékenység-2
.............
.............
tevékenység-n
```

Kiválasztási szerkezet

Egyágó kiválasztás:

```
ha <feltétel> akkor
   tevékenység
ha vége
```

Kétágó kiválasztás:

```
ha <feltétel> akkor
   tevékenység-1
egyébként
   tevékenység-2
ha vége
```

Ismétlési szerkezet

"amíg" ciklus:

```
amíg <feltétel> ismételd
   tevékenység
ciklus vége
```

"mígnem" ciklus:

```
ismételd
   tevékenység
mígnem <feltétel>
```

Ezt az eszközrendszert pszeudokódnak nevezzük. A pszeudokóddal - mint eddig is láttuk - algoritmusokat írhatunk le. Nem kezeljük ezt az eszközt túl mereven két szempontból sem. Egyrészt szükség esetén újabb elemi tevékenységeket vezethetünk be. Pl. ha

grafikus algoritmust akarunk kidolgozni, akkor szükségünk lehet a következő elemi tevékenységekre:

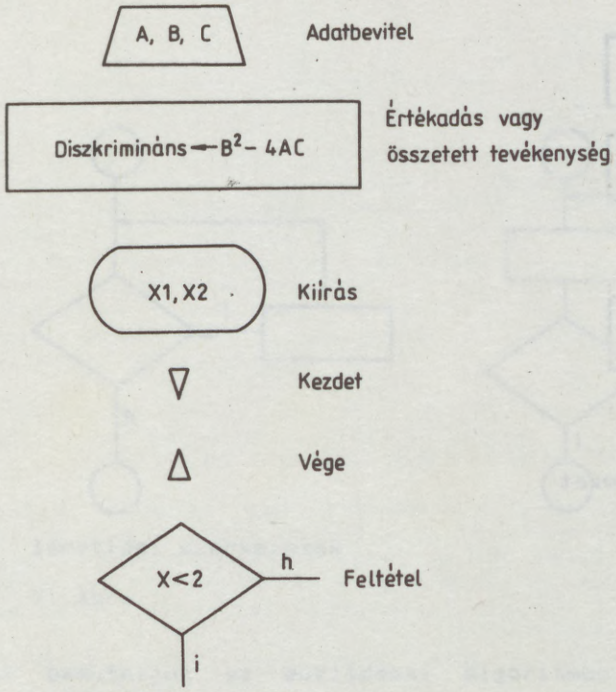
rajzolj (X,Y) koordinátájú pontot

rajzolj (A,B) és (C,D) végpontokkal szakaszt

Másrészt szabadon használhatunk összetett tevékenységeket is (mint az ösztöndíj számfejtésre vonatkozó példánkban), így különböző részletességű algoritmus vázlatokat írhatunk. Ezek a programtervezésben lesznek segítségünkre.

Az algoritmusokat grafikusan is szemléltethetjük.

Az elemi tevékenységeket különböző alakú síkidomokkal ábrázolhatjuk, a szükséges kiegészítő információt a síkidom belsejébe írjuk. Az algoritmus kezdetét és végét, valamint a feltételeket is ábrázoljuk (4. ábra).



Folyamatábra szimbólumok

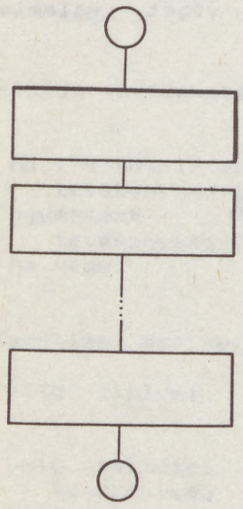
4. ábra

A folyamatábra szimbólumokat folyamatvonallal kötjük össze a

sorrendiség meghatározásához. A vonalakon felülről-lefelé, ill. balról-jobbra haladunk. Az eltérő haladási irányt nyílhegyekkel kell jelölni.
Mindegyik síkidomba egyetlen vonal vezet be és mindegyikből egyetlen vonal vezet ki. Kivételek:

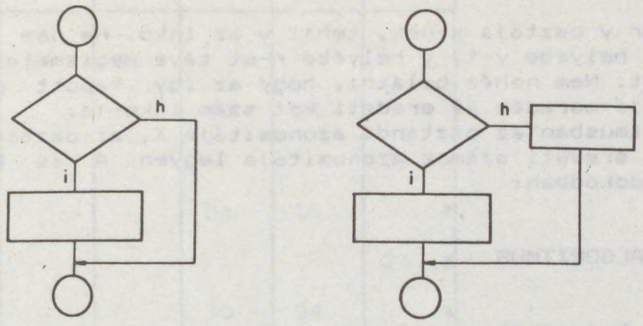
1) a kezdet szimbólumba nem vezet folyamatvonal,
2) a vége szimbólumból nem indul ki folyamatvonal,
3) a feltétel szimbólumból két folyamatvonal indul ki. Az egyikre az "i" (igaz), a másikra a "h" (hamis) jelzést írjuk.

Az egyes tevékenység szerkezetek folyamatábrás ábrázolását az 5., 6. és 7. ábrákon mutatjuk be. Az összetett tevékenységek kezdő és végpontját karika jelöli.



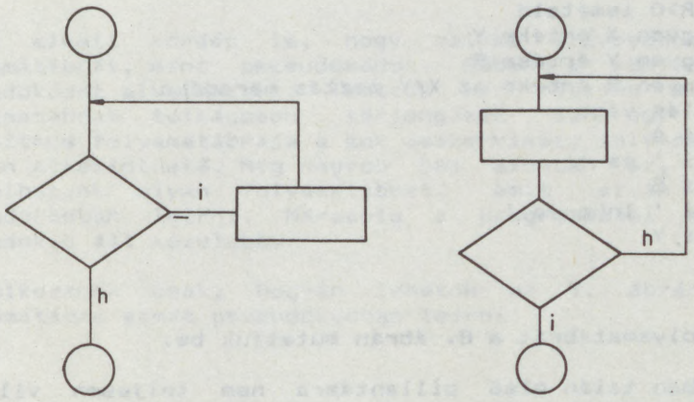
Tevékenység sorozat

5. ábra



Kiválasztási szerkezetek

6. ábra



Ismétlési szerkezetek

7. ábra

Most bemutatjuk az euklidészi algoritmus pszeudokódját és folyamatábráját.

Az algoritmus lényege egy egyszerű mardékos osztás ismétlése. Legyen x és y a két természetes szám, amelyek legnagyobb közös osztóját (lnko) meg akarjuk határozni. Tegyük fel, hogy x a nagyobb, majd osszuk el x-et y-nal, az osztás maradékát jelölje r.

Ezt a következőképp is írhatjuk (k jelöli a hányadost):

x - k*y = r

Ha itt r=0, akkor y osztója x-nek, tehát y az lnko. Ha nem ez a helyzet, akkor x helyébe y-t, y helyébe r-et téve megismételjuk a maradékos osztást. Nem nehéz belátni, hogy az így kapott utolsó zérustól különböző maradék az eredeti két szám lnko-ja.
Legyen az algoritmusban az osztandó azonosítója X, az osztóé Y és a maradéké R! Az eredeti számok azonosítója legyen A és B. Az algoritmus pszeudokódban:

```
EUKLIDESZI ALGORITMUS

kezdet
   olvasd A,B értékét
   ha A>B akkor
      legyen X értéke B
      legyen Y értéke A
   egyébként
      legyen X értéke A
      legyen Y értéke B
   ha vége
   legyen R értéke X
   amíg R>0 ismételd
      legyen X értéke Y
      legyen Y értéke R
      legyen R értéke az X/Y osztás maradéka
   ismétlés vége
   írd ki A
   írd ki ' és '
   írd ki B
   írd ki ' lnko-ja '
   írd ki Y
vége
```

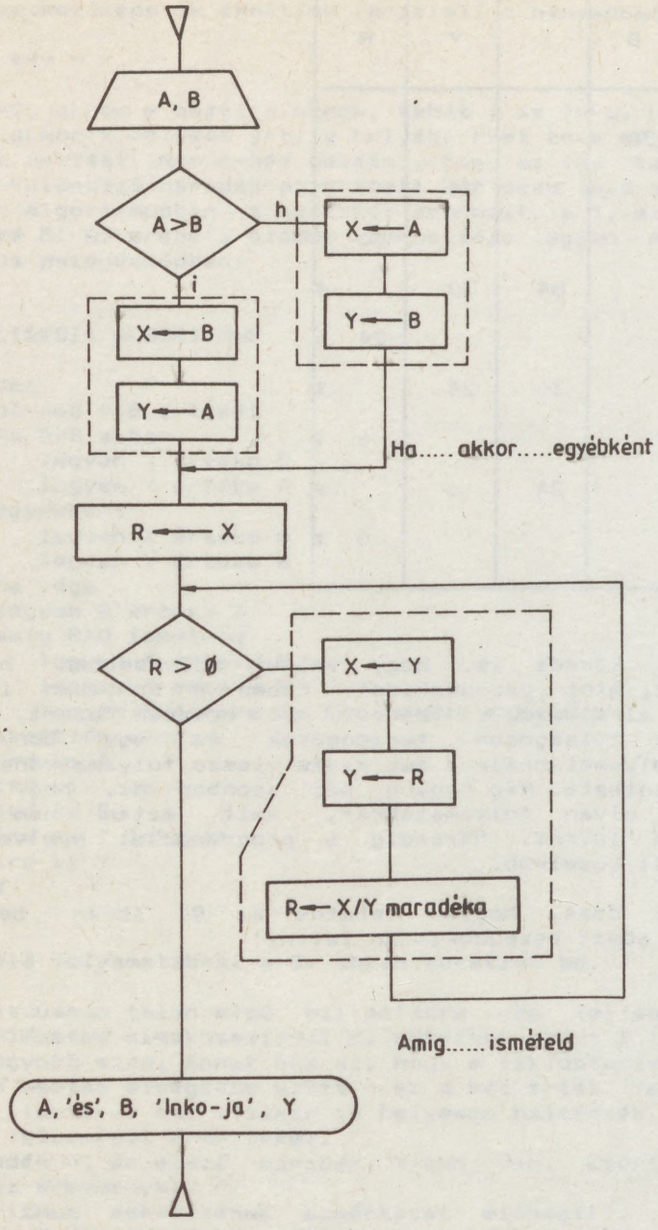
A megfelelő folyamatábrát a 8. ábrán mutatjuk be.

Az algoritmusban talán első pillantásra nem teljesen világos, hogy az előzetes diszkusszióval ellentétben miért X lesz a kisebb és Y a nagyobb szám. Ennek oka az, hogy a ciklustörzsben - még a maradékos osztás elvégzése előtt - ez a két érték felcserélődik. Hogy a ciklustörzs már először is helyesen hajtódjék végre, ezért indítunk felcserélt értékekkel.
Ha a maradék 0, az előző maradék Y-ban van. Ezért Y értékét íratjuk ki eredményül.
Az algoritmus működésének megértését elősegíti, ha papíron ceruzával "végrehajtjuk". Ehhez egy táblázatot rajzolunk, a táblázat oszlopai a tár rekeszeit ábrázolják. Minden oszlopban a legutoljára beírt érték az érvényes.A táblázat *-gal jelölt sorai tartoznak a ciklushoz.

A	B	X	Y	R	
84	30				
		30	84		
				30	
		84	30		*
				24	*
		30	24		*
				6	*
		24	6		*
				0	*

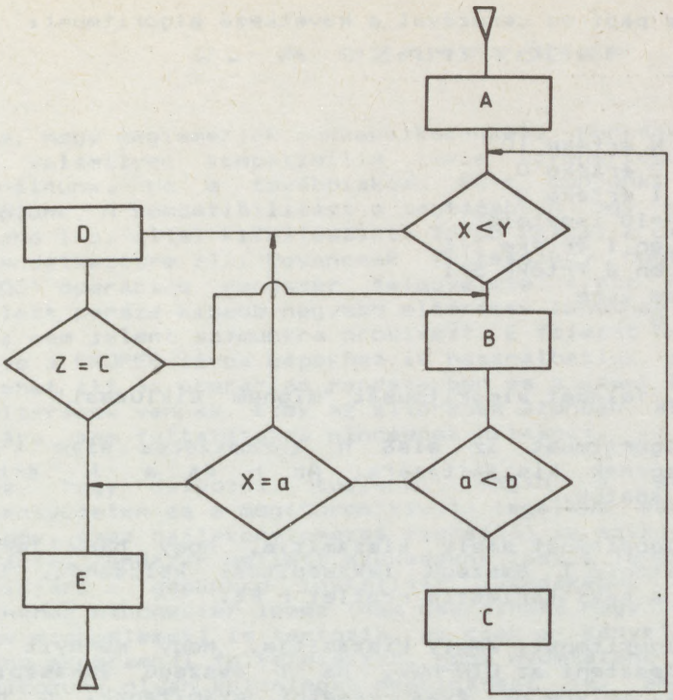
Talán alkati kérdés is, hogy valaki szívesebben használ-e folyamatábrát, mint pszeudokódot. Ebben a könyvben inkább a pszeudokódot alkalmazzuk. Tesszük ezt részben azért, mert a folyamatábrák túlságosan terjengősek és egy bonyolultabb algoritmus folyamatábrája a sok össze-vissza folyamatvonallal nem nagyon áttekinthető. Még nagyob baj azonban az, hogy könnyen rajzolhatunk olyan folyamatábrát, amit aztán nem tudunk pszeudokódban leírni. Márpedig a programozási nyelvekhez a pszeudokód áll közelebb.

Gondolkozzunk csak, hogyan lehetne az 9. ábrán bemutatott folyamatábra sémát pszeudokódban leírni!



Euklidészi algoritmus

8. ábra



Nehezen kódolható folymatábra

9. ábra

1.6 Feladatok

1. Tervezzen algoritmust amely a tár X és Y rekeszében tárolt értékeket felcseréli!

2. Rajzolja fel a másodfokú egyenlet megoldásának folyamatábráját!

3. Írjon pszeudokódot két szám számtani közepének kiszámítására!

4. Készítsen algoritmust egy egyenes körhenger térfogatának és felszínének kiszámítására!

5. Tervezzen algoritmust, amely a tár X és Y rekeszében tárolt számokat úgy rendezi át, hogy X-ben legyen a nagyobb.

6. Készítsen algoritmust, amely három szám közül kiválasztja a minimális értékűt.

7. Egy beolvasott számról döntse el, hogy 50 és 100 közé esik-e. Az eredménytől függően az "igen" vagy a "nem" szavakat írassa ki!

8. Hajtsa végre papíron ceruzával a következő algoritmust:

```
MIT CSINÁL

kezdet
   legyen N értéke 10
   legyen S értéke 0
   legyen I értéke 0
   amíg I<=10 ismételd
      legyen I értéke I+1
      legyen S értéke S+I
   ismétlés vége
   írd ki S
vége
```

9. Írja át a 8. feladat algoritmusát "mígnem" ciklussal!

10. Készítsen algoritmust az első n természetes szám k-adik hatványösszegének kiszámítására. Az n és a k értékek beolvasandó adatok.

11. Készítsen algoritmust amely kiszámítja, hogy hány hónapig kell törlesztenie T összegű lakásépítési kölcsönét, ha a kamat p% és a havi törlesztő részlet r Ft.

12. Készítsen algoritmust, amely kiszámítja, hogy mennyit kell havonta törleszteni az OTP-nek, ha T összegű lakásépítési kölcsönünket, melyre p% éves kamatot számítanak, n hónap alatt akarjuk visszafizetni.

2. A SZÁMÍTÓGÉP

Ideje, hogy megismerjük munkaeszközünket. Munkánkhoz IBM PC-t, vagy valamilyen kompatíbilis (vele egyenértékű) gépet kell használnunk. Ha a továbbiakban PC-t mondunk, ilyen gépre gondolunk. A kompatibilitást a legtágabban úgy értjük, hogy a Borland Inc. által kifejlesztett Turbo Pascal a gépen működőképes és rendelkezésre áll. Ugyancsak feltesszük, hogy DOS (MS-DOS, PC-DOS) operációs rendszer felügyelete alatt működik a gép. Emellett persze kisebb nagyobb eltérések lehetnek a gépek között, de ez nem jelent számunkra problémát. E fejezet néhány állítását kivéve a PROPER 16-os gépekhez is használhatjuk ezt a könyvet, jóllehet itt az operációs rendszerben és a gépek billentyűzetében is eltérések vannak. Ezek az eltérések azonban sem a programok írására, sem futtatásukra nincsenek hatással.

Ahhoz, hogy dolgozni tudjunk, magán a számítógépen, a billentyűzeten és a monitoron kívül legalább egy lemezegységre (floppy, azaz hajlékonylemezes meghajtó) is szükségünk van. A DOS operációs rendszer mellett szükségünk lesz a Turbo Pascal 3.0-s verziójára a gépünkre installált működőképes formában. Ha a gépünkhöz winchester lemez (nem cserélhető nagy tárolókapacitású merev mágneslemez) is tartozik, ez csak a kényelmünket fokozza. Néhány programnál feltételezzük, hogy nyomtatónk is van. Találkozunk olyan különleges megoldással is, amelyhez a Turbo Pascal 4.0-ás vagy 5.0-ás változatára lesz szükség.

2.1 A gép üzembe helyezése

A PC üzembe helyezéséhez tegyük a DOS-t tartalmazó lemezt, a rendszerlemezt a lemezegységbe (jó, ha a Turbo Pascal fordítóprogram is ezen a lemezen van), majd zárjuk le. Ezután kapcsoljuk be a hálózati kapcsolót! Ha a konfigurációnkhoz winchester lemez is tartozik és azon van a rendszer, nincs szükség külön rendszerlemezre.
Bekapcsolás után automatikusan egy tárteszt indul el, majd bejelentkezik a DOS. Kéri a napi dátumot, majd a pontos időt. Saját érdekünkben célszerű helyesen válaszolni ezekre a kérdésekre, bár az [ENTER] billentyű lenyomása is elég.

Ezután a gép a DOS felügyelete alatt üzemképes. Ezt a tényt a képernyőn feltűnő prompt jelzi. A prompt egy bejelentkezési szöveg, amely alapértelmezésben az aktuális lemezegység betűjeléből, az > jelből és az aláhúzásjelből áll, pl.:

```
A>_
```

vagy, ha winchesteren volt a rendszer:

```
C>_
```

Ha két lemezegységünk van, a második betűjele B.
Az aláhúzásjel a helyőr, mutatja, hogy a billentyűzeten lenyomott gombnak megfelelő jel hol tűnik fel a képernyőn. Azt a hasznos jelenséget, hogy a leütött billentyű jele a képernyőn is látszik echo-nak (visszhang) nevezzük. Az echo a bevitel vizuális ellenőrzését teszi lehetővé.

Az operációs rendszer legfontosabb feladata az, hogy a számítógépet a felhasználó számára könnyen kezelhetővé tegye. Olyan eszközöket tartalmaz, amelyek nélkül nehezen boldogulnánk. Ezek az eszközök a DOS parancsok révén használhatók. Nem lehet célunk a DOS mégoly vázlatos ismertetése sem, a legfontosabb parancsok ismerete nélkül azonban egyszerűen nem létezhetünk.

2.2 DOS parancsok

Kétféle DOS parancsot különböztetünk meg, külső és belső parancsokat. A belső parancsok a rendszer betöltése után állandóan a tárban vannak (rezidensek). Ezért bármikor kiadhatók, függetlenül attól, hogy a rendszerlemez az aktuális egységben van-e vagy sem. A külső parancsok nincsenek a tárban, ezeket végrehajtás előtt mindig újra és újra a rendszerlemezről tölti be a DOS. Ezért külső parancs kiadása előtt az aktuális meghajtóban kell lenni a rendszerlemeznek.

Új aktuális meghajtó

Ha gépünkhöz több lemezegység is tartozik, akkor szükség lehet az aktuális meghajtó megváltoztatására. A megfelelő parancs egyszerűen a meghajtó betűjeléből áll, amiután kettőspontot teszünk, majd lenyomjuk az [ENTER] billentyűt. Pl.:

```
A>b:[ENTER]
```

Ezután a B jelű meghajtó lesz az aktuális, amit az új prompt is mutat:

```
B>_
```

Itt jegyezzük meg, hogy

minden parancs beírása után le kell nyomni az [ENTER]-t ahhoz, hogy végrehajtódjon.

Programjainkat, adatainkat lemezeken állományokban (file) tároljuk. Az operációs rendszer tartja számon a lemezen állományainkat. Ezt úgy oldja meg, hogy két táblázatot kezel minden lemezen, az egyik a FAT (File Allocation Table), amely az állományoknak a lemezen való elhelyezkedésével kapcsolatos

adatokat tartalmazza, a másik a tartalomjegyzék (directory). Tudnunk kell, hogy a lemez az állományok tárolásának csak fizikai eszköze. Állományaink logikailag könyvtárakhoz tartoznak. Egy adott lemezen több könyvtár is elhelyezkedhet. Erre a kérdésre még visszatérünk. Most az egyszerűség kedvéért feltesszük, hogy a lemezünk egyetlen könyvtárat tartalmaz, a gyökérkönyvtárat.

A tartalomjegyzék kiíratása

Tegyük fel, hogy az aktuális meghajtó az A, és írjuk be:

```
dir
```

Az [ENTER] lenyomása után valami olyant látunk a képernyőn, mint a 10. ábrán.

```
Volume in drive A is ATS TURBO3
Directory of  A:\

TURBO    COM     39671    1-01-80   12:27a
TURBO    MSG      1536    1-01-80   12:57a
TINST    COM     29954    1-01-80   12:36a
TINST    MSG      4224    1-01-80   12:36a
TOROL    PAS       221    9-14-89    1:50a
ATNEVEZ  PAS       289    9-14-89    3:21a
PROG104  PAS      2452    2-03-89   11:37p
       7 File(s)     279552 bytes free
```

A gyökérkönyvtár tartalomjegyzéke

10. ábra

A dir belső parancs.

Ha a tartalomjegyzék olyan hosszú, hogy kifut a képernyőről, a "p" paraméter megadásával érhetjük el, hogy egyszerre csak egy képernyőoldalnyi kiírást kapjunk:

```
dir /p.
```

Bármely billentyű lenyomásával továbblapozhatunk. Ha nem az aktuális meghajtóban lévő lemezről akarunk tartalomjegyzéket kérni, ki kell írni a lemezegység betűjelét is:

```
dir b:
```

Ha csak egyetlen állományról akarunk információt kérni, beírjuk a teljes nevét:

```
dir a:format.com
```

A "com" az állománynév ón. kiterjesztése, amit ponttal kapcsolunk az állománynévhez.

Ezután nézzük milyen információkat kaptunk a tartalomjegyzékből a dir paranccsal.
Az első sorokban a lemezre, ill. az aktuális könyvtárra vonatkozó adatot találunk. Eszerint az A meghajtóban lévő lemeznek külön kötet azonosítója nincs. A második sorból kiderül, hogy a gyökérkönyvtártartalomjegyzékétlátjuk. A gyökérkönyvtár jele: \.
Ezután az állományok listája következik, a következő adatokkal:

Oszlop	Adat
1	az állomány neve
2	kiterjesztés
3	az állomány mérete
4	dátum
5	idő

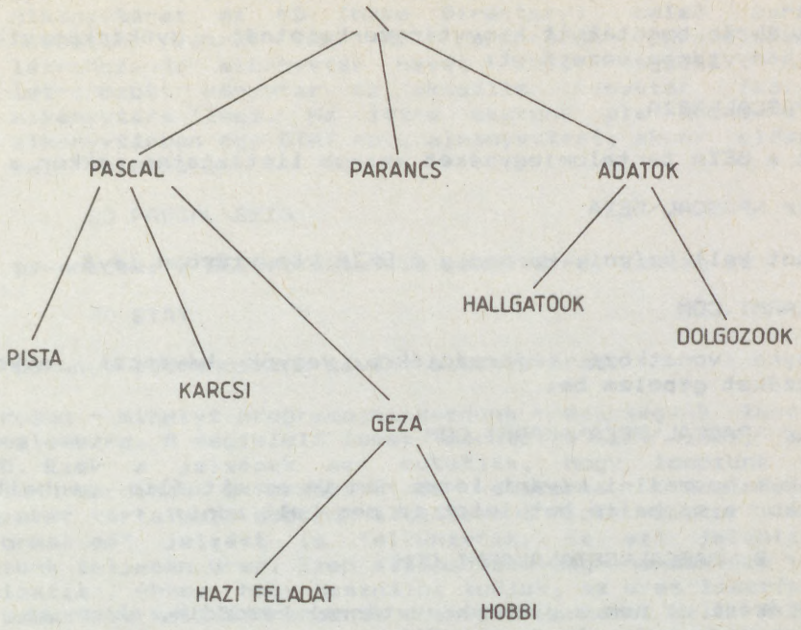
A kiterjesztést jobbára a rendszer rendeli az állományhoz, de magunk is megadhatjuk tetszésünk szerint. Némely kiterjesztés azonban a DOS számára fontos információt nyújt, pl. a COM, az EXE és a BAT kiterjesztésű állományokat parancsoknak tekinti, s végrehajtja, mint bármely külső DOS parancsot.
Az állomány mérete bájtokban (byte) értendő. A bájt az elemi információtárolási egység.
A dátum és az idő az állomány lemezre írásának időpontját mutatja. Ezt a rendszer automatikusan feljegyzi. Ezért célszerű a gép bekapcsolásakor pontos választ adni a dátumra és az időre vonatkozó kérdésekre. Így minden programunkról tudni fogjuk, hogy mikor készült.

A lista végén statisztikát látunk, amely mutatja, hogy összesen hány állományt tartalmaz a könyvtár és hány bájtnyi szabad terület áll még rendelkezésünkre.

A tartalomjegyzék listája két szempontból sem azonos a lemezen tárolt tartalomjegyzékkel. Először is nem szükségképpen tartalmazza a lemezen tárolt összes állományt. Léteznek rejtett állományok is, ezeket a dir parancs nem listázza. De a kiírt állományok esetében sem tűnteti fel az összes adatot, ami a lemezen megtalálható. Ezekkel az információkkal azonban itt nem is kell foglalkoznunk.

Valójában egy lemezen több könyvtár is lehet. Ilyenkor alkönyvtárakról, alkönyvtárak rendszeréről beszélünk.

Az alkönyvtárak hierarchikus elrendezésűek, mint pl. a 11. ábrán látható.



Alkönyvtárak

11. ábra

Itt a gyökérkönyvtár (root) alkönyvtárai a PASCAL, PARANCS, ADATOK nevű könyvtárak. A PISTA, a KARCSI, a GEZA a PASCAL alkönyvtárai; a HAZI FELADAT és a HOBBY a GEZA alkönyvtár alkönyvtárai.

Kérdés, hogy egy adott K könyvtárból mely állományokat érhetjük el és hogyan. Ha a könyvtárakat egy gráf csúcsaival, az "alkönyvtára" relációt a gráf éleivel ábrázoljuk, fagráfot kapunk, amelynek gyökere a gyökérkönyvtár. (Innen ered az elnevezése.) Tekintsük ennek a fának azt a részfáját, amelynek gyökere a K könyvtár. Ha az aktuális könyvtár a K, akkor csak azokban a könyvtárakban tárolt állományokat érhetjük el, amelyek a K gyökerű részfa csúcsait alkotják. A gyökérkönyvtárból tehát minden állomány elérhető.

A PASCAL könyvtárból pl. a PASCAL, a PISTA, a KARCSI, a GEZA, a HAZI FELADAT és a HOBBY könyvtárakban tárolt állományokhoz férhetünk hozzá.

Ha egy állomány nem az aktuális könyvtárban van, akkor nem érhetjük el közvetlenül, csak elérési úton keresztül. Az elérési út voltaképpen a fa két csúcsát összekötő út. Az elérési utat meghatározó formulában a könyvtárneveket a valóságnak megfelelő

sorrendben \ jellel (backslash) elválasztva soroljuk fel. Az állomány nevét az út végén adjuk meg.

A 11. ábrán bemutatott könyvtárszerkezetnél a gyökérkönyvtárból a GEZA könyvtárba vezető út:

\PASCAL\GEZA.

Ha most a GEZA tartalomjegyzékét akarom listáztatni, akkor a

dir \PASCAL\GEZA

parancsot kell beírni. Ha pedig a GEZA könyvtárban lévő

AKARMI.COM

állományra vonatkozó információkra vagyok kíváncsi, akkor a következőket gépelem be:

dir \PASCAL\GEZA\AKARMI.COM

Ha pedig a használni kívánt lemez nem is az aktuális meghajtóban van, akkor a meghajtó betűjelét is meg kell adni:

dir B:\PASCAL\GEZA\AKARMI.COM

Ha az elérési út nem a gyökérkönyvtárral kezdődik, akkor az útat meghatározó formula elején nem áll \.

Az aktuális könyvtár megváltoztatása

Az aktuális könyvtárat a CD (Change Directory) paranccsal változtathatjuk meg. A CD belső parancs. Paramétere egy elérési út, legegyszerűbb esetben egy közvetlen alkönyvtár neve. Ha a gyökérkönyvtárból az PARANCS-ba akarunk átlépni, akkor a

CD PARANCS,

ha GEZA-ba, akkor a

CD PASCAL\GEZA

parancsot kell begépelnünk. Ha valamely alkönyvtárból visszafelé, az őskönyvtárba akarunk lépni, könyvtárnévként .. (két egymásutáni pont)-ot kell megadni. Őskönyvtár alatt azt a könyvtárat értem, amelynek az adott könyvtár az alkönyvtára. Így GEZA-ból a PASCAL-ba a

CD ..

paranccsal léphetünk vissza.
Könyvtárváltás után a prompt is megváltozik, az aktuális elérési útat mutatja.

Alkönyvtár létrehozása

Alkönyvtárat az MD (Make Directory) belső paranccsal hozhatunk létre. Az MD parancsban paraméterként a létrehozandó alkönyvtár nevét kell megadni. Az így létrehozott könyvtár az aktuális könyvtár (közvetlen) alkönyvtára lesz. Ha létre akarunk pl. hozni a GEZA alkönyvtárban egy STAT nevű alkönyvtárat, akkor először ki kell jelölni a

CD PASCAL\GEZA

paranccsal a GEZA-t aktuális könyvtárrá. Ezután az

MD STAT

parancs létrehozza a kívánt alkönyvtárat.

Hamarosan - mihelyt programozni kezdünk - szükségünk lesz saját mágneslemezre. A megfelelő lemez mérete 5 1/4 inch, minősége DS,DD. Ezek a jelzések azt mutatják, hogy lemezünk mindkét oldalán tárolhat információt és kétszeres írássűrűségű. A lemezeket tartalmazó dobozon általában a "soft sector" vagy az "unformatted" jelzést is feltűntetik. Ez azt jelenti, hogy lemezünk teljesen üres. Ezen állapotában még nindenféle munkára alkalmatlan. Ahhoz, hogy használni tudjuk, az üres lemezfelületen egy beosztást kell létrehozni: a lemezt formázni kell. A formázás során alakul ki a lemez felületein az információ tárolás szerkezete - mintha csak egy üres papirost bevonalkáznánk, hogy tudjuk hová írni a sorokat.
A lemez felületén a formázáskor koncentrikus körök (sávok) alakulnak ki, amelyek szektorokra osztódnak. 40 vagy 80 sáv hozható létre lemezoldalanként (80 csak speciális minőségű nagy írássűrűségű, azaz HD jelű lemezen alakítható ki). A szektorok adattárolási kapacitása 512 bájt. A sávok 8 vagy 9 szektorra oszthatók. Számoljuk ki, hogy egy 40 sávos, 9 szektoros mágneslemez mennyi információt képes tárolni!
A 40 sáv 40*9 = 360 szektor. Szektoronként 512 bájt, továbbá mindkét lemezoldalt számításba kell venni, s ez összesen

360*1024 = 368640 bájt.

Mivel 1024 bájt 1 kbájt (kilobájt), mondhatjuk, hogy a lemez 360 kbájt tárolókapacitású. Hogy elképzelhessük, vegyük figyelembe, hogy 1 bájt 1 jelet (betűt, számjegyet, írásjelet) ábrázol. Ha egy nyomtatott sorra 64 betűhelyet számolunk, akkor ez

368640/64 = 5760 sor.

Egy oldalra 54 sort számítva ez

5760/54 = 107 oldal.

A winchester lemezek kapacitása jóval nagyobb, pl. 20 vagy akár

40 Mbájt (megabájt). 1 Mbájt = 1024 kbájt. Egy 20 Mbájtos lemez több mint 6000 könyvoldalnyi információt tárolhat.

A mágneslemez formázása

Lemezeinket az első használatba vétel előtt a FORMAT paranccsal kell formáznunk. A FORMAT külső parancs, ezért gondoskodni kell arról, hogy a rendszerlemez az aktuális meghajtóban legyen.

VESZÉLYFORRÁS!

A formázás nagyon veszélyes művelet. Ha olyan lemezt formázunk, amin már voltak állományok, akkor ezeket elveszítjük. Ha figyelmetlenek vagyunk, előfordulhat, hogy a formázáskor a rendszerlemez marad az aktuális meghajtóban. Megelőzhetjük a lemez tartalmának akaratlan tönkretételét, ha a tasakon található bevágást (írás engedélyező rés) leragasztjuk. Ezzel fizikailag akadályozzuk meg, hogy a lemezegység írni tudjon a lemezre.

Tegyük a formázatlan lemezt a B meghajtóba, a rendszerlemezt pedig az A-ba. (Ha csak egy meghajtónk van, a lemezeket cserélgetni kell.) Írjuk be a

```
FORMAT B: /V
```

parancsot és nyomjunk [ENTER]-t. Ezután az

```
Insert new diskette for drive B:
and strike ENTER when ready
```

üzenetet látjuk. Ha tehát még nem tettük volna a formázandó lemezt a B meghajtóba, tegyük meg, majd nyomjunk [ENTER]-t. Ezután a formázás elkezdődik. Miután véget ért, a

```
Volume label (11 characters, ENTER for none)?
```

promptot látjuk, ami a kötet azonosító címkét (lemeznevet) kéri. Beírhatunk válaszként egy legfeljebb 11 jelből álló nevet (pl.: BER ADATOK), vagy egyszerűen csak az [ENTER]-t nyomjuk le. Ekkor nem lesz neve a lemeznek. Ezután a

```
Format another? (Y/N)
```

Kérdésre kell válszolnunk. Ha még másik lemezt is akarunk formázni, akkor az Y-t, egyébként az N-et nyomjuk le.

A bemutatott FORMAT parancs 40 sávosra és 9 szektorosra formázza a lemezt. A /V paraméter a kötetcímke megadását teszi lehetővé.

Fontossága miatt még egy alakját bemutatjuk a FORMAT parancsnak:

```
FORMAT B: /V /S
```

Ezzel bizonyos különleges DOS állományokat is átmásolunk formázás közben, így a lemezről betölthető a gép bekapcsolását követően az operációs rendszer.

Lemezmásolás

A DISKCOPY külső paranccsal teljes lemezek tartalmát másolhatjuk át. Pl. az A meghajtóban lévő rendszerlemezről a következőképpen készíthetünk másolatot.

Tegyük az üres lemezt a B meghajtóba. Ezt a lemezt előzőleg még formázni sem kell. Írjuk be:

```
DISKCOPY A: B:
```

majd nyomjuk le az [ENTER]-t. A képernyőre kiírt promptokra tetszőleges billentyű lenyomásával kell válaszolnunk:

```
Insert SOURCE diskette in drive A:

Insert TARGET diskette in drive B:

Press any key when ready . . .
```

Jegyezzük meg a szóhasználatot:

source = forrás lemez, amit másolunk,
target = cél lemez, amire másolunk.

2.3 Állományok kezelése

Munkánk nagyobb részében nem teljes lemezekkel vagy könyvtárakkal foglalkozunk, hanem egyes állományokkal. A DOS számos paranccsal segíti az állományok kezelését is. Ezek közül is csak néhányat mutatunk be. Nem tárgyaljuk a parancsok minden lehetséges alakját, csak a legszükségesebb felhasználási lehetőségekre térünk ki.
Az állománykezelő parancsokban paraméterként állományneveket használunk. Az állománynevek megadásánál helyettesítő jeleket is használhatunk. A ? az állománynév bármely jelét, a * bármely összefüggő részét pótolhatja. Így pl.

```
Kov???
```

egyszerre jelöli a

Kovács, Kováts, Kovách, Kovakő

neveket. A

*.PAS

az összes PAS kiterjesztésű állományt,

.

az aktuális könyvtár összes állományát jelöli.

Állományok másolása

Egyes állományokat a COPY belső paranccsal másolhatunk. A VALAMI.PAS nevű állományt a következőképpen másolhatjuk át egy másik lemezre.

Tegyük a másolandó állományt tartalmazó lemezt pl. az A, a másik (formázott!) lemezt a B meghajtóba, majd írjuk be:

COPY A:VALAMI.PAS B:VALAMI.PAS

Ha a

COPY A:VALAMI.PAS B:AKARMI.MAS,

akkor lemásolt állomány neve AKARMI, kiterjesztése MAS lesz a B meghajtóban lévő lemezen.
Egyik alkönyvtárból másikba való másoláshoz az elérési útakat is meg kell adni. Pl. a

COPY A:\SAJAT\TURBO\FELIRAS.PAS B:\BERSZ\FELIRAS.PAS

parancs az A meghajtóban lévő lemezen alkönyvtárban található FELIRAS.PAS állományt ugyanilyen néven másolja át a másik lemez BERSZ alkönyvtárába.

Állományok törlése

Ha valamely állományra (pl. módosítás után) nincs többé szükségünk, a DEL (delete) paranccsal törölhetjük a lemezről. A DEL is belső parancs. A PELDA.BAK állomány törléséhez tegyük a lemezt az aktuális meghajtóba, és írjuk be a

DEL PELDA.BAK

parancsot. Ha a lemez nem az aktuális egységben van az sem baj. Ilyenkor a meghajtó betűjelét is meg kell adni:

DEL B:PELDA.BAK

Ha alkönyvtárban tárolt állományt akarunk törölni, jelöljük

ki az alkönyvtárat aktuális könyvtárrá a CD paranccsal és utána használjuk a DEL-t vagy használjuk az elérési útat:

DEL B:\PASCAL\PELDA.BAK

Helyettesítő jeleket alkalmazva egyszerre több állományt is törölhetünk. A

DEL *.BAK

parancs az összes BAK kiterjesztésű állományt törli az aktuális könyvtárból. Vigyázzunk! Ha a

DEL *.*

parancsot adjuk ki, az aktuális könyvtár összes állományát töröljük.

Egy állomány nevének megváltoztatása

Erre a RENAME belső parancsot használhatjuk. Ha a PELDA.BAK nevét MASPELDA.PAS-ra akarjuk változtatni, tegyük a kérdéses lemezt az aktuális meghajtóba és írjuk be:

RENAME PELDA.BAK MASPELDA.PAS

Ha az átnevezni kívánt állomány alkönyvtárban van, használjuk előzőleg a CD parancsot.

Állományaink között vannak olyanok, amelyek a processzor számára érthető formájúak - ilyenek pl. a rendszerlemez COM kiterjesztésű állományai. Más állományok szövegeket tartalmaznak, amelyeket a billentyűzeten látható jelek alkotják. Amíg egy ilyen szövegállomány a lemezen van, vajmi keveset látunk belőle. Ahhoz, hogy el tudjuk olvasni, a képernyőre kell íratni vagy ki kell nyomtatni.

Szövegállomány kiíratása a képernyőre

A TYPE (belső) paranccsal végezhetjük el ezt a feladatot. Ha a LEIRAS.TXT szövegállományt el akarjuk olvasni, tegyük az aktuális meghajtóba a megfelelő lemezt és írjuk be:

TYPE LEIRAS.TXT

A TYPE parancsban megadhatunk elérési útat is:

TYPE B:\PASCAL\LEIRAS.TXT

Ha a kiírt szöveg nem fér el a képernyőn, akkor az eleje kifut (ez a jelenség a scrolling). Ilyenkor a [CTRL] [S] billentyűkkel leállíthatjuk, majd újra megindíthatjuk a szöveg futását.
Ha a TYPE-ban megadott állomány mégsem szövegállomány, a

képernyőn mindenféle kriksz-krakszot látunk. Ilyenkor a [CTRL] [BREAK] gombokkal megszakíthatjuk a parancs végrehajtását.

Szövegállomány kinyomtatása

Ha van nyomtatónk, a PRINT paranccsal ki is nyomtathatjuk a szöveges állományokat. Vigyázat, a PRINT külső parancs, tehát gondoskodni kell arról, hogy a rendszerlemez az aktuális meghajtóban legyen. Tegyük a másik egységbe a szövegállományt tartalmazó lemezt és - miután meggyőződtünk arról, hogy a nyomtató be van kapcsolva és papír is van benne - írjuk be a

```
PRINT B:LEIRAS.TXT
```

parancsot. A DOS 3.0-s verziójától kezdve elérési útat is megadhatunk a PRINT parancsban.

Megjegyezzük, hogy a képernyő tartalmát közvetlenül is kinyomtathatjuk a [PRTSC] gomb lenyomásával. A gombot a [SHIFT]-tel együtt kell lenyomni.

2.4 A szövegszerkesztő (editor)

Sokmindenről volt már szó az állományokkal kapcsolatban, még most sem tudjuk azonban, hogyan hozhatunk létre egy állományt. A válasz könnyen kimondható: állományt programmal hozhatunk létre. Különböző típusú állományokat létrehozó programok találhatók a DOS-ban, a Turbo Pascal rendszerben, de magunk is írunk majd ilyeneket.
Most elsődlegesen olyan programok érdekelnek bennünket, amelyekkel szövegállományokat hozhatunk létre. Ilyen program használatával írhatjuk meg programjainkat.
Ezek a programok a szövegszerkesztők.
A DOS tartalmaz szövegszerkesztő programot is (EDLIN), azonban elég nehézkesen kezelhető, ezért megtanulását nem javasoljuk. Annál is inkább, mert a Turbo Pascal rendszert igazán kitűnő saját szövegszerkesztővel szerelték fel.
A szövegszerkesztő segítséget nyújt ahhoz, hogy a szöveget a képernyőn kialakítsuk, s biztosítja mindenfajta változtatás, javítás, sőt "ollózás" lehetőségét is. Lehetővé teszi a kész szöveg könyvtárba írását (így jön létre a szöveges lemezállomány). Módosíthatunk a segítségével már létező állományokat, egyesíthetünk több állományt egyetlen állománnyá.

Az editorhoz csak úgy férhetünk hozzá, ha először betöltjük a Turbo Pascal programot. Ennek a programnak a neve TURBO.COM. Tegyük az aktuális meghajtóba a megfelelő lemezt és írjuk be:

```
TURBO
```

A képernyőn a következőket fogjuk látni (12. ábra):

```
Turbo Pascal system          Version 3.01A
                                   PC-DOS

Copyright (C) 1983,84,85   BORLAND Inc.

Monochrome display


Include error messages (Y/N)?
```

A Turbo Pascal bejelentkezése

12. ábra

Jelenleg közömbös, hogy Y vagy N választ adunk-e, majd ha programot készítünk Y-nal fogunk válaszolni, hogy a programhibákkal kapcsolatos üzeneteket szövegesen írja ki a rendszer. Ha Y a válasz, akkor a lemezről beolvasódik a TURBO.MSG állomány is, amely a hibaüzenetek szövegét tartalmazza.

Ezután a Turbo Pascal főmenüje jelenik meg a képernyőn (13. ábra).

```
Logged drive: A
Active directory: \

Work file:
Main file:

Edit      Compile   Run   Save
Dir       Quit   compiler Options

Text:      0 bytes
Free: 63485 bytes
```

Főmenü

13. ábra

A főmenű használatára később még visszatérünk, most csak a jelenleg szükséges menüpontokat ismertetjük.

1) Logged drive

[L] leütése után új aktív meghajtót jelölhetünk ki. A

New drive:}

prompt után adjuk meg a használni kívánt meghajtó betűjelét! A rendszer nem jelzi vissza a változtatást, de nyomjuk pl. le a szóköz billentyűt, ha az aktualizált menüt akarjuk látni.

2) Active directory

Ha az [A] billentyűt leütjük, a

New directory:}

promptot kapjuk. Ezután beírhatunk egy elérési útat. Az elkészített állományt majd abba az alkönyvtárba tudjuk felírni, ahová az elérési út vezet.

3) Work file

Ezt a menűpontot kötelező használni. Ha a [W]-t lenyomjuk, a

Work file name:}

promptra válaszként az állomány nevét kell beírni. Megadhatjuk a kiterjesztést is. Ha nem tesszük, akkor a kiterjesztés automatikusan PAS lesz. Ha a megadott munkaállomány név PROBA.VLM, akkor tehát ilyen néven írhatjuk majd lemezre az állományt, ha a válaszunk csak PROBA, a teljes név PROBA.PAS lesz.
Irjuk be a PROBA állománynevet. Kiíródik a

Loading A:PROBA.PAS

üzenet, és ha az aktív könyvtárban létezik PROBA.PAS nevű állomány, akkor betöltődik. Ha még nem létezik, akkor a

New File

kiírást látjuk.

Ha már korábban létrehoztunk egy állományt (mondjuk MASIK néven) és nem írtuk lemezre, akkor a [W] lenyomása után a

Workfile A:MASIK.PAS not saved. Save (Y/N)?}

prompt kiírásával lehetővé válik, hogy felírhassuk (Y

válasz). Ha nincs szükségünk a MASIK állományra, akkor természetesen N-et nyomunk.

4) Edit

[E] leütésével lépünk a szerkesztési módba. Az üres képernyőn tetszőleges szöveget állíthatunk össze az editor parancsok használatával. Ha a szerkesztést befejeztük, a [CTRL]-t lenyomva tartva először a [K]-t, majd a [D]-t nyomjuk le. Ezután a szóközbillentyűt lenyomva visszakapjuk a főmenüt.

5) Save

Ha az [S]-et leütjük, a szerkesztett állomány az aktív meghajtóban lévő lemezre íródik a kijelölt aktív könyvtárba, a work file menüpontban megadott néven. Menetközben nincs mód e kijelöléseket megváltoztatni.

A [D] billentyű leütésével az aktuális könyvtár tartalomjegyzékét írathatjuk ki. Ez is Turbo parancs. Ahhoz, hogy a DOS dir parancsát használhassuk, előbb ki kéne lépni a Turbo Pascalból. Ez ugyan nem nagy ügy, de gondoskodni kell a munkaállomány mentéséről, s a Turbo újbóli indítása is időigényes. Ettől mentesít bennünket ez a menüpont.

6) Quit

Ha befejeztük a munkát a Turbo Pascallal és vissza akarunk térni a DOS-hoz, a [Q]-t kell lenyomni.

2.5 Munka a szerkesztővel

Ha a főmenüből az Edit menüpontot választjuk, belépünk a Turbo Editorba, a Turbo Pascal rendszer szövegszerkesztőjébe. Ha új állományt hozunk létre, akkor egy csaknem teljesen üres képernyőt látunk, csak a legfelső sorban látunk információt. Ez az állapotjelző (státusz) sor (14. ábra).

```
Line 0    Col 0    Insert    Indent    A:PROBA.PAS
```

A szerkesztő állapotjelző sora

14. ábra

Az egyes információk jelentése:

Line xx — A helyőr az állomány elejétől számított xx-edik sorban van.

Col xx — A helyőr a képernyő bal szélétől számított xx-edik pozición van.

Insert — Jelzi, hogy a leütött jel a helyőr pozícióján beszúródik a szövegbe. Ez azt jelenti, hogy a helyőrtől jobbra lévő szövegrész jobbra elmozdul. Ezt az üzemmódot kikapcsolhatjuk, ekkor az Overwrite szó kerül az állapotjelző sorba. Ennél a helyőr pozícióján lévő jel átíródik ha lenyomunk egy másik billentyűt.

Indent — Automatikus tabulálás van érvényben. Ez azt jelenti, hogy a begépelt sor végén az [ENTER]-t lenyomva nem a képernyő szélére ugrik vissza a helyőr, hanem arra a pozícióra, ahol az előző sor kezdődött.

A állapotjelző sor utolsó adata a szerkesztet állomány nevét mutatja.

Ezután a legfontosabb szerkesztési funkciókat mutatjuk be. A teljes listát az A) mellékletben közöljük.

A Turbo Pascal rendszer installációs programja lehetőséget nyújt az editor parancsok megváltoztatására. Mi az alapértelmezés szerinti állapotot közöljük. Ha ön úgy tapasztalja, hogy némely parancs nem úgy működik, ahogy e könyvben olvasható, próbálja meg kideríteni a parancsok tényleges működését.

A billentyűzeten úgy írunk, mint egy írógépen. Egy sorba elég sok (több mint 100) jelet írhatunk, bár a képernyőn legfeljebb 80 látszik egyszerre. Ha hosszabb sort írunk, a képernyő tartalma balra gördül. A hosszú sorok írásának lehetőségével ne éljünk! A szövegben szabadon mozoghatunk a helyőrmozgató billentyűkkel fel-le, jobbra-balra. A helyőrpozíción módosításokat eszközölhetünk: beszórhatunk és törölhetünk szövegrészeket.

Beszórás

Jelek beszórása a helyőr pozícióján Insert módban automatikusan megtörténik.

Ugyanígy teljes sorok is beszórhatók. A beszórandó sor előtti sor végén nyomjuk le az ENTER-t, és az új sort folyamatosan gépeljük. A beírt sor végén nem kell ENTER-t nyomni, ugyanis már ott van. Eléje szúrtuk be a sor összes jelét.

Törlés

A helyőrtől balra lévő jelet a DELETE felírató gombbal törölhetjük.

A helyértől jobbra lévő szót a [CTRL] [T] billentyűkkel törölhetjük.

Teljes sort a [CTRL] [Y] kombinációval törlünk.

Az edig ismertetett funkciók már elegendőek arra, hogy mindenféle javítást elvégezhessünk. A blokk parancsok további jól használható szerkesztési funkciókat szolgáltatnak.

Blokk alatt a szöveg egy összefüggő részét értjük. A blokk parancsok végrehajtása előtt ki kell jelölni az érintett blokkot. A kijelölt blokkot másolhatjuk, mozgathatjuk, törölhetjük, lemezre írhatjuk és lemezről beolvashatjuk.

Blokk kezdet kijelölése

A blokk kezdetének kijelöléséhez vigyük a helyőrt a blokk első jelére, majd először a [CTRL] [K], majd a [CTRL] [B] billentyűket nyomjuk le.

Blokk végének kijelölése

A blokkvéget a [CTRL] [K] és [CTRL] [K] kombinációk lenyomásával jelölhetjük meg, miután a helyőrt az utolsó jelre vittük.

Blokk másolása

Vigyük a helyőrt a szövegnek arra a pontjára, ahová az előzőleg kijelölt blokkot másolni akarjuk, majd nyomjuk le a [CTRL] [K], majd a [CTRL] [C] gombokat. A blokk átmásolódik és korábbi helyén is változatlan marad.

Blokk mozgatása

Vigyük a helyőrt a szövegnek arra a pontjára, ahova az előzőleg kijelölt blokkot mozgatni akarjuk. Ezután nyomjuk le a [CTRL] [K], majd a [CTRL] [V] billentyűket. A blokk a megadott helyen a szövegbe másolódik, eredeti helyéről pedig eltűnik.

Blokk törlése

A kijelölt blokkot a [CTRL] [K], [CTRL] [Y] gombokkal törölhetjük.

Blokk felírása lemezre

Az előzőleg kijelölt blokkot lemezre írhatjuk a [CTRL] [K] és a [CTRL] [W] gombokkal. A képernyőre írt prompt kéri az állománynevet. Ha nem adunk meg kiterjesztést, akkor a blokk PAS kiterjesztéssel kerül a lemezre.

Blokk olvasása lemezről

Valamely lemezen tárolt szövegállományt illeszthetünk teljes egészében a szerkesztett szövegünkbe. Vigyük a helyőrt arra a pontra, ahová az állomány tartalmát be akarjuk szúrni, majd nyomjuk le a [CTRL] [K] és a [CTRL] [R] billentyűket. Ezután prompt jelenik meg a képernyő tetején az állománynévre vonatkozólag. A név megadása után a lemezről leolvasott állomány a jelölt helyen megjelenik a képernyőn.

Sajnos a rendelkezésünkre álló terjedelem nem teszi lehetővé, hogy további szerkesztési parancsokat is tárgyaljunk. Kérjük az Olvasót, lapozza fel a C) mellékletet és - kisérletezzen. A szerkesztő alapos ismerete, sőt, készség szintű elsajátítása egy leendő profi programozó számára elengedhetetlen.

A Turbo szövegszerkesztő alapja egyébként a WordStar szövegfeldolgozó program, amelynek használatában szintén érdemes elmélyedni.

2.6 Feladatok

1. Helyezze üzembe a számítógépet, majd kérje be a rendszerlemez tartalomjegyzékét!

2. Hogyan tudná a tartalomjegyzékből csak a COM kiterjesztésű állományok listáját bekérni?

3. Másoljon magának egy saját rendszerlemezt!

4. Formázzon egy lemezt!

5. Nyisson a lemezén egy PASCAL nevű alkönyvtárat.

6. Másolja át a lemezére a Turbo Pascal rendszerlemezről a TINST.COM állományt!

7. Másolja át a PASCAL alkönyvtárba a Turbo Pascal rendszerlemez állományait!

8. Kérje le a PASCAL alkönyvtár tartalomjegyzékét!

9. Listáztassa a képernyőre a TURBO.MSG szövegállományt a Turbo Pascal rendszerlemezről!

10. Változtassa meg lemeze gyökérkönyvtárában szereplő TINST.COM állomány nevét AKARMI.MAS-ra!

11. Törölje a lemezén lévő AKARMI.MAS nevű állományt!

12. Ha van nyomtatója, nyomtassa ki a Turbo Pascal rendszerlemezről a TINST.MSG állományt!

13. Hogyan léphet vissza a PASCAL alkönyvtárból a gyökérkönyvtárba?

14. Biztosítsa, hogy az A legyen az aktív meghajtó. Töltse be a Turbo Pascalt és saját formázott lemezét tegye a B meghajtóba! Jelölje ki a Turbo főmenüvel a B meghajtót!

15. Legyen a munkaállomány neve WEORES, majd írja le a Turbo Editor segítségével a következő verset betű szerint!

Pityu és Pöszi
az óvodakertben mindenfélét sinálnak
ni mijen dicnók
a többi óvodások körülöttük álnak
nézi a Paidagógosz néni
pfuj meekkora dizsnók
űrlapot és hegyes tollat ragad
dühtől hullámozva ír:
Tüzdelt Zülők!
Máskor scináljanak jobb jerekeket.
És felelnek a zülők:
Kedves Paidagágász Néni!
Hun házasodunk hun meg elválunk
különb féle jerekekkel kísérletezünk.

Írjuk fel a kész szöveget a lemezünkre! Természetesen az ékezetes betüket a megfelelő ékezet nélkülivel pótolhatjuk.

16. Legyen a munkaállomány neve KRITIKA, és az Editorral írjuk bele a következő szöveget:

Weöres Sándor a legeredetibb kortárs költőnk, aki a szerves és szervetlen lét alapkérdéseinek filozófiai mélységei mellett a valóság apró rezdüléseit is érzékeli és tökéletesen képes érzékeltetni is. Ezt mutatja a következő vers. Csodálatos, ahogy a már idős költő szeme rányílik a valóságnak erre határvidékére. Csak azt mondhatjuk, hogy tökéletes.

17. A következő munkaállomány név legyen újra WEORES. Ekkor a korábban felírt vers beolvasódik és újra szerkeszthetjük. Rontsuk el a verset, azaz javítsuk ki benne a helyesírási hibákat, majd írjuk fel az új változatot is lemezre.

18. Nézzük meg a tartalomjegyzéket! WEORES.PAS és WEORES.BAK nevű állományt is látunk. A módosítás előtti állapot automatikusan megőrződik a BAK kiterjesztésű állományban.

19. Legyen ismét KRITIKA a szerkesztési munkaállomány. Építsük be mint blokkot a WEORES.BAK állományt a szöveg két bekezdése közé, majd írjuk fel lemezre.

20. Lépjünk ki a Turbo rendszerből és írassuk DOS paranccsal egymás után a képernyőre a készített állományokat!

21. Ha van nyomtatónk, nyomtassuk ki a KRITIKA.PAS állományt!

22. Töröljük a lemezről a KRITIKA.BAK és a WEORES.PAS állományt, majd a WEORES.BAK nevet változtassuk WEORES.PAS-ra!

23. Próbálja ki a felírt szövegeken a többi szerkesztőfunkciót is (blokk másolás, blokk mozgatás, stb.). Kísérletezzen a C) mellékletben található többi paranccsal is!

3. PROGRAMNYELV ÉS PROGRAM

Algoritmusainkat a számítógép számára programozási nyelven írjuk le. Jelenleg számos programozási nyelv ismeretes, de ezek közül csak néhányat használnak széles körben. Vannak géphez közeli nyelvek, amelyek utasításkészlete az adott számítógép processzorától függ. Ilyen nyelveken a gépek minden lehetőségét kihasználhatjuk és nagyon hatékony programokat készíthetünk, de manapság senki sem vetemedne arra, hogy egy műszaki számítási feladatot, vagy egy vállalati bérszámfejtést ilyen nyelven programozzon. Mindamellett a "profi" programozónak ismernie kell számítógépe assembly nyelvét. Az assembly nyelv - talán tól egyszerűen fogalmazva - voltaképpen emberi felhasználásra alkalmasabbá tett gépi kód.

A magasszintű programozási nyelvek jobban igazodnak a számítógépekkel elvégzendő tevékenységek természetéhez. A magasszintű nyelvek némelyike bizonyos problématípusokhoz igazodik. Ilyen pl. a COBOL, amely elsősorban nagy adat-, de kis számításigényű - ügyviteli - feladatok megoldására alkalmas; vagy a FORTRAN, amit műszaki - matematikai számítások elvégzésére terveztek. A PL/I, az Ada és a Pascal általános célú nyelvek. A C nyelv rendszerfejlesztésre is alkalmas.

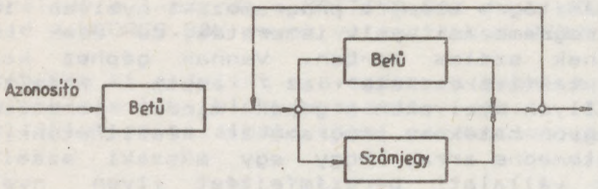
Beszélhetünk ezenkívül nagyon magas szintű nyelvekről és célnyelvekről.
A ProLog programozási nyelv a nagyon magas szintű nyelvek körébe tartozik. Bonyolult - a mesterséges intelligencia területére eső - problémák megoldására alkalmas.
A célnyelvek valamilyen szakterület konkrét igényeit elégítik ki. Ide sorolhatjuk pl. az adatbáziskezelő nyelveket is.

A programnyelvek egyaránt alkalmasak magának a tevékenységnek és azoknak az objektumoknak a leírására, amelyekkel a tevékenységek végrehajtódnak. Mivel a programokat gépek hajtják végre, nagyon szigorúan be kell tartani a formai követelményeket és nagyon pontosan kell ismerni az egyes nyelvi elemek jelentését.

3.1 Szintaxis és szemantika

A programírás formális szabályainak összességét a programnyelv szintaxisának nevezzük. Minden nyelvi forma meghatározott szerkezettel rendelkezik, ezt a szerkezetet adjuk meg a szintaktikai szabályokkal. Egy szintaktikusan helyes nyelvi szerkezethez jelentés tartozik. E jelentés az adott nyelvi alakzat szemantikája.
A nyelvi szerkezetek szintaxisát szintaxisábrákkal fogjuk megadni. A szintaxisábra felépítését és értelmezését egy példával mutatjuk be.

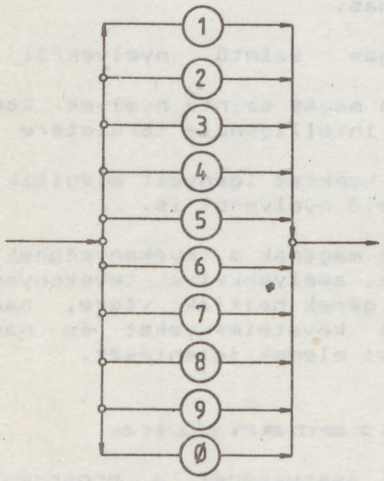
A nyelvi objektumokat azonosítóval jelölhetjük a programokban. Az azonositók betűvel kezdődő, betűk és számjegyek vegyes sorozatával folytatható jelsorozatok. Ennél a mondatnál áttekinthetőbb (és talán pontosabb) a 15. ábra.



Az azonosító szintaxisa

15. ábra

A szintaxisábrába a bal oldali irányított vonalon lépünk be, és a nyilak irányában haladva tetszőleges "sétát" tehetünk, végül a jobb oldali nyílon kijövünk. Az ábra minden bejárása egy helyesen képzett azonosítót ad.
A téglalap alakú dobozok olyan nyelvi elemeket jelölnek, amelyeket más szintaxisábra definiál. A "számjegy" szintaxisábráját a 16. ábrán adjuk meg.

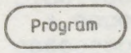


A számjegy szintaxisa

16. ábra

Nézze el nekünk az Olvasó, de a "betű" szintaxisát nem adjuk meg szintaxisábrával. A használható betűk az angol ábécé nagy és kisbetűi, valamint az aláhúzásjel. A program szövegében nem tesz különbséget kis- és nagybetű között a fordítóprogram.

Akor alakú dobozokba a nyelv alapjeleit írjuk. Ha az alapjel terjedelmesebb, akkor a 17. ábrán látható lekerekített síkidomot használjuk. Ezekbe a nyelv fenntartott szavait (kulcsszavait) írjuk.



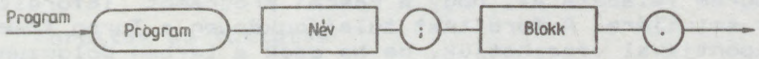
Fenntartott szavak ábrázolása

17. ábra

A nyelvi szerkezetek szemantikáját szavakban fogjuk megadni.

3.2 Az első program.

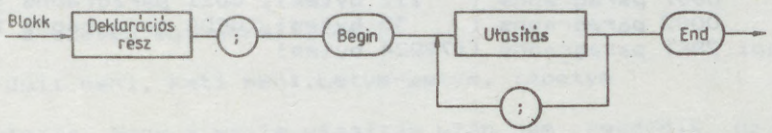
Minden Turbo Pascal program a "program" szóval kezdődik, ezután a program neve (azonosítója) áll. Ez a program első sora, a programfej. Ezután a programblokk következik, majd a program végét pont zárja le (18. ábra).



A program felépítése

18. ábra

A programblokk két része közül az első a programban használatos objektumokat írja le, ezt a részt deklarációs résznek nevezzük. Negyon egyszerű programokban - mint amilyen első programunk is lesz - a deklarációs rész elmaradhat.
A programblokk másik része az algoritmus leíró rész (19. ábra). A programot alkotó utasítások a "begin" és az "end" kulcsszavak között helyezkednek el.
Az utasítások a tevékenységleírás programnyelvi eszközei. Először az "írd ki" elemi tevékenységet megvalósító "writeln" utasítással találkozunk (1. program). A program részeit és az utasításokat pontosvesszővel kell elválasztani.



A programblokk

19. ábra

```
program elso;
begin
  writeln('Juli neni, Kati neni,');
  writeln('Letye-petye, lepetye');
  end.
```

Egyszerű program

1. program

A program azonosítója "elso", deklarációs részt nem tartalmaz. A writeln utasítások után zárójelben kell megadni a kiírandó értékeket, most ún. stringeket (fűzéreket) íratunk ki. A fűzérel voltaképpen szövegek.

Írjuk meg a programot az Editorral. Mint már korábban tapasztaltuk, ezzel egy szövegállományt kapunk. Ez az állomány - ami most a tárban van - egy Turbo Pascal forrásprogram. A forrásprogramot a számítógép nem tudja közvetlenül végrehajtani, hiszen nem gépi kódú utasításokból áll. A Pascal utasításoknak megfelelő tevékenységek pedig nem elemiek a processzor számára. A fordítóprogram feladata az, hogy a Pascal programot lefordítsa a processzor szintjére. A fordítást tulajdonképpen a Turbo főmenü Compile menüpontjával végezhetjük, de ha csak a tárban dolgozunk, akkor a Run menüpont is használható. Ekkor a fordítás befejeztével azonnal elkezdődik a program végrehajtása is.

Nyomjuk tehát le az R betűt! A 20. ábra mutatja a képernyőre íródó információt.

A Compiling szó után a fordítóprogram által adott információkat látjuk: a forrásprogram sorainak számát, majd a lefordított program (a tárgyprogram) tárbeli helyfoglalására vonatkozó adatokat.
A Running szó alatt a program működésének az eredményét találjuk.

```
Compiling
  5 lines


Code:        0007 paragraphs (   112 bytes), 0D21 paragraphs free
Data:        0002 paragraphs (    32 bytes), 0FDA paragraphs free
Stack/Heap: 2BE7 paragraphs (179824 bytes)


Running
Juli neni, Kati neni,
Letye-petye, lepetye
```

```
  1.0000000000E-02
00100011110101110000101000111101011100000 01111010 11111010
  1.0000000000E-01
01001100110011001100110011001100110011001100 01111101 11111101
  1.0000000000E+00
00000000000000000000000000000000000000000000 10000001 00000001
  1.0000000000E+01
00100000000000000000000000000000000000000000 10000100 00000100
  1.0000000000E+02
01001000000000000000000000000000000000000000 10000111 00000111
  2.0000000000E+00
00000000000000000000000000000000000000000000 10000010 00000010
  3.0000000000E+00
01000000000000000000000000000000000000000000 10000010 00000010
 -5.0000000000E+00
10100000000000000000000000000000000000000000 10000011 00000011
 -5.0000000000E-01
10000000000000000000000000000000000000000000 10000000 00000000
 -5.0000000000E+01
11001000000000000000000000000000000000000000 10000110 00000110
```

Fordítás és futtatás

20. ábra

Tessék megfigyelni, hogy a writeln utasítás a szöveg kiírásán kívül egy „új sor" tevékenységet is végez. Ha a program első kiírását átalakítjuk:

```
program elsol;
begin
  clrscr;
  writeln('Juli neni, Kati neni,');
  writeln('Letye-petye, lepetye');
  end.
```

A módosított program

2. program

akkor a szöveg egy sorba íródik:

Juli neni, Kati neni,Letye-petye, lepetye

Ez mutatja, hogy a write utasítás után nem kezdődik új sor. A clrscr utasítás a képernyőt törli. Ennek eredményeként most csak a program által kiírt eredményt látjuk, a fordítási információkat nem.
A tárbeli helyfoglalással kapcsolatban el kell mondanunk, hogy minden program három elkülönített területtel gazdálkodik. A kódszegmensben (code) a lefordított program utasításai vannak, az adatszegmensben (data) az adatok. Mindkét szegmens 64 kbájt, sem több, sem kevesebb. A tár fennmaradó része az ún. dinamikus tároló. Erről a későbbiekben majd még lesz szó.

A zárójeleken kívüli számok, mint pl. 2BE7 hexadecimális (16-os számrendszerbeli) számok.
Ha a programot hibásan gépeljük be, a fordítás közben hibajelzést kapunk. A fordítóprogram a szintaktikus hibákat jelzi. A Turbo Pascal rendszer betöltésekor az

Include error messages (Y/N)?

kérdésre adott Y válasz esetén a hiba oka szövegesen is kiíródik. A hibaüzeneteket a TURBO.MSG állomány tartalmazza - természetesen angolul. Nics azonban akadálya annak, hogy lefordítsuk. Ezt a legkönnyebben úgy tehetjük meg, hogy a munkaállományként kijelölt TURBO.MSG-t az Editorral átszerkesztjük (az angol kifejezéseket átírjuk magyarra). Az állományban a szokásostól eltérő jelek is előfordulnak, ezeket ne bántsuk! Mindenesetre másolt lemezen végezzük el a módosítást, az eredetit ne rontsuk el!

Ha hiba esetén az [ESC] (escape) gombot lenyomjuk, a programszöveg íródik a képernyőre, s a helyőrt a hiba helyén látjuk. Pontosabban annál a nyelvi elemnél, amit a hiba miatt a fordítóprogram már nem tudott felismerni. Ha a 3. programban elhagyjuk a két writeln utasítás közötti pontosvesszőt, az

Error 1: ';' expected. Press <ESC>

üzenetet látjuk. Az escape billentyű lenyomása után a második writeln "w"-je alatt villogó helyőr mutatja a hiba helyét.

A pontosvesszők miatt egyébként gyakran fordul elő szintaktikai hiba. Pedig a szabály egyszerű: két utasítás közé mindig pontosvesszőt kell tenni, utasítások belsejében nem fordulhat elő pontosvessző (kivéve a fűzéreket). Azonban a begin és az end pl. nem utasítások. A begin után és az end elé ezért nem kell kitenni a pontosvesszőt. De ha ott van sincs baj, ugyanis a fordítóprogram ezt úgy fogja fel, hogy a begin és a pontosvessző, valamint a pontosvessző és az end között egy "üres utasítás" áll. Ugyanakkor más alapszavak után hibát okozhat a pontosvessző.

Mi általában azt a gyakorlatot fogjuk követni, hogy esetenként felesleges pontosvesszőket is használunk. Ha ugyanis a program módosításánál az end elé egy új utasítást beszórunk, akkor fennáll a veszélye annak, hogy az előző sor végéről lefelejtjük a pontosvesszőt.

Programjainkban kommentárokat is használunk. Ezek a {, } zárójelek közé írt szövegek nem a számítógépnek szólnak, hanem a program olvasójának. A program megértését segítik.

3.3 Konstansok, változók, típusok

A következő programunk egy kör területét számítja ki (3. program).

```
program korterulet;

begin
  clrscr;
  write('A 25 cm sugaru kor terulete:');
  writeln(25*25*3.14);
  end.
```

Kör területének kiszámítása

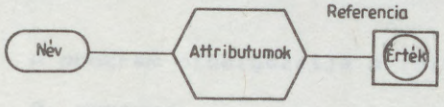
3. program

Ebben a programban a 25, a 3.14 és az 'A 25 cm sugaru kor terulete:' konstans. Ezek a konstansok mind különböző típusúak: a 25 egész, a 3.14 valós, az 'A 25 cm sugaru kor terulete:' pedig füzérkonstans. A programnak a legnagyobb hibája az, hogy csak a 25 cm sugarú kör területét számítja ki. Más esetben módosítani kell az utasításokat.
A megoldást a változók és az adatbevitel használata jelenti.

A változókat a nevükkel jelöljük, a név voltaképpen azonosító. Minden változó a tár egy területére utal, ahol az értéke van. A változóknak bizonyos jellemzőik is vannak (mint pl. a típus), amelyek a tárolt érték értelmezését határozzák meg. összefoglalva:

Változón egy négy összetevőből álló nyelvi objektumot értünk. Ezek az összetevők a következők:

1) név,
2) jellemzők (attributumok),
3) hivatkozás (referencia) és
4) érték.



A változó összetevőinek szemléltetése

21. ábra

A 3. programban vegyük a kör sugarát változónak, azonosítóul a "sugar" nevet választva. Ekkor a számítást a

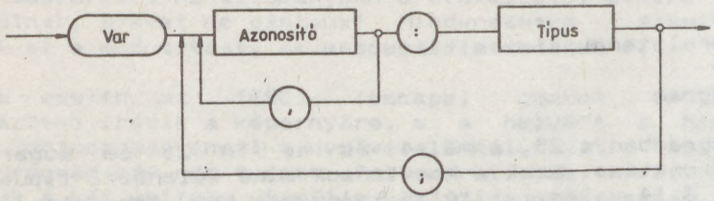
```
sugar*sugar*3.14
```

kifejezés írja le.

A változókat a programblokk elején deklarálni kell. A deklaráció a "var" kulcsszóval kezdődik. Utána adjuk meg a változók nevét és típusát. Az egész típus neve a Pascalban "integer".
Programunkat a

```
var sugar:integer;
```

deklarációval kell kiegészíteni. A deklaráció szintaxisát a 22. ábrán adjuk meg. Ha több változót deklarálunk azonos típussal, akkor az azonosítókat vesszővel elválasztva soroljuk fel. Egy deklaráción belül ugyanaz a típus többször is előfordulhat, sőt - a szabványos Pascaltól eltérően - egy programblokkon belül a var kulcsszó is többször előfordulhat.



A változódeklaráció szintaxisa

22. ábra

A deklarációval csak létrehozzuk a változó objektumot, ezzel még nem rendelünk hozzá értéket. A "sugar"-nak adatbevitel révén fogunk értéket adni. Az

```
olvasd <változó> értékét
```

elemi tevékenységet a readln utasítással írhatjuk le. A readln szintaxisa hasonló a writeln-éhez. A beolvasandó változó(k) nevét zárójelbe kell tenni, több változó esetén vessző az elválasztójel.
A 4. program a változó felhasználásával általánosítja a 3. programot.

```
program korteruleti;
var sugar:integer;
begin
   clrscr;
   readln(sugar);
   write('A',sugar,' sugaru kör terulete:');
   writeln(sugar*sugar*3.14);
   end.
```

A változó deklarálása és beolvasása

4. program

Figyeljük meg, hogy most a write utasítással több értéket íratunk ki. Először az 'A' füzért, majd a "sugar" változó értékét, végül a ' sugaru kor terulete:' füzért.
Ha a programot a gépen elkészítjük és lefuttatjuk ([R], Run menüpont), akkor a képernyő legfelső sorának elején a villogó helyőrt látjuk. Ezzel jelzi a gép, hogy adatra vár. Írjuk be a sugár értékét, majd nyomjunk [ENTER]-t. Ekkor folytatódik a program végrehajtása és egy pillanat alatt megkapjuk az eredményt.

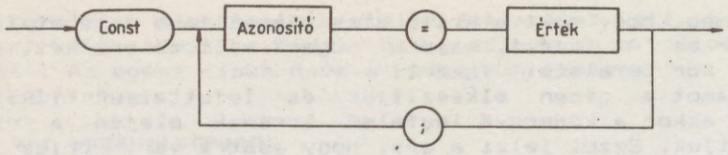
További két módosítást javaslunk. Célszerű volna, ha a program elindítása után nem kellene az üres képernyőt bámulva azon gondolkodni, hogy mi van ilyenkor. Írassuk ki a programmal!
Másrészt szebb lenne, ha ilyen közismert konstanst, mint a pi, a nevével használhatnánk. Konstansokat a deklarációs részben a "const" alapszó után definiálhatunk. A konstansok annyiban különböznek a változóktól, hogy értéküket nem a program végrehajtása közben kapják, hanem a fordítás alatt. Ezenkívül amíg egy változó értéke a program működésekor megváltozhat, a konstansé soha. De hát ezért konstans. Az 5. program a Majdnem Végleges Változat.

```
program korterulet2;

const pi=3.1415926;
var sugar:integer;

begin
   clrscr;
   writeln('Ezzel a programmal kor teruletet szamíthatja
ki.');
   writeln;
   write('Irja be a sugar erteket:');
   readln(sugar);
   writeln;
   writeln;
   write('A',sugar,' sugaru kor terulete:');
   writeln(sugar*sugar*pi);
   end.
```

A program útbaigazítja a felhasználót

5. program

Bemutattuk, hogyan definiálhatunk konstansokat. Most már nyugodtan bevallhatjuk, hogy erre nem is lett volna a jelen esetben szükség. A "pi" konstanst a Turbo Pascal eleve ismeri, ez egy ún. szabványos konstans. Az 5. programból tehát a konstansdeklarációt nyugodtan elhagyhatjuk.
Cserében bemutatjuk a konstansdeklaráció szintaktikáját (23. ábra).



A konstansdeklaráció szintaxisa

23. ábra

Láttuk, hogy a konstansoknak és a változóknak "típusuk" van. Ez a fogalom alapvető a programozási nyelvek fejlődésében. Típus alatt egy halmazt értünk, bizonyos - a halmazon definiált - műveletekkel együtt. A halmaz elemei a típust alkotó értékek.
A Pascal nyelvben a programozó sokféle típust létrehozhat, de van néhány típus, amely eleve rendelkezésünkre áll, ezek a szabványos típusok:

az egész (integer),
a valós (real),
a jel (char) és a
a logikai (boolean).

Amikor egy típusról beszélünk, a definíció szerint meg kell adni

1) magát a halmazt, azaz a típust alkotó értékeket,
2) a halmazon értelmezett műveleteket, vagyis az értékekkel végezhető elemi tevékenységeket.

A típushoz tartozó értékeket a programnyelvben használni kell, ezért szükségünk lesz

3) a típushoz tartozó konstansok szintaktikájára.

Végezetül meg kell mondanunk

4) milyen az értékek gépi megvalósítása.

Az utóbbira ugyan a Pascal programozásnál ritkán van szükség, de ha egyszer szükség van rá, akkor nem lehet semmi módon megkerülni.

3.4 Az egész típusok

Amíg a szabványos Pascal csak egyetlen egész típust ismer - ez az integer -, a Turbo Pascal ennél többet. Az egyes egész típusok között csak az értékhalmazban és a gépi ábrázolásban van különbség.

Az egész típusok értéktartománya

SHORTINT (csak a 4.0-s verziótól)	-128..127
BYTE	0..255
INTEGER	-32768..32767
WORD (csak a 4.0-s verziótól)	0..65535
LONGINT (csak a 4.0-s verziótól)	-2147483648..2147483647

Az értelmezett műveletek

Aritmetikai műveletek: +, -, *, div, mod
A - nemcsak kivonást, hanem negációt (ellentett előjelű érték képzését) is jelöl.
A div egészek osztása, azaz olyan osztás, amelynél a hányados is egész.
A mod a maradékképzés művelete. A mod B értéke az A B-re vonatkozó maradéka. E két műveletre teljesül a

```
j = k*(j div k) + j mod k
```

azonosság. Az aritmetikai műveleteket mutatja be a 6. program.

```
program egeszek;
begin
   writeln(23+42-55);
   writeln(5*123);
   writeln(123 div 5);
   writeln(123 mod 5);
   writeln((123 div 5)*5);
   writeln((123 div 5)*5 + 123 mod 5);
   writeln(5 div 123);
   writeln(32000-12000);
   writeln(32000+12000);
   writeln(32767);
   writeln(32767+1);
   end.
```

Műveletek egész értékekkel

6. program

Futtassuk le a programot és gondosan elemezzük az eredményeket. Lepődjünk meg, hogy két pozitív egész összegére negatív eredmény adódott. Ezt a rejtélyt majd a gépi ábrázolás ismeretében tudjuk felfedni.
Jegyezzük meg, hogy gondot jelent, ha a műveletvégzéskor az eredmény meghaladja a típus értékhatárait. Ilyenkor nem kapunk futás közbeni hibajelzést, csak az eredmény lesz hibás. Mindez a programozó felelőssége.

Van olyan egész számokkal végezhető aritmetikai művelet, amelynek az eredménye valós. Ez a / műveleti jelű osztási művelet. Mivel a / tulajdonképpen valós típusművelet, ezért bővebben ott tárgyaljuk.

Relációműveletek

A relációműveletek nem az egész típushoz kötődnek, hanem egy általánosabb tulajdonsághoz, a rendezettséghez. Rendezett halmazokon értelmezhetők a

<, <=, =, >=, >, <>

relációjelek. Jelentésük értelemszerű (<> jelentése: nem egyenlő). A kombinált jelek között nem lehet szóköz.
A relációjelek két egész (általánosan: két ugyanazon rendezett típushoz tartozó érték) között állhatnak, a relációműveletek eredménye logikai típusú.

További műveleteket csak a gépi ábrázolás ismeretében tudunk tárgyalni.

Az egésztípusú konstansok szintaxisa

Meg kell különböztetnünk a tízes és a tizenhatos számrendszerbeli írásmódot. Világosan kell látnunk, hogy mindegyik számrendszerben ugyanazokat az értékeket írjuk le, csak más alakban.

A tízes számrendszerbeli előjel nélküli egész a számjegyek egy nem üres sorozata. A szám elé a + vagy - előjelet írva előjeles egészhez jutunk. A tízes számrendszerbeli egész alatt előjeles vagy előjel nélküli egészt értünk. Az előjel nélküli és a pozitív számok szemantikája azonos.

Tizenhatos számrendszerben a tízes számrendszerbeli számjegyek mellett újabbakra is szükség van. Ezek a következők:

A tízes,
B tizenegyes,
C tizenkettes,
D tizenhármas,
E tizennégyes,
F tizenötös.

Egy tizenhatos számrendszerbeli szám mindig előjel nélküli, tízes számrendszerbeli számjeggyel kezdődő jelsorozat, amelyben a számjegyek és az A..F betűk keveredhetnek.
A számjeggyel való kezdést nem nehéz biztosítani, szükség esetén 0-t írunk a szám elejére.
Mivel tízes számrendszerben könnyebben tájékozódunk, meg kell tanulnunk a tizenhatos számrendszer számait tízesbe alakítani. A tizenhatos számrendszer alapszáma 16, tehát pl. a

C2AD

jelentése

$$C*16^3 + 2*16^2 + A*16^1 + D*16^0$$

azaz

$$12*4096 + 2*256 + 10*16 + 13*1 =$$

$$((12*16+2)*16+10)*16+13.$$

A számolás folytatását az Olvasóra hagyjuk.
Általában a négyjegyű wxyz hexadecimális szám átalakításához a

$$((w'*16 +x')*16 +y')*16 +z'$$

formulát használhatjuk. (A vesszőkkel a hexadecimális számjegyek decimális értékét jelöltük. Ha pl. w=D, akkor w'=13.)

Az egész számok gépi ábrázolása

A tárat alkotó kétállapotú elemek egyik állapotát a 0, a másikat az 1 számjegynek feleltetjük meg. Ezeket az értékeket biteknek nevezzük. A gépben tehát csak 0-kal és 1-esekkel - tehát kettes számrendszerben - fejezhetjük ki a számokat. A tárolási egység a bájt. Egy bájt 8 bitet tárol.

Az előjel nélküli egész típusok ábrázolása így elég kézenfekvő. A BYTE típusú értékeket egy bájtban ábrázoljuk. A minimális érték

00000000

a maximális pedig

11111111,

ami tízes számrendszerben felírva

```
  7   6   5   4   3   2   1   0
2 + 2 + 2 + 2 + 2 + 2 + 2 + 2  = 255.
```

A WORD típusú értékeket 16 biten, két szomszédos bájton ábrázoljuk. Két szomszédos bájt együttesét szónak nevezzük.

Az előjeles számok ábrázolásával csak annyi a gond, hogy az előjelet is 0-val vagy 1-gyel kell kifejezni, mivelhogy nincs más. A számok első bitje az előjelbit. A 0 a pozitív, az 1 a negatív számok előjelbitje. A SHORTINT típusú értékeket egy bájton, az INTEGER-eket egy szóban ábrázoljuk. Mivel a szám abszolút értékére most eggyel kevesebb bit jut, az előjeles egészek abszolút értéke fele a megfelelő előjel nélkülinek.
A LONGINT típusú számok ábrázolásához 32 bitet, két szomszédos szót használnak fel.

Ezekután könnyen megoldhatjuk a rejtélyt, miért lesz két pozitív szám összege negatív. A legnagyobb pozitív INTEGER típusú érték

```
01111111 11111111
```

ami tízes számrendszerben a kritikus 32767. Ha ehhez hozzáadunk 1-et:

```
 01111111 11111111
+                1
-------------------
 10000000 00000000
```

Az összeg láthatóan negatív számot ábrázol.

A tárban a bájtoknak címük van. A cím nem más, mint a bájtok sorszáma. Megjegyezzük, hogy az INTEGER típusú számok ábrázolásánál a bájtok sorrendje éppen fordított: elöl (az alacsonyabb címen) a szám kisebb helyi értékű része, hátul (a magasabb tárcímen) a magasabb helyi értékű rész van:

```
11111111 01111111
```

így az előjelbit a második bájt első bitje.
Az első bájtot LSB-nek (Less Significant Byte), a másodikat MSB-nek (Most Significant Byte) nevezik.

A gépi ábrázolás segítségével értelmezhetjük néhány speciális és bitenkénti műveletet.

Bitenkénti műveletek

Az egészeken értelmezett

not, and, or, xor

műveletek a számokkal bitenként hajtódnak végre:

not k — A k szám ábrázolásában minden bit az ellenkezőjére változik. Pl. ha k=00000001, akkor not k=11111110.

k and l — A művelet eredményében csak azok a bitek lesznek 1-es értékűek, amelyek mindkét számban 1-esek voltak. Ha k=01001110 és l=11100111, akkor k and l =01000110.

k or l — A művelet eredményében ott lesznek 1-es bitek, ahol legalább az egyik számban 1-es bit volt. Az előző k és l értékekkel: k or l =11101111.

k xor l — A művelet eredményében ott lesznek egyes bitek, ahol k-ban és l-ben különböző bitek voltak. Az előző értékekkel: k xor l=10101001.

Futtassuk le a 7. programot! Vegyük észre, hogy egy INTEGER szám negatívját a következőképpen is kiszámíthatjuk:

-n = not n + 1.

Ennek alapján könnyen felírhatjuk a negatív azámok gépi ábrázolását.

Speciális műveletek

A számot ábrázoló bitsorozatokat jobbra-balra léptethetjük:

n shl k — Az n szám bitjeit k értékkel balra toljuk. Ha n=00010101, akkor n shl 3 =10101000.

n shr k — Léptetés jobbra k-val. Az előző n-nel: n shr 3 =00000010.

Az egyes egész típusokhoz tartozó változókat a megismert SHORTINT, INTEGER, LONGINT, BYTE, WORD szabványos típusazonosítókkal deklaráljuk. Pl. a

```
var k,l: integer;
    a,b: byte;
```

változó deklarcióval két integer és két bájt típusú változót hozunk létre.

```
program bitenkenti;
begin
   writeln(not 200);
   writeln(893,' ',not 893 +1);
   writeln(893 and 711);
   writeln(893 or 711);
   writeln(893 xor 711);
   writeln(not (893 and 711));
   writeln((not 893) or (not 711));
   writeln(2 shl 8);
   writeln(512 shr 8);
   end.
```

Bitenkénti műveletek

7. program

Néhány egészek körében értelmezett függvényt is használhatunk, ezeket a 8. programban mutatjuk be.

```
program egeszfgv;
begin
   writeln(3,' ',succ(3));
   writeln(3,' ',pred(3));
   writeln(3,' ',sqr(3));
   writeln(-3,' ',abs(-3));
   end.
```

Egész függvények

8. program

A függvények argumentuma egész érték lehet, amit a függvény neve után zárójelben kell írni:

```
succ(k)   a k egész rákövetkezője,
pred(k)   a k egész megelőzője,
ord(k)    értéke k (csak a teljesség kedvéért),
abs(k)    k abszolút értéke,
sqr(k)    k négyzete (k*k).
```

Megemlítjük még, hogy a Pascal nyelv szabványos és legtöbb változatában ismeretes a

MAXINT

előre definiált konstans, amelynek értéke az INTEGER típus maximális értéke. Ez az érték függ a konkrét géptől és a nyelv megvalósításától. A Turbo Pascalban

MAXINT = 32767.

Ezt a konstanst a 4.0-s verzió nem ismeri.

3.5 Valós számok

A fizikai mennyiségek megadásához az egész számok általában nem alkalmasak. A Turbo Pascal 3.0-s verziója a REAL típust ismeri, de a 4.0-stól további valós típusok is használhatók: a SINGLE, a DOUBLE, az EXTENDED és a COMP.

A valós típusok értéktartománya

Mint a gépi reprezentációból majd látni fogjuk, a valós típusok értékhalmaza is csak véges sok értéket tartalmaz (ez annak következménye, hogy a számítógép véges rendszer). Létezik maximális és minimális valós érték, sőt, beszélhetünk szomszédos valós számokról és ezek különbsége is egy meghatározott érték. A valós számok használatakor tehát eleve meghatározott abszolót hibával kell számolnunk. Az abszolót hibakorlát mellett a gyakorlat szempontjából döntő fontosságó a számábrázolás relatív hibája, amit az értékes számjegyek száma határoz meg. Itt csak a REAL-lal foglalkozunk részletesen, a 4.0-sban használható egyéb típusokat csak vázlatosan jellemezzük.

REAL

Nagyságrendileg a -10^{38} .. 10^{38} intervallumba eső értékek ábrázolhatók, az abszolót pontosság 2^{-128}. A REAL típusú értékek max. 11 jegyre pontosak.

SINGLE

Nagyságrendileg a REAL-hoz hasonló, de csak kb. félakkora relatív pontosságot nyújt.

DOUBLE

15 jegyre pontos, nagyságrendileg

-10^{308} .. 10^{308}

közötti értékeket ábrázol.

EXTENDED

Az ábrázolási tartomány -10^{4932} .. 10^{4932}, pontossága 19-20 számjegy.

COMP

Csak egész értékű valós számokat tartalmaz (az abszolót pontosság 1). A szám 19-20 jegyre pontos, az értéktaromány

$$-2*10^{64} \ldots 2*10^{62} .$$

Műveletek valós számokkal

Valós értékekkel a szokásos

+, -, * és /

aritmetikai műveleteket és a

<, <=, =, >=, >, <>

relációműveleteket végezhetjük el. A valós osztás / művelete egész számokkal is végrehajtható, de ekkor az eredmény valós típusú lesz.
Az aritmetikai műveleteket a 9. program mutatja be.

```
program valos;
begin
   writeln((352.7-(12.3+0.71)*0.17)/2.0);
   writeln(2.3e28);
   writeln(2.3e-28);
   writeln(2.3e28+2.3e-28);
   writeln(2.3e28*1.2e22); {ez a muvelet hibat okoz}
   end.
```

Műveletek valós értékekkel

9. program

A program futtatásakor hibajelzést kapunk az utolsó writeln utasítás végrehajtásakor. A szorzás elvégzésénél tólcsordulás (overflow) lép fel, mert az eredmény meghaladja a REAL típusú maximális értéket, tehát nem ábrázolható.

Ellenkező esetben, ha a valós szám értéke az abszolót pontosság alá csökken, nem kapunk hibajelzést, csak az eredmény lesz zérus.

Használhatunk egy sor függvényt:

```
abs(x)    x abszolót értéke,
sqr(x)    x négyzete,
sqrt(x)   x négyzetgyöke,
```

```
sin(x)     x szinusza (x értékét ívmértékben kell
           megadni),
cos(x)     x koszinusza (x értékét ívmértékban kell
           megadni),
arctan(x)  az x tangensértéknek megfelelő szög
           (radiánban(a -pi/2 .. pi/2 intervallumból),
ln(x)      x természetes (e alapú) logaritmusa,
exp(x)     exponenciális függvény (e x-edik hatványa),
frac(x)    x törtrésze,
int(x)     x egész része(egész értékű valós).
```

A frac(x) és az int(x) között az

$$x = int(x) + frac(x)$$

összefüggés teljesül.

Bizonyára feltűnt az Olvasónak, hogy a műveletek közül hiányzik a hatványozás. Ezt a hiányt részben programozással pótoljuk (az egészek egészkitevős hatványainak kiszámításánál), másrészt pozitív alapot tetszőleges valós kitevőre emelhetünk az ln és az exp függvényekkel, ui:

$$x = \exp(r*\ln(x)).$$

A valós függvényeket a 10. program lefuttatásával tanulmányozhatjuk.

```
program valosfuggv;
begin
   writeln(sqrt(sqr(3.2)-4.0*1.02));
   writeln(exp(ln(16.0)/4.0));
   writeln('sin pi/2=',sin(pi/2));
   writeln('arctg 1 = ',arctan(1));
   writeln(3.18,' = ',int(3.18),' + ',frac(3.18));
   end.
```

Valós függvények

10. program

A valós konstansok írásmódja

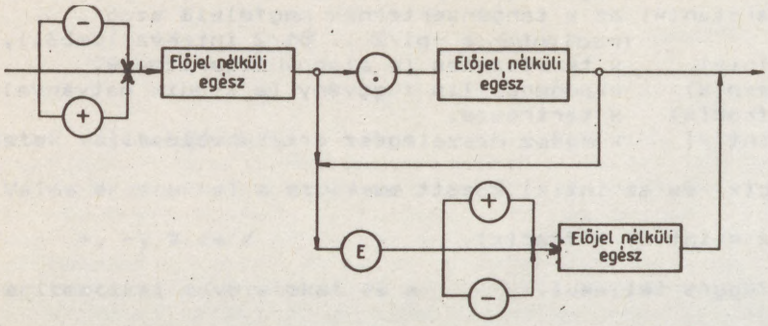
Kétféle szintaxist is használhatunk. Írhatjuk a valós számot tizedestört vagy kitevős alakban. Mindkét alakot tartalmazza a 24. ábra.

Eszerint helyes szintaxisúak a következő valós számok:

0.0, -328.002, 4e2, 0.12e-20,

hibásak:

12., .028, 43,6, 47e0.3



A valós konstans szintaxisa

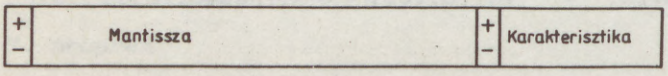
24. ábra

Ilyen alakban adhatjuk meg a valós változók értékét adatbevitelkor is.

A valós számok gépi ábrázolása

A valós számokat ún. lebegőpontos alakban ábrázolják. Ennek a számok normál alakú írásmódja az alapja, amelynél megadunk egy 1 és 10 közötti tizedestörtet (mantissza) és ezt 10 valamely egészkitevős hatványával szorozzuk. 10 kitevőjét a szám karakterisztikájának nevezzük.

A lebegőpontos számábrázolásban a mantiszát és a kitevőt adják meg, tehát a valós számot egy számpár reprezentálja (25. ábra).



A lebegőpontos számábrázolás elve

25. ábra

A mantissza rendszerint fixpontos tört ábrázolású. A kettedespontot az első és második bit közé kell képzelni, ennélfogva a mantissza értéke 1-nél kisebb (az első bit az előjelbit). A lebegőpontos számot akkor nevezzük normalizáltnak, ha a mantissza első és második bitje különbözik. Ebből következik, hogy a normalizált alakban a mantissza értéke 1 és 0.5 közé esik (miért?).
A Turbo Pascal a REAL értékeket hasonló módon ábrázolja 6 szomszédos bájton. Az első bájt a mantissza LSB-je, az 5. az

MSB. A karakterisztikát a 6. bájt tárolja. A 26. ábrán bemutatjuk néhány REAL típusú érték tárbeli ábrázolását, de a szemléletesség kedvéért a mantissza bájtjait ellenkező sorrendben rajzoltuk le.

```
  1.0000000000E-02
00100011110101110000101000111101011100000 01111010 11111010
  1.0000000000E-01
01001100110011001100110011001100110011001100 01111101 11111101
  1.0000000000E+00
00000000000000000000000000000000000000000000 10000001 00000001
  1.0000000000E+01
00100000000000000000000000000000000000000000 10000100 00000100
  1.0000000000E+02
01001000000000000000000000000000000000000000 10000111 00000111
  2.0000000000E+00
00000000000000000000000000000000000000000000 10000010 00000010
  3.0000000000E+00
01000000000000000000000000000000000000000000 10000010 00000010
 -5.0000000000E+00
10100000000000000000000000000000000000000000 10000011 00000011
 -5.0000000000E-01
10000000000000000000000000000000000000000000 10000000 00000000
 -5.0000000000E+01
11001000000000000000000000000000000000000000 10000110 00000110
```

REAL értékek a tárban

26. ábra

Két apró turpisság miatt nézi a kedves Olvasó tanácstalanul az ábrát. Az egyik a mantissza ábrázolásában van. A mantissza mindig normált és abszolót értéket tárol. Tehát az első bit mindig 1-es értékű. De ha ezt tudjuk, akkor ezt a bitet nem is kell tárolni, helyette az előjelbitet látjuk, ami negatív számnál 1, pozitív számnál 0. (A mantissza abszolót értékét, tehát a negatív számok csorbítatlanul mutatják.)

A másik cselt a karakterisztikánál leplezhetjük le. Itt egyáltalán nincs előjelbit. A tárolt értékből úgy kapjuk meg a tényleges mantisszát, ha kivonunk 128-at.
A karakterisztika 2 kitevőjét jelenti.

Ha tehát a mantissza 0, a karakterisztika 130, akkor a tárolt érték pozitív. Ezért az ábrázolt szám mantisszája 0.5, karakterisztikája 130-128=2, tehát a tárolt érték

$$0.5*2^{2} = 2.$$

A valós típusú változókat a REAL típusnévvel deklaráljuk, pl.:

```
var sugar, magassag: real;
```

A valós számoknak a tárban többnyire csak egy közelítő értékét találjuk. Ezért az "=" relációt ne használjuk valós értékek összehasonlítására. Ne azt vizsgáljuk pl., hogy egy valós változó értéke egyenlő-e 0-val, hanem hogy abszolút értékben elég kicsi-e:

```
abs(x) < 1.0E-8.
```

Nem mindig biztonságos ilyen abszolút korlátot sem használni, jobb, ha ehelyett relatív eltérést vizsgálunk. Így az

```
x = y
```

reláció helyett pl. az

```
abs(x-y)/max(x,y) < 1.0E-8
```

kifejezést tanácsos használni. Ezzel voltaképp azt vizsgáljuk, hogy x és y értéke az első 8 számjegyben megegyezik-e. (max(x,y) az x és y közül a nagyobbikat jelöli.)

3.6 A karakter típus

A számítógéppel nemcsak számokat, hanem szöveges információkat is feldolgozunk. A fordítóprogram és az Editor is szövegeken végez műveleteket. A szövegek betükből, írásjelekből, számjegyekből épülnek fel. Ezeket az objektumokat jeleknek, karaktereknek neveztük. Ezt a halmazt a Pascal nyelv típusként kezeli.

A karakterek halmaza

A használható jelek halmazát mindig a használatos kódrendszer határozza meg. Az ASCII (American Standard Code for Information Interchange) kódrendszer jeleit használhatjuk, pontosabban azt a készletet, amit eszközeink (billentyűzet, monitor, nyomtató) számunkra hozzáférhetővé tesznek. Az ASCII kódtáblázatot a B) mellékletben közöljük.

A karakterek száma legfeljebb 256 lehet.

Műveletek karakterekkel

A karakterek halmaza rendezett, így a relációműveletek értelmezettek.

Értelmezett a karakter típuson a következő néhány függvény:

succ(c)	a c karakter rákövetkezője,
pred(c)	a c karakter megelőzője,
ord(c)	a c rendszáma (kódja), ennek a függvénynek az értéke nem karakter, hanem egész,
chr(i)	az i egész értékhez rendelt karakter.

```
program karakterfuggv;
  begin
    writeln('"a" rakovetkezoje:',succ('a'));
    writeln('"c" megelozoje:',pred('c'));
    writeln('"a" kodja= ',ord('a'));
    writeln('a 237 kodu karakter ',chr(237));
    end.
```

Karakterfüggvények

11. program

A karakterkonstansok írásmódja

A karaktert felsővesszők közé kell tenni. Pl.

'F', 'j', 'f', '7', '?', '$'.

f és F különböző karakterek. A felsővessző maga is karakter, amit ugyancsak ábrázolni kell. Különleges szerepe miatt az írásmódja is kivételes:

'''',

tehát megduplázva tesszük felsővesszők közé.
Vannak olyan jelek, amelyekhez nem tartozik látható jelalak. Ilyen pl. az új sor karakter vagy a csengőjel. Ezeket a jeleket kontroll karaktereknek nevezzük. A kontroll karaktereket nem tudjuk az előbbi módon karakterkonstansként megadni.

A Turbo Pascalban erre más lehetőségek vannak.
Egyik a ^ kontrolljel használatából adódik. A kontrollkaraktereket a billentyűzetről a [CTRL] billentyű segítségével tudjuk beírni. Ha a [CTRL] billentyűt lenyomva tartva leütünk egy másik billentyűt is, ezzel a bevitt jel valamilyen kontrollkarakter. Pl. a [CTRL] [G]-t lenyomva a

csengő megszólal. Ugyanezt a hatást érjük el a programban a ^G kombinációval, a

```
writeln(^G)
```

utasítás végrehajtásakor.

A # jellel bármelyik karaktert kifejezhetjük, ehhez csak a megfelelő kódot kell ismerni. Ha tudjuk, hogy a csengőjel kódja 7, akkor a ^G helyett #7 is használható:

```
writeln(#7).
```

Az A betű kódja 65, tehát 'A' és #65 is ugyanazt a karakterkonstanst jelöli.

A karakterek gépi ábrázolása

A karaktereket kódjukkal ábrázoljuk. A kód egy bájt típusú érték. Egy karakter ábrázolásához tehát egy bájtra van szükség. A ^G karakter tárbeli ábrázolása a 7 értékű

```
00000111
```

bitsorozat.

A karakter típusú változókat a CHAR típusnévvel deklaráljuk:

```
var c, jel: char;
```

3.7 A logikai értékek

A logikai értékek halmaza

Ez a halmaz kételemű, a false és a true (hamis és igaz) logikai értékekből áll: az igazságértékek halmaza.

Logikai műveletek

A Pascalban szokásos not, and, or műveleteken kívül a Turbo Pascalban az xor is használható.

NOT a tagadás művelete:

A	not A
false	true
true	false

Az AND az "és" művelet (konjunkció), a következő értéktáblázattal definiáljuk:

		B	
		false	true
A	false	false	false
	true	false	true

OR a megengedő "vagy" művelet (diszjunkció):

		B	
		false	true
A	false	false	true
	true	true	true

A XOR a kizáró "vagy" művelet:

		B	
		false	true
A	false	false	true
	true	true	false

A logikai értékek halmaza rendezett, megállapodás szerint

false < true.

Emiatt a relációműveletek az igazságértékek halmazán értelmezettek. Két esetet emelnénk ki mint gyakorlatilag jelentős műveletet, <= és az = relációkat.

<= az implikáció (következtetés) művelete. A <= B jelentése: ha A, akkor B.

		B	
		false	true
A	false	true	true
	true	false	true

= az ekvivalencia (egyenértékűség) művelete. A = B jelentése: akkor és csak akkor A, ha B.

A \ B	false	true
false	true	false
true	false	true

Értelmezett a logikai értékekre a pred, succ, ord függvény.

```
pred(true) = false,
succ(false)= true,
ord(false) =0,
ord(true)  =1.
```

Az egész számok vizsgálatánál jól használható az odd predikátum (predikátumon olyan függvényt értünk, amelynek értékkészlete az igazságértékek halmaza).

odd(k) = true akkor és csak akkor, ha k páratlan.

A logikai konstansok szintaxisa

Az írásmód, mint eddig is láttuk: false, ill. true.

A logikai értékek gépi ábrázolása

A false értéket a 0,
a true-t az 1 értékű bájt reprezentálja.

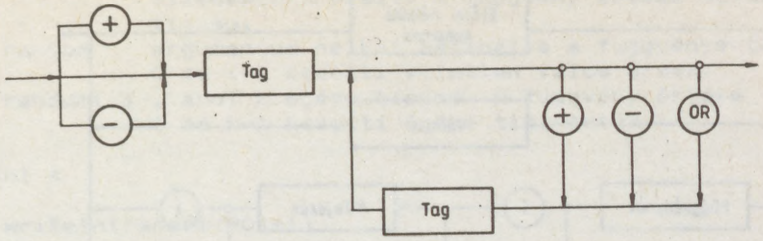
A logikai változókat a BOOLEAN típusnévvel deklaráljuk, pl.:

```
var kesz, hibavolt: boolean;
```

A boolean típusú változók értékét - a többi szabványos típussal ellentétben - nem lehet adatbevitel révén meghatározni.

3.8 Kifejezések

A változókból, konstansokból és függvényekből műveleti jelekkel és zárójelekkel felépített formulákat kifejezéseknek nevezzük. A kifejezés értékének típusától függően beszélünk egész vagy valós aritmetikai kifejezésről, karakterkifejezésről, ill. boolean kifejezésről. A kifejezés képzési szabályait szintaxisábrák segítségével több lépésben adjuk meg. A 27. ábrán a kifejezést tagokból építjük fel.

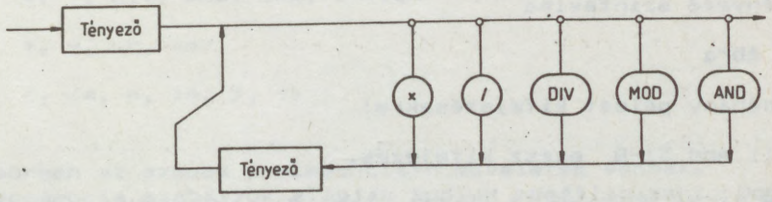


A kifejezés szintaxisa

27. ábra

Természetesen az or művelet csak akkor jöhet számításba, ha a tagok logikai értékűek. Real vagy integer tagoknál pedig a + és - használható.

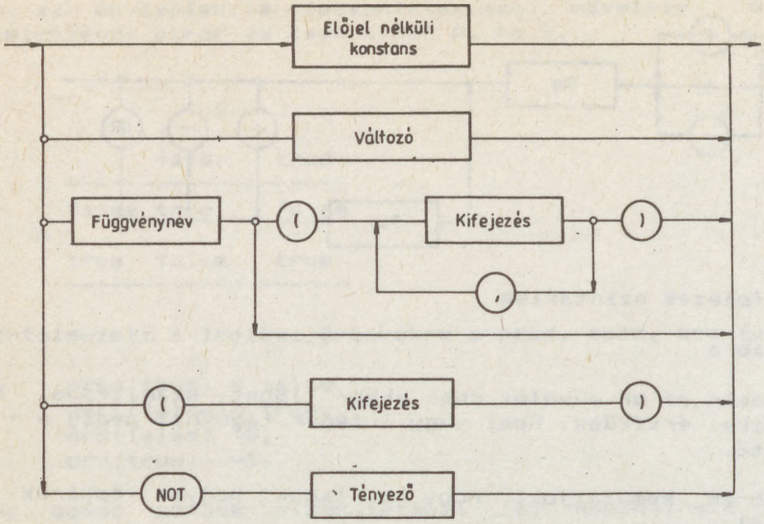
A 28. ábrán bemutatjuk, hogy a tagok hogyan épülnek fel tényezőkből.



A tag szintaxisa

28. ábra

A * és a / valós és egész, a mod és a div csak egész, az and pedik kizárólag logikai értékekkel használható. A valós és egész értékeket keverhetjük, a tag értéke ilyen esetben valós típusú lesz.
A tényező felépítése bonyolúltabb (29. ábra).



A tényező szintaxisa

29. ábra

Nézzünk néhány példát kifejezésekre!

```
7*(11 mod 3)-8  egész kifejezés.
```

7*(11 mod 3) és 8 tagok, 7 és (11 mod 3) tényezők, 11 mod 3 kifejezés.

```
sqrt(abs(A-B) + ord(succ('F'))) valós kifejezés.
```

Az egész kifejezés egyetlen függvényhívás, tehát tényező és tag is. A függvény argumentuma valós, tehát A és B közül legalább az egyik valós, hogy a különbséggel együtt abs(A-B) is valós legyen, mert az ord függvény értéke egész.

```
chr(7*(11 mod 3)+ 12) karakterkifejezés.

not(G and H) or (x<2.3)  boolean kifejezés.
```

Létezik még néhány számokkal kapcsolatos függvény a Turbo Pascalban, amit eddig nem tárgyaltunk, de a kifejezésekben ezeket is használhatjuk.

```
round(x)  az x valóst egészre kerekíti. A függvény
          értéke egész típusú,
trunc(x)  az x valós értékét csonkítja, elhagyja a
```

tizedestört részt. A függvény értéke egész típusú.

random argumentum nélkül használva a függvényérték 0 és <=1 közötti véletlen valós érték.

random(k) , ahol k egész típusú. A függvény értéke 0 és k-1 közötti egész tip. érték.

Például a

```
writeln(random(90)+1)
```

utasítással kiírathatunk egy lottószámot.

A kifejezések felépítésénél tekintettel kell lenni a műveletek precedenciájára (elsőbbségi szabályaira). Ezeket a szabályokat burkoltan tartalmazza a szintaxis, de nem árt tudatosítani. A következő táblázat csökkenő precedenciával sorolja fel a műveleteket:

- (ellentett képzés), not

*, /, div, mod, and, shl,shr

+, -, or, xor

<, <=, =, >=, >, <>

Egy sorban az azonos precedenciájú műveletek vannak.
A precedencia szabályok alapján tudjuk megállapítani, hogy a gép hogyan számítja ki egy kifejezés értékét. A magasabb precedenciájú műveletektől halad az alacsonyabbak felé. Ha azonos precedenciájú műveletekből több is van, akkor általában balról jobbra haladva végzi a műveleteket, hacsak a fordítóprogram nem optimalizálja a kifejezések kiértékelését. Ha ezt a művelet végrehajtási sorrendet meg akarjuk változtatni, zárójeleznünk kell.

Két példát szeretnénk mutatni, mindkettő lehetséges hibaforrás. Tegyük fel, hogy a

$$\frac{P}{Q*W}$$

törtkifejezést kell átírni Pascal nyelvre. A nevezőt zárójelbe kell tenni:

```
P/(Q*W),
```

mert P/Q*W a balról-jobbra szabály miatt

```
P
-*W
Q
```

A másik - ennél gyakoribb - hibát a relációkból felépített logikai kifejezések felírásánál követik el a kezdők. Mivel a relációműveletek kisebb precedenciájúak, zárójelbe kell tenni őket. Az

```
(x<0.0) and (x>-2.5)
```

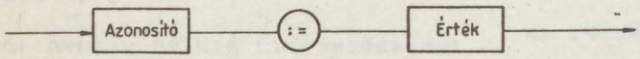
kifejezés szintaktikusan hibátlan, a gyakran helyette írt

```
x<0.0 and x>-2.5
```

annál kevésbé.

3.9 Az értékadás

A változók nemcsak adatbevitel, hanem értékadás révén is értéket kaphatnak. Az értékadás szintaxisát a 30. ábrán adjuk meg.



Az értékadás szintaxisa

30. ábra

Az A:=x értékadást így olvassuk: "legyen A értéke x".
Az értékadásban használt változókat előzőleg deklarálni kell. A jobb oldali kifejezésben pedig csak olyan változó lehet, amely korábban már értéket kapott (akár adatbevitel, akár értékadás révén).

Egy változónak nem adhatunk tetszőleges értéket, csak olyat, amelyik a típusával kompatíbilis. A

```
var x,y: real;
    i,j: integer;
    l,m: boolean;
    p,q: char;
    a,b: byte;
```

deklarációt feltételezve az

```
x:= y;
i:= j;
l:= m;
p:= q;
a:= b;
```

```
i:= a;
b:= i;
x:= j;
```

értékadások helyesek. A b:=i végrehajtása okozhat ugyan hibát, ha i>255, ilyenkor b csak i LSB-jét fogja tárolni, de a fordítóprogram elfogadja az ilyen értékadásokat. Hibás viszont az

```
i:= y;
a:= p;
l:= a;
```

és a hasonló értékadások mindegyike.

Az A és B objektumok típusa az értékadás szempontjából kompatíbilis (vagyis az A:=B értékadás helyes), ha

K1) A és B típusa azonos,

K2) A és B mindketten egész típusóak,

K3) A és B mindketten valós típusóak (csak a 4.0-s verziótól kezdve),

K4) A valós és B egész típusó.

Ha a továbbiakban újabb kompatibilitási szabályokat fogalmazunk meg, folytatólagosan fogjuk számozni. A következő a K5-ös lesz.

A kompatibilitást esetenként típusváltó függvényekkel biztosítjuk. Ilyenek az ord, a chr, a round és a trunc. Pl. az

```
i:= round(x);
b:= ord(p);
q:= chr(a);
```

értékadások helyesek.

3.10 Feladatok

1) Adja meg egy lakóház "szintaxisát" szintaxisábrával! A lakóház alapból, pincéből, földszintből, 0 vagy több emeletből, padlásból és tetőből áll. A földszinten és az emeleteken 1 vagy több lakás van. Minden lakás előszobából, konyhából, fürdőszobából, WC-ből és 1 vagy több szobából áll.

2) A következő konstansdeklarációk közül melyekben van szintaktikus hiba és miért:

```
const nagyszam=200;
const kisszam=0 or 1 or 2;
```

```
const elsobetu:'A';
const nagy= 200.0;
      kicsi=-200.0;
```

3) Definiálja a következő konstansokat:
1/8-ot, az Avogadro-féle számot, a g-t, és a 7-et valósként, egészként és karakterként, továbbá a csengőjelet.

4) Írja át a fejezet programjait változók használatával és próbálgassa a műveleteket különböző adatokkal!

5) Határozza meg a következő kifejezések értékét:

```
1-2-3-4-5,
1*4*2-4*2+3,
sqr(3*4-2),
(7 div 3 -2)*5,
3*(9 mod 2)-5,
trunc(2.7+3.9/2.0+5.2) mod 3.
```

6) Határozza meg a következő kifejezések értékét:

```
A+B div trunc(D)+3,
D-0.75/(E+140.0)*70.0,
sqrt(abs(B-A)+succ(ord(F))),
chr(10*A mod sqr(B)),
not(A>=B) and G,
(odd(B) or (sqr(B)>C)) and (G and not (abs(B)<>6)),
```

feltéve, hogy A=7, B=-6, C=30, D=10.86, E=1e3, F='*', G=true.

7) Ha a következő formulák valamelyikét hibásan írtuk volna Pascal nyelven, tessék kijavítani:

$$\frac{a+b}{c}$$ sqrt(A+B/C),

$$ax^2 + bx + c$$ A*X*X+B*X+C,

$$(6.02 * 10^{23})(\ln(1+e^{(1-x)}))$$ 6.02e23(ln(1+exp(x)-1.))

8) Írja fel Pascal nyelven a következő kifejezéseket:

$$4a, \quad 6a-2x, \quad -\sin\left[2\,(f_c + f_m)t - \frac{\pi}{2}\right], \quad \arcsin(x).$$

9) Tegyük fel, hogy A integer, B és R valós, Q boolean típusú változók. Melyek hibásak a következő értékadások közül és miért:

```
A:=2+3.0;
A+2:=B;
3:=2+1;
A:-2-(-(-1+2)-3)-2*4;
A:=2*-3;
B:=(-6.73)2;
Q:=2+B=6*8;
R:=6.4 div 8;
Q:=2+B:=8;
```

10) Írjon programot tetszőleges méretű és anyagú téglatest tömegének meghatározására!

11) Mit ír ki a következő program?

```
program namit;
var p,q,r,s,eredmeny:boolean;
begin
  p:=true;
  q:=p;
  r:=false;
  s:=r;
  writeln;
  writeln;
  eredmeny:=not p;
  writeln(eredmeny);
  eredmeny:=not not q;
  writeln(eredmeny);
  eredmeny:=q or s;
  writeln(eredmeny);
  eredmeny:=p and s;
  writeln(eredmeny);
  eredmeny:=p and q and s;
  writeln(eredmeny);
  eredmeny:=(p or r) and q;
  writeln(eredmeny);
  end.
```

12) Mit csinál a következő program?

```
program leresc;
var a,b,w: integer;
begin
  clrscr;
  write('a=');
  readln(a);
  write('b=');
  readln(b);
  w:=a;
  a:=b;
  b:=w;
  writeln;
  writeln('a=',a);
```

```
writeln('b=',b);
end.
```

13) Írjon programot gépkocsi átlagfogyasztásának számítására. A program adatai: a km-számláló előző állása, a km-számláló állása tankoláskor, a tankolt mennyiség. Tegyük fel, hogy minden tankoláskor teletöltjük a tartályt. Készítsen értelmes, szöveges kiíratást!

14) Készítsen programot jövedelemadó-számításra. A program bemenő adata a jutalom bruttó összege, és az adókulcs. Az eredményt a következőképpen írassa ki:

```
Brutto jutalom      xxxxx Ft
Jovedelemado nn%    xxxxx Ft
Nyugdijjarulek       xxxx Ft
-------------------------------
Kifizetendo         xxxxx Ft
```

A nyugdíjjárulék a brutto jövedelem 10%-a. Minden értéket külön kell kerekíteni.

15) A pincében egy henger alakú hordóban tároljuk a fűtőolajat. Készítsünk programot, amellyel felmérhetjük a készletet, valamint a hordóból hiányzó mennyiséget. Ki kell számítani a beszerzendő mennyiség árát is.
A program bemenő adatai: a hordó átmérője és magassága, az olajfelszínnek a hordó felső szélétől mért mennyisége, az olaj literenkénti ára.
Tervezzen tetszetős és a felhasználót irányító adatbevitelt és áttekinthetően írassa ki az eredményt is!

4. ÖSSZETETT TEVÉKENYSÉGEK PROGRAMOZÁSA

A szokásos tevékenységszerkezetek felépítéséhez szükséges eszközöket a Pascal nyelv tartalmazza, sőt, néhány újabbal tovább könnyíti a programozók munkáját.

4.1 Tevékenység sorozat

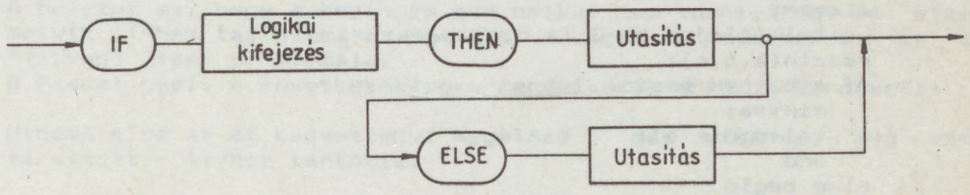
A tevékenységsorozatokat általában nem kell külön jelölni, elég az utasításokat egyszerűen egymás után írni. Elválasztáshoz elég a pontosvessző, de áttekinthetőbb a programunk, ha minden utasítást külön sorba írunk.
Gyakran szükség van arra, hogy egy utasítássorozat utasításait szintaktikusan egy összetett utasítássá egyesítsük. Ilyenkor a begin ... end kulcsszavakat használjuk:

```
                                  begin
utasítás1;   utasítás-              utasítás1;   egyetlen
utasítás2;   sorozat,               utasítás2;   összetett
.........    több                    .........    utasítás
utasításn;   utasítás               utasításn;
                                    end
```

Minden olyan helyen, ahol a szintaxis utasítást ír elő, összetett utasítást is használhatunk, de utasítássorozatot nem.

4.2 Az if utasítás

Az egy- és a kétágú kiválasztási szerkezeteket az if utasítással programozhatjuk. Az if utasítás szintaxisát a 31. ábrán mutatjuk be.



Az if utasítás szintaxisa

31. ábra

A then utáni utasítás itt akkor és csak akkor hajtódik végre, ha a logikai kifejezés értéke true. Ügyeljünk arra, hogy az else kulcsszó előtt nem állhat pontosvessző (utasítás belsejében vagyunk, nem a végén).
A következő 12. programban az a, b, c értékek közül a minimálisat választjuk ki if utasításokkal.

```
program minimum3;

var a,b,c,min: integer;

begin
  clrscr;
  writeln('Irjon be 3 egészszámot!');
  readln(a,b,c);
  if a<b then
     min:=a
  else
     min:=b;
  if min>c then
     min:=c;
  writeln('A minimális érték',min);
  end.
```

A minimális érték kiválasztása

12. program

Az if utasítás ágaiban bármilyen utasítás használható, összetett utasítás is. Ezt látjuk a 13. programban, amely nemcsak a minimális értéket határozza meg, de megadja a változót is, amelyben a minimális érték van.

```
program masminimum3;

var a,b,c,min: integer;
    valtozo: char;

begin
  clrscr;
  writeln('Irjon be 3 (a,b,c) egészszámot!');
  readln(a,b,c);
  if a<b then begin
     min:=a;
     valtozo:='a';
     end
  else begin
     min:=b;
     valtozo:='b';
     end;
  if min>c then begin
```

```
        min:=c;
        valtozo:='c';
        end;
    writeln('A minimalis ',valtozo,'=',min);
    end.
```

összetett utasítás az if-ben

13. program

Az if után bonyolultabb logikai kifejezést is használhatunk. Ha a then utáni tevékenységet a 0.0 és 0.8 közötti x-ekre kell csak végrehajtani, akkor az

```
if (x>=0.0) and (x<=0.8) then ...
```

a megoldás.
Az if szerkezet ágaiban álló utasítás maga is lehet if utasítás. Ilyenkor kétértelmű szerkezetek is létrejöhetnek, ami programban megengedhetetlen. Tekintsük az

```
if a>0 then if b>0 then b:=a+1 else a:=b+1
```

szerkezetet. Az egyik értelmezési lehetőség (begin és end használatával téve egyértelművé):

```
if a>0 then begin
   if b>0 then
      b:=a+1
   else
      a:=b+1;
   end
```

a másik:

```
if a>0 then begin
   if b>0 then
      b:=a+1;
   end
else
   a:=b+1
```

A helyzet az, hogy a begin és end nélkül nem tudni, hogy az else melyik if-hez tartozik, az elsőhöz vagy a másodikhoz. Ez a "fityegő else" problémája.
A Pascal nyelv a következőképpen rendelkezik erről a kérdésről:

Minden else az őt közvetlenül megelőző - más else-vel még nem társított - if-hez tartozik.

Ez az első értelmezés érvényességét jelenti.
Egy további példát mutatunk egymásba ágyazott if utasításokra, hogy a szabályt világosabbá tegyük:

```
if feszultseg>6 then
   if feszultseg>12 then
      if feszultseg>24 then
         writeln('A feszultseg veszelyesen magas.')
      else
         jelzes:=magas
   else
      jelzes:=normalis
else
   writeln('A feszultseg tulsagosan alacsony.');
```

Az írásmóddal is szemléltetjük az if-ek és else-k összetartozását, de nem az írásmód - ez a gépnek teljesen közömbös - hanem a fityegő else-re vonatkozó szabály, amit a fordítóprogramba építettek, határozza meg a programrészlet működését.

4.3 A case szerkezet

A case utasítást kettőnél többágó kiválasztási tevékenység programozására használjuk. A 14. programban számmal kódolt hónapok nevének szöveges kiírását oldjuk meg ezzel a szerkezettel.

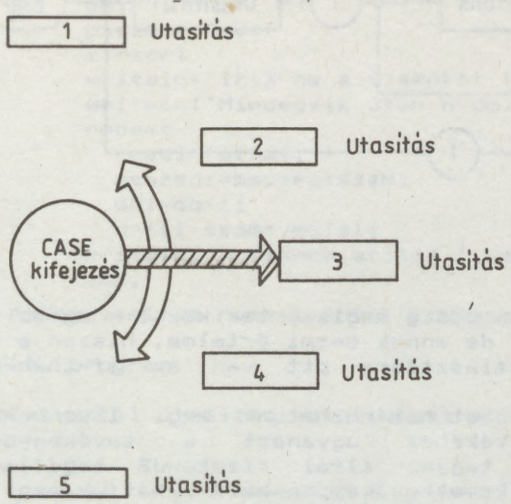
```
program datumatalakito;

var ev,ho,nap: integer;

begin
  clrscr;
  write('Evszam:');
  readln(ev);
  write('Honap (szammal):');
  readln(ho);
  write('Nap:');
  readln(nap);
  writeln;
  write('A mai datum',ev);
  case ho of
     1: write('. januar');
     2: write('. februar');
     3: write('. marcius');
     4: write('. aprilis');
     5: write('. majus');
     6: write('. junius');
     7: write('. julius');
     8: write('. augusztus');
     9: write('. szeptember');
    10: write('. oktober');
    11: write('. november');
    12: write('. december');
    end;
```

```
    writeln(nap,'.');
    end.
```

A hónapok kiíratása

14. program

Ez a szerkezet úgy működik, hogy mindig a ho változó értékével azonos értékkel megjelölt utasítás hajtódik végre. Ezt a végrehajtási mechanizmust a 32. ábrán szemléltetjük. Az egyes ágakban tetszőlegesen bonyolult összetett utasítások is lehetnek.



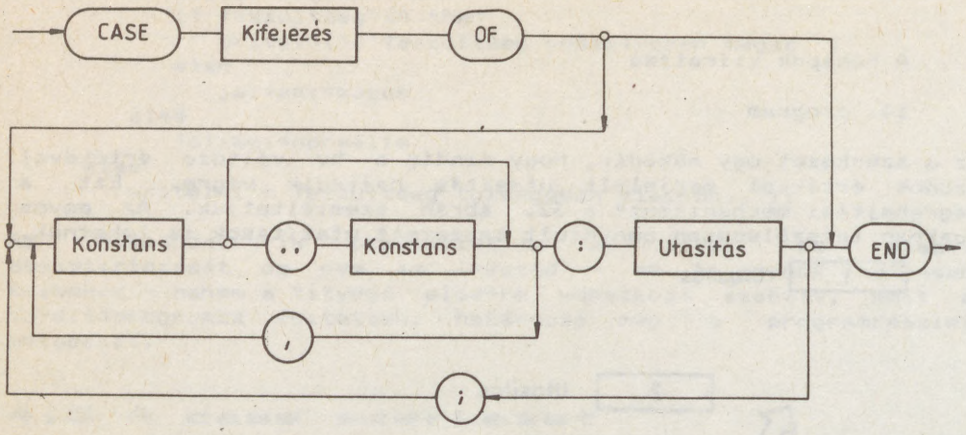
A case szerkezet működése

32. ábra

Ha ho értéke hibás és nem fordul elő a case utasításban, akkor a Turbo Pascal szabálya szerint a case hatástalan utasítás lesz. (A szabványos Pascalban és több megvalósításban ilyenkor hibajelzést kapunk.) Ha a nem definiált értékekhez is akarunk tevékenységet rendelni, akkor az else kulcsszót használhatjuk itt is. A 14. program egyáltalán nem ír ki semmit a hónap helyett, ha pl. ho=13. Ilyenkor írassuk vissza a hibás sorszámot, utána 3 kérdőjellel. Ehhez a következő módosítást kell elvégezni:

```
    ..........................
    ..........................
      12: write('. december');
      else
        write('. ',ho,'???');
      end;
    ..........................
    ..........................
```

A 33. ábra a case utasítás teljes szintaxisát mutatja be.



A case szintaxisa

33. ábra

A case kifejezés típusa - az eddig megismertek közül - egész vagy karakter lehet. Boolean is, de ennek semmi értelme, hiszen a true és false szerinti esetszétválasztásra ott van az if-then-else szerkezet.
A .. jelöléssel intervallumokat határozhatunk meg. Ilyenkor az intervallum mindegyik értékéhez ugyanazt a tevékenységet rendeljük. Egy szervezet tagjai által fizetendő tagdíjat a jövedelemtől függően pl. a következőképpen határozhatjuk meg:

```
case jovedelem of
       0..4000 : tagdij:=0;
    4001..7000 : tagdij:=30;
    7001..10000: tagdij:=80;
   10001..15000: tagdij:=150;
   15001..30000: tagdij:=300;
   else          tagdij:=600;
   end;
```

4.4 Ismétlési szerkezetek

Hátultesztelő ciklusszerkezetet a repeat-until utasítással valósíthatunk meg. Az utasítás használatát a 15. programban mutatjuk be. A program valós számokból számít átlagot. összegzi a számokat és - mivel nem tudjuk előre, hogy hány számunk lesz - számlálja őket. A tevékenység a 0 számra fejeződik be (végjel). Mivel a végjelet csak utólag észleljük (a ciklus hátultesztelő), a "szamlalo"-t módosítani kell, hiszen a végjelet nem kell figyelembe venni az átlagszámításnál.

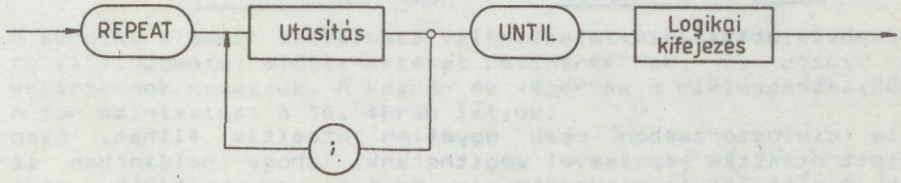
Mielőtt hozzáfognánk egy ciklikus program kipróbálásához, írjuk a program elejére a {$U+} compiler direktívát (lásd: D) melléklet). Ekkor a hibás, végtelen ciklust tartalmazó program futását megszakíthatjuk a [CTRL] [C] gombokkal.

```
program atlag1;
  const vegjel=0.0;
  var db:integer;
      szam:real;
      osszeg:real;
  begin
    db:=0;
    osszeg:=0.0;
    clrscr;
    writeln('Irja be a szamokat (vegjel:',vegjel:2:0,')!');
    writeln('Mindegyik utan nyomjon [enter]-t');
    repeat
      readln(szam);
      osszeg:=osszeg+szam;
      db:=db+1;
      until szam=vegjel;
    writeln('A szamok atlaga ',osszeg/(db-1.0):8:2);
    end.
```

Átlagszamítas repeat-until ciklussal

15. program

A repeat-until utasítás szintaxisát a 34. ábrán mutatjuk be.



A repeat-until szintaxisa

34. ábra

A ciklustörzs több utasításból állhat, ismételt végrehajtása abbamarad, ha az until utáni logikai kifejezés értéke true. Mivel a ciklus hátultesztelő, a ciklustörzs akkor is végrehajtódik egyszer, ha a kilépési feltétel eleve teljesül.

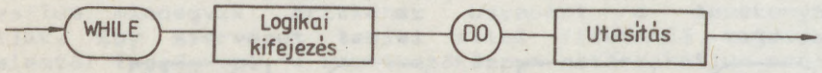
Elöltesztelő ismétlési szerkezetek programozásához a while utasítás áll rendelkezésünkre. Az átlagszámítást - néhány részletben apró eltéréssel - a 16. programban a while utasítással fogalmaztuk meg.
Mivel az ismétlési feltétel vizsgálata az adatként bevitt értéktől függ, a "szam" olvasása meg kell előzze a feltételt.

```
program atlag1;
  const vegjel=0.0;
  var db:integer;
      szam:real;
      osszeg:real;
  begin
    db:=0;
    osszeg:=0.0;
    clrscr;
    writeln('Irja be a szamokat (vegjel:',vegjel:2:0,')');
    writeln('Mindegyik utan nyomjon [enter]-t');
    readln(szam);
    while szam>vegjel do begin
      osszeg:=osszeg+szam;
      db:=db+1;
      readln(szam);
      end;
    writeln('A szamok atlaga ',osszeg/db:8:2);
    end.
```

Átlagszámítás while ciklussal

16. program

A szintaxis a 35. ábrán látható.



A while utasítás szintaxisa

35. ábra

A while ciklustörzsében csak egyetlen utasítás állhat. Ezen összetett utasítás képzésével segíthetünk, ahogy példánkban is tettük. A cilkustörzs mindaddig újra és újra végrehajtódik, amíg a logikai kifejezés true értékű. Ha a ciklusba való belépés előtt false az értéke, akkor a ciklustörzset egyszer sem hajtja végre a gép, a while ciklus hatástalan lesz.

4.5 A for ciklus

A legtöbb programozási nyelvben találkozunk egy sajátos ismétlési szerkezettel, amelynél a ciklustörzs egy meghatározott értéksorozat minden elemével pontosan egyszer végrehajtódik. Az azonos for kulcsszó ellenére lényeges szintaktikai és szemantikai eltéréseket tapasztalunk. A Pascalban a for utasítás meglehetősen egyszerű, amint azt a 17. program is tanusítja. A példában az értéksorozat kiíratásával mutatjuk be a ciklust.

```
program sorozat;
var i: integer;
begin
  clrscr;
  for i:=1 to 10 do
    write(i,' ',);
  writeln;
  end.
```

A for ciklus

17. program

A kiírt sorozat

1 2 3 4 5 6 7 8 9 10

mutatja, hogy a ciklustörzset ezekkel az értékekkel és ilyen sorrendben hajta végre a gép. Esetünkben a ciklus törzse az egyetlen

```
write(i,' ')
```

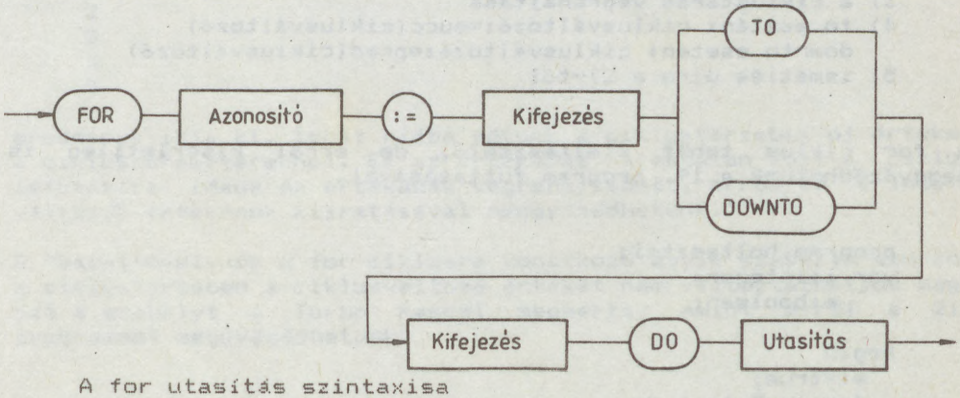
utasítás. Ha a

```
for i:=10 downto 1 do
  write(i,' ');
```

alakot használjuk, akkor a sorrend fordított lesz:

10 9 8 7 6 5 4 3 2 1.

A sorozat elemeit tartalmazó változót (itt i) ciklusváltozónak, a to (ill. downto) előtti értéket kezdőértéknek, az utánit pedig végértéknek nevezzük. A kezdő- és végérték a ciklusparaméterek. A for szintaxisát a 36. ábrán látjuk.



A for utasítás szintaxisa

36. ábra

A ciklustörzs csak egyetlen utasításból állhat. Ez a szabály most sem jelent érdembeli megszorítást, hiszen bármilyen bonyolult összetett utasítás is egyetlen utasításnak számít szintaktikus szempontból.

A ciklusváltozó és a ciklusparaméterek típusának meg kell egyezni. A valóson kívül minden eddig ismert típus alkalmazható, így pl. a 18. programnál ' betűkön futtatjuk végig a ciklusváltozót, s kiíratjuk a betű kódját.

```
program betukodok;
var c:char;
begin
  clrscr;
  for c:='A' to 'Z' do
     write(c,' kodja ',ord(c):3,'    ');
  if (ord(c) mod 4)=0 then writeln;
  writeln;
  end.
```

A betűk kódjának kiíratása

18. program

Az if utasítással vezéreljük, hogy soronként négyesével írja ki a kódokat a program. Az ord(c) után a :3 szerepe: a szám mindig 3 hely szélességben íródik ki (ha csak egyjegyű, akkor két szóköz előzi meg).

A for utasítás szemantikáját a következőképpen írhatjuk le:

1) ciklusváltozó:= kezdőérték
2) ha a ciklusváltozó> végérték, akkor vége
3) a ciklustörzs végrehajtása
4) to esetén: ciklusváltozó:=succ(ciklusváltozó)
   downto esetén: ciklusváltozó:=pred(ciklusváltozó)
5) ismétlés újra a 2)-től

A for ciklus tehát elöltesztelő, de erről kísérletileg is meggyőződhetünk a 19. program futtatásával.

```
program holtesztel;
var i:integer;
    e:boolean;

begin
  e:=true;
  for i:=3 to 1 do
     e:=false;
```

```
    if e then
       writeln('eloltesztelo')
    else
       writeln('hatultesztelo');
    end.
```

A feltételvizsgálat helye

19. program

A for működésével kapcsolatos másik kérdés, hogy a ciklusparaméterek értéke a végrehajtás során módosítható-e. Ezt eldönthetjük a 20. program futtatásával.

```
program paraméterek;
var i, kezd, veg:integer;
begin
   kezd:=1;
   veg:=5;
   for i:=kezd to veg do begin
      writeln(i);
      kezd:=4;
      veg:=10;
      end;
   end.
```

A paraméterek vizsgálata

20. program

A program az

```
1
2
3
4
5
```

eredményt írja ki, tehát hiába adtunk a ciklustörzsben új értéket a ciklusparamétereknek. Ez az értékadás hatástalan volt a ciklus lefutására. (Maga az értékadás végrehajtódott, erről az érintett változók értékének kiíratásával meggyőződhetünk.)

A Pascal nyelvnek a for ciklusra vonatkozó egyik szabálya szerint a ciklustörzsben a ciklusváltozó értékét nem változtathatjuk meg. Ezt a szabályt a Turbo Pascal megsérti, amint erről a 21. programmal meggyőződhetünk.

```
program ejnye_Borland;
var i:integer;
```

```
begin
  for i:=1 to 5 do begin
    write(i,' ');
    i:=i+1;
    end;
  writeln;
  end.
```

A ciklusváltozó kezelése

21. program

Ha lefuttatjuk a programot, semmiféle hibajelzést nem kapunk, sőt, a következő eredményt látjuk:

1 3 5 7 9.

Hm. Mit mondjak? Ez bizony nem szép!

4.6 Feladatok

1) Készítsen programot másodfokú egyenlet megoldására! Vizsgálja meg, hogy az egyenlet valóban másodfokú-e, van-e valós gyöke, hány különböző gyök van!

2) Írjon programot az

$$ax + by = p$$
$$cx + dy = q$$

lineáris egyenletrendszer megoldására. Az adatok: a, b, c, d, p, q. Ha lehet, számítsa ki x és y értékét, ha nem, döntse el, hogy az egyenletek ellentmondóak-e vagy nem függetlenek.

3) Készítsen programot a jövedelemadó kiszámítására. Adatként az adóköteles jövedelem összegét kell megadni, az adó sávosan számítandó:

jövedelem	adókulcs
0 - 55000	0%
55001 - 70000	17%
70001 - 100000	23%
100001 - 150000	29%
150001 - 240000	35%
240001 - 360000	42%
360001 - 600000	49%
600001 -	56%

4) Legyen 3 valós szám az adat. Ha e számok lehetnek egy háromszög oldalainak hosszúságai, akkor számítsa ki a kerületet és a területet. A terület kiszámítására a Hérón-képletet használhatja:

$$T=\sqrt{s(s-a)(s-b)(s-c)},$$

ahol a, b, c a háromszög oldalhosszai, s a kerület fele.

5) Állapítsa meg (futtatás nélkül), hogy mit csinál a következő program:

```
program vicces;
var sor, i: integer;
begin
  for sor:=1 to 3 do begin
    writeln;
    for i:=sor downto 0 do
      case i of
        0: writeln('nem ugat hiaba');
        1: writeln('harom kutya');
        2: writeln('ket kutya');
        3: writeln('egy kutya');
        end;
    end;
  end.
```

6) Könnyű belátni, hogy a

$$2^x - x - 10 = 0$$

egyenletnek valahol 3 és 4 között van gyöke, ugyanis

$$2^3 - 13 = -5, \text{ de } 2^4 - 14 = 2.$$

Írjon programot, amely megkeresi a gyököt, az azt tartalmazó intervallum ismételt felezéseivel. Akkor álljon le a program, ha a gyököt tartalmazó algoritmus hossza kisebb 0.001-nél.

7) Írjon programot, amely kiszámítja az

$$\left(1 + \frac{1}{n}\right)^n$$

sorozat k-adik elemét. k értékét adatbevitellel határozzuk meg.

8) Egy n természetes szám faktoriálisát n! jelöli és az 1*2*3*...*n szorzatot értjük alatta. Készítsen programot n!

kiszámítására. Vigyázzon, n! értéke nagyon gyorsan nő az n függvényében!

9) Készítsen programot, amely 0.1 lépésközzel elkészíti az

$$\frac{1}{\sin x+\cos x}$$

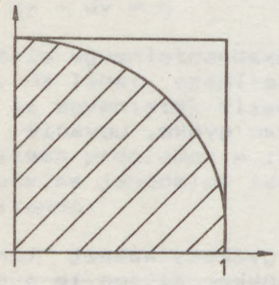
függvény értéktáblázatát a 0..1.5 intervallumban.

10) Az OTP-től felvett lakásépítési kölcsönét mennyi idő alatt törleszti adott éves kamat és adott havi részlet mellett? Feltételezheti, hogy a kamatot havonta számolják az aktuális adósságállomány után.

11) Mekkorának kell választani a havi törlesztőrészletet, ha lakásépítési kölcsönét adott éves kamat mellett n év alatt fizeti vissza?

12) Dolgozatok eredményéről statisztikát kell készíteni. Készítsen programot, amely összeszámolja az 1-es, 2-es stb. dolgozatokat és átlagot is számol.

13) A random függvény segítségével állítson elő véletlen (x,y) koordinátákat, ahol mindegyik szám a 0..1 intervallomba esik. Határozza meg, hogy n ilyen számpár közül hány esik az egységsugarú kör negyedébe (37. ábra).



37. ábra

Hogyan tudná ebből a pi közelítő értékét meghatározni? (A találati valószínűség nyilván a területekkel arányos.)

14) Szimuláljon kockadobásokat az 1..6 intervalumbeli véletlen egész számokkal! Egyszerre két kockával dobva milyen valószínűséggel dobunk 2-t, 3-at, 4-et, ..., 12-t? Nagy számú dobásból a valószínűség közelítőleg megállapítható. Pl. a 2-es dobás valószínűsége = a 2-es dobások száma/az összes dobás száma.

15) A Fibonacci-sorozat képzési szabálya a következő:

$$a_1 = 1,\ a_2 = 1,\ \ldots,\ a_{i+2} = a_{i+1} + a_i$$

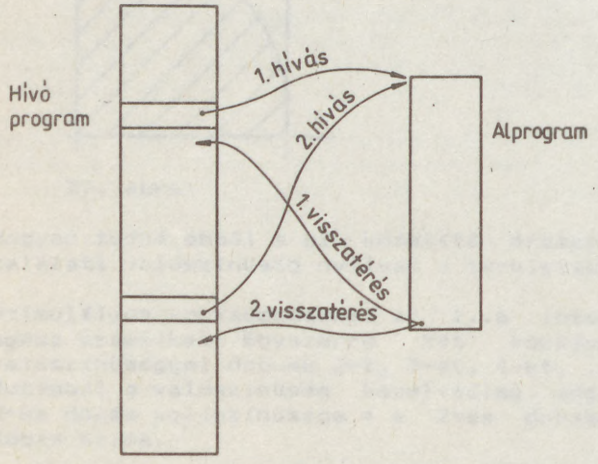
Írjon programot az első n elem kiszámítására!

5. ALPROGRAMOK

A legbonyolultabb programokat is felépíthetjük elemi tevékenységekből, sokszorosan összetett szerkezetekkel. Az ilyen programok azonban nem áttekinthetők, nehezen érthető a működésük. Hibátlanul elkészíteni és utólagosan módosítani egy néhány ezer utasításos programot szinte képtelenség. Világosabbá és rövidebbé tehetjük programunkat, ha bizonyos résztevékenységeket elemi tevékenységként kezelünk. Erre nyújtanak lehetőséget az alprogramok.

Az alprogramok névvel ellátott összetett tevékenységek, amelyeket az elemi tevékenységekhez hasonlóan használunk. Ez azt jelenti, hogy bár nyelvi szinten az alprogramokban megfogalmazott tevékenységek nem elemiek, a felhasználó számára - aki készen kapja vagy magának gyártja ezeket - használat közben elemi tevékenységeknek tűnnek.

Az alprogram egy jól meghatározott tevékenység önálló meghatározott programja, amit egy másik programon belül egyetlen utasításként írunk le. Ezen utasítás hatására az alprogram teljes egészében végrehajtódik, majd fut tovább a másik program. Az alprogramnak egy másik programban való aktivizálását az alprogram hívásának nevezzük. Az alprogramok önmagukban nem is használhatók, mindig szükség van egy másik programra, amelyik aktivizálja őket. Egy alprogramot aktivizáló programot gyakran nevezzük meghajtó programnak. Az alprogram tevékenységének befejeződése után a hívó program mindig az aktuális hívó utasítást közvetlenül követő utasítás végrehajtásával folytatódik. Az alprogram és meghajtója közötti vezérlésátadás-visszatérés mechanizmusát a 38. ábrán szemléltetjük.



A meghajtó és az alprogram kapcsolata

38. ábra

Azok a programok, amelyek másik programon belül nem használhatók, csak önállóan, főprogramok. Az eddigi programjaink valamennyien főprogramok voltak.

Nemcsak főprogram hívhat alrogramokat, hanem alprogram is, sőt, arra is látunk majd példát, hogy egy alprogram önmagát hívja.

Az alprogramok hívása általában nemcsak a 38. ábrán szemléltetett vezérlésátadási mechanizmust foglalja magában. A meghajtó program a hívással egyidőben információt is átadhat az alprogramnak, amit az alprogram feldolgoz, majd visszatéréskor visszaadja a hívó programnak. Az információátadást paraméterekkel valósítjuk meg.

Kétféle alprogramot használhatunk, eljárásokat és függvényeket.

5.1 Eljárások

Az általánosabb értelemben vett alprogramok az eljárások, feladatuk egy meghatározott - és önmagában is értelmes - tevékenység elvégzése. Két alapvető dolgot kell elmondani,

1) hogyan definiáljuk,
2) hogyan használjuk

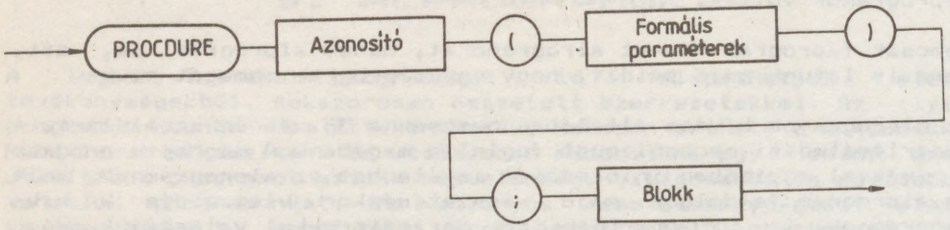
az eljárásokat.

Egyszerűbb esetben - s a Turbo Pascal 3.0-s verziójánál lényegében más lehetőség alig van - az alprogramokat egy főprogram blokkjában, a deklarációs részben a változók deklarálása után definiáljuk. A definíció az eljárásfejjel kezdődik.
Az eljárásfejet a procedure kulcsszó vezeti be, majd az eljárás neve, utána - ha vannak - zárójelben a paraméterek leírása következik, amelyek az eljárás és a hívó program információs kapcsolatát határozzák meg. A definícióban szereplő paramétereket formális paramétereknek nevezzük. A formális paraméterek megfelelő specifikációjával adjuk meg, hogy az eljárásnak milyen információkat kell kapnia a hívó programtól a híváskor és milyen értékeket juttat vissza a főprogramnak, ha elvégezte feladatát.
Az eljárásfej után az eljárásblokk következik, amely tartalmazhat deklarációkat (akár alprogram definíciókat is), majd begin és end között a tevékenység programja (ez az eljárástörzs). A 39. ábrán az eljárás szintaxisát mutatjuk be.

Az eljárásblokk és programblokk szintaktikusan teljesen azonos nyelvi szerkezet. Ezért a továbbiakban néha egyszerűen blokkról beszélünk.
Az eljárásfej ismerete elegendő, hogy az eljárást használni tudjuk. Az eljárás hívásakor ugyanis a nevét adjuk meg, és a formális paraméterek helyébe főprogrambeli objektumokat, aktuális paramétereket helyettesítünk:

```
eljárásnév(aktuális paraméterek);
```

az eljárásnévből és az aktuális paraméterekből álló utasítást eljárásutasításnak nevezzük.



Az eljárásdefiníció szintaxisa

39. ábra

Ezek után tekintsünk egy egyszerű példát. A 22. program egy eljárás definícióját mutatja be a meghajtó programmal együtt. A "jelsor" eljárás a paraméterként megadott karakterből egy sornyit kiír (pontosabban 70-et). összetettebb programokban célszerű kommentárokat használni. Ezek a {, } közé írt magyarázatok megkönnyítik a program olvasását.

```
program jelhajto;
  var ch: char;

  procedure jelsor(x:char);
    const jelszam=70;
    var i: integer;
    begin
      for i:=1 to jelszam do
        write(x);
      writeln;
    end; {a jelsor eljaras vege}

  begin {a foprogram torzse}
    clrscr;
    jelsor('?');
    writeln;
    for ch:='a' to 'f' do
      jelsor(ch);
  end.
```

Eljárás és hívása

22. program

Próbálja ki a programot! Módosítsa az egy sorba kiírt jelek számát!
A 22. programban az eljárás formális paramétere a char típusú x érték. Ez azt jelenti, hogy a főprogram eljárás utasításában valamilyen konkrét karakter típusú értéket kellett a helyébe írni, ahogy azt tettük.

Amilyen jel az aktuális paraméter, olyan jelekből álló sort kapunk a képernyőn a hívás eredményeként.
Eljárásunkat könnyen továbbfejleszthetjük, hogy a kiírt jelek számát is szabadon változtathassuk. A jelszam értékét is paraméternek választhatjuk, természetesen ekkor ellenőrizni is kell, hogy értéke elfogadható legyen (23. program).

```
program massorhajto;
  var ch:char

  procedure massor(x:char;hossz:integer);
    const maxhossz=80;
          minhossz=1;
    var i: integer;
    begin
      if hossz<minhossz then
        hossz:=minhossz;
      if hossz>maxhossz then
        hossz:=maxhossz;
      for i:=1 to hossz do
        write(x);
      writeln;
    end; {massor vege}

  begin {foprogram}
    massor('*',27);
    writeln;
    for ch:='1' to '9' do
      massor(ch,ord(ch)-ord(0));
  end.
```

Eljárás két paraméterrel

23. program

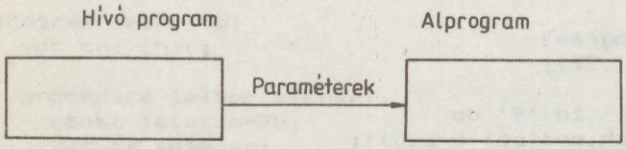
A massor második hívásánál a hossz paraméter értéke mindig a kiírt számjegy értékével azonos. A kódrendszernek azt a tulajdonságát használtuk fel, hogy a számjegyek kódjai a számjegyeknek megfelelő sorrendben egymás után következő értékek. Ez alapkövetelmény minden - Pascalban használatos - kódrendszerrel szemben. Azt sem kellett tudnunk, hogy a '0'-nak mi a kódja. Ennek az eljárásnak a működése nem függ a konkrét kódrendszertől, más körülmények között is használható (hordozható) alprogram.

Nincs szükségképpen minden eljárásnak paramétere. A paraméter nélküli eljárások hívásánál a zárójelet is elhagyjuk. Az eljárásutasításban pontosan annyi aktuális paraméternek kell szerepelni, ahány formális paramétert az eljárásfejben előírtunk. Az eljárásfejet nagyon gondosan kell kialakítani, a paraméterezés határozza meg az eljárás más programokhoz való hozzákapcsolásának lehetőségét.

Az eljárások ugyanúgy épülnek fel, mint a programok, ugyanúgy definiálhatunk bennük konstansokat, változókat, de még alprogramokat is.

5.2 Értékparaméterek

Amikor a formális paraméterek helyébe behelyettesítjük az aktuális paramétereket, ügyelni kell arra, hogy a helyettesítést a formális paraméter típusának megfelelően végezzük el. Ha egy formális paraméter integer típusú, nem tudunk real típusú értéket átadni vele. A bájt és integer típusok itt is kompatíbilisek. Írhatunk bájt típusú formális paraméter helyébe integernek deklarált változót, csak az alsó bájtban (LSB) lévő értéket adja át.
A jelsor és a massor eljárások a főprogramtól átvesznek értékeket, de vissza nem adnak semmit. Az olyan paraméterátadási mechanizmust, amely csak az alprogram felé közvetít értéket, érték szerinti paraméterátadásnak nevezzük. Érték szerinti paraméterátadást értékparaméterekkel valósítunk meg.



Érték szerinti paraméterátadás

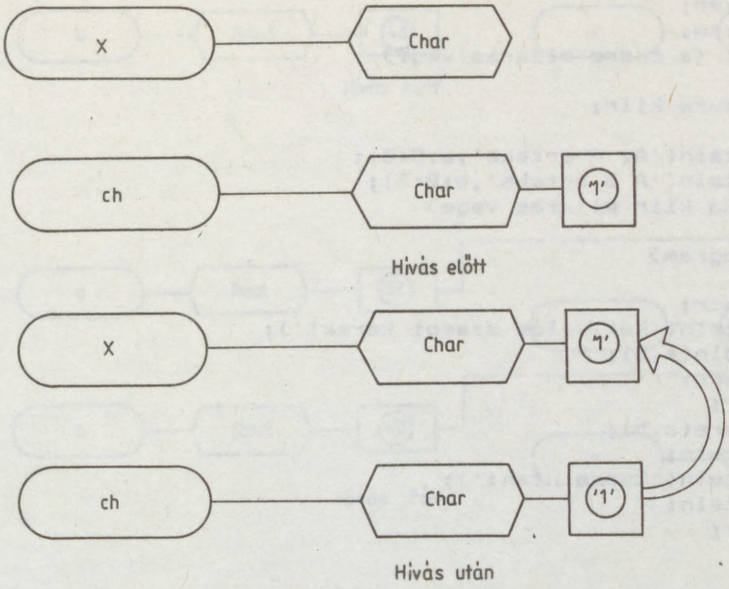
40. ábra

Az értékparaméterek helyébe aktuális paraméterként kifejezést írhatunk (speciálisan nyilván konstans, ill. változó is megfelel).

A jelsor eljárás első hívásakor a '?' konstans az aktuális paraméter, a következő hívásoknál a ch nevű változó. A változók szemléltetésére bevezetett rajzokkal követhetjük nyomon az érték szerinti paraméterátadást a 41. ábrán.

Az eljáráshívás előtt nem tartozik a formális paraméterekhez tárterület. Híváskor a jellemzőinek megfelelő tárterület kapcsolódik hozzájuk, s az aktuális paraméter értéke ide átmásolódik. Ezután az alprogram minden kapcsolatát elveszíti az aktuális paraméterrel. Akárhogy változtatja meg végrehajtás közben a formális paraméter értékét, ennek az aktuális paraméterre semmi hatása sincs. Ugyanakkor a főprogram semmi módon soha nem is szerezhet tudomást az alprogramban deklarált változók (az alprogram saját változóinak) értékéről. Ezeket a változókat használja feladatának végrehajtása során, amikor visszaadódik a vezérlés a főprogramba, ezek a változók megszűnnek létezni. Olyan ez, mint amikor egy feladat megoldása közben

valamilyen kalkulációt egy papírfecnin végzünk el, majd összegyűrve a szemétkosárba dobjuk.



Az értékparaméter átvétele

41. ábra

Figyeljük meg, hogy az érték szerinti paraméterátadás során az átadott érték helyigénye a tárban megduplázódik, mert két különböző helyen is tárolódik.

5.3 Változóparaméterek

Jóllehet az eljárások feladata egy tevékenység megvalósítása, nem kizárt, hogy ez a tevékenység egy vagy több érték létrehozására irányul. Ilyen esetben meg kell oldani a kiszámított értékek visszaadását.
A változóparaméterek a főprogram és az alprogram között mindkét irányú paraméterátadásra alkalmasak. A változóparaméterekkel történő paraméterátadást referencia szerinti átadásnak nevezzük.

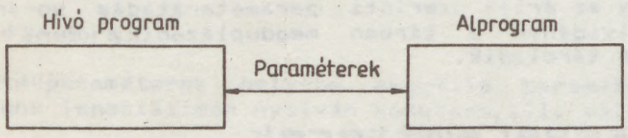
A 24. programban definiált csere eljárásnál példát látunk változóparaméterek alkalmazására.

```
program cserel
  var a,b: real;

  procedure rcsere(var x,y:real);
```

```
    var w: real;
    begin
      w:=a;
      a:=b;
      b:=w;
    end; {a csere eljaras vege}

  procedure kiir;
  begin
    writeln('Az A erteke',a:8:3);
    writeln('A B erteke',b:8:3);
  end; {a kiir eljaras vege}

  {foprogram}
  begin
    clrscr;
    writeln('Ket valos szamot kerek!');
    readln(a,b);
    clrscr;
    kiir;
    rcsere(a,b);
    writeln;
    writeln('Csere utan:');
    writeln;
    kiir;
  end.
```

Változók értékének felcserélése

24. program

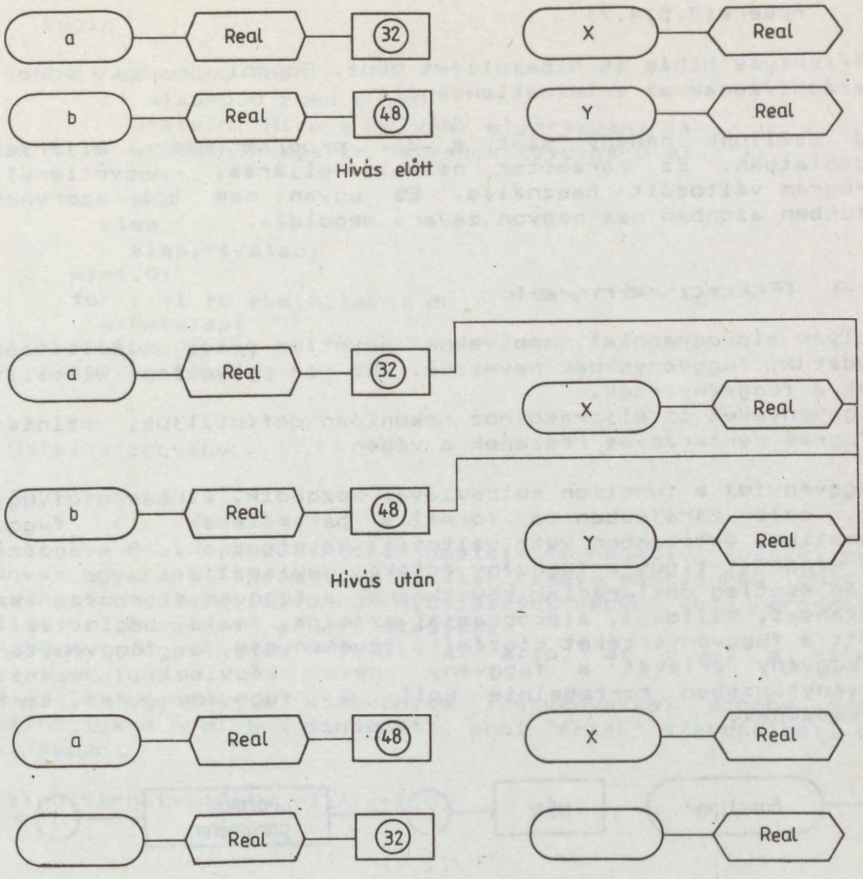


Referencia szerinti paraméterátadás

Referencia szerinti paraméterátadás

42. ábra

Az rcsere eljárás két valós változó értékét cseréli fel. Az "x", "y" formális paraméterek itt változóparaméterek, amit az eléjük írt var jelez. A változóparaméterek helyére csak változók azonosítóit írhatjuk aktuális paraméterként (sem konstanst, sem kifejezést!), s az eljárás tulajdonképpen ezekhez a (főprogrambeli) változókhoz tartozó tárolóhelyekkel dolgozik. Minden, amit az eljárástörzsben a formális paraméterekkel megfogalmazunk, híváskor az aktualis paraméterekkel történik meg. Kövessük nyomon a paraméterátadás folyamatát rajzban (43. ábra).



A változóparaméter átadási mechanizmusa

43. ábra

A referenciaátadás kifejezésben a referencia tulajdonképp a tárolóhelyre való hivatkozást jelenti. A formális és aktuális paraméterek ugyanazon a tárolóhelyen osztoznak. Ez azt is jelenti, hogy ugyanazon tárterülethez különböző neveken is hozzáférünk. Ezt a jelenséget a programozásban álnévképződésnek (aliasing) nevezzük. A szabványos Pascalban csak a változóparaméterek használatakor képződhet álnév, a Turbo Pascalban erre külön nyelvi eszközt is találunk.

Vigyázat!
Változóparaméter helyébe nem írhatunk értéket (konstanst vagy kifejezést). Az

```
csere(3.5,4.7)
```

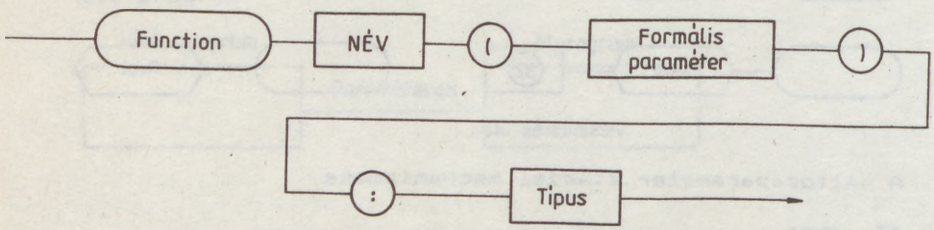
eljáráshívás hibás és hibajelzést okoz. Gondoljuk végig ennek az eljáráshívásnak az értelmetlenségét!

Hadd szóljunk néhány szót a 23. program kiir eljárásával kapcsolatban. Ez paraméter nélküli eljárás, közvetlenül a főprogram változóit használja. Ez ugyan nem túl szerencsés, esetükben azonban nem nagyon zavaró megoldás.

5.4 Függvények

Az olyan alprogramokat, amelyeknek egyetlen érték előállítása a feladatuk, függvényeknek nevezzük. Ez az egyetlen előállított érték a függvényérték.
A függvényeket az eljárásokhoz hasonlóan definiáljuk, szintén a főprogram deklarációs részének a végén.

A függvényfej a function kulcsszóval kezdődik, ezután a függvény neve, majd zárójelben a formális paraméterek (a függvény matematikai értelemben vett változói) következnek. A végzárójel után megadott típus a függvény értékét deklarálja.
Ezután esetleg deklarációk következnek a függvényalprogram saját konstansai, változói, alprogramjai számára, majd begin és end között a függvényértéket kiszámító tevékenység, a függvénytörzs.
A függvény értékét a függvény neve képviseli, ezért a függvénytörzsben szerepelnie kell a függvénynévre történő értékadásnak.



A függvénydefiníció szintaxisa

44. ábra

Az eljáráshívástól eltérően a függvényhívás nem utasítás, hanem érték.
A 25. program valós számok egészkitevős hatványainak kiszámítására mutat be egy függvényt.

```
function rhatvanyi(alap: real;
                   kitevo:integer):real;
var w:real;
    i:integer;
```

```
begin

  if kitevo<0 then
    if alap=0.0 then begin
      writeln('Hiba a HATVANY eljarasban:');
      writeln('Negativ hatvanykitevo es 0 alap');
      halt;
      end
    else
      alap:=1/alap;
  w:=1.0;
  for i:=1 to abs(kitevo) do
    w:=w*alap;
  rhatvanyi:=w;
end; {rhatvanyi}
```

Hatványfüggvény

25. program

Az eljárás a real típusú értékek pozitív és negatív egészkitevős hatványát egyaránt kiszámítja. Negatív kitevő esetén nem lehet 0 a hatványalap értéke. Ilyenkor hibajelzést kapunk és a programot megállítjuk. Erre való a halt utasítás.
Az rhatvanyi függvénynév r-jével az alap, az i-vel a kitevő típusára utalunk.
A definiált függvényt a szabványos függvényekkel azonos módon használhatjuk a nyelvben mindenütt, ahol "érték" szerepelhet, pl. értékadásban:

```
z:=3.9*rhatvanyi(x,-3)/(x+1.0)
```

5.5 Az include direktíva

Elkészítettünk egy alprogramot, amit minden bizonnyal több programban is használhatunk. Csak az a baj, hogy az alprogram itt van a könyvben, majdani programjaink pedig a mágneslemezre kerülnek az Editor jóvoltából.

Persze megtehetjük, hogy a könyvet olyan helyre tesszük, hogy mindig kéznél legyen és mindig bemásoljuk - ha kell - a deklarációs részbe. Ez nem tól csábító megoldás - nem beszélve arról, hogy olyan, mintha valaki lovakat fogna be a Rolls Roys-a elé.

Jobb megoldás, ha az Editor blokkparancsait használjuk. Az alprogramot felírjuk lemezre és szerkesztés közben a [CTRL] [K], [CTRL] [R] parancsokkal beolvasva beszórjuk a programunk megfelelő helyére. Ezzel a másolást nem magunk végezzük el, hanem a számítógépre bízzuk.
De ekkor is ott van a már régen elkészített és ezerszer használt

alprogram szövege az esetleg egyébként is hosszó programunkéban, foglalja a szerkesztő tárterületét és nehezíti a program áttekintését a képernyőn.

Igazibb megoldás is van. Nem kell sem kézzel, sem az Editor segítségével a főprogramba másolni az alprogramot. Elegendő megmondani az állomány nevét, ahogy a lemezen található és azt, hogy hová kell beilleszteni. A beillesztést nem az editor, hanem majd a fordítóprogram fogja fordítás közben elvégezni.

Erre az I (include) compiler direktívát használjuk.
A program megfelelő helyére a

```
{$I állománynév}
```

fordítóprogramnak szóló utasítást kell elhelyeznünk és gondoskodni arról, hogy fordításkor az állomány valóban kéznél legyen. Ha akadálya van, hogy az aktív meghajtóban legyen a kérdéses állomány, állománynév helyett elérési útat is megadhatunk.
Tegyük fel, hogy az rhatvanyi alprogram írásakor a hatvany.pas munkaállomány nevet választottuk, így a függvény ilyen néven található a könyvtárban. A 26. program ekkor minden további nélkül lefordítható és végrehajtható.

```
program hatvanyok;
  var k: integer;
      r: real;

{$I hatvany.pas}

  begin
    clrscr;
    r:=0.0:
    writeln('H A T V A N Y O K');
    writeln;
    write('kitevo:          ');
    for i:=2 to 7 do
      write(k:10);
    writeln;
    writeln;
    repeat
      for k:=1 to 7 do
         write(ihatvanyr(r,k):10:4);
      writeln;
      r:=r+0.01;
    until r>=3.0;
  end.
```

Alprogram használata include állományból

26. program

A program a 0..3 intervallumbeli valós számok pozitív egészkitevős hatványait számolja 2-től 7-ig, 0.01-es lépésközzel. Mivel a valós számokat csak közelítőleg ábrázolja a gép, az until feltételvizsgálatba nem írhatunk "="-t, mert esetleg végtelen ciklusba kerül a program.

Az include direktíva használatára alapozva létrehozhatunk alprogram könyvtárakat, ahonnan nemcsak a szükséges eljárások és függvények, de bizonyos közös konstans éa változó deklarációk is beszerkeszthetők fordítás közben különböző programokba. Ez nagyon hasznos dolog. Még jobb volna, ha ezeket az alprogramokat nem kellene újra és újra mindig lefordítani, minden újabb felhasználáskor. Ha egyszer lefordítanánk, és lefordított állapotban tárolnánk a lemezen, és csak hozzákapcsolnánk a mindenkori főprogramhoz annak lefordítása után.

Ilyen megoldás a Turbo Pascal 4.0-ás verziójától kezdve az ún. UNIT-ok használatával lehetséges, de a 3.0-ásban még nem. Más Pascal megvalósításokban definiálhatunk külső alprogramokat, és ezzel a nyelvi mechanizmussal szeparáltan lefordított eljárások használhatók, - sőt, sok esetben még más programnyelven megírt és lefordított alprogramokat is felhasználhatunk. Ehhez a főprogramban csak az eljárásfejet kell megadni, a blokkot az EXTERNAL kulcsszóval kell helyettesíteni:

```
procedure FORTRANprog(var i,j:integer);
  external;
```

A Turbo Pascalban is használhatók külső alprogramok, de csak olyanok, amelyeket assemblerben írunk meg. Ilyenkor az alprogram definíciójában a neve mellett az EXTERNAL kulcsszó után meg kell adni az állománynevet is:

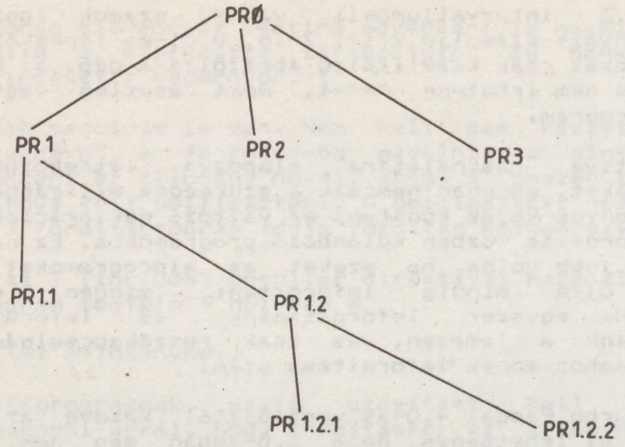
```
Procedure Plot(x,y:integer);
  external 'PLOT';
```

Ezután a gépi kódú alprogramot ugyanúgy használhatjuk programunkban, mintha Pascalban írtuk volna meg.

Mivel a gépi kódú programozás rejtelmeinek ismertetése messze meghaladja könyvünk kereteit, így külső alprogramokat nem fogunk készíteni.

5.6 A program szerkezete

A bonyolult programok szerkezete is áttekinthetővé válik, ha célszerűen definiált alprogramok segítségével építjük fel.
A program tervezésekor a programozandó tevékenységet nem bontjuk le közvetlenül elemi tevékenységekre, hanem magasabb szintű tevékenységekből állítjuk össze. A következő lépésben ezeket a magasabb szintű tevékenységeket bontjuk tovább, s esetleg csak sok lépésben jutunk az elemi tevékenységekig. A 45. ábrán egy ilyen program tervezési folyamatát mutatjuk be.



Hierarchikus programépítés

45. ábra

Az ábra nemcsak a tervezés folyamatát, a kész program szerkezetét is elárulja. A főprogramban 3 alprogramot definiálunk, a pr1 alprogramnak két alprogramja van, a pr1 alprogram pr12 alprogramjában további két alprogramot kell deklarálni.
A pr11, pr121, pr122, pr2 és pr3 alprogramokat már elemi tevékenységek segítségével kell megfogalmazni (a szabványos eljárásokat és függvényeket, mint pl. a sin(x), elemi tevékenységeknek tekintjük).
A program szövegének szerkezetét a 46. ábra mutatja.

Kérdés, hogy ilyen - egyáltalán nem rendkívüli - szerkezetű program esetben egy adott pontján milyen alprogramok és milyen változók érhetők el. Az erre vonatkozó nyelvi szabályokat nevezzük hatásköri szabályoknak.
Először is meg kell különböztetnünk lokális és globális objektumokat.

Globális objektumok

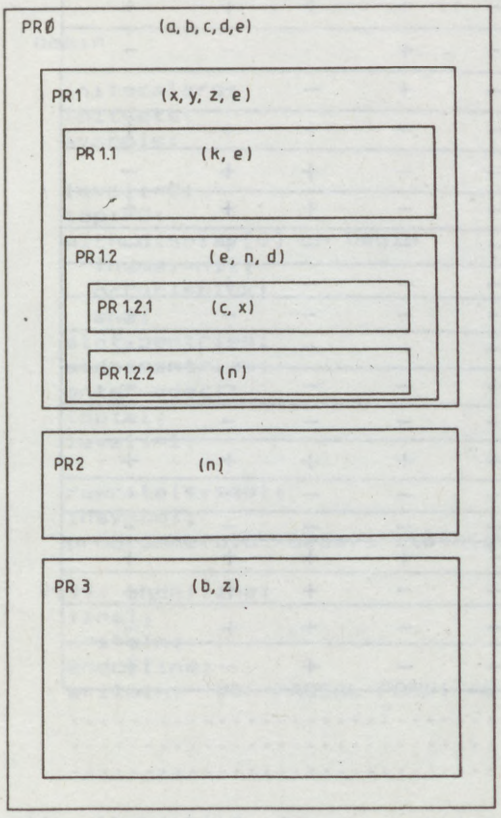
(változók, konstansok, alprogramok) azok, amelyeket a főprogramban deklaráltunk. Ezek a program bármely pontján hozzáférhetőek, kivéve, ha valamely alprogramban ugyanazt a nevet még egyszer deklaráljuk. Ez esetben ebben és csakis ebben az alprogramban (valamint az ezen alprogramban deklarált alprogramokban) csak az új objektum férhető hozzá.

Lokális objektumok

az alprogramokban deklarált objektumok.

A 24. programban az a és b globális változók, az x, az y és a w pedig lokálisak a csere eljárásra nézve. A csere és a kiir

eljárásnevek szintén globálisak. Jegyezzük meg, hogy a formális paraméterek mindig lokálisak.
A Pascal program blokkos szerkezetét bemutató ábrán nyomonkövethetjük a változók hatáskörét. Az egyes blokkok elején feltüntettük az ott deklarált azonosítókat.
A hatásköri szabályok nagyon célszerűek, sok munka alól mentesítenek bennünket pl. akkor, ha forráskönyvtárból include-dal szerkesztünk a programunkhoz kész alprogramokat. Mivel az objektumok szempontjából minden alprogram külön világ, nem kell tudnunk, hogy belül milyen nevek szerepelnek. Az előfordulhat, hogy az alprogram használatához szükség van valamely globális objektumra, de ez a főprogramban figyelembe vehető - természetesen ilyen körülményekre az alprogram használati utasításában ki kell térni. Minél kevesebb külön információt kell tudni egy alprogramról a paraméterezésén kívül, annál egyszerűbb a használata.



A program blokkstrukturája

46. ábra

	Blokk							
Név	PRØ	PR1	PR2	PR3	PR1.1	PR1.2	PR1.2.1	PR1.2.2
a	+	+	+	+	+	+	+	+
b	+	–	+	+	+	+	+	+
c	+	+	+	+	+	+	–	+
d	+	+	+	+	+	–	–	–
e	+	–	+	+	–	–	–	–
x	–	+	–	–	+	+	–	+
y	–	+	–	–	+	+	+	+
z	–	+	–	–	–	+	+	+
e1	–	+	–	–	+	+	+	+
k	–	–	–	–	+	–	–	–
Z11	–	–	–	–	+	–	–	–
l	–	–	–	–	–	+	+	+
u12	–	–	–	–	–	+	+	–
d12	–	–	–	–	–	+	+	+
c121	–	–	–	–	–	–	+	–
X121	–	–	–	–	–	–	+	–
U122	–	–	–	–	–	–	–	+
U	–	–	+	–	–	–	–	–
b3	–	–	–	+	–	–	–	–
Z3	–	–	–	+	–	–	–	–
PR1	+	+	+	+	+	+	+	+
PR2	+	–	+	–	–	–	–	–
PR3	+	–	–	–	–	–	–	–
PR11	–	+	–	–	+	+	+	+
PR12	–	+	–	–	–	+	+	+
PR121	–	–	–	–	–	+	+	–
PR122	–	–	–	–	–	+	–	+

A nevek hatásköre

47. ábra

A 47. ábra táblázata mutatja a nevek hatáskörét abban a

képzeletbeli programban, amelynek szerkezetét a 46. ábrán vázoltuk. A több blokkban is deklarált neveket indexeléssel különböztetjük meg.
A nagyméretű programok elkészítése elképzelhetetlen eljárások nélkül. A programozásmódszertani kutatások szerint a programok megvalósítási időszükséglete, költsége a program hosszának exponenciális függvényeként növekszik. Ez a törvényszerűség egyszerűen kizárja a nagy programrendszerek elkészítésének lehetőségét. De csak a monolit (egyetlen tömbben megírt, eljárások nélküli) programok esetében. Az is bizonyított, hogy alprogramok alkalmazásával, a program hierarchikus felépítésével az idő- és költségfüggvény csaknem lineárissá szelidíthető. A 27. program egy Pascal fordítóprogram, pontosabban a főprogramjának egy részlete. (Az R 40-es számítógépen használatos Pascal 8000 Pascal nyelven megírt fordítóprogramjáról van szó.) Ez a program több mint 6000 utasítást tartalmaz és 257 alprogramból áll – a főprogram már egyszerű.

```
begin

  initscalars;
  initsets;
  symbols;

  level:=0;
  top:=0;
  with display[0] do begin
    fname:=nil;
    occur:=blck;
    end;
  stdtypentries;
  stdnamentries;
  enterundecl;
  top:=1;
  level:=1;

  rewrite(sysqo);
  insymbol;
  programme(blockbegsys+statbegsys-[casesy]);

9999: endofline;
  final;
  writeln;
  endofline;
  writeln(' **A PASCAL FORDITAS BEFEJEZODOTT**');
  ....................................
  ....................................
  ....................................
```

Egy tipikus főprogram

27. program

Még egyetlen dologról nem szóltunk a hatáskört illetően. Minden objektum hatásköre a deklarációjával kezdődik. Ezért, ha pl. egy alfa nevű alprogram felhasználja a beta alprogramot (az eljárástörzsben előfordul a beta eljárásutasítás), akkor a betát az alfa előtt kell definiálni.

5.7 Rekurzió

Egy egyszerű példával mutatjuk be a rekurzió lényegét.
Az n természetes szám n! faktoriálisát a következőképpen definiáljuk:

$$n! = \begin{cases} 1, & \text{ha } n=0 \\ n*(n-1)!, & \text{ha } n>0 \end{cases}$$

Ez a definíció rekurzív, mert a faktoriális fogalmának meghatározásához a faktoriális fogalmát is felhasználtuk. Ugyanakkor világosan kell látnunk, hogy ez a definíció mégsem semmitmondó. Hiszen, ha 3! értékére vagyunk kíváncsiak, akkor a definíció szerint:

```
3! = 3*2! =
     3*2*1! =
     3*2*1*0! =  (0!=1)
     3*2*1*1 = 6.
```

A rekurzív definíció mindig két részből áll:

alapesetek konkrét meghatározásából
és

redukciós szabályokból,

amelyek az általános esetet egyszerűbbekre vezetik vissza. Az n!-nál csak egyetlen alapesetet és egyetlen redukciós szabályt kellett használnunk. Mivel a redukciós szabály a definiálandó fogalmat használja, egy rekurzív függvény vagy eljárás önmagát hívja. Első példaként írjuk fel az n! értékét kiszámító függvény programját (28. program).

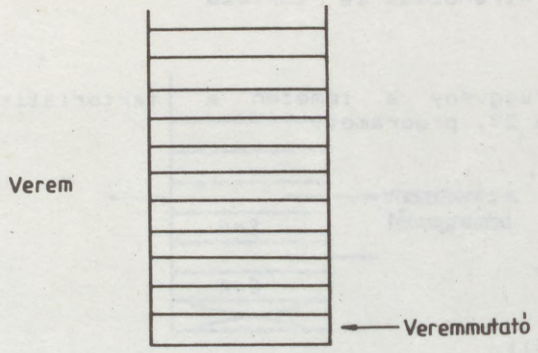
```
function faktorialis(n:integer):integer;
  begin
    if n=0 then
      faktorialis:=1
    else
      faktorialis:=n*faktorialis(n-1);
  end;
```

Rekurzív függvény

28. program

A program utasításai pontosan a rekurzív definíciót fogalmazzák meg.

A rekurzív alprogramok működésének megértéséhez többet kell tudni arról, hogyan kezeli a nyelv az eljárásokat. A lokális változókat egy ún. veremtárban helyezi el. A verem olyan tárolási szerkezet, amelyből mindig a legutoljára betett érték olvasható ki először (ezért a "Last In, First Out" kezdőbetűiből LIFO tárnak is nevezik). A kiolvasható értéket a veremmutató ("teteje") jelzi. Ha a verem üres, akkor a veremmutató a verem aljára mutat (48. ábra).



Veremtár

48. ábra

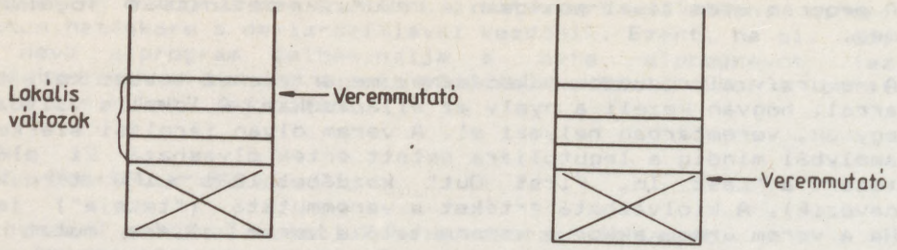
A vermet két tevékenységgel kezeljük:

1) a verem tetejére új értéket tehetünk (push down),

2) a verem legfelső elemét kiemelhetjük (pop up).

Alprogram hívásakor a hívott alprogram lokális változói a verem tetejére íródnak (a visszatérési címmel együtt). Amíg az alprogram működik, ezek a változók végig a verem tetején vannak, kilépéskor a veremmutató visszalép a korábbi helyére (a lokális változókhoz ezért nem lehet többé hozzáférni). A verem állapotát az alprogramba való belépéskor és kilépéskor a 49. ábrán láthatjuk.

A rekurzió ugyanezt a nyelvi mechanizmust használja. Amikor az alprogram meghívja önmagát, minden alkalommal vermeli a lokális változókat és a visszatérési címet.



A lokális változók létrehozása és törlése

49. ábra

Legyen a faktorialis függvény a lemezen a faktorialis.pas eljárásban és tekintsük a 29. programot.

```
program fakt3;
var k:integer;

{$I faktorialis.pas}

  begin
    k:=faktorialis(3);
    writeln('3!=',k);
  end.
```

A rekurzív függvény hívása

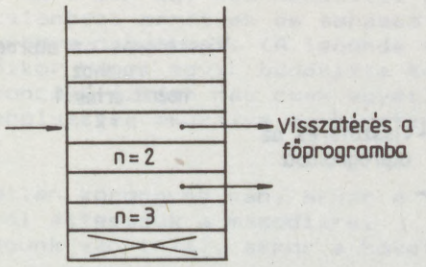
29. program

Kövessük nyomon a faktoriális hívásait a 50. ábra alapján!

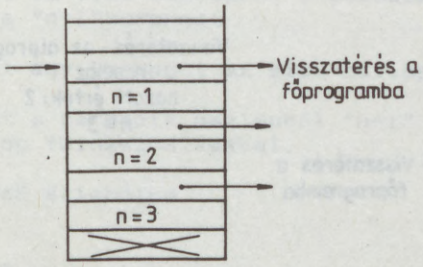
A faktoriálisok kiszámítását csak egyszerűsége és követhetősége miatt választottuk példának. Az ilyen feladatokat hatékonyabb ciklusokkal megoldani.

A rekurziót elsősorban nehéz problémák elegáns megoldására használhatjuk, mint pl. a "Hanoi tornyai" közismert rejtvény. Ennek lényege az, hogy három oszlop közül az elsőre n db korongot fűzünk fel (51. ábra).

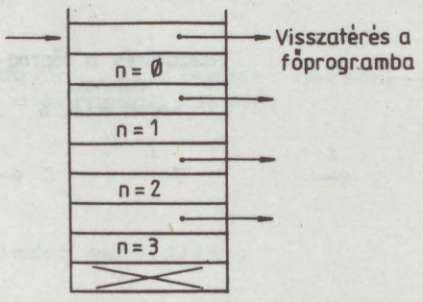
Hívás a főprogramból
faktoriális (3)



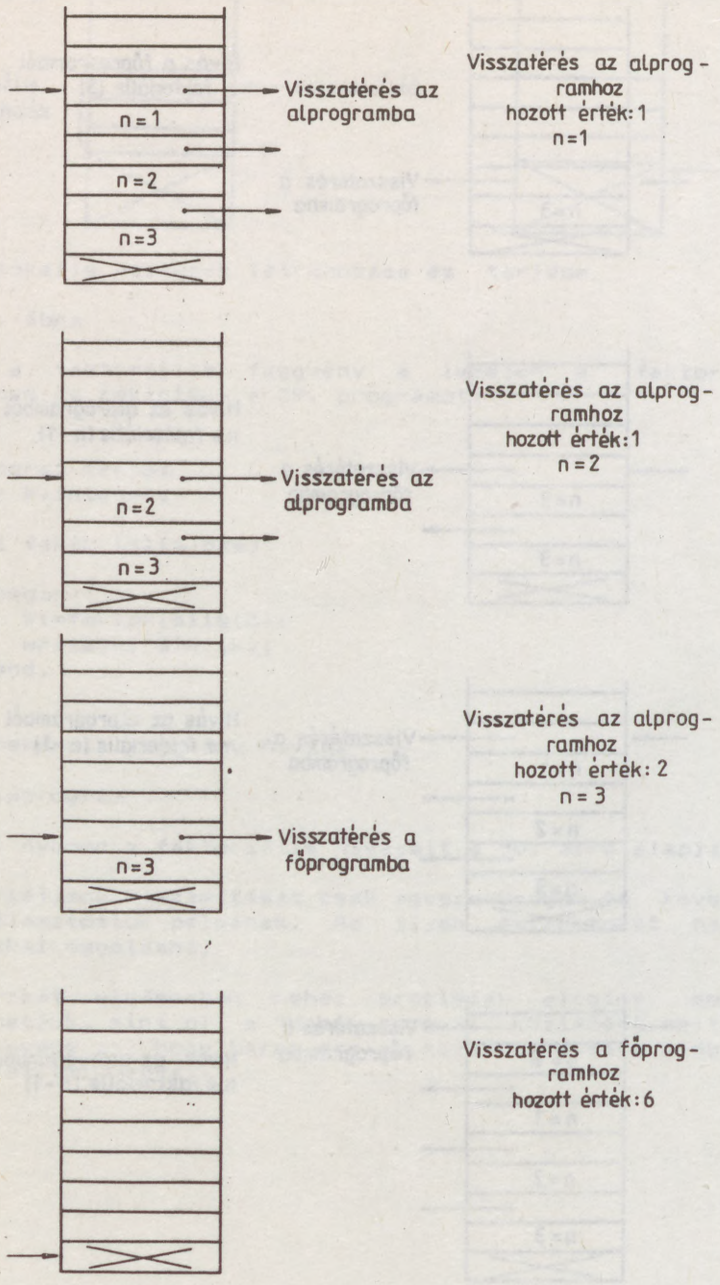
Hívás az alprogramból
n * faktoriális (n-1)



Hívás az alprogramból
n * faktoriális (n-1)

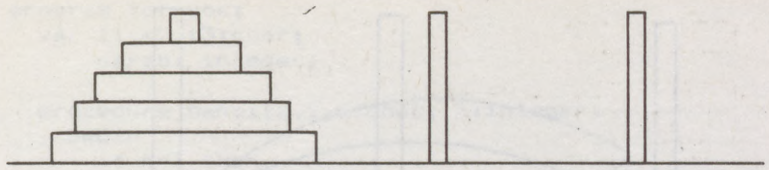


Hívás az alprogramból
n * faktoriális (n-1)



A rekurzív hívások nyomon követése

50. ábra



Hanoi tornyai

31. ábra

Az a feladat, hogy az első oszlopról egyenként átrakjuk a másodikra az összes korongot. Eközben használhatjuk a harmadik oszlopot is, mert van egy kis nehezítés a dologban.
A korongok különböző merétűek és sohasem - még átmenetileg sem - tehetünk kisebbre nagyobbat. (A legenda szerint akkor lesz vége a világnak, amikor Hanoi egyik buddhista templomában 64 db korongot átraknak a boncok, minden nap csak egyetlen korongot mozgatva.)
A feladat megoldását rekurzíve a következőképpen fogalmazhatjuk meg:

Ha csak egyetlen korongunk van, akkor a megoldás triviális: az első oszlopról áttesszük a másodikra.
Ha "n" korongunk van (n>1), akkor a következőképpen redukáljuk a problémát:

1. tegyünk át az első oszlopról a második oszlop segítségével a harmadikra "n-1"korongot

2. tegyünk át egy korongot az első oszlopról a másodikra

3. tegyünk át a harmadik oszlopról "n-1" korongot a másodikra az első oszlop felhasználásával.

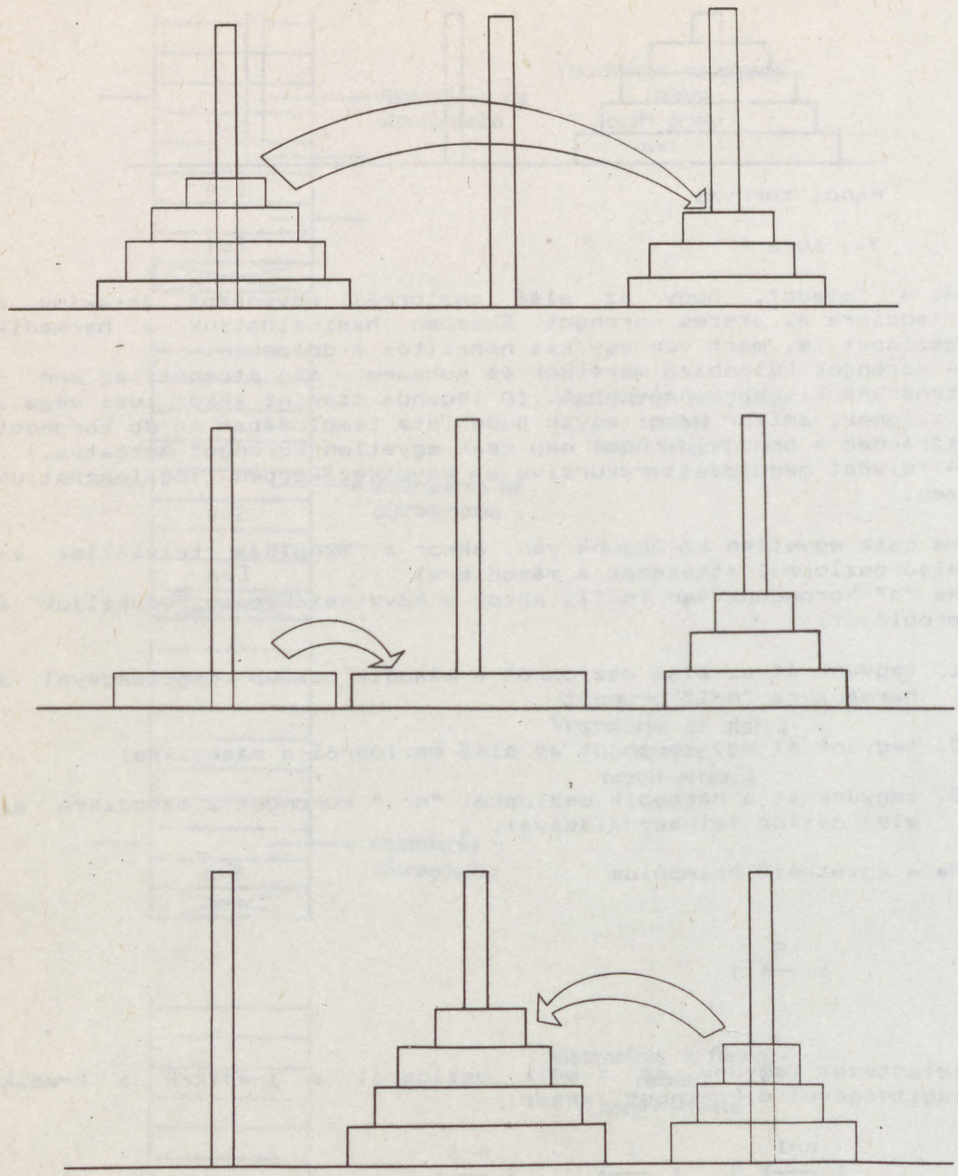
Ha a következő szimbólum

$$i \xrightarrow[k]{m} j$$

jelentése: tegyünk az i-edik oszlopról a j-edikre a k-adik segítségével m korongot, akkor:

$$1 \xrightarrow[2]{n-1} 3 \qquad 1 \xrightarrow{1} 2 \qquad 3 \xrightarrow[1]{n-1} 2$$

írja le a feladat megoldását.



A probléma redukciója

52. ábra

Definiáljuk a "hanoi" eljárást az előzőek segítségével, és a főprogram feladata csak a korongok számának megadása, és az eljárás hívása legyen (az oszlopokat rendre a, b, c jelölje).

```
program tornyok;
  var t1,t2,t3:char;
      darab: integer;

  procedure hanoi(x,z,y:char; n:integer;
    begin
      if n=1 then
        writeln('Tegyen egy korongot ',x,'-rol ',y,'-ra')
      else begin
        hanoi(x,z,y,n-1);
        writeln('Tegyen egy korongot ',x,'-rol ',y,'-ra');
        hanoi(z,y,x,n-1);
      end;
    end; {Hanoi}

  {a foprogram torzse}
  begin
    clrscr;
    write('Irja be a korongok szamat:');
    readln(darab);
    hanoi('a','b','c',darab);
  end.
```

A hanoi eljárás és meghajtó programja

30. program

darab=4 esetén a következőket írja ki a program:

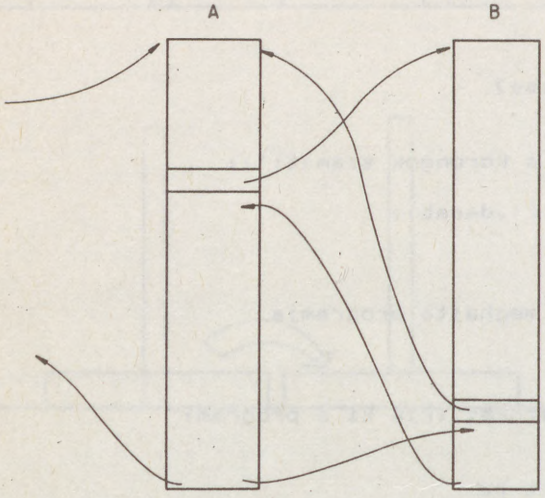
```
tegy egy korongot a-rol c-re
tegy egy korongot a-rol b-re
tegy egy korongot c-rol b-re
tegy egy korongot a-rol c-re
tegy egy korongot b-rol a-ra
tegy egy korongot b-rol c-re
tegy egy korongot a-rol c-re
tegy egy korongot a-rol b-re
tegy egy korongot c-rol b-re
tegy egy korongot c-rol a-ra
tegy egy korongot b-rol a-ra
tegy egy korongot c-rol b-re
tegy egy korongot a-rol c-re
tegy egy korongot a-rol b-re
tegy egy korongot c-rol b-re
```

Bármennyire elegánsak is a rekurzív programok, nem mindig hatékonyak. Könnyen előfordul, hogy kifogyunk a memóriából, mert a rekurzív eljárás minden híváskor vermeli a változókat. Ha a rekurzió túl sokszor hajtódik végre, akkor a

Heap/Stack collision

hibajelzéshez vezethet.
Sok esetben viszont kénytelenek vagyunk rekurzióhoz folyamodni, mert nem találunk a feladat megoldására más algoritmust. Néha nem is létezik más. Vannak olyan problémák, amelyekről bebizonyították, hogy csak rekurzíve oldhatók meg.

Nemcsak akkor beszélünk rekurzióról, ha egy alprogram önmagát hívja, hanem akkor is, ha két alprogram hívja egymást kölcsönösen (53. ábra). Ilyenkor korutinokról beszélünk.



Korutinok kapcsolata

53. ábra

Az egymást hívó alprogramok deklarálása nem oldható meg a szokásos módon, hiszen, ha az A alprogram hívja a B-t, akkor B-t az A előtt kell deklarálni:

```
.............
procedure B(paraméterek);
  ............
  end;
procedure A(paraméterek);
  ............
  B hívása
  ............
  end;
```

Mivel azonban B is hívja A-t, ezért az A-t a B előtt kell deklarálni, ami nyilván lehetetlen.
Az ellentmondást a FORWARD kulcsszóval oldhatjuk fel. Az egyik

alprogramnak először csk a fejét adjuk meg, törzsét a FORWARD szóval helyettesítjük:

```
procedure B(paraméterek);
  forward;

procedure A(paraméterek);
  eljárástörzs
  end;

procedure B;
  eljárástörzs
  end;
```

A B eljárás tényleges definiálásakor a paraméterezést már nem kell még egyszer megadni.

5.8 Mellékhatás

A változóparaméterek használatával hatást gyakorolunk a főprogrambeli változókra, megváltoztatjuk értéküket.
Más mód is elképzelhető a globális változók értékének megváltoztatására, elég egy egyszerű értékadás az alprogram törzsében az illető változóra.

Ha egy alprogram törzsében valamely globális változó értékét megváltoztatjuk, mellékhatásról beszélünk.

Figyeljük meg a 31. program viselkedését!

```
program mellékhatás;
  var a,b,c: real;

  function kobgyok(x:real):real;
    var xx:real;
    begin
      a:=x;
      xx:=1.0;
      while abs((x-xx)/xx)>1e-9 do begin
        x:=xx;
        xx:=(2.0*x+a/sqr(x))/3.0;
      end;
      kobgyok:=xx;
    end; {kobgyok}

  {a foprogram torzse}
  begin
    clrscr;
    readln(a,b);
    clrscr;
    writeln('a=',a:4:1);
    writeln('b=',b:4:1);
```

```
        c:=kobgyok(b);
        writeln;
        writeln('a=',a:4:1);
        writeln('b=',b:4:1);
        writeln('c=',c:4:1);
      end.
```

Mellékhatást okozó alprogram

31. program

Ha a program az

```
 1  8
```

bemenetet kapja, a kiírás:

```
          a= 1.0
b= 8.0

a= 8.0
b= 8.0
c= 2.0
```

Az "a" globális változó megváltozását a kobgyok függvény mellékhatása okozta, a függvénytörzs

```
    a:=x;
```

utasítása miatt.

A mellékhatások nehezen áttekinthetővé teszik a programot, ezért célszerű kerülni őket.

5.9 Feladatok

1) Készítsen alprogramokat az arcsin(x) és az arccos(x) függvények kiszámítására!

2) Írjon alprogramot másodfokú egyenlet megoldására. Használja alprogramját akárhány másodfokú egyenlet megoldására. Akkor álljon le a program, ha mindhárom együttható 0.

3) Írjon programot, amely különféle síkidomok területét számítja ki. Minden idomért külön eljárás legyen a "felelős", egy alkalmas karakterrel adja meg a főprogram számára, hogy milyen síkidommal akar dolgozni (pl.:'n' négyzet, 'k' kör stb).

4) Definiáljon függvényt valós szám valóskitevős hatványozására. Írassa ki a segítségével 3 hatványait a 0 és a 2 kitevő között, a kitevőt 0.1-enként változtassa.

5) Definiálja a faktoriális függvény nem rekurzív változatát és használja az

$$\binom{n}{k}$$

értékének meghatározásához!

6) A binomiális együtthatókra igaz az

$$\binom{n-1}{k-1}+\binom{n-1}{k}=\binom{n}{k}$$

összefüggés, ezenkívül tudjuk, hogy

$$\binom{n}{0}=1,\ \binom{n}{n}=1,\ \text{és ha } n<k,\ \text{akkor } \binom{n}{k}=0.$$

Írjunk rekurzív függvényt az összefüggések alapján a binomiális együtthatók kiszámítására!

7) Az egészkitevős hatványozásra írt rhatvanyi függvényünk nem túl hatékony, mert pl. a nyolcadik hatványt 8 szorzással számítja ki, pedig először kiszámíthatná a második hatványt, a második hatványból a negyediket, majd a negyedikből és a másodikból a nyolcadikat. Ez csak 3 szorzás. Ennek az elvnek a felhasználásával írjon rekurzív függvényt az egészkitevős hatványozásra!

8) Mit csinál a következő program? Ha nem tud gép nélkül rájönni, futtassa le és magyarázza meg az eredményt!

```
program talany;

  procedure beki;
    var ch:char;
    begin
      readln(ch);
      if ch<>'.' then beki;
      writeln(ch);
    end;

  begin
    clrscr;
    writeln('Irj be jeleket,');
    writeln('mindegyik utan [ENTER]-rel,');
```

```
    writeln('a veget ponttal jelezd!');
    beki;
    writeln;
  end.
```

9) Tekintsük a következő eljárást:

```
procedure csere(x,y: real);
  var w:real;
  begin
    w:=x;
    x:=y;
    z:=w;
  end;
```

Ha p értéke 3.7, q értéke 4.3, mi lesz az értékük a

```
csere(p;q)
```

eljáráshívás után?

10) A kiterjesztett ASCII kódrendszerben a 127 és 255 közötti kódok némelyike olyan grafikus karaktereknek felel meg, amelyekből derékszögű négyszögeket lehet kirajzolni (négyféle sarokelem, vízszintes és függőleges vonalkák). Készítsen egy eljárást, amely megadott méretű téglalapot rajzol a képernyő bal felső sarkába.

11) Meghatározhatunk a képernyőn egy téglalapot úgy, hogy megmondjuk a bal felső és a jobb alsó csúcsának koordinátáit. Koordinátákon itt sor és oszlopszámot értünk. Írjon egy általánosabb téglalaprajzoló eljárást ezen az elven (négy paraméterrel). Használhatja a

```
gotoxy(xhely,yhely)
```

eljárást, amellyel a kiírás helyét a képernyőn bárhol szabadon megválaszthatja. A

```
gotoxy(10,8);
write('*');
```

utasításokkal pl. a *-ot a képernyő 8-adik sorában a 10-edik helyre írhatja.

12) Írjon programot, amely az előző téglalaprajzoló alprogram felhasználásával n db véletlen elhelyezésű és alakó téglalapot rajzol. A képernyőről "kilógó" téglalapokat ismerje fel és ezek helyett másikat rajzoltasson.

6. A PROGRAMOZÁS MÓDSZERTANÁRÓL

Hamarosan újabb nyelvi eszközöket ismerünk meg és előfordulnak hosszabb, bonyolultabb problémák is. Ugyanakkor már elég sokat megtanultunk, hogy lássunk valamit a programozás nehézségeiből. Talán még nem késő - és bizonyára nem korai - azokról a módszerekről beszélni, amelyek szisztematikusabbá teszik munkánkat.
A programozás módszereiről kötetek írtak. Különböző programozási technológiákat fejlesztettek ki, amelyek a szoftvergyártást igazi ipari tevékenységgé próbálják szervezni. Az ilyen kérdéseknek tolnyomó részét még érinteni sem tudjuk ebben a fejezetben. A programozás magasszintű szellemi alkotómunka, így igazából nem is hisszük, hogy technológiákba merevíthető. Ez azonban nem jelenti azt, hogy ne volnának a szakmának olyan szabályai, amelyek az alkotóerőt egyrészt támogatják azáltal, hogy irányítják a működését, másrészt korlátozzák is.

Az első gondolat amit fel kell vetnünk, a jó programozási szokások kifejlesztésének a kérdése. A kezdő programozó boldog, ha a nehezen összetákolt programja egyáltalán működik. Ha kap egy feladatot, azon nyomban nekiesik a gépnek, behívja az Editort és elkezdi írni a programot. Valamit összehoz, aztán megnézi, hogy működik-e. Persze nem! Ezzel folytatódik a programbütykölés. Még szerencse, hogy a Pascal nyelv - a BASIC-kel ellentétben - nem támogatja, ellenkezőleg, megnehezíti a bütykölők dolgát.

Hát ez így nem megy!

Az első és legfontosabb szokás, amit ki kell fejlesztenünk magunkban, hogy a programot gondosan megtervezzük. Ennek a munkának három fázisa van:

1) A probléma megfogalmazása

   Fogalmazzuk meg egyértelműen a feladatot. Tisztázzuk, hogy milyen tartalmú és formájú eredményt kell kapnunk, ehhez milyen adatok állnak rendelkezésünkre. Elemezzük, hogy mely adatok szükségesek az eredmény egyes részeinek a meghatározására. Elegendőek-e az adatok?
   Ne kezdjünk addig az algoritmus tervezéséhez, amíg nem értjük világosan mit kell tenni.

2) Az algoritmus tervezése

   Fogalmazzuk meg egyértelműen, hogy milyen tevékenységeket kell elvégezni, hogy az adatokból megkapjuk az eredményeket. Bontsuk a tevékenységeket résztevékenységekre, ehhez használjuk a megismert tevékenységszerkezeteket, egyébként világos mondatokban fogalmazzunk. Igyekezzünk az egyes

résztevékenységeket, mint leendő alprogramokat megfogalmazni, gondoljunk a paraméterezésre is (mi az eljárás inputja és mi az outputja). Ezáltal minden tervezési szinten tisztázzuk a résztevékenységek kapcsolatát. El kell jutni olyan szintig a tervezésben, ahol már a végső résztevékenységek közvetlenül programozhatók.

3) Programírás

Sohase írjunk hosszú programot. A program hosszával rohamosan nő a hiba lehetősége. Ne használjunk olyan konstrukciókat, amelyek működésében nem vagyunk biztosak. (Itt felmerül az önképzés fontossága.) Használjuk a nyelv leírását, ha kételyeink vannak. Miután egy alprogram elkészült, nézzük át és újra gondoljuk végig, hogy működik.

A programírásnak is megvannak a maga hasznos szokásai. Adjunk a programunkban deklarált objektumoknak "beszélő" neveket. Könnyebb nyomon követnia program működését, ha "nev", "ber", "jovado", stb azonosítókat használunk, mintha x-et, y-t és z-t írnánk. Egyszer olyan konstansdeklarációval találkoztam, amely kifejezetten félrevezető volt:

```
const tiz=8;
```

Egy tízes számrendszerbeli probléma megoldására készült programot valaki átírt nyolcas számrendszerre és az alapszámot megváltoztatta. Mennyivel helyesebb lett volna ha az eredeti programban a

```
const alapszam=10;
```

deklarációt használják!

Nagyon megkönnyíti a program áttekintehetőségét a bekezdések helyes használata. Ahogy egy if utasítás ágait, vagy egy ciklus törzsét beljebb kezdjük írni, ezzel a program szerkezetét tesszük láthatóvá.

Használjunk kommentárokat. Eddigi programjainkban ritkán kellett élni ezzel a lehetőséggel - inkább csak tájékozódási pontok megjelölésére alkalmaztuk. Nagyobb programoknál a tervezés során létrejött mondatokat használhatjuk kommentárként. Ez egyszerre több szinten is olvashatóvá teszi a programunkat. Ne felejtsük! a kommentárok saját magunknak szólnak és mindazon kollégáinknak, akik kapcsolatba kerülnek a programunkkal. Világosan írjuk le az algoritmust. Aki a kommentárokon spórol, később óriási energiát fordíthat arra, hogy saját - már elfelejtett - szándékait kibogarássza, ha módosítania kell, vagy tovább kell fejlesztenie a programot.
Többször tapasztaltam, hogy néhányan először megírják a programot, majd ezután - mert ez kötelező - kiegészítik kommentárokkal. Ezek a kommentárok olyanok is. Felszínesek, nem a lényeget ragadják meg, formálisak és nem egyszer félrevezetőek.

Higgyük el, jó kommentárt csak a program írásakor tudunk írni, utólag már nem. Annyit sem ér, mint a vasárnapi edédre kedden megivott sör.

6.1 A jól struktúrált program

A Pascal struktúrált programozási nyelv, támogatja a jól struktúrált programok írását és megnehezíti azok dolgát, akik programbütyköléshez szoktak. Megjegyezzük, hogy struktúrálatlan nyelveken is lehet jól felépített programokat írni, csak nehezebb. A struktúrálatlan nyelveket a nyelvi eszközök hiánya jellemzi, amit a goto ugró utasítással pótolnak.

Goto utasítást a Pascalban is találunk, azonban valójában nagyon ritkán indokolt alkalmazni. A goto használatához címkék szükségesek. A címkével megjelölhetjük a program egy pontját, azután a program egy másik helyéről a megjelölt helyre ugrunk. A programban használt címkéket - más objektumhoz hasonlóan - deklarálni kell. Nincs szándékunkban a goto szintaxisát és szemantikáját részletezni, így egy példán mutatjuk be az elmondottakat.

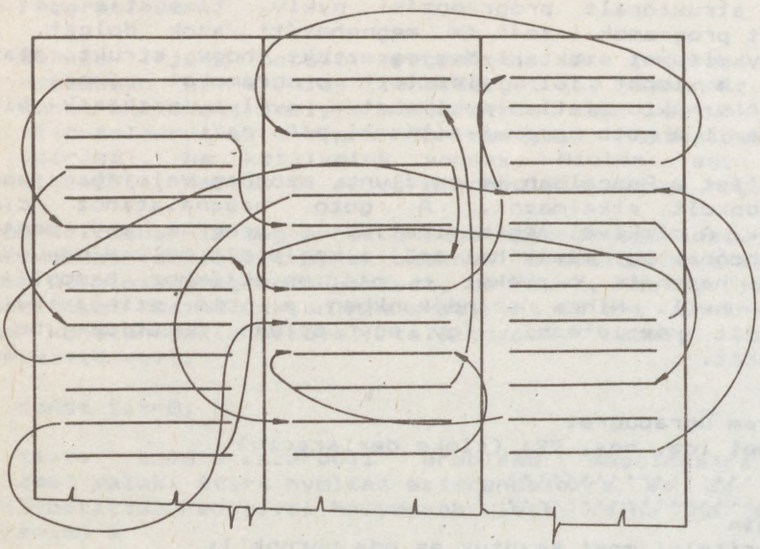
```
program ugrabugra;
  label ide, oda, 99; {cimke deklaracio}
  { ..................
    .................. }
  begin
    writeln('most kezdtuk es oda ugrunk');
    goto oda;
    { ..................
      .................. }
  ide:
    writeln('itt vagyunk');
    writeln('a vegere megyunk');
    { ..................
      .................. }
    goto 99;
  oda:
    writeln('ott vagyunk');
    writeln('visszaugras kovetkezik');
    goto ide;
    { ..................
      .................. }
99:
end.
```

Ugrások a programban

32. program

Struktúrálatlan nyelvekben a goto utasítást a hiányzó

kiválasztási és ismétlési szerkezetek programozására indokolt használni.
A dolog eddig rendben is van. A baj ott kezdődik, ha egy programot elkezdenek javítgatni, módosítgatni úgy, hogy elugranak valahová, ott beírnak pár programsort, azután visszaugranak. Ha ezt többször megteszik, a program az ugrásokkal valahogy úgy fog kinézni, mint a 54. ábra.



programlista ugrásokkal

54. ábra

Nos az ilyen program minden, csak nem jól struktúrált.
A struktúrált programokban a program szövege visszatükrözi a tevékenység időbeliségét: valamely A tevékenységet leíró utasítások akkor és csak akkor állnak a programlistában a B tevékenység utasításai előtt, ha a B tevékenység az A után hajtódik végre. Ezért, ha elő is fordul goto utasítás, mindig a program vége felé ugrunk.
A Turbo Pascalban a goto helyett gyakrabban alkalmazzuk az

```
exit
```

utasítást, amivel az aktuális blokkból léphetünk ki. Pl. ha egy eljárásban olyan feltételeket észlelünk, ami miatt az eljárás nem tudja elvégezni a feladatát, exit-tel visszaléphetünk a hívó blokkba. Ha olyanok a feltételek, hogy a program tevékenysége is meghiúsul, akkor a

```
halt
```

utasítással leállíthatjuk a működését. Ez a két utasítás a goto használatát a legfontosabb esetekben feleslegessé teszi.

Ebben a könyvben a goto utasítást egyetlen alkalommal sem fogjuk használni.

6.2 A top-down módszer

A struktúrált programozás feltételezi a programok felülről-lefelé történő kifejlesztését. Ez megfelel annak a folyamatnak, amit az algoritmustervezéssel kapcsolatban már korábban leírtunk. Nézzünk itt egy kis példát!
Készítsünk algoritmust amely leírja, hogyan tudunk valakit telefonon felhívni. A Posta műszaki helyzetének jóvoltából arra is fel kell készülni, hogy nincs vonal. Az is lehet, hogy foglalt a szám, ill. nem veszik fel.
Ha most nem felülről-lefelé kezdjük a megoldást, hanem azzal foglalkozunk, hogy mi a teendő, ha pl. nincs vonal, akkor elvesztünk. Miért?
Ha nincs vonal, helyére tesszük a kagylót, kicsit várunk, majd újra próbálkozunk. Kialakult egy ciklus. Ugyanez a helyzet akkor is, ha foglalt a szám. Igen ám, de lehet, hogy az egyik próbálkozásnál vonal nincs, utána foglalt jelzést kapunk. (Ilyenkor szokták azt kérdezni a strukturálatlan nyelvhez szokott hallgatók, hogy "Ugrás nincs?")
A top-down módszer nem engedi, hogy túl korán elvesszünk a részletekben. Nincsenek részletek. Legfelső szinten a következőképpen írhatjuk le a tevékenységet:

```
Ismételd
  próbálkozz meg egy hívással
mígnem sikerül beszélni
```

Ezután a "próbálkozz meg egy hívással" tevékenység már könnyen leírható egy kiválasztási szerkezettel:

```
vedd fel a kagylót;
ha nincs búgó hang, akkor
   tedd le a kagylót
egyébként,
   ha foglalt a szám, akkor
      tedd le a kagylót
   egyébként,
      ha nem veszik fel, akkor
         tedd le a kagylót
      egyébként beszélj
```

A top-down programozásban az algoritmus tervezése és a programozás nem különül el egymástól élesen. Ezt egy egyszerű példán szemléltetjük.
Tegyük fel, hogy egyszeregytáblázat nyomtatásához akarunk egy kis programot írni. A tevékenységeket az elképzelt eredmény alapján határozhatjuk meg (55. ábra).

A tevékenységet fokozatos finomításokkal tesszük egyre részletesebbé, mígnem elkészül a program.

```
0. szint: nyomtassuk ki az egyszeregytáblát!

1. szint: nyomtassuk ki a fejlécet;
          nyomtassuk ki a táblázat többi részét

2. szint: nyomtassuk ki a fejlécet;
          {nyomtassuk a tablazat tobbi sorat}
          for i:=1 to 10 do
             nyomtassuk ki az i-edik sort
```

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
.	.	.	.	.	.	.	.	.	.	.
10	10	20	30	40	50	60	70	80	90	100

A nyomtatandó táblázat

55. ábra

```
3. szint: nyomtassuk ki a fejlécet;
          {a tablazat tobbi sora}
          for i:=1 to 10 do begin
             {az i-edik sor}
             write(i:3);
             nyomtassuk az i-edik sor folytatását
             end;

4. szint: nyomtassuk ki a fejlécet;
          {a tablazat tobbi sora}
          for i:=1 to 10 do begin
             {az i-edik sor}
             write(i:3);
             {az i-edik sor folytatasa}
             for j:=1 to 10 do
                write(i*j:4);
             writeln;
             end;

5. szint: {a fejlec}
          write('   ');
          for j:=1 to 10 do
             write(j:4);
          writeln;
          writeln;
          {a tablazat tobbi sora}
```

```
for i:=1 to 10 do begin
   {az i-edik sor}
   write(i:3);
   for j:=1 to 10 do
      write(i*j:4);
   writeln;
   end;
```

Egyszeregytáblázat nyomtatása

33. program

Még a deklarációkat kell csak elkészíteni és a program kész.

Most természetesen csak a szemléltetés kedvéért haladtunk ilyen kis lépésekkel. Általában egyszerre egy-egy alprogramot intézünk el.
Figyeljük meg, hogy lett az algoritmus mondataiból a programban kommentár!
Arra is vigyázzunk, hogy ne írjunk túl sok kommentárt. Egy olyan program, amelynek minden sorát kommentárral kísérjük, ugyancsak nehezen olvasható.

6.3 A programtesztelés

Megírása után programunk valószínűleg nem mentes a hibáktól. A következő lépés a hibák szisztematikus felderítése és kijavítása. A programozásnak ezt a szakaszát általában "programbelövés"-nek nevezik. Az angol "debugging" (kb. hibátlanítás) szerencsésebb elnevezés. Kalmár László egy programozási konferencián a "kibogarászás" kifejezést javasolta, de sajnos nem terjedt el. Így maradunk a belövés vagy a tesztelés kifejezés mellett.

Figyelmetlenségből eredő hibáink egy része a már korábban említett szintaktikai hibák közül kerül ki. E hibákat a fordítóprogram felismeri, az okot és többnyire a hiba helyét is jelzi. A szintaktikus hibák kijavítása után a programot lefordítja a fordító és elindítható a végrehajtása. (Célszerű a Run menüpontot használni.)

A program futása közben is kapunk hibajelzéseket, ilyen pl.:

Floating point overflow

vagy

Division by zero attempted

amit akkor kapunk, ha valamely valós számmal végzett művelet eredményeképpen kapott érték nem ábrázolható, mert meghaladja a real típus maximális értékét, ill. ha nullával való osztást kellene végrehajtani a gépnek.
Célszerű, ha az általunk készített alprogramokba is beépítünk

hiba elleni védelmet és hiba esetén megfelelő hibaüzenetet íratunk ki.

Ha a programunk már működőképes, tehát sem szintaktikai, sem pedig futás közbeni hibaüzeneteket nem kapunk, még nem biztos, hogy jó. Lehetnek benne logikai hibák. A logikai hibákat a leggyakrabban a tervezés fázisában követjük el. Pl a számításra egy hibás képletet alkalmazunk, vagy az algoritmust nem gondoljuk át kellőképpen, így nem azt végzi, amit hiszünk róla. Az is megeshet, hogy a program írásakor követünk el olyan hibát, amitől a program még működőképes, de nem azt számolja, amit kell. Pl. a másodfokú egyenlet gyökeinek kiszámításakor a következőt írhatjuk:

```
diszkr:=sqr(b)-4.0*a*c
if diszk>=0.0 then begin
  gyok1:=-b+sqrt(diszkr)/(2.0*a);
  gyok2:=-b-sqrt(diszkr)/(2.0*a);
```

Itt a hibát az okozta, hogy kifelejtettünk egy zárójelpárt, s a helyes gyökképlet helyett a

$$-b + \frac{\sqrt{b^2 - 4ac}}{2a}$$

kifejezéssel számoltunk. Semmiféle hibajelzést nem kapunk, csak az eredmény rossz.
Ilyenkor a várt és a tényleges eredmény eltérései árulkodhatnak a hiba helyéről és jellegéről. Ha a várt eredmény számszerű, akkor mindig készítsünk kézzel kiszámolt mintapéldákat.
Sokszor segít a hiba kiderítésében, ha a programunkba több helyre beszórunk kiíratásokat. Kiíratjuk a szükséges - vagy akár az összes - változó értékét, hogy felderítsük, hol romlott el valami.
Ciklusokat tartalmazó programokkal néha előfordul, hogy "megbolondulnak" és nem akarnak megállni. Egy hibás kilépési feltétel a felelős a végtelen ciklusért, legtöbbször ha valós értékek egyenlőségére akarjuk leállítani a ciklust. Ilyenkor gondot okozhat a program megállítása. Beolvasási és kiírási műveletek végrehajtása közben a [CTRL] [C] lenyomásával megszakíthatjuk a futó programot, máskor nem. Ha ciklust tartalmazó programot akarunk belőni, mindig használjuk az U compiler direktívát. Ha a program elején elhelyezzük a {$U+} jelzést (user interrupt):

```
{$U+}
program veszelyes;
.................
.................
```

akkor a [CTRL] [C]-vel bármikor megszakíthatjuk a program futását. Ha a programunk már biztonságosan működik, ne felejtsük el eltávolítani, mert nagyon lelassítja a programot. Az U direktíva alapértelmezés szerint kikapcsolt {$U-} állapotban van. Az input-output műveletek megszakíthatóságáért is egy compiler direktíva a felelős, a C direktíva, de ennek alapértelmezése {$C+}, ezért nem kell gondoskodnunk a bekapcsolásáról.
Használni fogjuk majd bizonyos hibák futás közbeni felismerésére az R direktívát is. Alapértelmezés szerint kikapcsolt {$R-} állapotban van. Ha bekapcsoljuk, akkor pl. hibajelzést kapunk, ha egy bájt típusú változóba 255-nél nagyobb integer értéket akarunk írni.

A nagyobb - több eljárást tartalmazó - programok belövését szisztematikusan, a programtervezéshez hasonlóan felülről lefelé végezhetjük el. Először a főprogramot írjuk meg, a főprogramban hívott eljárásokat üres eljárásokkal szimuláljuk. Az üres eljárás az eljárásfejből áll, az eljárásblokk pedig nem tartalmaz utasítást, vagy csak egy szöveget ír ki jelezve, hogy végrehajtódott (34. program).

```
procedure ures(var x: real; k:integer);
  begin
    writeln('meghívott eljaras: ures');
    writeln('kapott paraméterek: x=',x,' k=',k);
    writeln('visszaadott ertek: x=',x);
  end;
```

Üres eljárás

34. program

Ennek a módszernek előnye, hogy a tesztelés már korán, a tervezés szakaszában elkezdődhet. Ahogy a munkában előrehaladunk, üres eljárásainkat rendre az igaziakkal váltjuk fel. Így minden alkalommal a teljes program működését ellenőrizzük.

Vannak olyan alprogramok, amelyeket könyvtárosítási céllal, magunk és mások számára készítünk, hogy több különböző programban is felhasználhatók legyenek. Ezeket külön kell belőni. Az alprogramok belövéséhez meghajtó programot kell írni. A meghajtó programnak különféle feltételek mellett kell kipróbálnia az eljárást.

A programok tesztelésénél lehetnek előre meghatározott célok, mint pl. ciklusoknál meg kell nézni hogy a kilépési feltételek megfelelően működnek-e, nem fut-e tól a ciklus, illetve nem történik-e idő előtti kilépés. Sokszor célszerű azonban egy nagyobb adathalmazon is kipróbálni a programot. Ezt az adathalmazt pl. a random véletlenszám-generátorral is előállíthatjuk.

Világosan látnunk kell, hogy a programtesztelés módszere korántsem ad választ arra a kérdésre, hogy jó-e a program. Nagyobb és bonyolultabb programoknál sohasem mondhatjuk a tesztelés befejezése után sem, hogy mindig, minden körülmények között hibátlanul fog működni. Annál is inkább, mert a program "jósága" intuitív fogalom, még azt sem tudjuk, hogy mit értsünk pontosan alatta. Állapodjunk meg abban, hogy akkor nevezünk jónak egy programot, ha az mindenben a specifikációjának megfelelően működik.
A 60-as évek óta komoly erőfeszítések történnek arra nézve, hogy matematikai eszközökkel bebizonyítsák egy programról, hogy (ebben az értelemben) jó. Ennek a munkának az eredményeként született már olyan program, amély adatként megkap egy matematikailag megfogalmazott programspecifikációt, majd egy Pascal nyelvű programot, és bizonyítja a program helyességét (vagy hibás voltát). Csak az a baj, hogy a specifikációt nehezebb elkészíteni, mint a programot.

Így egyelőre meg kell maradnunk a gyakorlatban a jó öreg programbelövés mellett, és belenyugodni abba, hogy programjaink valószínűleg jók, vagy legalábbis, sok és nagy hibát nem tartalmaznak.

6.4 A program dokumentálása

A programdokumentáció kérdését a személyi számítógépek elterjedése a korábbihoz képest más megvilágításba helyezte. Egy nagyszámítógépes rendszerben működő program használatához a felhasználónak kézikönyveket kellett tanulmányoznia (ez sok embert - akinek szüksége lett volna rá - elriasztott a számítógépek használatától).

Természetesen a kézikönyvekre most is szükség van. Azonban a személyi számítógépek programjai párbeszédesek és megfelelő promptok kiírásával irányítják a felhasználó munkáját. A kézikönyvek anyagát, a promptokhoz fűzött bővebb magyarázatokat pedig nem egyszer a felhasználói program HELP funkciójával - magától a programtól is - megkérdezhetjük.

Természetesen mi még messze vagyunk attól, hogy ilyen felhasználói programokat írjunk. Azonban - amint eddig is tettük - kis programjainkat is "felhasználóbarát" módon kell megírnunk. Mondja el a program, hogy mi a funkciója, ehhez milyen adatot kér és milyen bánásmódot igényel.
A hivatásos programozók által írt programok zömét különféle helyeken változatlan formában használják olyan emberek, akiknek semmi részük sem volt az elkészítésében, sőt, nem is számítástechnikai szakemberek. Pl. a bérszámfejtést, a raktári nyilvántartásokat, a szállodai szobafoglalást feldolgozó programok ilyenek. Számukra szakmai zsargon nélküli, világos és gyakorlatias dokumentációt kell készíteni. A felhasználói dokumentációnak a következőket kell tartalmaznia:

1) A felhasználási terület és a program tevékenységének leírása.

2) A program által igényelt adatok leírása.

3) A kiírt eredmények tartalmi és formai leírása.

4) A program kezelésének részletes leírása.

5) A kiírt üzeneteknek, hibajelzéseknek a magyarázata és a választevékenységek leírása.

6) A felhasználási lehetőségek és korlátok ismertetése.

A forrásprogram szövegének is dokumentálnia kell önmagát. Minden olyan információt tartalmaznia kell, ami lehetővé teszi

1) a használat során észlelt hibák javítását,

2) a felhasználási környezet változásával kapcsolatos módosítások, a program karbantartás végrehajtását,

nemcsak a program szerzője, hanem bármely programozó számára, aki a jövőben ilyen feladatot kap.
Mindehhez szükségesek a kommentárok, a beszélő azonosítók, a program olvashatóságát elősegítő tagolás, az alprogramok használata (méghozzá az egy funkció - egy alprogram elv betartásával).
A programok elején adjuk meg kommentárként a program feladatának meghatározását, környezeti feltételeit (pl. ha external alprogramokat használ, akkor a megfelelő könyvtár elérhető legyen).
Különösen kritikus a több program által is felhasználható könyvtári alprogramok dokumentálása. Állapodjunk meg abban, hogy minden ilyen célú alprogramunk elején a következő információkat adjuk meg kommentárok formájában:

1) A funkció leírása

   Röviden meghatározzuk, hogy milyen tevékenységet végez az alprogram.

2) Az alprogram hatása

   Leírjuk a működés eredményét, pl., hogy milyen értékeket ad vissza a hívó blokknak.

3) Paraméterek

   felsoroljuk a paramétereket, mindegyiknek megadjuk a szerepét, típusát és a paraméterátadás irányát (input vagy output paramétere az eljárásnak).

Globális objektumok

Ha az alprogramnak szüksége van valamilyen globális konstansra, globális változóra, stb, akkor ennek az azonosítóját és szerepét leírjuk.

Igényelt alprogramok

Ha az alprogram más könyvtári alprogramokat használ, ezeket fel kell sorolni.

Példa

Bemutatjuk egy konkrét példán, hogyan kell az alprogramot használni.

Hibajelzés

Ha az alprogram hibajelzést ad, akkor e hibajelzés szövegét és a hibafeltételt is leírjuk.

Megjegyzés

Bármilyen fontos körülményt megadhatunk itt, amit a korábbiakban nem volt módunk közölni.

A 34. program listáját a dokumentáció kedvéért mutatjuk be.

```
function lg (hatv: real):real;

{ A fuggveny funkcioja:

     Az lg fuggveny a hatv ertek 10-es alapu
     logaritmusat szamitja ki.

  Hatasa:

     A függvény értéke a parameter logaritmusa.

  Parameter:

     hatv (be): a valos szam amelynek a logaritmusara
     szuksegunk van.

  Globalis objektumok: -

  Igenyelt alprogramok: -

  Pelda:

     k:= lg(1000);
     k értéke 3.0 lesz.
```

```
    Hibajelzes: -

    Megjegyzes:

       A parameter 0 vagy negativ ertekeinel futas
       kozbeni hibajelzest kapunk.
  ------------------------------------------------------}

  begin
    lg:=ln(hatv)/ln(10.0);
  end;
```

Dokumentált alprogram

35. program

6.5 Feladatok

1) Vizsgálja meg és értékelje eddigi programjait a programozási stílus szempontjából!

2) Fogalmazza meg önmaga számára pontosan a bekezdések használatának szabályait. Használja ezután következetesen ezeket a szabályokat!

3) Tegyük fel, hogy ön egy társkereső GMK-tól megrendelést kap a házasságközvetítés számítógépes támogatására. Definiálja a problémát! Engedje szabadon a fantáziáját!

4) A háztartásban mindig szükség van bizonyos anyagok (élelmiszerek, mosószerek stb.) készletezésére. Tervezzen programot, amely a készletek nyilvántartását végzi.

5) Tegyük fel, hogy a következő utasításokhoz a megfelelő deklarációk rendelkezésre állnak. Keresse meg az esetleges szintaktikai hibákat és javítsa ki az utasításokat!

a)
```
while x<1 and y<>2 do
  readln(ch);
```

b)
```
if jolett and (i<=n)
  then
else writeln('kesz');
```

c)
```
if szamlalo:=0 then
  writeln('nincs tobb adat);
```

d)
```
case i of begin
  1,2: j:=j+1; k:=k+1;
    3: j:=sqrt(j+k);k:=0;
  4,5: j:=abs(j+k); k:=-1;
  end;
```

e) if szam<10
then a:=a+1;
else b>=b+1;

6) Keresse meg és javítsa ki a következő program hibáit! Melyek a szintaktikai és melyek futás közben felismert hibák?

```
programhibák;
{
 Ez a program az elso k db termeszetes szam
 osszeget hatarozza meg. k adat.}

  var k, sum: integer;
    readln(k);
    sum:=1
    for i=1 to k do
      sum=k
    writeln('Az osszeg 1-tol k-ig'sum)
  end.
```

7) Az n természetes szám faktoriálisát a következőképpen is definiálhatjuk:

$0! = 1$

$n! = n*(n-1)*(n-2)*...*2*1$

Tekintsük ezt a következő alprogram specifikációjának:

```
function fact(n:integer):integer;
  var i, szorzat:integer;
  begin
    i:=0;
    szorzat:=1;
    repeat
      i:=1+i;
      szorzat:=szotzat*i
      until i=n;
    fact:=szorzat;
  end;
```

Jó-e ez a program (megfelel-e a specifikációnak)?

7. ADATSZERKEZETEK

7.1 Az adattípusok áttekintése

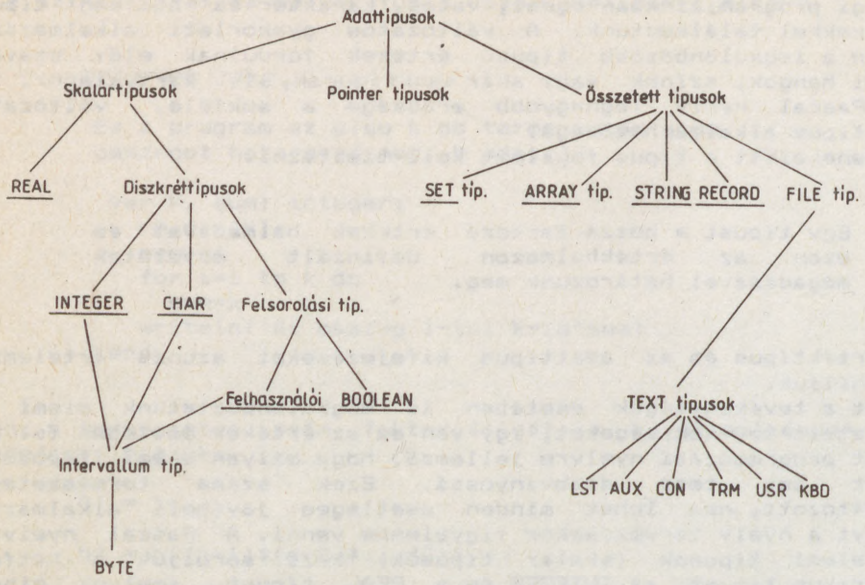
Eddigi programjainkban egész, valós, karakter és boolean típusú értékekkel találkoztunk. A változatos gyakorlati alkalmazások során a legkülönbözőbb típusú értékek fordulnak elő: szavak, zenei hangok, színek vagy akár sakkfigurák, ill. kártyalapok.
A Pascal nyelv legnagyobb erőssége a sokféle, változatos adattípus alkalmazhatósága.
Mindenekelőtt a típus fogalmát kell tisztázni.

> Egy típust a hozzá tartozó értékek halmazával és ezen az értékhalmazon definiált műveletek megadásával határozunk meg.

Az értéktípus és az adattípus kifejezéseket azonos értelemben használjuk.
Amint a tevékenységek esetében is megkülönböztetünk elemi és összetett tevékenységeket, így van ez az értékek esetében is. Egy adott programozási nyelvre jellemző, hogy milyen elemi típusokat vezet be, tesz szabványossá. Ezek száma természetesen korlátozott, nem lehet minden esetleges jövőbeli alkalmazási igényt a nyelv tervezésekor figyelembe venni. A Pascal nyelvben az elemi típusok (skalár típusok) közé soroljuk a kétféle numerikus típust, az INTEGER és a REAL típust, amelyek minden Pascal implementációban szabványosak. Az INTEGER típust fixpontos egész, a REALt pedig lebegőpontos számok alkotják. A típushoz tartozó értékek halmaza implementációfüggő.
A nyelvben hozzáférhető elemi jelek alkotják a CHAR típust, amely a Turbo Pascal esetében ASCII kódváltozat.
Implementációs szinten definiált ún. felsorolási típus a BOOLEAN típus, amely a FALSE és a TRUE értékekből áll.
A Pascal értéktipizálási lehetőségeinek igazi ereje nem is az elemi típusok sokféleségében rejlik. A nyelv egy sor lehetőséget ad új típusok létrehozására.
Lehetőség van pl. a szabványos felsorolási típus mellett saját felsorolási típusok definiálására is.
Az INTEGER, CHAR, BOOLEAN és a programozó által definiált felsorolási típusokat diszkrét típusoknak nevezzük. (Figyelem! a skalár típusok közül a REAL nem diszkrét típus!) A diszkrét típusok egy intervallumának megadásával intervallumtípusokat definiálhatunk. Ilyen intervallumtípus a Turbo Pascalban szabványos BYTE.

A skalártípusokból néhány szerkesztési mechanizmussal összetett adattípusokat hozhatunk létre. Ilyenek a

1. tömbtípusok,
2. recordtípusok,
3. halmaztípusok,
4. file-típusok.



Pascal adattípusok

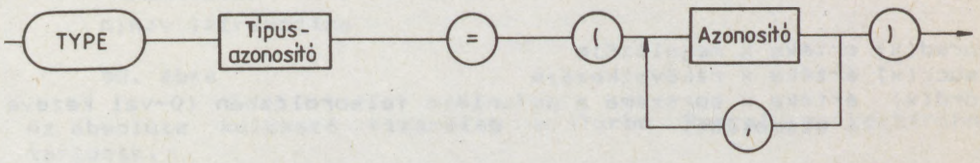
56. ábra

A szabványos Pascal definíciója szerint az összetett típusok tárfoglalása a packed minősítővel csökkenthető (pakolt adatszerkezetek). A Turbo Pascalban a packed használata nem tilos, de hatástalan, mivel a fordítóprogram automatikusan optimalizálja a tárigényt.
Új szabványos tömbtípus a string típus, amellyel a fűzérváltozókat határozhatjuk meg. Az adatállományok definiálására való file-típusok közül minden Pascal változatban szabványos a text típus.
A pointertípussal dinamikus adatszerkezeteket tudunk konstruálni. A dinamikus adatszerkezeteket a program futása közben hozzuk létre (nem úgy, mint pl. a tömböket a deklarációval).
A Pascal adattípusokat a 40. ábrán tekinthetjük át.
A nyelvben előre definiált (elemi) adattípusokat a rajtuk - ugyancsak előre - definiált műveletekkel (elemi tevékenységekkel) dolgozhatjuk fel. Ha a felhasználó a rendelkezésre álló nyelvi eszközökkel egy új adattípust definiál, az ő dolga az adattípushoz tartozó értékhalmazon megvalósítani a megfelelő műveleteket is. Az új műveleteket eljárásokkal, ill. függvényekkel határozzuk meg.
Ezzel hozzuk létre ténylegesen az új típust, mivel az adattípusok

a típushoz tartozó értékek halmazából, és a halmazon definiált műveletekből állnak.

7.2 A diszkrét típusok

Az egyszerű (skalár) típusok a valós típusok kivételével alkotják a diszkrét típusok osztályát. Közös jellemzőik: értelmezett rajtuk az ord, a succ és a pred függvény, for ciklusok ciklusváltozói és ciklusparaméterei, case kifejezés és case címkék típusai lehetnek.
A diszkrét típusok körét új felsorolási típusok létrehozásával bővíthetjük. Egy felsorolási típust a típushoz tartozó értékek tényleges felsorolásával adhatunk meg. Az új típusokat a programblokk elején a konstansokhoz és a változókhoz hasonlóan deklarálni kell. A típusdeklarációs alapszó a type (57. ábra).



Felsorolási típus deklarálása

57. ábra

A felsorolási típus értékeit azonosítók alkotják. Ezeket az azonosítókat soroljuk fel zárójelben. Pl. a hét napjainak típusa:

```
type naptipus= (hetfo,kedd,szerda,csutortok,pentek,
                szombat,vasárnap);
```

Itt a típusnév a "naptipus". A típusneveket a szabványos típusnevekhez hasonlóan változódeklarációkban használhatjuk:

```
var munkanap,szabadnap:naptipus;
```

Az így deklarált változóknak értéket adhatunk:

```
munkanap:=pentek;
szabadnap:=succ(munkanap);
```

relációs műveletekben használhatjuk,

```
if munkanap<pentek then
  ..................
```

A típus értékeinek rendezettségét a felsorolási sorrend határozza meg. Így pl. hetfo<szerda és vasarnap>pentek.

A felsorolási adattípusnál használt neveknek egyedieknek kell

lenni. Ez azt jelenti, hogy a következő deklaráció hibás

```
type havertipus=(Fero,Lacus,Tamás,Jeno,Jozsi,Denes);
     hazibuli=(Gyuri,Eszter,Jeno,Kata,Lacus,Mari,Tamas);
```

mert a Jeno és a Lacus értéket két különböző felsorolási típus is tartalmazza. A tilalmat indokolja, hogy az első deklarációból

```
Lacus<Jeno,
```

a másodikból

```
Jeno<Lacus
```

következne.

Három függvény is értelmezett a felsorolási típusokon: a PRED, SUCC és ORD.

pred(x) értéke x megelőzője
succ(x) értéke x rákövetkezője
ord(x) értéke x sorszáma a definiáló felsorolásban (0-val kezdve a számolást)

A tárban a megfelelő ord érték, azaz a 0-val kezdődő sorszám (a deklarációban meghatározott sorrend szerint) ábrázolja a felsorolási típusú értéket. Erről a 36. programmal könnyen meggyőződhetünk.

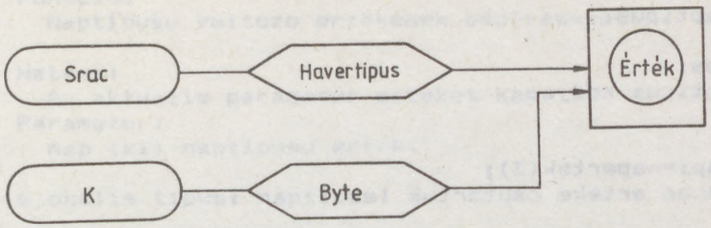
```
program felsabr;

{A felsorolasi típus ertekeinek gepi abrazolasa.}

  type havertipus=(Fero,Lacus,Tamas,Jeno,Jozsi,Denes)

  var srac: havertipus;
      k: byte absolute srac;
  begin
    clrscr;
    for srac:=Fero to Denes do
      writeln(k);
  end.
```

A felsorolási értékek ábrázolása

36. program

A program magyarázatra szorul, mert az absolute kulcsszót még nem használtuk.
Az absolute kulcsszóval a változók tárbeli elhelyezését

befolyásolhatjuk, illetve álneveket képezhetünk. A deklarációban a típusnév után írhatjuk az absolute szót majd egy tárcímet, vagy egy korábban már deklarált változó azonosítóját. Ekkor a deklarált változónk a tár adott helyén, vagy a másik változóval azonos tárcímen tárolja az értékét (58. ábra). Az absolute kulcsszóval a változó referencia komponensét határozzuk meg.



Álnév létrehozása

58. ábra

Az absolute kulcsszó kizárólag a Turbo Pascal eszköztárához tartozik.

A kedves Olvasóban bizonyára már hosszabb ideje mocorog a kérdés, mire használhatók ezek a furcsa értékek? Ha már a hét napjait pl. úgyis a 0, 1, ..., 6 számok ábrázolják a tárban, akkor minek ez a bonyolult nyelvi mechanizmus, mi is kódolhatnánk így. És ha a programban azt írnánk: 3, tudnánk, hogy ez a csütörtököt jelenti. Hát ez az. Ha ugyanakkor a havertípus értékeit is kódolnánk, meg a hónapokat is, akkor honnan tudnánk, hogy melyik 3-as jelent csütörtököt, melyik áprilist, melyik Jenot és melyik egyszerűen 3 egészt.
Ennek a nyelvi eszköznek éppen az a rendeltetése, hogy mentesítsen bennünket a kódolástól és - azáltal, hogy a program szövegében mindent az igazi nevén nevezhetünk - érthetőbbé, könnyebben olvashatóvá tegye a programot.
Bár léteznek függvények és műveletek, amelyek minden felsorolási típusra értelmezve vannak, beolvasni és kiírni nem tudjuk ezeket az értékeket. Ha szükségünk van rá, el kell készítenünk a megfelelő eljárásokat. Elkészíthetjük az ord függvénynek az adott felsorolási típusra vonatkozó inverz függvényét is. Ez egy a chr-hez hasonló típusváltó függvény lesz. Szabványos Pascalban nincs más lehetőségünk, de - amint majd látni fogjuk - Turbo Pascalban ezek a függvények automatikusan létrejönnek.
Készítsük el példaként a naptípus típusváltó függvényét (37. program).

```
function napertek(kod:integer):naptipus;
{
 Funkcio:
```

```
    Integer tipusu erteket a megfelelo naptipusu
    ertekke alakit at.

  Eredmeny:
    A fuggveny naptipusu erteket ad vissza.

  Parameter:
    kod (be), az ertek 0-val kezdodo sorszama.

  Globalis objektum:
    type naptipus.

  Hibajelzes:
    Nem naptipus kodja.

  Pelda:
    munkanap:=napertek(3);
    A munkanap erteke csutortok lesz.

  Megjegyzes:
    Az aktualis parameter erteke csak 0 es 6
    kozotti ertek lehet.
}
```

```
begin
function napertek(kod:integer):naptipus;
begin
  case kod of
    0: napertek:=hetfo;
    1: napertek:=kedd;
    2: napertek:=szerda;
    3: napertek:=csutortok;
    4: napertek:=pentek;
    5: napertek:=szombat;
    6: napertek:=vasarnap;
    else begin
      writeln('Nem NAPTIPUS kodja');
      halt;
      end;
  end;
end;
```

Típusváltó függvény

37. program

Az ord és napertek függvény kapcsolatát az

nap = napertek(ord(nap)),

ill. ha 0<=k<=6, akkor a

```
k = ord(napertek(k))
```

összefüggés mutatja.
A típusváltó függvénnyel könnyen elkészíthetünk egy adatbeolvasó eljárást (38. program).

```
procedure readnap(var nap:naptipus);
{
 Funkció:
   Naptipusu valtozo ertekenek beolvasasa.

 Hatasa:
   Az aktualis parameter erteket kap.
 Parameter:
   nap (ki) naptipusu ertek.

 Globalis tipus: naptipus.

 Igényelt alprogram: function napertek.

 Pelda:
   readnap(munkanap);
   A munkanap erteket a billentyuzetrol keri.

 Megjegyzes:
   Az ertek beirasat az eljaras iranyitja.
_____________________________________________________}

 var k: integer;
 begin
   writeln(' naptípusu ertek beolvasasa');
   write('1:hetfo,2:kedd,3:szerda,4:csutortok,');
   write('5:pentek,6:szombat,7:vasarnap');
   readln(k);
   nap:=napertek(k-1);
   end;
```

Beolvasó eljárás

38. program

Hasonlóképpen készíthetünk kiíró eljárást, ha szükséges.

A Turbo Pascalban tulajdonképpen nem is kell elkészitenünk a típusváltó függvényeket, mert minden deklarált felsorolási típushoz a fordítóprogram a típusváltó függvényt is elkészíti. A függvénynév azonos a típusnévvel. A

```
naptipus(4)
```

függvényhívás értéke pl. péntek.

A real kivételével minden típusazonosítót felhasználhatunk típusváltásra. Pl.

```
integer(kedd)
```

értéke 1.
Mivel a "napertek" függvény helyett a "naptípus" függvény is használható, ezzel a beolvasó eljárás használati feltételei is egyszerűbbek lesznek.

7.3 Intervallum típusok

A diszkrét típusú értékekből intervallum típusokat képezhetünk. Az intervallumot a határaival adjuk meg:

```
alsóérték..felsőérték.
```

A megfelelő típusazonosítót a type kulcsszó után definiálhatjuk. Pl.:

```
type kisegesztipus=1..100;
     munkanaptipus=hetfo..pentek;
     nagybetutipus='A'..'Z';
```

Az intervallum határai csak konstansok lehetnek.
Az intervallum típusok altípusok. Ez azt jelenti, hogy öröklik annak a típusnak a műveleteit, amelyből képeztük őket.
Az altípusokkal kapcsolatban megfogalmazzuk a következő kompatibilitási szabályt:

K5) Minden típus kompatibilis az altípusaival és egy adott típus különböző altípusai is kompatibilisek egymással.

Ha bekapcsoljuk az R compiler direktívát, akkor futás közben hibajelzést kapunk, ha egy intervallum típusú változónak az intervallumán kívül eső értéket akarunk adni.

A byte-típus szabványos intervallum típus, voltaképpen a

```
type byte=0..255;
```

deklarációnak felel meg.

7.4 A stringek

Az eddig megismert adattípusokat egyszerű vagy skalár adattípusoknak nevezzük. A stringek összetett adattípusok, a típust alkotó értékek összetett értékek.

String alatt olyan adattípust értünk, amelynek minden értékét maximált számú char típusú érték összessége alkotja.

A stringkonstans fogalmával már megismerkedtünk, egy karakterfűzért neveztünk stringkonstansnak.
A legfeljebb 20 betűs szöveget tartalmazó stringváltozók deklarálásához először a

```
type str20=string[20];
```

típusdeklarációra van szükség. A szögletes zárójelben lévő 20-as számmal határoztuk meg a típust alkotó jelsorozatok maximális hosszát. Az str20 típus értékhalmazát alkotó értékek az összes 20, 19, ..., 3, 2, 1 és 0 ASCII karakterből alkotott szöveg. (A 0 hosszúságú szöveg az üres szüveg, amit két egymás melletti felsővesszővel írhatunk le: ''.) A string maximális hosszát meghatározó egésztípusú konstans értéke 1 és 255 között lehet.

Az str20 típusazonosítóval deklarálhatunk azután változókat:

```
var cim, uzenet:str20;
```

A stringváltozókat pl. értékadásokban használhatjuk:

```
cim:='A stringek'

uzenet:='Varj ram a sarkon, Cucus!'
```

A stringváltozók mindig a maximális méretnek megfelelő helyet foglalják a tárban, plusz egy bájtot ami az aktuális hosszt tartalmazza. Tehát az str20 típusazonosítóval deklarált változók tárigénye 21 bájt. A 59. ábrán a cím tárbeli ábrázolását mutatjuk be, a bájtok tartalmát 10-es számrendszerben megadva.

Az "uzenet"-be írt érték hossza nagyobb 20-nál. Ilyenkor csak az első 20 jel tárolódik, ezért a valtozó értéke 'Varj ram a sarkon, C' lesz.

8	A	u	S	T	R	I	N	G

A string ábrázolása

59. ábra

A stringben tárolt szöveghez karakterenként is hozzáférhetünk. Ehhez az elérni kívánt karakter indexét kell megadni a változónév után szögletes zárójelben. Pl. a "cim" 5. karaktere

```
cim[5]
```

A 39. programmal ellenőrizheti az 59. ábrát.

```
program cimabrazolas;
  type str20=string[20];
  var i:integer;
      cim:str20;
  begin
    clrscr;
    cim:='A string';
    writeln(cim);
    writeln;
    for i:=0 to 20 do
      write(ord(cim[i]),' ');
    writeln;
  end.
```

A string gépi ábrázolása

39. program

A stringekkel stringkifejezéseket képezhetünk. A stringkifejezések stringkonstansokból, stringváltozókból, stringfüggvényekből stringműveletekkel építhetők fel.
A + jel a stringek körében a konkatenáció műveleti jele. A hangzatos név nem takar egyebet, mint, hogy két szövegből egyszerű egymás mellé írással egy egyesített szöveget hozunk létre. Pl. a

```
'Letepik'+' a '+'viragot'
```

konkatenáció eredménye a

```
'Letepik a viragot'
```

szöveg. Ha a konkatenáció eredményeként kapott string hossza nagyobb 255-nél, futás közbeni hibajelzést kapunk. A konkatenációt a később ismertetendő concat stringfüggvénnyel is végrehajthatjuk, de a + műveleti jel kényelmesebb.
A relációműveletek is értelmezettek a stringek halmazán. Ez azt jelenti, hogy a stringek rendezett halmazt alkotnak. Ez a rendezettség a stringet alkotó karakterek kódértékén alapuló lexikografikus rendezés. Ez azt jelenti, hogy két string közül az a "nagyobb", amelyikben balról jobbra haladva először találunk nagyobb kódú karaktert, mint a másikban. Ezért pl.

```
'vagtam' < 'vagtat'.
```

Ha két szöveg közül az egyik hosszabb, de a rövidebb utolsó jeléig azonosak, akkor a hosszabb a "nagyobb":

```
'Turbo' < 'Turbo Pascal'.
```

A relációműveletek a konkatenációnál kisebb prioritásúak. Ezért a

```
'Turbo' < 'Turbo'+' '+'Pascal'
```

kifejezésben nem kell zárójelet használni.
Mint említettük, a stringkifejezésekben stringfüggvényeket is használhatunk. A Turbo Pascalban a következő stringfüggvények szabványosak:

COPY

Hívása: copy(str,poz,db)

A copy függvény értéke az str paraméterrel megadott stringnek a poz pozíción kezdődő db számú karakterből álló részszövege. Pl. a

```
copy('mely fiu es leany is',6,12)
```

függvényhívás értéke a

```
'fiu es leany'
```

string. A poz és a db integer típusú kifejezések lehetnek. Ha poz értéke meghaladja a str string hosszát, akkor a függvény értéke az üres string lesz. Ha a megadott részstring túlnyúlik a str stringen, vagyis poz+db értéke nagyobb a string hosszánál, akkor a string végéig terjedő rövidebb részszöveg lesz a függvény értéke. Pl. a

```
copy('mely fiu es leany is',13,12)
```

értéke csak a

```
'leany is'
```

string.
Ha a poz paraméter értéke nincs 1 és 255 között, akkor futás közben hibajelzést kapunk.

CONCAT

Hívása: concat(str1,str2[,strn])

A paraméterként megadott függvények konkatenációja lesz a függvény értéke. Használata semmivel sem nyújt többet, mint a + műveleti jel.

LENGTH

Hívása: length(str)

A függvény egész típusú értéket ad vissza, a paraméterként megadott string hosszát. Pl. a

```
length('szine mint szemhejade');
```

függvényérték 21. A length(str) és az ord(str[0]) függvények azonos értéket adnak.

POS

Hívása: pos(str,reszstr)

A pos függvény végignézi az str paraméterben megadott stringet, hogy előfordul-e benne a reszstr részszövegként Ha igen, akkor az első előfordulás kezdőpozíciója lesz a függvény értéke, ha nem, akkor 0. A

```
pos('mint rebben a virag','virag')
```

függvényhívás eredménye 15, a

```
pos('ha szelben terdepel','virag')
```

értéke pedig 0.

Felhívjuk az Olvasó figyelmét arra, hogy függvény értéke string típusú érték is lehet. Ezt azért kellett kiemelnünk, mert más összetett értékről ezt nem mondhatjuk majd el.

Néhány szabványos stringkezelő eljárást is használhatunk.

DELETE

Hívása: delete(strg,poz,db)

Az eljárás törli az strg paraméterben megadott szövegből a poz-ban megadott sorszámú karaktertől kezdődő db hosszúságó részstringet. A poz és a db pareméterek integer kifejezések lehetnek, strg változóparaméter. Ha poz>length(strg), akkor strg értéke változatlan marad. Ha poz<=length(strg), de poz+db>length(strg), akkor string végét töröljük az strg[poz] karaktertől kezdve.
Ha poz értéke nem 1 és 255 közötti, akkor futás közben hibajelzést kapunk.

Példák:

Ha strg értéke 'oly felve rebben maris', akkor a

```
delete(strg,11,7)
```

hívás után 'oly felve maris', a

```
delete(strg,18,20)
```

után 'oly felve rebben' lesz.

INSERT

Hívása: insert(reszstr,strg,poz)

Az eljárás a reszstr-ben adott szöveget a poz karakterpozíciótól kezdődő részstringként az strg stringbe illeszti. Az strg stringváltozó, a reszstr stringkifejezés, poz integer kifejezés lehet.
Ha poz>length(strg), akkor konkatenáció történik. Ha az eredmény hosszabb, mint az strg változó maximális hossza, akkor csonkított stringet kapunk (a karakterfüzér jobb vége elveszik). Ha poz értéke az 1..255 tartományon kívül esik, akkor futás közben hibajelzést kapunk.

Példa:

Ha a reszstr értéke 'lila ', és strg='Letepik a viragot', akkor az

```
insert(reszstr,strg,11)
```

hívás után 'Letepik a lila viragot' lesz strg értéke.

STR

Hívása: str(ertek,strg)

Ezzel az eljárással integer és valós értékeket alakíthatunk stringekké. Az ertek paraméter formátumozott egész vagy valós kifejezés lehet, ahogy a write utasításba is írnánk. Az strg változóban a kifejezés értékét a megadott alaki feltételeknek is megfelelő string alakjában kapjuk meg.

Példák:

Legyen pl. k=2. Akkor a

```
str(11*k:4,strg)
```

hívás után strg értéke ' 22' lesz. Ha x értéke -2.2e-1, akkor a

```
str(x:8:5,strg)
```

strg-be a '-0.02200' string kerül.

VAL

Hívása: val(strg,valt,jel)

A val eljárással az strg stringet valós vagy egész értékké alakíthatjuk át (attól függően, hogy a valt változóparaméter milyen típusú). A kapott számértéket a valt változóban kapjuk meg. Az eljárás természetesen csak olyan stringeket

képes numerikus értékké alakítani, amelyek numerikus konstansok szintaktikusan helyes stringjei azzal a megszorítással, hogy sem a szám elején, sem a végén, nem lehetnek felesleges szóköz karakterek.
A valt paraméter integer vagy real típusú változó lehet. A jel paraméter helyébe csak integer típusú változót helyettesíthetünk. Ha az átalakítandó string szintaktikusan helyes, akkor a jel értéke a hívás után 0 lesz. Hiba esetén az első hibásnak észlelt jel pozícióját tartalmazza és ez esetben valt értéke definiálatlan marad.

Példák:

Ha az strg értéke '510', akkor a

```
val(strg,k,j)
```

hívás után k=510, j=0 lesz.
Ha strg='85 ', akkor az előző hívás j értékét 3-ra állítja.
Ha z valós változó és strg értéke '-2.2e-1', akkor z értéke -0.022 lesz, j pedig 0.

A string típusok és a char típus kompatíbilisek egymással, így kifejezésekben vegyesen használhatók. Stringváltozó bármikor kaphat értékül karaktert, de fordítva vigyázni kell. Karakter típusú változó csak olyan stringet kaphat értékül, amelynek hossza pontosan 1, egyébként hibajelzést kapunk futás közben.

Ha alprogramnak adunk át stringet változóparaméterként, akkor szigorú típusellenőrzés történik. Ez azt jelenti, hogy az aktuális és formális paraméterek hosszának meg kell egyezni. Ezért az ellenőrzésért a V compiler direktíva a felelős, amely alapértelmezés szerint bekapcsolt {$V+} állapotban van. Ha kikapcsoljuk, akkor nem okoz problémát a hosszban nem azonos aktuális paraméterek átadása.

7.5 Szövegfeldolgozás

Szöveges adatok feldolgozására a számítástechnikai alkalmazások számos területén van szükség. A Turbo Pascalban rendelkezésünkre álló string adattípus és a számos rendelkezésre álló eljárás és függvény lényegesen megkönnyíti az ilyen feladatok programozását.

Ha olyan stringkezelő alprogramokat akarunk írni, amelyeket különféle programokban használhatunk, célszerű egyetlen stringtípus használata mellett kikötni. Legyen a

```
strings
```

ez a típus (saját szabványos stringtípusunk), amelynek hosszát deklarált konstansnévvel állítjuk be:

```
const stringhossz=79;
```

```
type strings=string[stringhossz];
```

így összes stringünk hosszát a 79-es szám módosításával egyszerre megváltoztathatjuk. Azért tartjuk a 79-es értéket célszerűnek, mert a képernyő sorai 80 pozíciósak. Így még odafér a sor végére az újsor karakter is.

Ezek utánfogalmazzuk meg a feladatot: készítsünk olyan eljárást, amellyel egy szöveg valamely részszövegét minden előfordulási helyen egy meghatározott másik szöveggel cseréljük ki. Legyen az eljárás neve "helyettesit", paraméterei

1) az alapszöveg, amiben a helyettesítést elvégezzük "miben", ez nyilván változóparaméter (hiszen a megváltozott értéket vissza kell adnia a hívó programnak);

2) a részszöveg, amit ki akarunk cserélni "mit" értékparaméter;

3) az új szöveg: "mire", ugyancsak értékparaméter.

Az eljárásfej tehát a következő lesz:

```
procedure helyettesit(var miben:strings;
                      mit,mire:strings;);
```

Az eljárás algoritmusát a következőképpen fogalmazhatjuk meg:

```
repeat
  k:következő előfordulás kezdete(a "miben" szövegben
                                  a "mit" szövegnek)
  "mit" helyettesítése "mivel" k-tól kezdve
mignem nincs több előfordulás
```

A helyettesítendő szöveg előfordulásainak felkutatására célszerű lesz egy függvényt készíteni. Ehhez a pos függvényt nem tudjuk hatékonyan felhasználni, mert azzal csak az első előfordulást kapjuk meg. Viszont a szövegeket indexeléssel karakterenként is végignézhetjük, így a függvény ezen a módon megvalósítható.
Most a top-down módszernek megfelelően feltesszük, hogy létezik egy ilyen függvény. Legyen az azonosítója "kovetkezo", paraméterei:

1) "szoveg" stringkifejezés, amelyben az előfordulást keressük;

2) "resz" stringkifejezés, amit keresünk;

3) "honnan" integer kifejezés, annak a karakternek az indexe, ahonnan a keresést kezdjük.

A függvényérték típusa integer, értéke a részszöveg első előfordulásának kezdő pozíciója a "honnan" paraméterben megadott értéktől kezdve a keresést. Ha nem fordul elő, akkor legyen 0 a függvényérték.
A függvényfej a következő lesz:

```
function kovetkezo(szoveg,resz:strings;
                   honnan:integer):integer;
```

Ezzel a helyettesítési algoritmus programozható (40. program).

```
procedure helyettesit(var miben:strings;
                      mit,mivel:strings);
  var k: integer;
  begin
    k:=0;
    repeat
      k:=kovetkezo(miben,mit,k+1);
      if k>0 then begin
        delete(miben,k,length(mit));
        insert(mivel,miben,k);
        end;
    until k=0;
  end;
```

Részszövegek kicserélése

40. program

Még a "kovetkezo" függvényt kell elkészíteni (41. program).
A megadott pozíción kezdve a keresés a következőképpen történik:

```
i:= honnan
while nincs meg és van még do begin
 if nincs több then
   hol:=0
 else begin
   if i-től egyezik then
    hol:=i
   else
     i:=i+1
end
kovetkezo:=hol
```

A "nincs meg és van még" feltételt a "hol" értékével kifejezhetjük. Ugyanis a while ciklusban akkor kap értéket, ha egy előfordulás megvan, vagy ha kiderül, hogy nincs több. Ha tehát előzőleg végrehajtjuk a

```
hol:=-1
```

értékadást, akkor a

```
hol<0
```

feltétel megfelelő.
Akkor nincs több előfordulás, ha már a részszöveg hosszánál

közelebb értünk a szöveg végéhez, azaz ha

```
i> length(szoveg)-length(resz)+1.
```

Nehezebbnek látszik az "i-től egyezik" feltétel kiértékelése. Egy újabb ciklusban összehasonlítjuk a jeleket a szöveg i-edik és a részszöveg első karakterétől kezdve. A ciklus a részszöveg hosszáig fut. Előzőleg az egyezik feltételt true-ra állítjuk, s ha akármelyik karakternél eltérés van, false-ra állítjuk át:

```
j:=1
egyezik:=true
while egyezik és (j<=length(resz)) do
  if szoveg[i+j-1]=resz[j] then
    j:=j+1
  else
    egyezik:=false
```

A "szöveg" indexelésénél a -1 azért szükséges, mert az i-edik karaktertől indulunk, j kezdőértéke pedig 1.

Ezzel már összerakhatjuk az alprogramot.

```
function kovetkezo(szoveg,resz:string;
                   honnan:integer):integer;
  var hol,i,j:integer;
      egyezik:boolean;
  begin
    hol:=-1;
    i:=honnan;
    while hol<0 do
      if i> length(szoveg)-length(resz)+1 then
        hol:=0
      else begin
        j:=1;
        egyezik:=true;
        while egyezik and (j<=length(resz)) do
          if szoveg[i+j-1]=resz[j] then
            j:=j+1
          else
            egyezik:=false;
        if egyezik then
          hol:=i
        else
          i:=i+1;
      end;
    kovetkezo:=hol;
  end;
```

Részszövegek felkutatása

41. program

Ezzel a kitűzött feladatot megoldottuk. Használatkor ne feledkezzünk meg arról, hogy globális konstanst és típust kell deklarálnunk, továbbá arról sem, hogy a "kovetkezo" függvényt előbb kell deklarálni, mint a "helyettesit" eljárást. Az alprogramokat lemezen tároljuk 'kovetk.pas' és 'helyette.pas' nevű állományokban.

7.6 Feladatok

1) Készítsen kiíró eljárást a naptípus értékeihez!

2) Definiáljon adattípust a hónapokhoz. Használja a hozzá tartozó típusváltó függvényt a beolvasó eljárás elkészítéséhez!

3) Ha a hónapok adattípusát definiálta, készítse el a nyári hónapokhoz a megfelelő altípust.

4) Készítse el a

```
function binbyte(x:byte):str8
```

függvényfejjel a binbyte függvényt, amely a bájt típusú x érték bitképét (8 hosszúságú 0-kból és 1-esekből álló stringet)!

5) Írjon programot, amely különböző - adatként megadott - byte típusú értékek bitképét kiírja a binbyte függvény segítségével.

6) Egy k integer változó alsó bájtját a lo(k), felső bájtját a hi(k) függvény adja meg. Írjon programot, amely egészek bitképét szemlélteti!

7) Szemléltesse a 6. feladat programjával a 32767+1 összeadást!

8) Írjon eljárást, amely valamely adott számrendszerben megadott számból meghatározza a 10-es számrendszerbeli ábrázolást. Legyen az eljárásfej a következő:

```
procedure tizesre(masszam: strings;
                  alap:integer;
                  var tizes:integer;
                  var hiba:boolean);
```

Ha az alapszám nagyobb 10-nél, akkor az ábécé nagybetűit használjuk a számjegyek ábrázolására. A "hiba" értéke legyen true, ha a megadott szám hibás (az alapnál nagyobb értékű számjegyet tartalmaz).

9) Írjon függvényt amely 10-es számrendszerben megadott számot adott más számrendszerbe alakít. A függvényfej:

```
function masszam(tizes:integer;
                 alap:integer):strings;
```

10) Használja fel a 8. és 9. feladat alprogramjait olyan eljárás készítésére, amely bármilyen számrendszerben adott számot bármely számrendszerbe alakít. Az eljárásfej legyen:

```
procedure valtas(szam1:strings;
                 alap1,alap2:integer;
                   var szam2:strings;
                   var hiba:boolean);
```

11) Alakítsa át az előző alprogramokat úgy, hogy "tizedeseket" kezeljenek!

12) Készítsen alprogramot, amely egy szöveg betűit nagybetűkre változtatja!

13) Módosítsa a "kovetkezo" és a "helyettesit" eljárásokat úgy, hogy számolják a helyettesítések számát!

14) Módosítsa a "kovetkezo" és a "helyettesit" alprogramokat úgy, hogy a helyettesítendő szöveget akkor is felismerje, ha nagy- és kisbetűk is előfordulnak benne. Ha csupa nagybetűből áll a helyettesítendő szöveg, akkor ilyennel helyettesítse. Ha csak a kezdőbetű nagy, akkor nagy kezdőbetűs helyettesítést végezzen. Minden más esetben a helyettesítő szöveg kisbetűs legyen.

15) Készítsen eljárást, amely egy stringet megfordít. Ha hívás előtt 'BANYAI JANOS' a paraméter értéke, utána 'SONAJ IAYNAB lesz.

16) Használja a 15. feladat eljárását a palindrom négyzetszámok felkutatására az integer típusú értékek körében. Palindrom alatt általában olyan szöveget értünk, amely visszafelé olvasva is ugyanaz. Palindrom négyzetszám pl. 121.

17) Készítsen függvényt, amely két szöveget összehasonlít és 0-t ad vissza, ha nem talál eltérést, k-t, ha a k-adik karakterek különböznek először. A függvény ne tegyen kis- és nagybetűk között különbséget!

18) Írjon alprogramot quick index készítéséhez! Az alprogram egy adott szövegsort rendezzen át úgy, hogy egy adott résszöveggel kezdődjön (természetesen csak akkor, ha az adott résszöveg előfordul benne). A szöveg eredeti kezdetét * jelölje! Ha pl. az adott szöveg:

'Programozas Turbo Pascalban',

az adott résszöveg pedig 'Pascal', akkor a kapott string:

'Pascalban * Programozas Turbo'.

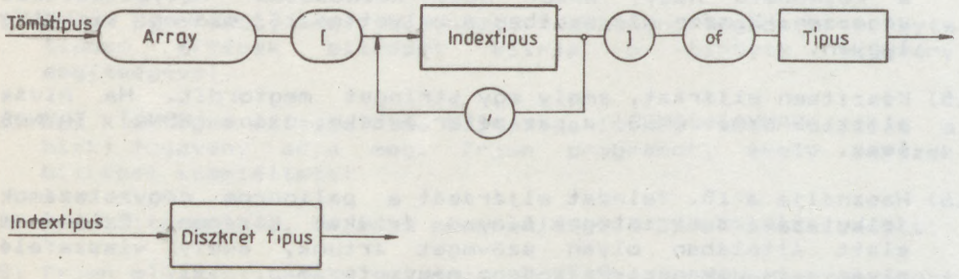
8. TÖMBÖK

8.1 A tömbök definiálása és ábrázolása

A stringekre emlékeztető, de általánosabb összetett adattípusok a tömbök.

> Tömb alatt olyan adattípust értünk, amelynek értékeit meghatározott számú, azonos típusú elem összessége adja.

A tömbben tárolt értékeket egyenként ugyanúgy indexeléssel érhetjük el, mint a stringeknél a stringet alkotó karaktereket. Az indexek diszkrét típusú kifejezések lehetnek. A tömb elemeinek közös típusát a tömb bázistípusának nevezzük. Az eddig megismert típusok bármelyike lehet tömb bázistípusa – még tömb is.
Tömböket az array kulcsszóval hozhatunk létre (60. ábra). Az indextípust szögletes zárójelben adjuk meg, ezután az of kulcsszót, majd a bázistípust írjuk. Ha több indextípust adunk meg, akkor többdimenziós tömböt hozhatunk létre. A többdimenziós tömbök elemeinek kiválasztásához több indexkifejezést kell meghatároznunk.



A tömbtípus szintaxisa

60. ábra

A tömbtípust típusdeklarációban és változódeklarációban használhatjuk (az utóbbi alkalmazást nem javasoljuk). Egy egészekből álló 100 elemű tömböt pl. a következőképpen hozhatunk létre:

```
const maxindex= 100;
type tombtipus= array[1..100] of integer;
var x,y,z: tombtipus;
```

Ezzel az x, y, z tömböket hoztuk létre.

Sokszor célszerű az indextípust is típusnévvel megadni:

```
const minindex= 1;
      maxindex= 100;
type  indextipus= minindex..maxindex;
      tombtipus= array[indextipus] of integer;
var x,y,z: tombtipus;
```

Az x, y és z tömbök 100 eleműek.
Sokféle tömbtípust definiálhatunk, mindegyikkel más és más értékhalmazt hozunk létre. Mindegyik értékhalmazt a bázistípus értékeinek a rendezett n-esei alkotják, ahol n az indextípus számossága. Ezt a

```
type ttip=array[1..3] of (A,B,C);
```

típussal szemléltetjük, melynek összes értékét az A,B,C elemekből álló összes 3 elemű sorozat alkotja:

```
A, A, A          A, A, B
A, A, C          A, B, A
A, B, B          A, B, C
A, C, A          A, C, B
A, C, C          B, A, A
B, A, B          B, A, C
B, B, A          B, B, B
B, B, C          B, C, A
B, C, B          B, C, C
C, A, A          C, A, B
C, A, C          C, B, A
C, B, B          C, B, C
C, C, A          C, C, B
C, C, C
```

A ttip tömbtípus értékhalmazát tehát a

$$(A,B,C)\times(A,B,C)\times(A,B,C) = (A,B,C)^3$$

Descartes-szorzat elemei képezik. Általában, ha a bázistípus értékhalmaza az S halmaz, az indextípus számossága pedig n, akkor a tömbtípust az

$$S^n$$

halmaz elemei alkotják. Ez az előbb definiált "tombtipus" esetében egy

$$65536^{100}$$

számosságú halmazt jelent. Ennyi különböző 100 elemű egész (bázis-) típusú tömb létezik.

Tömbkonstansokat nem használhatunk a programokban, erre a nyelv nem ad lehetőséget. Egyetlen vonatkozásban fordul elő ehhez hasonló eszköz a tömbváltozók kezdőértékének beállításánál. Ez a konstansdeklarációhoz hasonló. A Turbo Pascal terminológiában a kezdőértékadást "típusos konstans" definíciónak nevezik, de véleményünk szerint ez az elnevezés hibás és félrevezető. A

```
type t5tipus=array[1..5] of integer;
const x: t5tipus= (3,0,-2,1,7);
```

deklarációval egy x tömbváltozót (nem tömbkonstanst) definiáltunk, s egyben értéket is adtunk neki. Csak éppen nem a programvégrehajtás, hanem fordítás közben kap x értéket. A kapott értéket azonban (a konstansokétól eltérően) futás közben megváltoztathatjuk.

A tömbváltozók értékét megadhatjuk értékadó utasítással, pl.:

```
x:=y;
```

Ilyenkor az y tömb elemei átmásolódnak az x tömb megfelelő elemeibe. Gyakoribb azonban a tömbök elemenkénti feldolgozása. Az elemeket indexkifejezéssel határozzuk meg. Az indexkifejezést a tömbazonosító után szögletes zárójelbe tesszük, típusának a deklarációban adott indextípussal kell megegyezni.

Példák:

```
x[7], y[89], z[k], z[3*k-i], x[i div 3]
```

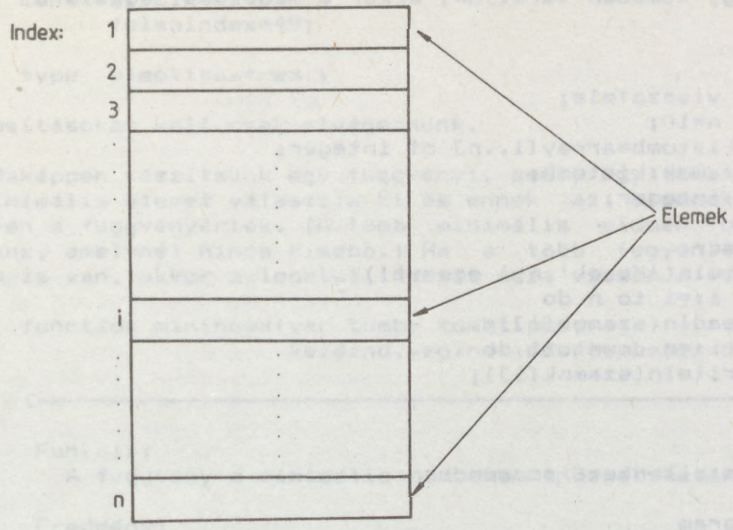
Feltesszük, hogy k és i integer típusúak. Ha az indexkifejezés aktuális értéke a deklarált tartományon kívül esik, csak akkor kapunk futás közben hibajelzést, ha az R compiler direktívát bekapcsoltuk. Mindig éljünk ezzel a lehetőséggel programbelövés közben, ha tömböket használunk. Egyébként nehezen derítjük fel programunk hibás működésének okát, ugyanis az index túlfutása miatt ilyenkor más változók értéke megy tönkre.

A tömbök a tárban elemenként, egybefüggő tárterületen helyezkednek el (61. ábra).

Az elemek által elfoglalt tárrész mérete is változó, az integer értékek 2, a valósak 6 bájtot foglalnak. A legkisebb indexű elem a legkisebb tárcímen helyezkedik el. Ez a tömb báziscíme. A tömb deklarálásával együtt automatikusan képezi a fordító az egyes elemek címét kiszámító címfüggvényt. Egydimenziós tömböknél a címfüggvény egyszerű:

```
B+(i-1)*h,
```

ahol B a báziscím, h a bázistípus értékeinek tárigénye bájtokban, i a tömbelem aktuális indexe.



Tömb ábrázolása a tárban

61. ábra

Az - egyébként ritkán használható - értékadáson kívül egyetlen elemi tevékenység sem létezik a nyelvben a tömbökhöz. Még a beolvasást és kiírást is programoznunk kell.

Eljárásnak és függvénynek lehet tömbparamétere, tehát írhatunk olyan alprogramokat, amelyek egy teljes tömbbel végeznek műveletet. Vigyáznunk kell azonban a kompatibilitásra. Csak az azonos típusú tömbök kompatibilisek. Nem azonos típusú pl. a következőképp deklarált x és y:

```
var x:array[1..100] of integer;
    y:array[1..100] of integer;
```

de igen, ha a deklaráció

```
var x,y:array[1..100] of integer;
```

Formális paramétereket - az ilyen bajokat megelőzendő - nem is szabad így deklarálni, csak típusnévvel. Célszerű ezért mindig típusneveket deklarálni és szokjunk hozzá, hogy változókat csak típusnévvel deklarálunk.

8.2 Programozás tömbökkel

Kezdjük egy olyan programmal, amit - bármennyire egyszerű is - tömbök nélkül nagyon nehézkesen tudnánk megírni. A feladat: olvassunk be 10 számot és írassuk ki ellenkező sorrendbe őket. Ha

a számokat egy tömbben tároljuk, akkor a megoldás egyszerű (42. program).

```
program visszafele;
  const n=10;
  type kistomb=array[1..n] of integer;
  var szamok:kistomb;
      i:integer;
  begin
    clrscr;
    writeln('Kerek',n,' egeszt');
    for i:=1 to n do
      readln(szamok[i]);
    for i:=n downto 1 do
      writeln(szamok[i]);
  end.
```

Kiíratás ellenkező sorrendben

42. program

A Pascal tömbjeivel kapcsolatosan komoly gondot jelent, hogy az elemek száma konstans, s ezt a számot tulajdonképpen már a program fordítása előtt megadjuk azáltal, hogy az index alsó és felső értékét rögzítjük. A program végrehajtásakor ezen már semmiképpen sem tudunk változtatni.
Ennek ellenére a tömbkezelő eljárásainkat és függvényeinket többnyire meg tudjuk írni elég általánosan, hogy különböző programokban is használhassuk. Arra kell törekednünk, hogy minél kevesebb előfeltételt építsünk az alprogramba, minden lehetséges információt paraméterekkel kell megadni. Ezért ne csak magát a tömböt adjuk át paraméterként, hanem az indexek alsó és felső határát is.
Globális objektumok - elsősorban globális típusok - használatával is növelhetjük az általánosságot. A legtöbb felhasználói programban ugyanis egyféle tömböt használunk. Alprogramjainkban erre a "tombtipus"-ra hivatkozunk és a bázistípust sem kell az alprogram írásakor okvetlenül rögzíteni. Használhatjuk a főprogramban deklarálandó "elemtipus"-t. Hasznát vehetjük egy globális "indextipus"-nak is. Ha pl. az alprogrammal egy 100 elemű egész tömböt akarunk feldolgozni, akkor a főprogramban a következő deklarációkat kell használni:

```
const alsoindex=1;
      felsoindex=100;

type  elemtipus=integer;
      indextipus=alsoindex..felsoindex;
      tombtipus=array[indextipus] of elemtipus;
```

Ha pedig egy 50 elemű valós tömböt akarunk használni, de 0-val akarjuk az indexelést kezdeni, akkor a

```
const alsoindex=0;
      felsoindex=49;

type  elemtipus=real;
```

módosításokat kell csak elvégeznünk.

Példaképpen készítsünk egy függvényt, amely egy tömb elemei közül a minimális elemet választja ki és ennek az elemnek az indexe legyen a függvényérték. (A tömb minimális elemén olyan elemet értünk, amelynél nincs kisebb.) Ha a több (egyenlő) minimális elem is van, akkor a legelső indexét adja vissza a függvény.

```
function minindex(var tomb: tombtipus;
                  kezdind,vegind:indextipus):indextipus;
{-------------------------------------------------------------

 Funkció:
   A függvény a minimális tömbelem indexét határozza meg.

 Eredmény:
   A függvény értéke a minimális elem indexe.

 Paraméterek:
   tömb (be): a vizsgálandó tömb,
   kezdind (be): a legkisebb index,
   vegind (be): a legnagyobb index.

 Globális típusok:
   indextipus: a használt tömb indextípusa,
   elemtipus: a tömb bázistípusa, értelemszerűen csak
              rendezett típus lehet,
   tombtipus: a tömb típusa.

 Példa:
   min:=a[minindex(a,51,100)]
   min értéke az a tömb 51-től a 100-adikig terjedő
   elemeinek a minimuma.

 Megjegyzés:
   Több minimális elem esetén a legelőször megtalált
   indexét kapjuk meg.

-------------------------------------------------------------}
 var minelem:elemtipus;
     i,minind:indextipus;
 begin
   minind:=kezdind;
   minelem:=tomb[minind];
   for i:=succ(kezdind) to vegind do
     if tomb[i]<minelem then begin
```

```
            minelem:=tomb[i];
            minind:=i;
          end;
        minindex:=minind;
      end;
```

Minimális tömbelem meghatározása

43. program

Talán észrevette az Olvasó, hogy a "tomb" paramétert változóparaméternek deklaráltuk. Ez elvileg felesleges, hiszen a tömböt nem változtatja meg az alprogram, és egyébként sem akarunk semmilyen értéket visszaadni. Viszont a tömbök sokszor nagy tárfoglalásúak és érték szerinti paraméterátadáskor az aktuális paraméter értéke átmásolódna az alprogram területére. Ezért a szükséges tárigény megduplázódna. Mindössze azért használunk változóparamétert, hogy ennek a kétszeres helyfoglalásnak elejét vegyük.

A minindex függvényt különféle főprogram környezetben használhatjuk. Először használjuk arra, hogy egy 100 nevet tartalmazó névsorból az ábécé szerinti első nevet kiválasszuk (44. program). Ehhez olyan tömbtípust kell létrehoznunk, amelyiknek stringek az elemei. Tegyük fel, hogy a minindex függvényt a 'minindex.pas' állományban találhatjuk meg az aktuális könyvtárban.

```
program elsonev;

  const alsoindex= 1;
        felsoindex=100;

  type  elemtipus= string[30];
        indextipus=alsoindex..felsoindex;
        tombtipus= array[indextipus] of elemtipus;

  var nev :tombtipus;
      elso:elemtipus;
        i:indextipus;

  {$I minindex.pas}

  begin
    clrscr;
    {adatbevitel}
    for i:=alsoindex to felsoindex do begin
      write(i,'. nev: ');
      readln(nev[i]);
      end;
    elso:=nev[minindex(nev,alsoindex,felsoindex)];
    writeln;
```

```
    writeln('A nevsorban az elso: ',elso);
  end.
```

Az első név megkeresése

44. program

Az adatbevitel újabb kérdéseket vet fel. Egy tömböt nem mindig azért deklarálunk 100 eleműnek, mert valóban 100 elemű. Az elemszámot általában úgy határozzuk meg, hogy az adott alkalmazás szempontjából elegendő eleme legyen. A feltöltött elemek száma nagyon sokszor kisebb a tömb méreténél. Ez azt jelenti, hogy - for ciklussal történő adatbevitelkor - megadjuk a tényleges adatmennyiséget. Vizsgáljuk, hogy ez nem több-e a deklarált tömbméretnél és ha igen, figyelmeztetést íratunk ki. Természetesen legfeljebb a tömb méretének megfelelő mennyiségű adatot viszünk be.

E feladatokhoz esetleg megéri egy beviteli eljárást írni (45. program).

```
procedure tombread(var tomb:tombtipus;
                   kezdind,vegind:indextipus;
                   hibajel:boolean);

{-------------------------------------------------------

 A kommentárok elkészítésére tisztelettel felkérjük
 az Olvasót.

-------------------------------------------------------}

  var i,v: indextipus;
      adatszam,meret:integer;

  begin
    clrscr;
    write('Kerem a beviendo adatok szamat: ');
    readln(adatszam);
    meret:=ord(vegind)-ord(kezdind)+1;
    if adatszam>meret then begin
      hibajel:=true;
      v:=vegind;
      writeln('Az adatok szama nagyobb a tombmeretnel');
      writeln;
      end
    else begin
      hibajel:=false;
      v:=indextipus(ord(kezdind)+adatszam-1);
      end;
    for i:=kezdind to v do begin
      write(ord(i),'. adat: ');
```

```
        readln(tomb[i]);
        end;
    end;
```

Általános beolvasó eljárás

45. program

Ez az eljárás minden lehetséges indextípust képes kezelni. A szabványos Pascalban nem tudtunk volna ilyen általánosan fogalmazni - hacsak meg nem követeltük volna a szükséges típusváltó függvény definiálását a főprogramban. Mivel a Turbo Pascalban a típusváltó függvények automatikusan rendelkezésre állnak, így az indextípus visszaalakítása nem okozott gondot.

Ha a bázistípus értékeit a readln eljárással nem tudjuk beolvasni, akkor a

```
readln(tomb[i])
```

utasítást a megfelelően megírt "elemread" eljárás

```
elemread(tomb[i])
```

hívására kell cserélni.

Az előzőleg megírt alprogramjainkat felhasználhatjuk a szerző megszégyenítésére, kimutatva, hogy a hét mely napján lustálkodott a legtöbbet (46. program).

```
program kismunka;

  type naptipus=(hetfo, kedd, szerda, csutortok,
                 pentek, szombat, vasarnap);
  const alsoindex=hetfo;
        felsoindex=szombat;
        napnev:array[naptipus] of string[9]=('hetfo',
               'kedd','szerda','csutortok','pentek',
               'szombat','vasarnap');

  type elemtipus=integer;
       indextipus=naptipus;
       tombtipus= array[indextipus] of elemtipus;

  var sorok:tombtipus; {a naponta megirt sorok szama}
      rossznap:indextipus;
      hiba:boolean;

  {$I minindex.pas}

  {$I tombread.pas}
```

```
begin
  tombread(sorok,alsoindex,felsoindex);
  rossznap:=minindex(sorok,alsoindex,felsoindex,hiba);
  writeln;
  writeln('A legrosszabb nap ',napnev[rossznap],'volt.');
  writeln('Akkor csak ',sorok[rossznap],' sort irtam.');
end.
```

A minimális hatékonyságú munkanap

46. program

Az Olvasó figyelmét felhívjuk a "napnev" tömb kezdőértékének meghatározására, és arra, hogy milyen szerepet játszik ez a tömb a felsorolási típus értékének megfelelő szöveg kiírásában.

Végezetül egy újabb probléma megoldásában mutatjuk be a minindex függvény használatát. Szövegek filológiai elemzésénél az egyik rutinvizsgálat az egyes betűk gyakoriságának a vizsgálata. A szöveget stringek tömbjeként ábrázoljuk. Tegyük fel, hogy rendelkezésünkre áll a "gyakorisag" nevű eljárás. Ez előállít egy tömböt, amely a betűgyakoriságokat tartalmazza, és az összes betű számát is megadja (a relatív gyakoriságok kiszámításához).

```
program betustatisztika;

type strings=string[79];

const alsoindex='A';
      felsoindex='B';

type elemtipus=real;
     indextipus=alsoindex..felsoindex;
     tombtipus=array[indextipus] of elemtipus;

var sor:strings;
    betuszam:tombtipus;
    osszesbetu:real;
    min,i:char;

{$I gyakoris.pas}

{$I minindex.pas}

begin
  osszesbetu:=0.0;
  for i:=alsoindex to felsoindex do
    betuszam[i]:=0.0;
  repeat
    readln(sor);
    gyakorisag(sor,betuszam,osszesbetu);
    until length(sor)=0;
  for i:=alsoindex to felsoindex do
```

```
    betuszam[i]:=betuszam[i]/osszesbetu;
  min:=minindex(betuszam,'A','Z');
  writeln('A legritkabban elofordulo betu "',min,'"');
  writeln('relativ gyakorisaga ',betuszam[min]:5:2);
  end.
```

A minimális betűgyakoriság

47. program

Az inputot tekintve a program nem tökéletes, hiszen egy ilyen feldolgozást valószínűleg mágneslemezen tárolt adatállománnyal végeznénk. Mivel ezt csak később tudjuk tárgyalni, itt meg kell elégednünk a szöveg sorainak egymás utáni begépelésével. A beolvasást végző repeat-until ciklus az üres stringre áll le (egyszerűen [ENTER]-t kell nyomnunk).
A program elején a betűk számlálását a számlálók nullázásával készítettük elő. A "gyakorisag" eljárástól azt várjuk el, hogy az aktuális paraméter betűinek számát hozzáadja az ugyancsak aktuális paraméterként átadott számlálókhoz. Az eljárásnak az üres stringet is helyesen kell tudni kezelni.

Általánosan használható - ún. generikus - alprogramokat körültekintően kell megfogalmazni. Az általánosságnak korlátai is lehetnek, sokszor maga az elvégzendő tevékenység korlátozza azon típusok számát, amelyekre érdemes tekintettel lenni. Eddigi példáink a generikus eljárások alkalmazhatóságának egy nagyon széles spektrumát mutatták be. Általában nem fogunk a továbbiakban ilyenfokó általánosságra törekedni.

8.3 Többdimenziós tömbök

A "tömbszerkesztés" egy lehetőség összetett érték létrehozására. Az összetett tevékenységek képzési szabályai nemcsak elemi tevékenységekre, hanem tetszőlegesen bonyolult összetett tevékenységekre is alkalmazhatók. Hasonló a helyzet a tömböknél is. Nemcsak egyszerű (skalár) típusokból hozhatunk létre tömböket, hanem összetett értékekből, azaz tömbökből is:

```
var t:array[0..4] of array[2..5] of integer;
```

Ez a tömbtípus olyan tömböket határoz meg, amelyek 5 eleműek. Az elemeket 0-tól 4-ig indexeléssel választjuk ki. Minden elem egy tömb, amelynek 2..5 az indextípusa, egészek az elemei (62. ábra).

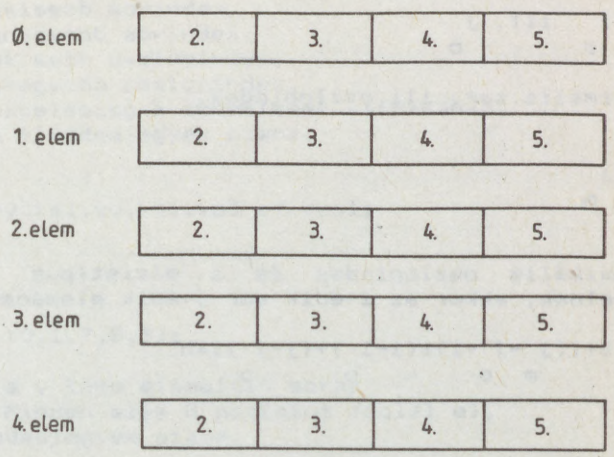
Egy ilyen tömb elemeire kettős indexeléssel hivatkozunk, pl.:

```
t[2][4].
```

Ez a jelölés meglehetősen nehézkes, ezért szívesebben tekintjük az ilyen strukturát kétdimenziós tömbnek, amelynek elemeit két indexszel (sor és oszlop) határozzuk meg (63. ábra).

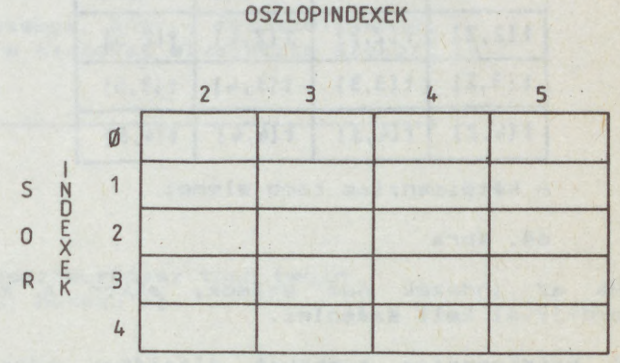
A kétdimenziós tömb deklarációjában két indextípust adunk meg. Az indextípusokat vesszővel választjuk el. Pl.:

```
t: array[0..4,2..5] of integer;

a: array[1..10,'a'..'z'] of boolean;

m: array[boolean,boolean] of boolean;
```



Tömbök tömbje

62. ábra



Kétdimenziós tömb

63. ábra

A kétdimenziós tömb elemeit két index megadásával tudjuk kiválasztani,

```
t[2,4]
```

jelenti a t tömb második sorának negyedik elemét. Az elemek logikai elhelyezkedését a 64. ábrán mutatjuk be.

A kétdimenziós tömb elemeit a tárban sorfolytonosan kell elképzelni. Ha B a báziscím, akkor a t[i,j] elem címe

```
B+(4*i+j-2)*2.
```

Általában, ha

```
i   ill. j
 o        o
```

a minimális sor, ill oszlopindex,

```
j
 m
```

a maximális oszlopindex és a bázistípus értékei h bájtot igényelnek, akkor az i-edik sor j-edik elemének címe

```
B+((j -j +1)*(i-i )+(j-j ))*h
     m  o        o      o
```

t[Ø,2]	t[Ø,3]	t[Ø,4]	t[Ø,5]
t[1,2]	t[1,3]	t[1,4]	t[1,5]
t[2,2]	t[2,3]	t[2,4]	t[2,5]
t[3,2]	t[3,3]	t[3,4]	t[3,5]
t[4,2]	t[4,3]	t[4,4]	t[4,5]

A kétdimenziós tömb elemei

64. ábra

Ha az indexek nem számok, akkor a megfelelő indexértékek ord-jával kell számolni.

A kétdimenziós tömböket általában kétszeres for ciklusokkal kezeljük. A 48. programmal sorokba és oszlopokba rendezetten írathatunk ki egy kétdimenziós valós elemű tömböt.

```
procedure tomb2write(var tomb:tombtipus;
                     mins,maxs:integer;
                     mino,maxo:integer;
                     hossz,tortresz:integer);

{-------------------------------------------------------------

 Funkcio:
   Valos elemu tomb kiiratasa.

 Parameterek:
   tomb (be) a kiiratando tomb,
   mins (be) a legkisebb sorindex,
   maxs (be) a legnagyobb sorindex,
   mino (be) a legkisebb oszlopindex,
   maxo (be) a legnagyobb oszlopindex,
   hossz (be) mezoszelesseg a tombelemek kiirasanal,
   tortresz (be) a tizedesjegyek szama.

 Globalis tipus:
   tombtipus= array[ks..vs,ko..vo] of real;


 Példa:

   tomb2write(v,1,10,1,7,8,3);

   A hivas kiirja a v tomb elemeit10 sorba
   es 7 oszlopba. Minden elem 8 poziciot foglal el,
   amibol 3 a tizedesjegyek szama.

 Hibajelzes:
   Ha nem fer el a kepernyo egy soraban a tomb egy
   sora, "A sor tul hosszu" hibajelzest kapjuk.

 Megjegyzés:
   A programozo felelossege, hogy a mezoszelesseget
   ugy adja meg, hogy a szamokat elválaszto szokozt is
   tartalmazza.

-------------------------------------------------------------}

const sorhossz=79;
var i,j:integer;

begin
  if (maxo-mino+1)*hossz>sorhossz then begin
    writeln('A sor tul hosszu');
    exit;
    end;
  for i:=mins to maxs do begin
     for j:=mino to maxo do
        write(tomb[i,j]:hossz:tizedes);
```

```
        writeln;
        end;
    end;
```

Kétdimenziós tömb kiíratása

48. program

Ezzel az eljárással nagyobb tömböket is kiírathatunk, amint ezt a 49. programban bemutatjuk (feltesszük, hogy a kiíró eljárást a 'tomb2w.pas' állomány tartalmazza).

```
program tombfeldolgozas;

  const sorkezd=1;
        sorveg=10;
        oszlkezd=1;
        oszlveg=20;

  type inds=sorkezd..sorveg;
       indo=oszlkezd..oszlveg;
       tombtipus=array[inds,indo] of real);

  var M:tombtipus;
  ..................
  ..................

  {$I tomb2w.pas}

  begin

  ..................
  ..................
  ..................
  ..................
  writeln('1 - 10 oszlopok');
  writeln;
  tomb2write(M,1,10,1,10,7,2);
  writeln;
  writeln('11 - 20 oszlopok');
  writeln;
  tomb2write(M,1,10,11,20,7,2);
  ..................
  ..................

  end.
```

Nagyobb tömb kiíratásának szervezése

49. program

A Pascal adatszerkezetek egy szép példájaként bemutatunk egy lehetőséget a sakkjáték leírására alkalmas objektumok létrehozására. A játékot egymást követő állások sorozataként írhatjuk le. Minden állást a sakkfigurák elhelyezkedésével adunk meg. A sakktábla kétdimenziós tömb, amelynek minden eleme (mezője) vagy üres, vagy valamilyen sakkfigurát tartalmaz. Az üres mezőt és a figurákat egyetlen adattípusnak tekintjük. Ezt a következő deklarációval definiáljuk:

```
type figuratipus= (ures, fgyalog, fhuszar, ffuto,
                   fbastya, fvezer, fkiraly, sgyalog,
                   shuszar, sfuto, sbastya, svezer,
                   skiraly);
```

A sakktábla sorait betűkkel, oszlopait számokkal szokás megadni:

```
type sorindtip=(a,b,c,d,e,f,g,h);
     oszlindtip=1..8;
```

Ezekután definiálhatjuk a játék állásainak leírására alkalmas típust és a megfelelő változót:

```
type allastipus=array[sorindtip,oszlindtip] of figuratipus;

var allas: allastipus;
```

A figurák mozgatását értékadásokkal szimulálhatjuk. Minden lépésnél az "allas" bizonyos elemei változnak meg. Ha pl. a világos vezér c2-ről h7-re lép, a következő értékadásokat kell leírni:

```
allas[c,2]:= ures;
allas[h,7]:= fvezer;
```

Természetesen a lépés végrehajthatóságát előzetesen vizsgálni kell. Meg kell nézni, hogy a c2-n valóban a világos vezér áll-e, végrehajtható-e fizikailag a lépés (azaz nem áll-e az útban valamilyen figura). Más kérdés - s ez már a játék stratégiájához tartozik -, hogy valóban ezt kell-e lépni. A lépés fizikai végrehajtását is célszerű eljárásra bízni. A feltételek meglétét az eljárástörzsben kell ellenőrizni.

A kétdimenziós tömbök mintájára három, sőt többdimenziós tömböt is deklarálhattunk. A gyakorlatban háromnál több dimenzióra ritkán van szükség.
A háromdimenziós tömböket meghatározott számú lap sorozatának képzeljük, ahol minen lap egy kétdimenziós tömb. Tekintsük pl a következő deklarációt:

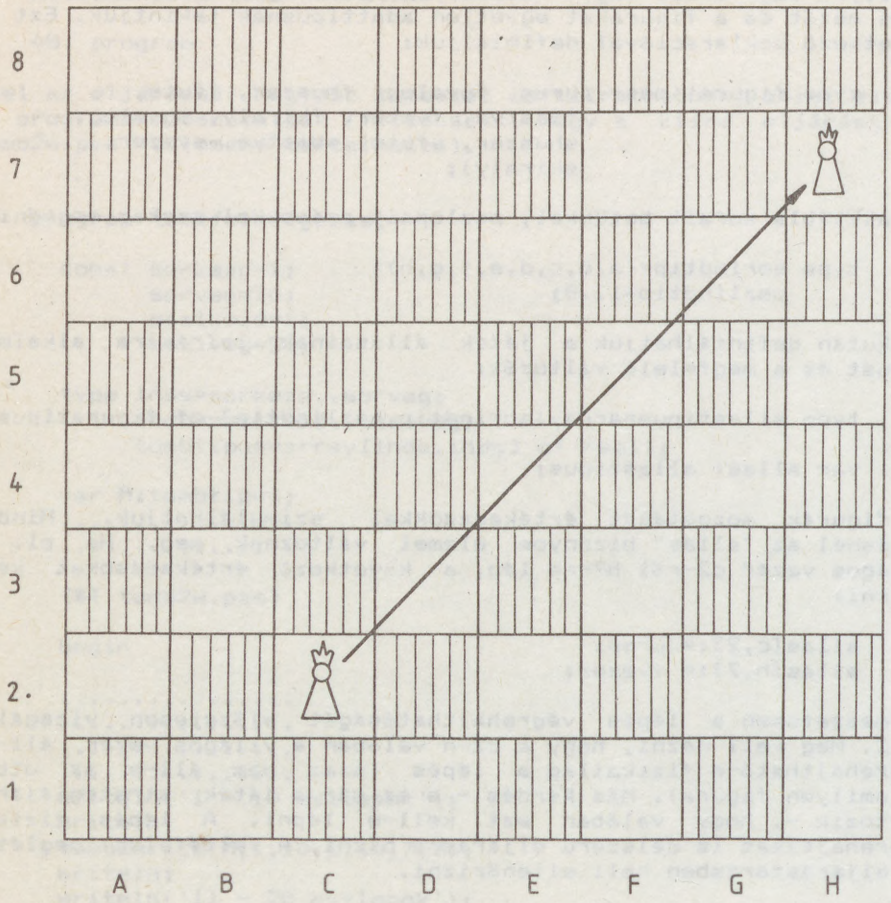
```
type tomb3tipus=array[1..4,1..3,1..3] of integer;

var tomb:tomb3tipus;
```

A tömb 3 lapból, minden lap 3 sorból, ill. 3 oszlopból áll (66.

ábra). Az indexek sorrendje mindig lap, sor, oszlop. A

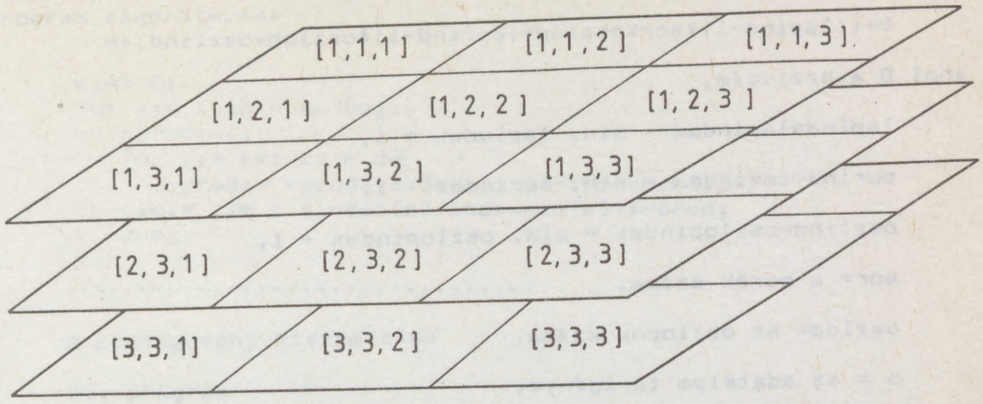
```
tomb[3,1,2]:=10;
```

értákadással a 3. lapon az 1. sor 2. elemébe írtunk 10-et.



A vezér lépése

65. ábra



Háromdimenziós tömb

66. ábra

A háromdimenziós tömb elemei a tárban lapfolytonosan helyezkednek el (67. ábra).

		tomb[1,1,1]	báziscím
	1. sor	tomb[1,1,2]	
		tomb[1,1,3]	
		tomb[1,2,1]	
1. lap	2. sor	tomb[1,2,2]	
		tomb[1,2,3]	
		tomb[1,3,1]	
	3. sor	tomb[1,3,2]	
		tomb[1,3,3]	
		tomb[2,1,1]	
	1. sor	tomb[2,1,2]	
		tomb[2,1,3]	
		tomb[2,2,1]	
2. lap	2. sor	tomb[2,2,2]	
		tomb[2,2,3]	
		tomb[2,3,1]	
	3. sor	tomb[2,3,2]	
		tomb[2,3,3]	
		tomb[3,1,1]	
	1. sor	tomb[3,1,2]	
		tomb[3,1,3]	
		tomb[3,2,1]	
3. lap	2. sor	tomb[3,2,2]	
		tomb[3,2,3]	
		tomb[3,3,1]	
	3. sor	tomb[3,3,2]	
		tomb[3,3,3]	

Háromdimenziós tömb a tárban

67. ábra

A háromdimenziós tömb egy elemének címfüggvénye:

B+((lapind-1)*sor*oszlop+(sorind-1)*oszlop+oszlind)*h

ahol B a báziscím,

lapind=lapindex - min. lapindex + 1,

sorind=sorindex - min. sorindex + 1,

oszlind=oszlopindex - min. oszlopindex + 1,

sor= a sorok száma,

oszlop= az oszlopok száma,

h = az adatelem tárigénye.

Ezt a formulát könnyen általánosíthatjuk n-dimenziós tömbökre. Legyen

i_k a k-adik dimenzió indexe,

i_{k_o} a k-adik index minimális,

i_{k_m} a k-adik index maximális értéke.

Legyen továbbá $r_k = i_{k_m} - i_{k_o} + 1$, a k-adik dimenzió számossága,

akkor az $i_1, i_2, \ldots, i_k, \ldots, i_n$ indexekkel meghatározott elem címe:

$$B+((i_1-i_{1_o})*r_2*\ldots*r_n+(i_2-i_{2_o})*r_3*\ldots*r_n+(i_n-i_{n_o}))*h =$$

$$B + h*\sum_{1}^{n}(i_k - i_{k_o})\prod_{k+1}^{n} r_j$$

Talán az Olvasó számára a matematikai képletnél világosabb a 50. program algoritmusa.

```
cim:=0;
for k:= 1 to n do begin
   rprod:=1;
   for j:= k+1 to n do
      rprod:= rprod*(indexmax[j]-indexmin[j]-1;
   cim:= cim + (index[k]-indexmin[k])*rprod;
   end;

.....
```

A címfüggvény kiszámítása

50. program

8.4 Mátrixok és vektorok

A matematikában a rendezett szám n-eseket (n elemű) vektoroknak nevezzük. A rendezettség azt jelenti, hogy pl. a

(2,0,-3,1) és a (0,2,-3,1)

vektorok különbözők, két n elemű vektor akkor egyenlő, ha a megfelelő elemeik rendre egyenlőek.
Az n elemű vektorok halmazán műveleteket definiálhatunk. Két vektor összegén, ill. különbségén a megfelelő elemek összegéből, ill. különbségéből álló vektort értjük. Két vektor skalárszorzata a vektorok megfelelő elemeinek szorzatösszege. A skalárszorzat nem igazán vektorművelet, hiszen eredménye nem vektor.
A számítástechnikában a vektorokat egydimenziós tömbökkel ábrázoljuk, amelyek indextípusa 1..n, bázistípusa real. Így elegendő a vektorkezelő alprogramokban a vektor azonosítója mellett a méretét (n értékét) megadni paraméterként. A vektorok összeadását a 51., kivonását az 52. program definiálja.

```
procedure vektorosszeg(var x,y,osszeg:vektortipus;
                       meret:integer);
  i:integer;

  begin
    for i:= 1 to meret do
      osszeg[i]:= x[i] + y[i];
    end;
```

Vektorok összege

51. program

```
procedure vektorkivonas(var x,y,kulonbseg:vektortipus;
                                meret: integer);
  i:integer;

  begin
    for i:= 1 to meret do
      kulonbseg[i]:= x[i] - y[i];
    end;
```

Vektorok különbsége

52. program

Az x és y vektorok skalárszorzatat az 53. program állítja elő.

```
function skalarszorzat(var x,y:vektortipus;
                               meret:integer):real;
  var i:integer;
      s:real;

  begin
    s:=0.0;
    for i:=1 to meret do
       s:=s+x[i]*y[i];
    skalarszorzat:=s;
    end;
```

Skalárszorzat

53. program

A mátrixok a matematikában számtáblázatok, amelyekkel - ugyanúgy, mint a vektorokkal - műveleteket végezhetünk. A mátrix egy elemét két indexszel (sor- és oszlopindexszel) határozzuk meg. Egy 3*3-mas mátrixot (amely 3 sorból, ill. 3 oszlopból áll) a következőképpen adhatunk meg:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Két mátrix összegén, ill. különbségén a megfelelő elemek

összegéből, ill. különbségéből álló mátrixot értjük. Beszélhetünk mátrixok szorzatáról is. Az A és B mátrixok

C = A x B

szorzatát csak akkor értelmezzük, ha az A mátrix sorainak száma egyenlő a B mátrix oszlopainak számával. Ilyenkor a C szorzatmátrix c_{ij} eleme az A mátrix i-edik sorának és a B mátrix j-edik oszlopának a skaláris szorzata. A mátrixműveleteket az 54. 55. és 56. programok definiálják.

```
procedure marixosszeg(var A,B, osszeg:matrixtipus;
                      sor,oszlop:integer);
  var i,j:integer;

  begin
    for i:= 1 to sor do
      for j:= 1 to oszlop do
        osszeg[i,j]:= A[i,j] + B[i,j];
    end;
```

Mátrixok összegzése

54. program

```
procedure matrixkivonas(var A,B, kulonbseg:matrixtipus;
                        sor,oszlop:integer);
  var i,j:integer;

  begin
    for i:= 1 to sor do
      for j:= 1 to oszlop do
        kulonbseg[i,j]:= A[i,j] - B[i,j];
    end;
```

Mátrixok kivonása

55. program

```
procedure matrixszorzas(var A,B,szorzat:matrixtipus;
                        sor1,oszl1,sor2,oszl2:integer);
  var i,j,k:integer;
      z:real;

  begin
    if oszl1<>sor2 then begin
```

```
        writeln('Hiba! Kiserlet nem komformabilis');
        writeln('matrixok szorzasara');
        halt;
        end
      else begin
        for i:=1 to sor1 do
          for j:= 1 to oszl2 do begin
            z:=0.0;
            for k:=1 to oszl1 do
              z:=z+A[i,k]*B[k,j];
            szorzat[i,j]:=z;
            end;
       end;
```

Mátrixszorzás

56. program

A mátrixszorzás programjában a z segédváltozót a szorzatmátrix elemei helyett használjuk az összegzéshez. Így megtakarítjuk a címfüggvény többszöri kiértékelését.
A mátrixok közül kiemelnénk az nxn-es típusú ún. (n-edrendű) négyzetes mátrixokat. Ezek körében a szorzás korlátlanúl elvégezhető, s a szorzat is nxn-es mátrix. A szorzás asszociatív, de nem kommutatív művelet, azaz

$$Ax(BxC) = (AxB)xC$$

mindig teljesül, de általában

$$AxB <> BxA.$$

A szorzásnak van egységeleme az n-edrendű négyzetes mátrixok körében. Az egységmátrixban minden elem 0, kivéve azokat, amelyek sor és oszlopindexe egyenlő (főátlóbeli elemek). Ezek 1-esek. Pl. a negyedrendű egységmátrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Az egységmátrix jele E. Mátrixokkal osztani ugyan nem lehet, de beszélhetünk egy mátrix inverzéről. Ha valamely A mátrixhoz létezik olyan A' mátrix, hogy

$$AxA' = A'xA = E,$$

akkor az A' mátrixot az A inverzének nevezzük. Nincs minden
-1
mátrixnak inverze. Ha egy A mátrix inverze létezik, A jelöli.
A mátrixokat a gyakorlatban pl. a többismeretlenes lineáris egyenletrendszerek megoldásánál használják. Egy

$$a_{11} y_1 + a_{12} y_2 + a_{13} y_3 + a_{14} y_4 = c_1$$

$$a_{21} y_1 + a_{22} y_2 + a_{23} y_3 + a_{24} y_4 = c_2$$

$$a_{31} y_1 + a_{32} y_2 + a_{33} y_2 + a_{34} y_4 = c_3$$

$$a_{41} y_1 + a_{42} y_2 + a_{43} y_3 + a_{44} y_4 = c_4$$

négyismeretlenes egyenletrendszert mátrixszorzással az

$$A \times y = c$$

alakban írhatunk, ahol

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

az egyenletrendszer együtthatómátrixa,

$$y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

az ismeretlenekből álló 4x1-es mátrix (vagy 4-elemű oszlopvektor),

$$c = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix}$$

a konstansok vektora. Ha az A mátrix invertálható, akkor az egyenletrendszer megoldását az inverzzel való szorzással megkaphatjuk:

$$A^{-1} \times A \times y = A^{-1} \times c,$$

de a bal oldalon

$$A^{-1} \times A \times y = E \times y = y,$$

így

$$y = A^{-1} \times c.$$

A mátrix invertálásának, ill. az egyenletrendszerek megoldásának technikai részleteit nem áll módunkban tárgyalni. Erről a témáról a lineáris algebrával és a numerikus analízissel foglalkozó könyvekben olvashatunk.

8.5 Speciális és ritka mátrixok

A műszaki gyakorlatban sokszor kell olyan mátrixokkal dolgozni, amelyek különleges szerkezetűek. Ilyenek pl. a szimmetrikus mátrixok. Egy S szimmetrikus mátrixokat a főátlóra való tükrözés nem változtatja meg: bármely i, j indexpárral teljesül az

$$s_{ij} = s_{ji}$$

egyenlőség. Ez a helyzet pl. az

$$S = \begin{bmatrix} 3 & -4 & 0 \\ -4 & 5 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

mátrixnál. Felesleges ilyenkor a kétszer előforduló elemeket kétszer tárolni, elegendő, ha csak a

```
 3
-4   5
 0   1   0
```

"háromszöget" tároljuk. Ennek különösen a valóságos problémákban előforduló nagyméretű mátrixoknál van nagy jelentősége, hiszen

$$n^2 * h \quad \text{helyett} \quad \frac{n*(n+1)}{2} * h$$

a tárigény (h a bázistípus értékeinek tárfoglalása). Ugyanakkor azt szeretnénk, hogy úgy kezelhessük ezt a mátrixot, mintha az összes eleme jelen volna a szokásos elrendezésben.
Az elemeket egy sv egydimenziós tömbbe írhatjuk be, pl. sorfolytonosan:

```
sv[1]    3      S[1,1]
sv[2]   -4      S[2,1] = S[1,2]
sv[3]    5      S[2,2]
sv[4]    0      S[3,1] = S[1,3]
sv[5]    1      S[3,2] = S[2,3]
sv[6]    0      S[3,3]
```

Az így leképezett kétdimenziós tömb elemeit kényelmes egy S(i,j) függvényhívással meghatározni, igy csaknem a megszokott szintaxissal dolgozhatunk a helytakarékosan tárolt szimmetrikus mátrixszal, mintha minden trükk nélkül ábrázolnánk.
Szükségünk van ehhez egy

```
k = L(i,j)
```

függvényre, amely az eredeti i, j indexepárt az sv tömb k indexére képezi le. Az L függvény argumentumaira előírjuk az

```
i >= j
```

feltételt (ami szükség esetén i és j felcserélésével biztosítható). Könnyű belátni, hogy k értéke a megelőző sorokban álló elemek összege plusz j:

$$k = 1 + 2 + 3 + \ldots + (i-1) + j,$$

ahol az első i-1 természetes szám összegét zárt alakban felírva a

$$k = \frac{i*(i-1)}{2} + j$$

összefüggést kapjuk.
Ezután az S(i,j) függvény megírása nem jelent gondot (57. program).

```
function S(sor,oszlop:integer):real;
  var t:integer;
  begin
    if sor<oszlop then begin
      t:=sor;
      sor:=oszlop;
      oszlop:=t;
      end;
    S:= sv[sor*(sor+1) div 2 +oszlop];
    end;
```

Szimmetrikus matrix elemeinek kiolvasása

57. program

Az S függvény használatának feltétele, hogy az sv globális tömbben sorfolytonosan legyenek elhelyezve az eredeti S mártix "alsó háromszögének" elemei. Az sv tömb kezdő indexe 1, az elemek száma n*(n+1)/2 (ahol n az S mátrix rendje).
Ha egy szimmetrikus mátrixszal pl. egy v vektort szorzunk meg, akkor a következőképpen használjuk a mátrix helyett az S függvényt:

```
.....................
.....................

for i:= 1 to n do begin
  z:=0.0;
  for j:= 1 to n do
    z:= z+ S(i,j)*v[j];
  p[i]:= z;
  end;

.....................
.....................
```

Szorzás szimmetrikus mátrixszal

58. program

A előzőhöz hasonló feladatot jelent az alsó, ill. a felső háromszögmátrixok ábrázolása. Az alsó háromszögmátrixban a főátló feletti, a felsőnél a főátló alatti elemek zérusok. Alsó háromszögmátrix pl.:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 1 & -2 & 0 & 0 \\ 3 & 0 & -1 & 0 \end{bmatrix}$$

Az ismert nullákat felesleges tárolni. Az előző S-hez hasonló A függvény most minden i<=j-re 0-t ad vissza, az alsó háromszög elemeit pedig egy sv tömbből olvashatjuk ki (59. program).

```
function A(sor,oszlop:integer):real;
  begin
    if sor<=oszlop then
      A:=0.0
    else
      A:=sv[(i-1)*(i-2) div 2 +j];
    end;
```

Alsó háromszögmátrix elemének visszanyerése

59. program

Hasonlóan kezelhetjük a felső háromszögmátrixokat is.

Az olyan mátrixokat, amelyekben viszonylag kevés zérustól különböző elem fordul elő, ritka mátrixoknak nevezzük. Egy tipikus ritka mátrix:

$$\begin{bmatrix} 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 & -3 & 0 & 0 & 0 & 0 \end{bmatrix}$$

A mátrix 63 eleméből csak 9 különbözik 0-tól. A nullákat felesleges tárolni, csak az a baj, hogy a nemzérus elemek elhelyezkedésében semmi szabályosság nem látszik.
Egyetlen megoldásnak az látszik, ha a nullától különböző elemek mellett azok indexeit is tároljuk. Így a példában szereplő mátrixot az 68. ábrán láthatóan ábrázolhatjuk.

1	1	1
1	4	2
2	2	3
2	8	-1
5	3	1
5	6	5
6	8	2
6	9	2
7	5	-3

sor osz- elem
     lop

Ritka mátrix tömbábrázolása

68. ábra

Ez egy kx3-mas tömb, ahol k a mátrix nullától különböző elemeinek száma. Ehhez a tárolási szisztémához kicsit nehézkesebb egy R függvényeljárást szerkeszteni, amelyik az R(i,j) hívásnál visszaadja az i-edik sor j-edik elemét, de nem lehetetlen. A 68. ábrán látható tömb sorait kell végignézni, hogy megtalálható-e benne az i, j értékpár. Ha igen, akkor a 3. érték, ha nem, akkor 0 a megfelelő mátrixelem.
Ha a ritka mátrix real bázistípusú, akkor két tömbre van szükség. A kx2-es indextömbre és egy k elemű valós vektorra. Ha az indextömb valamely q-adik sora tartalmazza az i, j értékpárt, akkor a vektor q-adik eleme, egyébként 0.0 a mátrixelem. Legyen it az indextömb, et az elemtömb azonosítója. Az R függvény egy lehetséges megvalósítását az 60. programban mutatjuk be.

```
function R(sor,oszlop:integer):real;
  var q:integer;
      megvan:boolean;
```

```
begin
  megvan:=false;
  R:=0.0;
  q:=1;
  while it[q,1]<sor do q:=q+1;
  while (it[q,1]=sor) and not megvan do
    if it[q,2]=oszlop then begin
      R:=et[q];
      megvan:=true;
      end
    else
      q:=q+1;
  end;
```

Ritka mátrix elemének meghatározása

60. program

A program helyes működéséhez "strázsát" kell állítanunk az it tömb végére. A strázsa a maximális sorindexnél nagyobb érték lehet, tehát a példában 8 már megfelelő, de legyen pl. 99 (69. ábra).

q	it		et
1	1	1	1.0
2	1	4	2.0
3	2	2	3.0
4	2	8	-1.0
5	5	3	1.0
6	5	6	5.0
7	6	8	2.0
8	6	9	2.0
9	7	5	-3.0
10	99	0	

Az indextömb lehatárolása strázsával

69. ábra

Ha ugyanis a keresett sorindex pl. 7 és a 7. sorban nincs zérustól különböző elem, nem fejeződne be a

```
while it[q,1]<sor do;
```

keresőciklus. A strázsára azonban megáll, s mivel 8>sor minden sorindexre igaz, a második while ciklus egyszer sem hajtódik végre, s a függvény az ez esetben helyes 0.0 értéket adja vissza.

Az ennél az algoritmusnál időigényes sorindex szerinti keresést megtakaríthatjuk, ha egy rt referenciatömböt vezetünk be a sorindexek szerint (70. ábra).

sor	rt		oit	q	et
1	1	2	1	1	1.0
2	3	2	4	2	2.0
3	0	0	2	3	3.0
4	0	0	8	4	-1.0
5	5	2	3	5	1.0
6	7	2	6	6	5.0
7	9	1	8	7	2.0
			9	8	2.0
			5	9	-3.0

Ábrázolás referenciatömbbel

70. ábra

Most csak az oszlopindexeket kell indextáblában tárolni (oit). Az rt tömb sorindexe azonos az ábrázolt ritka mátrix sorindexével. Ha pl. a 2. sor j-edik elemét kívánjuk kiolvasni, akkor rt 2. sorának első eleme mutatja, hogy oit-ben a 3. elemmel (q értéke) kezdődnek a ritka mátrix nemzérus értékeit meghatározó oszlopindexek, az rt[2,2] 2 értéke pedig azt jelenti, hogy e sorban 2 db nullától különböző elem van (és így oit-ben két oszlopindex található). Ha e két index valamelyike j-vel egyenlő, akkor a megfelelő et[q] a ritka mátrix keresett eleme. Ha nincs ilyen, akkor 0.0. Így működik az 61. program.

```
function R(sor,oszlop:integer):real;
  var q,qk,qv:integer;
      megvan:boolean;
```

```
}
begin
  megvan:= false;
  R:=0.0;

  {oszlopindex kereseshez oit-beli
   kezdoindex beallitasa}

  qk:= rt[sor,1];

  {ha az adott sorban nincs nemzerus elem,
   0.0-val visszater}

  if qk=0 then exit;

  {vegindex beallitasa es kereses}

  qv:=qk + rt[sor,2] -1;   q:=qk;
  while (q<=qv) and not megvan do
    if oit[q]=oszlop then begin
      R:=et[q];
      megvan:=true;
      end
    else
      q:=q+1;
  end;
```

A mátrixelem visszanyerése

61. program

Gondolja meg az Olvasó, hogy az rt tömb második oszlopára voltaképpen nincs szükség, a q végértékét enélkül is meghatározhatjuk az első oszlopban tárolt értékekből. Hogyan?

8.6 Két alkalmazás

Egy üzemben m-féle alapanyagból n-féle terméket állítanak elő. Ismeretes a termékek fajlagos anyagigénye, amit a T technológiai mátrixban adunk meg:

$$T = \begin{bmatrix} t_{11} & t_{12} & \dots & t_{1n} \\ t_{21} & t_{22} & \dots & t_{2n} \\ . & . & \dots & . \\ t_{m1} & t_{m2} & \dots & t_{mn} \end{bmatrix}$$

ahol a

$$t_{ij}$$

elem megadja, hogy a j jelű termék egységnyi mennyiségének előállításához mennyit kell az i anyagból felhasználni. A napi termelési programot egy p programvektorral határozzuk meg. A vektor j-edik eleme a j termékből gyártandó mennyiség. Kérdés, hogy az adott gyártási program megvalósításához mekkora az anyagigény.
Az anyagigényt egy m elemű a tömbben akarjuk megkapni, amelynek i-edik eleme az i anyag mennyiségét tartalmazza.
Könnyű belátni az

$$a = T \times p$$

összefüggés helyességét. Ellenőrizzük a következő konrét példán. Legyen

$$T = \begin{bmatrix} 3 & 0 & 1 \\ 1 & 2 & 2 \\ 2 & 3 & 0 \\ 1 & 1 & 1 \end{bmatrix} \qquad p = \begin{bmatrix} 15 \\ 20 \\ 10 \end{bmatrix}$$

Ekkor az 1. anyagból szükséges mennyiség:

az 1. termékhez 15*3,
a 2. termékhez 20*0,
a 3. termékhez 10*1,

azaz az a vektor első eleme

15*3 + 20*0 + 10*1.

Ugyanezt kapjuk a mátrixszorzással is.

A program szerkezete egyszerű:

adatbeolvasás
mátrixszorzás
eredménykiírás

A technológiai mátrix adatai állandók, ezeket nem kívánjuk újra és újra beírni. A T tömbnek kezdőértéket adunk egy const deklarációban (ekkor a megadott értékek a fordítás során beépülnek a programba). Csak a programvektor elemeit kell beolvastatni. A szorzáshoz használjuk fel a 8.4 alfejezetben megadott matrixszorzás eljárást, tegyük fel, hogy az

'mszorzas.pas' állományban találjuk. A feladatot a 62. program oldja meg.

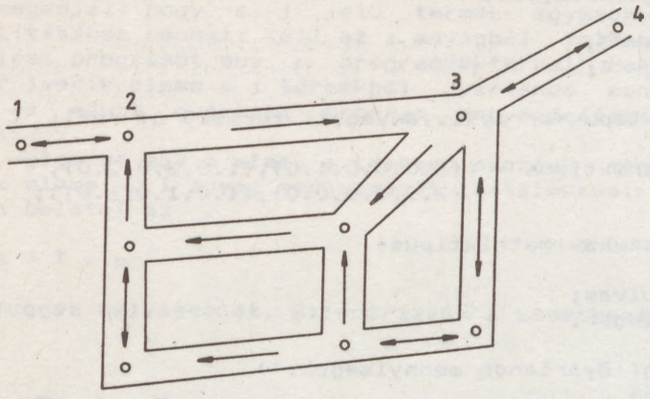
```
program anyagszukseglet;

  const termek=3;
        anyag= 4;

  type matrixtipus=array[1..anyag,1..termek] of real;

  const T:matrixtipus = ((3.0,0.0,1.0),(1.0,2.0,2.0),
                         (2.0,3.0,0.0),(1.0,1.0,1.0));

  var progv,szuksv:matrixtipus;

  procedure olvas;
    var j:integer;
    begin
      writeln('Gyartando mennyisegek:')
      writeln;
      for j:=1 to termek do begin
        write(j,'-edik                              termekbol:');
readln(progv[j]);
      end;
    end; {olvas}

  {$I mszorzas.pas}

  procedure kiir;
    var i:integer;
    begin
      writeln('Anyagszukseglet:');
      writeln;
      for i:=1 to anyag do
        writeln(i,'-edik anyagbol:',szuksv[i]:8:2);
    end; {kiir}

  begin
    clrscr;
    olvas;
    matrixszorzas(T,progv,szuksv,anyag,termek,termek,1);
    kiir;
    end.
```

Anyagszükséglet számítás

62. program

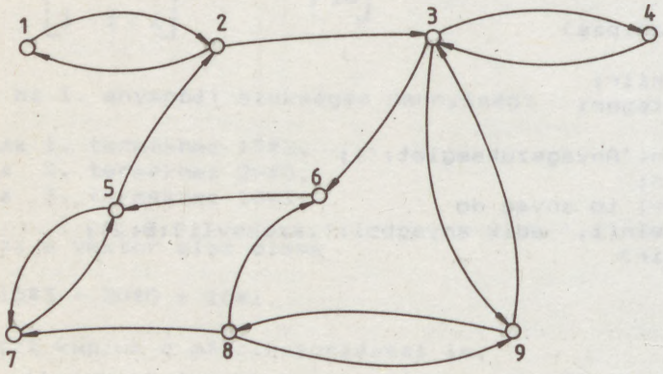
A program jól szemlélteti, hogy a "könyvtári" eljárásokat néha csak némi áldozat árán tudjuk használni. Ezt az áldozatot most a matrixtípus egységes használatával hoztuk, ami miatt a progv és a szuksv vektorok helyfoglalása szükségtelenül megnőtt.

A következő probléma a városi közlekedéstervezéssel kapcsolatos. A kérdés az, hogy eljuthatunk-e gépkocsival a város bármelyik pontjáról bármely másik pontra? A probléma megfogalmazásához tekintsük a 71. ábrán látható térképet.



Közlekedési vázlat

71. ábra



A közlekedési hálózat gráfja

72. ábra

A problémát az egyirányú utcák jelentik. Elképzelhető az utcák olyan egyirányúsítása, hogy lesznek a városban olyan területek ahonnan nem lehet kijutni, mert minden utca befelé vezet, de olyan régiók is, ahová bemenni képtelenség, mert csak kifelé egyirányúsított utcák vannak. Jobban kifejezi a lényeget a térképnél az 72. ábrán látható gráf.

Minden útkereszteződést a gráf egy csúcsával jelölünk és az i

csúcsból a j csúcsba egy irányított élt húzunk, ha összeköti őket az adott irányításnak megfelelő utca. Ha ez az utca kétirányú, akkor két - ellenkezőleg irányított - élt rajzolunk be.

A közlekedési hálózat vizsgálatához a gráf szerkezetét le kell írni a számítógép számára. Erre több lehetőség is van, de a legegyszerűbb és a mi esetünkben leginkább gyümölcsöző a csúcsmátrixszal való ábrázolás. E mátrixban minden sor és minden oszlop a gráf egy csúcsának felel meg. Az i-edik sor j-edik eleme akkor és csak akkor 1, ha az i-edik csúcsból a j-edikbe vezet irányított él. Az 72. ábra gráfjának mátrixa:

$$K = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

A közlekedési hálózatok mátrixai tipikus ritka mátrixok. Példánkban is a 81 elem közül csak 16 a nullától különböző. Ábrázolásuknál célszerű a 8.5 alfejezetben megismert technikákat alkalmazni.

Nem világos, hogyan használhatnánk a K mátrixot feladatunk megoldására. K elemei azt mutatják meg, hogy mely csúcsokból juthatunk el milyen másik csúcsba egyetlen élen. A K (megfelelően definiált szorzás melletti) négyzetéről be fogjuk látni, hogy ez a mátrix azt mutatja meg, hogy az egyes csúcsokból milyen csúcsokba juthatunk el két élen keresztül (melyek a "két saroknyira" lévő kereszteződések).

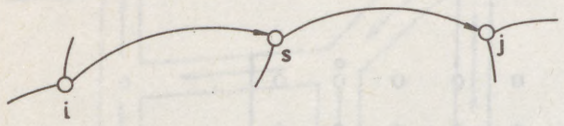
K^2
A K mátrix valamely k2[i,j] elemét úgy kaptuk, hogy a K i-edik sorát és j-edik oszlopát szoroztuk össze skalárisan. Ez azt jelenti, hogy

```
k2[i,j]=k[i,1]*k[1,j]+k[i,2]*k[2,j]+...+k[i,9]*k[9,j].
```

Ez a szám csak akkor lehet 0-tól különböző, ha van olyan s index (s=1, 2, ..., 9), hogy

k[i,s] = 1 és k[s,j] = 1

teljesül. ez pedig azt jelenti, hogy van olyan s csúcs, amelybe i-ből egyetlen élen eljutunk, ugyanakkor s-ből j-be is egyetlen élen eljutunk (73. ábra). Ez pedig éppen azt jelenti, hogy i-ből j-be két él hosszúságú úton eljuthatunk.



Nemzérus szorzótényezőknek megfelelő élek

73. ábra

Lehet hogy több ilyen út is létezik, azonban ez számunkra érdektelen. Módosítsuk ezért az összeadás szabályait a mátrixszorzásra vonatkozólag úgy, hogy

1 + 1 = 1 (*)

2
legyen. Ekkor a K mátrix k2[i,j] eleme akkor és csak akkor 1, ha létezik az i csúcsból a j-be vezető 2 élből álló irányított útvonal.

s
Hasonlóan láthatjuk be, hogy K valamely ks[i,j] eleme akkor és csak akkor 1, ha létezik az i csúcsból a j-be vezető s folytatólagosan irányított élből álló útvonal.
Az úthálózat pedig akkor megfelelő, ha bármely két i, j csomópontja között van valamilyen hosszúságú irányított útvonal, vagyis a

0 1 2 n-1
K , K , K , ..., K

mátrixok közül legalább egyikben 1 a megfelelő mátrixelem értéke. K nulladik hatványa E azt a triviális tényt fejezi ki ebben a sorban, hogy minden csúcs elérhető "önmagából" 0 élből álló úton. A legmagasabb hatványkitevő n-1, ahol n a csomópontok száma. Belátható ugyanis, hogy ha egy n csúcsú gráfban valamely csúcs n-1 él hosszú úton nem érhető el, akkor egyáltalán nem érhető el. Ha az összeadást a

2 n-1
H = E + K + K + ... + K

összegzésnél a (*) szabálynak megfelelően végezzük el, akkor az

úthálózat akkor és csak akkor összefüggő, ha a H mátrix minden eleme 1. (Pontosabban: a szóban forgó összefüggőséget a gráfelméletben erős összefüggőségnek nevezik.)

A kitűzött problémát tehát a következő algoritmussal oldhatjuk meg:

```
a K mátrix (nem 0) elemeinek megadása,
a ritka mátrix megfelelő szerkezetű tárolása
a H mátrix előkészítése (H:=E+K)
G:=K a hatványozáshoz
for i:= 2 to n-1 do
   G:=GxK
   H:=H+G
H vizsgálata, esetleges 0 elemeinek kiírása
```

Két dolgot kell még megbeszélni, egy rosszat és egy jót. A rossz: a H mátrixot nem kezelhetjük ritka mátrixként hiszen éppen azt várjuk tőle, hogy minden eleme 1 legyen. A jó: a K és a G mátrixnál a zérustól különböző elemek értékét sem kell tárolni, hiszen mindegyik érték 1. A mátrixok szorzására és összeadására készített alprogramjaink használata problematikus, hiszen a mátrixelemeket meghatározó K(i,j), G(i,j) függvényeket nem tudjuk paraméterként átadni. Csak a nyelv 5.0-ás változatában jelennek meg olyan eszközök, amelyek ezt lehetővé teszik. Ehelyett a létező alprogramok módosítását javasoljuk, de elvégzését az Olvasóra hagyjuk. Mivel a programban két ritka mátrixot is kezelni kell, javasoljuk a 60. program módosítását is (ezt a megváltozott tárolási szerkezet is indokolja). Legyen az új függvény feje

```
function elem(rt:reftipus; oit:indtabtipus;
              csucs,el:integer):byte;
```

Ne használjunk et tömböt az új függvényben (hiszen csak 1-es nemzérus elem lehet).
Ezek után nézzük a 63. programot.

```
program uthalozat;

  const csucsszam=50;
        elszam=125; {csucsonként 5 ellel szamolva}

  type matrixtipus=array[1..csucsszam,1..csucsszam] of byte;
       reftipus=array[1..csucsszam,1..2] of integer;
       indtabtipus=array[1..elszam] of integer;

  var H:matrixtipus;
      kreftab,greftab:reftipus;
      koszlindtab,goszlindtab:indtabtipus;
      i,j:integer;
```

```
    csucs,el:integer;

{$I elem.pas}

{$I mosszeg.pas}

{$I mszorzas.pas}

procedure olvas;

  type strings=string[79];

  var i,j,q: integer;
      ok:boolean;
      egysor:strings;

  {$I kovetk.pas}

  procedure elrak(hova:integer;
                  var db:integer; var jo:boolean);
    var i,j,k:integer;
        resz:strings;
        oind:integer;

    begin
      db:=0;
      j:=1;
      while j>0 do begin
        k:=kovetkezo(egysor,',',j);
        if k<>0 then begin
          resz:=copy(egysor,j,k-j);
          i:=1;
          while resz[i]=' ' do
            resz:=copy(resz,i+1,length(resz)-1);
          oind:=val(resz);
          if (oind>csucsszam) or (oind<1) then begin
            jo:=false;
            exit;
            end;
          koszlindtab[hova]:=oind;
          hova:=hova+1;
          db:=db+1;
          end
        else
          j:=0;
        end;
      end; {elrak}

  begin
    ok:=true;
    repeat
      clrscr;
      if not ok then
        writeln('Maximum ',csucsszam,' csucs kezelheto');
```

```
      write('A csomopontok szama:');
      readln(csucs);
      ok:=false;
      until (csucs>1) and (csucs<=csucsszam);
    writeln;
    write('Irja be soronként (vesszovel elvalasztva)');
    writeln('az 1-es elemek oszlopindexeit!');
    q:=1;
    el:=0;
    for i:=1 to csucs do begin
      repeat
        ok:=true;
        kreftab[i,2]:=0;
        write(i,'-edik sor:');
        readln(egysor);
        elrak(q,szam,ok);
        until ok;
      el:=el+szam;
      if el>elszam then begin
        writeln;
        writeln('HIBA');
        writeln('A megengedettnel tobb utca ;
        halt;
        end;
      q:=q+szam;
      kreftab[i+1,1]:=q;
      kreftab[i,2]:=szam;
    end; {olvas}
begin
  clrscr;
  olvas;

  { H:=E+K }
  for i:=1 to csucs do begin
    for j:=1 to csucs do
      H[i,j]:=elem(kreftab,koszlindtab,i,j);
    if H[i,i]=0 then H[i,i]:=1;

  { G:=K }
  greftab:=kreftab;
  goszlindtab:=koszlindtab;

  for k:=1 to csucs-1 do begin

    { G:= GxK }
    mszorzas(greftab,goszlindtab,kreftab,
             koszlindtab,csucs,el);

    { H:= H+G }
    for i:=1 to csucs do
       for j:=1 to csucs do
         if H[i,j]=0 then
           H[i,j]:=H[i,j]+elem(greftab,goszlindtab,i,j);
    end;
```

```
{H vizsgalata, kiiras}
clrscr;
ok:=true;
for i:=1 to csucs do
  for j:=1 to csucs do
    if ok then
      ok:=ok and (H[i,j]=1)
    else
      if H[i,j]=0 then
        writeln('Nem kozelitheto meg ',i,'-bol',j);
if ok then begin
  writeln('Az uthalozat kifogastalan,')
  writeln('mindenhonnan mindenhova el lehet jutni');
  end;
end.
```

Az úthálózat vizsgálata

63. program

8.7 Feladatok

1. Írjon alprogramot, emely egy valós tömb elemeit összegzi!

2. Olvastasson be zérustól különböző valós számokat (végjel:0), és az 1. feladat alprogramjának felhasználásával számítsa ki az átlagukat és a szórást. Szórás alatt az

$$\left(\sum_{1}^{n} \frac{x_i^2}{n} - M^2 \right)^{\frac{1}{2}}$$

értéket értjük, ahol x_i az elemeket, n az elemek számát, M pedig az átlagot jelöli.

3. Írjon alprogramot, amely egy tömb 3 legkisebb elemét határozza meg!

4. Egy sportversenyen a 100 méteres gyorsúszás eredményeit - a számítógépes kiértékelésnél - egy tömbben tárolják. Az index a rajtszámmal azonos. Határozza meg a 3. feladat alprogramjával a dobogós versenyzőket! Írassa ki a nyertesek nevét is (feltéve, hogy rendelkezésére áll a nevek tömbje is).

5. Készítsen eljárást amely egy tömb elemeit több sorba, soronként k-asával írja ki!

6. Az egydimenziós v tömb elemei nagyság szerint növekvő sorrendben helyezkednek el. Írjon eljárást, amely megfordítja ezt a sorrendet!

7. Készítsen programot, amely egy v tömb elemeit 3 hellyel ciklikusan balra lépteti. (Pl. ha az eredeti tömb

   21 22 23 24 25 26 27 28 29

   akkor a léptetés után

   24 25 26 27 28 29 21 22 23

   lesz.)

8. Általánosítsa a 7. feladat algoritmusát úgy, hogy k hellyel léptesse balra ciklikusan a tömb elemeit.

9. Egy egydimenziós tömb elemei tetszőleges valós számok. Határozzuk meg azt az összefüggő résztömböt, amelynek az elemösszege maximális.

10. Egy tömb elemeit rendezzük növekvő sorrendbe úgy, hogy mindig csak a szomszédos elemeket cseréljük fel, ha szükséges. Fejezzük be a rendezést, ha a tömb már rendezett.

11. Írjunk egy boolean értékű függvényt (predikátumot), amely akkor és csak akkor ad vissza true értéket, ha a paraméterét képező egydimenziós tömb elemei növekvő sorrendben rendezettek.

12. Használja a 43. program minindex függvényét és a két változó értékének felcserélésére készített eljárást rendező alprogram szerkesztésére. Az alprogram az i-edik helyen álló elemet cserélje fel az i-edik legkisebb elemmel i=1-től kezdve.

13. Írja meg általánosan valamelyik rendezőprogramot és használja egy névsor ábécé szerinti rendezésére!

14. Egy felmérésben 8 kérdés szerepelt azonos súllyal, mindegyik választ a 0..9 számjegyekkel értékelték. Azonos eredményűnek tekintünk két felmérést, ha - sorrendtől eltekintve - ugyanolyan pontszámokat kaptak. Pl.

   43011225 és 05213241

   azonos eredményt jelentő értékelések, de

   42111225

már más (hiába egyenlő a pontszámok összege). Írjunk programot, amely megszámolja, hogy hány azonos eredmény született!

15. Írjon a 4.6.3 feladat alapján eljárást a jövedelemadó kiszámítására. A kiválasztási szerkezetek helyett használjon tömböt!

16. Két rendezett tömbből állítson elő egy harmadikat, amely a két eredeti tömb összes elemét ugyancsak rendezetten tartalmazza.

17. Adott egy rendezetlen tömb. Válasszon ki egy véletlen elemet és határozza meg a nála kisebb és nagyobb elemek számát!

18. Írjon programot, amely egy inputként megadott összegről eldönti, hogy milyen címletekben fizethető ki a legkevesebb bankjegy, ill. pénzdarab felhasználásával.

19. Írjon eljárást, amely egy kétdimenziós tömb adott két sorát felcseréli!

20. Egy négyzetes mátrix transzponáltját a sorok és oszlopok felcserélésével képezzük. A

$$\begin{bmatrix} 3 & 7 & 1 \\ 2 & 0 & 4 \\ 5 & 6 & 2 \end{bmatrix}$$

mátrix transzponáltja pl.:

$$\begin{bmatrix} 3 & 2 & 5 \\ 7 & 0 & 6 \\ 1 & 4 & 2 \end{bmatrix}$$

Készítsen eljárást, amely a paraméterként adott mátrixot önmagában transzponálja!

21. Írjon programot, amely egy mátrix sorait és oszlopait összegzi!

22. Határozza meg egy mátrix soraiban az elemek minimumát, majd e minimális elemek maximumát!

23. Keressen egy mátrixban olyan elemet, amely a sorában minimális, ugyanakkor az oszlopában maximális elem.

24. A lineáris algebrában a mátrixok átalakítására gyakorta használt művelet a bázistranszformáció. Az elemi bázistranszformációt a következőképpen kell végrehajtani.

Választunk az A mátrix s-edik oszlopában egy g=a $\neq$0 elemet.
rs
g-t generáló elemnek nevezzük. A transzformált mátrix elemei:

$$a_{ij} = \begin{cases} 1, \text{ ha } j=s \text{ és } i=r, \\ 0, \text{ ha } j=s \text{ és } i\neq r, \\ d_j = \dfrac{a_{ij}}{g}, \text{ ha } j\neq s \text{ és } i=r, \\ a_{ij} - d_j * a_{is}, \text{ ha } j\neq s \text{ és } i\neq r. \end{cases}$$

Készítsünk alprogramot az elemi bázistranszformáció elvégzésére!

25. Egy A négyzetes n-edrendű mátrix inverzét - ha létezik - meghatározhatjuk bázistranszformációval. Ehhez végezzünk n elemi bázistranszformációt a mátrixon és az n-edrendű egységmátrixon is úgy, hogy rendre az a[1,1], a[2,2], ..., a[n,n] főátlóbeli elemeket választjuk generáló elemnek. (Ha a soron következő főátlóbeli elem 0 volna, akkor sor, ill. oszlopcserékkel elérhetjük, hogy 0-tól különböző érték kerüljön a helyére. Ha mégsem, akkor a mátrix nem invertálható. Az egyszerűség kedvéért azonban most ne folytassuk a számolást, ha a soron következő generáló elem 0.) Az egységmátrixon ugyanazokat a műveleteket végezzük mint az A mátrixon, azaz

$$e_{ij} = \begin{cases} d_j = \dfrac{e_{ij}}{g}, \text{ ha } i=r \\ e_{ij} = e_{ij} - d_j * a_{is}, \text{ ha } i\neq r. \end{cases}$$

A számítás elvégeztével A inverzét az egységmátrix helyén kapjuk (a helyén pedig az egységmátrixot).

26. A 25. feladat alapján készített mátrixinvertáló eljárás segítségével készítsen programot n ismeretlenes, n egyenletből álló lineáris egyenletrendszer megoldására. Ha a mátrixot a leírt módon nem tudja invertálni, nem jelenti azt, hogy az egyenletrendszernek nincs megoldása.

27. Antiszimmetrikus mátrixnak olyan A négyzetes mátrixot nevezünk, amelynek elemeire az

$$a_{ij} = -a_{ji}$$

összefüggés érvényes. Pl.:

$$\begin{bmatrix} 0 & 1 & -2 \\ -1 & 0 & 3 \\ 2 & -3 & 0 \end{bmatrix}$$

A főátlóban álló elemek az $a_{ii} = -a_{ii}$ követelmény miatt nyilván nullák.
Tervezzen gazdaságos tárolási szerkezetet antiszimmetrikus mátrixok számára és írjon függvényt az elemek visszanyerésére!

28. Alakítson ki alkalmas tárolási szerkezetet felső háromszögmátrix elemeinek tárolására. Készítsen eljárást a mátrix beolvasására és az elemek visszanyerésére!

29. Ritka mátrixok tárolásánál az 70. ábra szerint kétdimenziós referenciatömböt használtunk. Ha elhagyja a referenciatömb 2. oszlopát, még elegendő információja marad az elemek visszanyeréséhez. Írja át ennek megfelelően az 60. programot!

9. HALMAZOK ÉS ALKALMAZÁSAIK

9.1 Halmazok és halmaztípusok

Meghatározott elemek összességét halmaznak nevezzük. A matematikában a halmazokat az elemeikkel adjuk meg. Pl.

A={a, b, c}

az a halmaz, amelynek a, b és c az elemei. Ha adott egy halmaz, akkor beszélhetünk annak részhalmazairól. Akkor mondjuk, hogy egy X halmaz részhalmaza az Y halmaznak, ha X minden eleme az Y halmaznak is eleme. Az A halmaz részhalmazai:

0, {a}, {b}, {c},

{a,b}, {a,c}, {b,c}, {a,b,c}.

A 0 a halmazelméletben az üres halmaz jele, azaz olyan halmazé, amelynek egyetlen eleme sincs. A részhalmazra adott definíció értelmében az üres halmaz minden halmaznak részhalmaza. Az {a,b,c} a definíció értelmében szintén részhalmaza A-nak.
Két halmazt egyenlőnek nevezünk, ha azonosak az elemeik. Ezért a következő halmazok egyenlőek:

{a,b,c} = {c,a,b} = {c,a,b,a,a,c}.

Állapodjunk meg abban, hogy nem is tüntetünk fel kétszer egy elemet a halmaz megadásakor. Nem számít az sem, hogy milyen sorrendben adjuk meg az elemeket.
A matematikai halmazok számítástechnikai megvalósítására a halmaztípusokat használjuk. Ahogy a realtípus elemei valós számok, egy tömbtípusé tömbök, a halmaztípus elemei halmazok. Mégpedig egy meghatározott alaphalmaz részhalmazai. A halmaztípus deklarálásakor ezt az alaphalmazt adjuk meg. A matematikától eltérően a Pascal nyelvben nem lehet bármilyen objektum egy halmaz eleme. A halmazok csak azonos típusú elemkből állhatnak. Ezt a típust a halmaz bázistípusának nevezzük. A bázistípus csak diszkréttípus lehet. Az alaphalmazt a bázistípus megadásával határozzuk meg a set alapszó után:

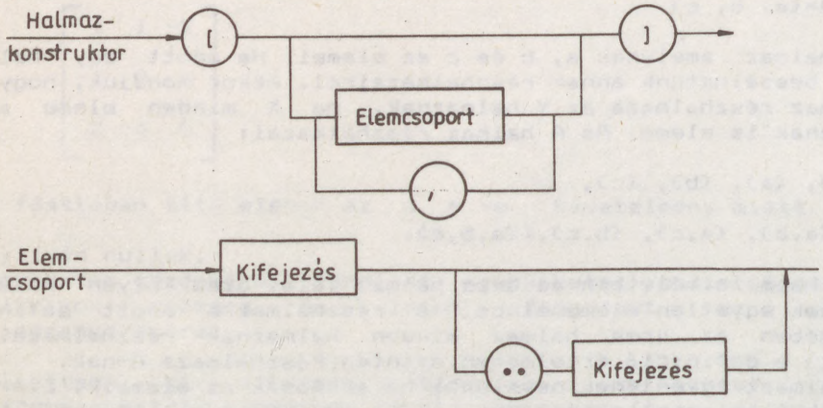
```
type szamjegyhalmaz=set of 0..9;
     betuhalmaz=set of 'A'..'B';
     szinhalmaz=set of (kek,sarga,zold,voros);
     jelhalmaz=set of char;
     szamhalmaz=set of byte;
```

Ezekkel a típusazonosítókkal pl. a következő halmazváltozókat deklarálhatjuk:

```
var szamjegyek          :szamjegyhalmaz;
```

```
betuk,maganhangzok:betuhalmaz;
szinek,festekek    :szinhalmaz;
szamok,lottoszamok,nyeroszamok:szamhalmaz;
irasjelek          :jelhalmaz;
```

Egy halmaztípusú értéket - azaz az alaphalmaz egy részhalmazát - halmazkonstruktorral adhatunk meg. A halmazkonstruktor szintaxisát az 74. ábrán mutatjuk be.



A halmazkonstans szintaxisa

74. ábra

Példák:

[] az üres halmaz, minden halmaztípus értékei közé tartozik.

[0,2,4,6,8] a számjegyhalmaznak, de egyben a számhalmaznak is értéke.

[100,101,200,201] a számhalmaztípushoz hozzá tartozik, de a számjegyhalmazhoz nem.

['I'..'N'] a betűhalmazhoz tartozó érték, a
['I','J','K','L','M','N'] halmaz tömörebb jelölése.

[1,10..19,21,100..119] a számhalmaz típushoz tartozó érték.

[kek] a színhalmaz típus egy értéke.

Egy adott halmaztípushoz tartozó értékek halmaza az alaphalmaz részhalmazainak a halmaza. Ezt a halmazt a matematikában hatványhalmaznak nevezik. Egy A halmaz hatványhalmazát a

$$2^{A}$$

szimbólummal szokás jelölni. Így pl. a számjegytípus értékhalmaza a

$$2^{[0..9]}$$

hatványhalmaz. (Most és a továbbiakban is a Pascal jelölést fogjuk használni a halmazok megadására.) A hatványhalmaz számosságát az alaphalmaz számosságából meghatározhatjuk. Ha az A halmaz számosságát A -val jelöljük, akkor a számjegyhalmaz értékeinek számossága

$$2^{[0..9]} = 2^{10} = 1024.$$

A bázistípus számossága legfeljebb 256 lehet. Ennél erősebb megszorítás is érvényes, a bázistípus minden x értékére az ord(x) nem lehet 0-nál kisebb és 255-nél nagyobb.

Megjegyzés:

A tömbtípus értékhalmazát a bázistípus értékhalmazának hatványaként határoztuk meg. Most hatványhalmazról beszélünk. Ne keverjük össze ezt a két fogalmat. Az [a,b] halmaz (második) hatványa az elempárok

$$[a,b]^2 = [(a,a),(a,b),(b,a),(b,b)]$$

halmaza, míg ugyanezen halmaz hatványhalmaza:

$$2^{[a,b]} = [0,[a],[b],[a,b]].$$

A példák között láttunk olyan halmazt, amely különböző típusok értékhalmazához is hozzá tartozott. Ez felveti a halmaztípusok kompatibilitásának kérdését.

K6 Két halmaztípus kompatibilis, ha bázistípusaik kompatibilisek.

Kompatibilis típusú halmazváltozók és halmazkonstansok esetén az értékadás művelete használható. Pl. az

```
maganhangzok:=['A','E','I','O','U'];
betuk:=maganhangzok;
irasjelek:=['.',',','?','!'];
```

értékadások helyesek. Ugyanakkor a

```
betuk:=irasjelek;
```

értékadás a változók típusának kompatibilitása ellenére is hibás, mert a jobb oldali érték nem eleme a betűk változó típusával meghatározott értékhalmaznak.

9.2 Programozás halmazokkal

Az értékadáson kívül a szokásos halmazelméleti műveletek is rendelkezésünkre állnak a halmazok kezelésére. Képezhetjük halmazok egyesítését, metszetét és különbségét.

Halmazműveletek

Egyesítés (jele: +)

Az A és B halmazok A+B egyesítése az a halmaz amelynek minden eleme az A vagy a B halmaznak is eleme. Ha pl.

A=[1,2,6,8] és B=[2,3,7,8],

akkor

A+B=[1,2,3,6,7,8].

Metszet (jele: *)

Az A és B halmazok A*B metszetén azt a halmazt értjük, amelynek elemei A-nak is és B-nek is elemei. Az előző A, B halmazokkal:

A*B=[2,8]

Külömbség (jele: -)

Az A és B halmazok A-B külömbségén azt a halmazt értjük, amely az A halmaznak azokból az elémeiből áll, amelyek nem tartoznak B-hez is.

A-B=[1,6]

Használhatjuk az =, <>, <=, >= relációjeleket, továbbá a speciális "in" relációjelet. A relációjelek jelentése a következő:

A=B halmazok egyenlősége (A és B egyenlő),
A<>B halmazok különbözősége (A és B nem egyenlő),
A<=B tartalmazás (A-t tartalmazza B),
A>=B tartalmazás (A tartalmazza B-t).

Pl. az

[1,3,5,7]<>[1,3,5,7,9]

és az

[1,3,5,7]<=[1,3,5,7,9]

relációk értéke true.

Az in relációval állapíthatjuk meg, hogy valamely objektum eleme-e egy halmaznak vagy sem. Az "eleme" reláció egy kifejezés és egy halmaz között állhat fenn:

```
kifejezés in halmaz.
```

A kifejezés típusának a halmaz bázistípusával kompatibilisnek kell lenni. Pl. a

```
betu in maganhangzok
```

reláció helyes, ha a betu változó char típusú, és igaz a következő értékadások után:

```
maganhangzok:=['A','E','I','O','U'];
betu:='E';
```

A halmazok elegáns programozási megoldásokra adnak lehetőséget sok olyan esetben, amikor egyébként if-ekkel kellene dolgoznunk. Ha pl. egy szövegben az előforduló magánhangzókat akarjuk megszámolni, jó szolgálatot tesz az in reláció (64. program).

```
function maganhangzok(sor:strings):integer;

  type jelhalmaztipus=set of char;
  const maganhalmaz:jelhalmaztipus=['A','a',
                        'E','e','I','i','O','o','U','u'];
        s:integer=0;
  var i:integer;

  begin
    for i:=1 to length(sor) do
       if sor[i] in maganhalmaz then
         s:=s+1;
    end;
```

A magánhangzók számlálása

64. program

Erdekes és tanulságos a következő számelméleti alkalmazás is, a prímszámok keresése. Erre a legjobb algoritmus ma is az a régi görög eljárás, amit "Eratoszthenész szitája" néven emlegetnek. Tekintsük a 2..255 természetes szám halmazát (a 65. programban a "szita" halmazváltozó értéke). Vegyük sorra a számokat és hagyjuk el mindig a halmazból a soron következő szám többszöröseit. A megmaradó számok a prímszámok.
A halmaz elemeit a szam ciklusváltozójú for ciklus veszi sorra. Ha egy szám eleme a "szita"-nak, akkor prímszám. Ezután az i szerinti ciklusban távolítjuk el az éppen megtalált prím többszöröseit. Itt halmazműveletet végzünk, a szita halmazból kivonjuk a többszörösök egyelemű [i*szam] halmazait.

```
program Eratoszthenesz;

  const maxertek=255;
  type szamhalmaztipus=set of 2..255;

  const szita:szamhalmaztipus=[2..255];

  szam,i:integer;

  begin
    clrscr;
    writeln('A primszamok ',maxertek,'-ig:');
    for szam:=2 to maxertek do
      if szam in szita then begin
        writeln(szam);
        for i:=2 to maxertek div szam do
          szita:=szita-[i*szam];
        end;
  end.
```

Eratoszthenész szitája

65. program

Figyelmeztetjük az Olvasót, hogy programja csak a fordítás után első ízben működik helyesen. Az első futás után ugyanis a szita értéke már megváltozik, a kezdőértéket pedig fordítás közben kapja.

9.3 A bittérképes ábrázolás

A halmazok ábrázolásához legfeljebb 32 bájt tárterületet használ a Turbo Pascal rendszer. Ez 8*32=256 bitet jelent, és emiatt kell az alaphalmaz számosságára vonatkozó felső korláttal számolnunk. Az alaphalmaz minden elemét egy bit ábrázolja. Ha az alaphalmaz egy tetszőleges x értékét tekintjük, akkor az x-et ábrázoló bit sorszámát az

```
ord(x)
```

függvényérték határozza meg. Ha kis sorszámú elemeket tartalmaz a bázistípus, akkor csak a szükséges számú bájtot foglalja el az ábrázolás. A

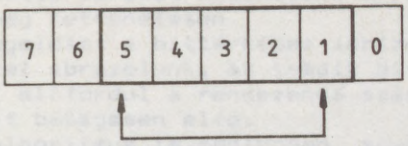
```
type htipus=set of 1..5;
var jegyek:htipus;
```

deklarációk vannak érvényben, akkor csak egyetlen bájtra van szükség (75. ábra).
Az ábrán a bájtnak azt a részét, amelyet a deklarált halmazok ábrázolásához ténylegesen használunk, vastag vonallal jelöltük. Ha a bázistípus valamely értéke eleme a "jegyek" halmaznak, akkor

a megfelelő bit értéke 1, egyébként 0. Ha végrehajtottuk a

```
jegyek:=[1..3,5]
```

bitpozíciók:



a bájt ábrázoláshoz használt része

A halmazhoz tartozó tárterület

75. ábra

értékadást, akkor az 1-es, 2-tes, 3-mas és 5-ös bitek értéke lesz 1. Az ábrázolt halmaz tárképét az 76. ábrán láthatjuk.

bitpozíció:

7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0

A halmaz értékének ábrázolása

76. ábra

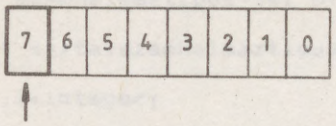
Mivel a halmaz ábrázolásához csak egyetlen bájtra volt szükség, a többi 31 bájtot más célra lehet használni. Így csak 3 bit maradt kihasználatlanul, a 0, 6 és 7-es sorszámú.
Most tekintsük a következő deklarációt:

```
type szamjegyhtipus=set of 7..9;
var x:szamjegyhtipus;
```

Az x alaphalmaza csak 3 elemet tartalmaz, ábrázolására mégis két bájtra van szükség (77. ábra).
Ez esetben a 16 bájtból 13 veszendőbe megy. Meg lehetett volna írni úgy is a fordítóprogramot, hogy ilyen esetben takarékosabban gazdálkodjon a tárral, de ennek az az ára, hogy a lefordított program mérete és végrehajtási ideje is valamivel megnő. Ezért nem indokolt ilyen megoldást választani. A teljesen üres bájtokat azonban a struktúra elején is meg lehet takarítani.

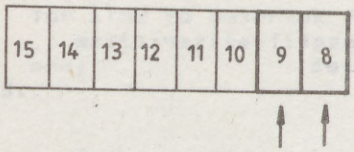
Az első bájt

bitpozíció:



A második bájt

bitpozíció:



Halmazábrázolás két bájton

77. ábra

A halmazok ábrázolási módját bittérképes ábrázolásnak nevezzük. Ha a halmazt konstansokkal határozzuk meg, akkor a

```
type htipus=set of minelem..maxelem;
```

típusú halmazok ábrázolásához szükséges bájtok számát a következőképpen határozhatjuk meg:

```
hossz:= ord(maxelem) div 8 - ord(minelem) div 8 + 1
```

Ha adott a halmaz egy tetszőleges h eleme, akkor a bájt sorszáma amelyben a h-t ábrázoló bit van

```
bajt:= ord(h) div 8 + ord(minelem) div 8,
```

és a bájton belül a bit sorszáma

```
bit:= ord(h) mod 8.
```

Bár a kezelhető halmazok mérete a Turbo Pascalban meglehetősen korlátozott, az előző formulákkal byte bázistípusú tömbökben magunk is megvalósíthatunk bittérképeket.
Ez a tárolási szerkezet néha nagyon hatékony algoritmusok írását teszi lehetővé. [1]-ben olvashatunk egy ilyen esetről. 1 és 27000 közötti egészeket kell nagyság szerint növekvő sorrendbe rendezni. A tárban azonban csak legfeljebb 4 kbájt szabad terület

áll rendelkezésre. A rendezési idő kritikus, nem lehet 1-2 percnél hosszabb. Tudjuk még, hogy a rendezendő számok egyike sem fordulhat egynél többször elő.
Mit lehet tenni? A szabad tárterületen legfeljebb 2000 integer értéket tudunk tárolni. Hol van ez több tízezres nagyságrendhez?! Ha háttértárakat használunk a rendezéshez, akkor az időszükséglet nő meg rettenetesen.
A megoldást a bittérképes ábrázolás adja. Minden számot egyetlen bittel ábrázolunk, az i-edik bit értéke akkor és csak akkor 1, ha az i előfordul a rendezendő számok között. Mivel 27000/8=3375, 4 kbájt bőségesen elég.
Az algoritmus is meglepően egyszerű. Először nullázni kell a szükséges tárrészt, majd egymás után beolvassuk a számokat és 1-esre állítjuk a megfelelő bitet. Ha minden számot beolvastunk, akkor sorban megvizsgáljuk a biteket és ha 1-est találunk, a megfelelő számot kiírjuk.
A számok természetesen mágneslemezen vannak és lemezre kell írni is őket. Mivel a lemezes állományok írásával, olvasásával csak a későbbiekben fogunk foglalkozni, beérjük a billentyűzettel és a képernyővel (az algoritmust úgyis csak néhány számmal próbáljuk ki).

```
program rendezes;

  const max=3375; {27000/8}

  type bajtvektortipus=array[1..3375] of byte;
       bitvekttipus=array[0..7] of byte;

  const bitek:bitvekttipus=(1,2,4,8,16,32,64,128);

  var bajtvektor:bajtvektortipus;
      bajt,bit,szam:integer;
      j,h:byte;

  begin

    {nullázás}
    for bajt:=1 to max do
       bajtvektor[bajt]:=0;

    {beolvasas, vegjel=0}
    repeat
      readln(szam);
      if szam>0 then begin
        bajt:=szam div 8 +1;
        bit:=szam mod 8;
        h:=1;
        for j:=1 to bit do
           h:=h*2;
        bitvektor[bajt]:=bitvektor[bajt] or h;
        end;
      until szam=0;
```

```
      for bajt:=1 to max do
         for bit:=0 to 7 do
            if (bitvektor[bajt] and
                bitek[bit])<>0 then
               writeln((bajt-1)*8+bit);
      end.
```

Rendezés bitvektorral

66. program

A 66. program nem használja ki, hogy a Pascalban halmazokkal is dolgozhatunk. Még kényelmesebben oldhatjuk meg a rendezési feladatot, ha halmazt használunk.
Mivel a halmazunk legfeljebb 32 bájtos és 256 elemű, halmazok tömbjével dolgozunk. A szükséges típus

```
type halmaztomb=array[1..844] of set of 0..255;
```

A módosított algoritmus a 67. program.

```
program rendezesuj;

  const max=105; {27000/256}
  type halmaztip=set of 0..255;
       halmaztomb=array[1..max] of halmaztip;

  var halmaz,szam:integer;
      hvektor:halmaztomb;
      elem:bajt;

  begin
    for halmaz:=1 to max do
      hvektor[halmaz]:=[];
    repeat
      readln(szam);
      if szam<>0 then begin
        halmaz:=szam div 256 +1;
        elem:=szam mod 256;
        hvektor[halmaz]:=hvektor[halmaz]+[elem];
        end;                                    until szam=0;
    for halmaz:=1 to max do
       for elem:=0 to 255 do
         if elem in hvektor[halmaz] then
           writeln((halmaz-1)*256+elem);
    end.
```

Bitvektor megvalósítása halmazzal

67. program

9.4 Feladatok

1. Legyen H1 és H2 halmaztípusú változók, amelyekre a

```
H1:=['A'..'C','D']
```

és a

```
H2:=['A','C','D','E']
```

értékadások végrehajthatók. Írja meg a szükséges deklarációkat!

2. A H1 és H2 halmazok 1. feladatbeli értékei mellett határozza meg a következő kifejezések értékét:

```
H1*H2

H1-H2

H1<>H2

(H1=H2) or not ('A' in (H1-H2))

H1=['A','D','B','C']
```

3. Ha L boolean típusú változó, H típusa pedig set of byte, a következő értékadások közül melyek érvényesek:

```
H:=H+1

H:=H+[1]

L:=H in [1]

L:=H = [1]+[2]
```

4. Legyen H nagybetűk egy halmaza. Írjunk programot, amely H elemeit kinyomtatja!

5. Bizonyos programozási nyelvek eszköztárában szerepel egy ún. halmaziterátor. Ez olyan ismétlési szerkezet, amely a ciklustörzset egy adott halmaz minden elemére pontosan egyszer hajtja végre. Ilyen pl. a

```
forall i in H do ciklustörzs
```

utasítás, ahol H egy halmaz és az i változó típusa a H bázistípusa. A Turbo Pascalban ilyen utasítás nincs. Hogyan hozhatunk létre mégis egy ilyan szemantikájú ismétlési szerkezetet?

6. Az 5. feladat alapján megadott szerkezetben összegezzük egy számhalmaz elemeit!

7. Definiálja az italok halmazát. Adja meg a lakásában megtalálható italok halmazát, majd a következő koktélok összetevőinek halmazát:

   Royal: cherry brandy, gin, martini, maraschino

   Aphrodité: triple-sec, gin, brandy, martini

   Puszta: tokaji szamorodni, barackpálinka, mecseki

   Baccardi: portoriko rum, triple-sec, citromlé

   Wembley: whisky, vermut, ananászlé

   Készítsen programot, amely kiírja, hogy mely koktélokat tudja kikeverni a rendelkezésére álló készletből!

8. Írjon programot, amely megszámolja, hogy egy adott szövegben hány nagybetű, hány kisbetű, hány számjegy, hány írásjel és hány speciális karakter van.

9. Egy beszámolón három kérdésre kell válaszolni a hallgatóknak. A vizsgáztató az A, B és C betűkkel értékeli a válaszokat. Az értékelékelés a következő szabályok szerint történik:

   1. Ha a hallgató legalább egy A-t kapott és nem kapott C-t, akkor az értékelés "jól megfelelt".

   2. Ha a hallgató nem kapott A-t és legalább egy C-t kapott, akkor "nem felelt meg".

   3. Minden más esetben "megfelelt" a minősítés.

   Írjon programot, amely hallgatónként meghatározza a minősítést! A program inputja hallgatónként a következő:

      név 1.osztályzat 2.osztályzat 3.osztályzat.

   A végjel legyen a név helyett a "vege" szó!
   Eredményül a névsort kérjük, a nevek mellett a "jól megfelelt", "megfelelt", "nem felelt meg" minősítésekkel.

10. REKORDOK

10.1 Rekord és rekordtípusok

Gyakran van szükség olyan összetett adatszerkezetre, amelyben - a tömböktől eltérően - nem szükségképpen azonos típusú értékeket egyesítünk. Ilyen adatszerkezetek a rekordok. A rekordok talán a legjellemzőbb Pascal adatszerkezetek, egyaránt jól használhatók ügyviteli, műszaki, és gazdasági feladatok programozásánál.
A record szó eredileg feljegyzést jelent. A zsebnaptárunk regiszteres részében, ahol barátainkról, barátnőinkről és üzletfeleinkről tárolunk "recordokat", egy-egy feljegyzés a következő információkból áll:

1. név,
2. cím,
3. telefonszám,
4. munkahely,
5. munkahelyi telefonszám,
6. szoktunk-e képeslapot küldeni.

Ha ezeket az adatokat Pascal nyelven akarjuk feldolgozni, akkor a következő adattípusokat használhatjuk:

1. név: string[20]
2. cím: string[40]
3. telefonszám: string[9]
4. munkahely: string[30]
5. munkahelyi telefonszám: string[9]
6. szoktunk-e lapot küldeni: boolean

Ezzel voltaképpen egy rekordszerkezetet határoztunk meg.

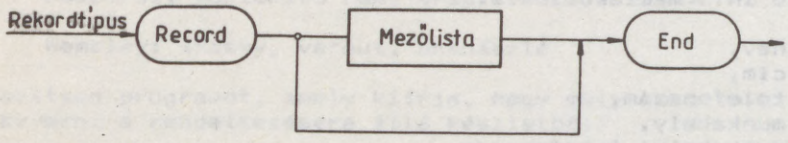
A Pascalban rekord alatt meghatározott számú nem szükségképpen azonos típusú érték összességét értjük. Egy rekordtípust meghatároz a benne tárolt adatok száma és a típusaik. A rekordot alkotó adatelemeket a rekord mezőinek nevezzük. A rekordtípus definiálásakor megadjuk az egyes mezők nevét és típusát:

```
type adatrecord= record
                   nev:          string[20];
                   cim:          string[40];
                   telefon:      string[9];
                   munkhely:     string[30];
                   munkhelyitel: string[9];
                   lapkuldes:    boolean;
                   end;
```

Egy raktári anyagnyilvántartásban a

```
type anyagtipus = record
                    anyagszam: string[8];
                    megnevezes:string[30];
                    egyseg:    (kg,l,db,m,m2);
                    egysegar:  real;
                    mennyiseg: real;
                    end;
```

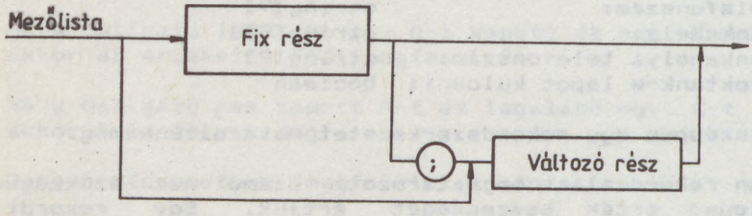
A rekordtípus deklarációjának szintaxisát a 78. ábrán mutatjuk be.



Rekordtípus

78. ábra

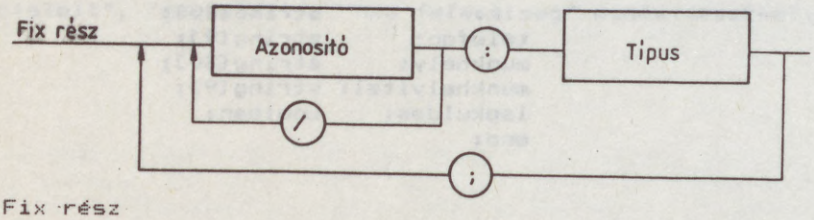
A mezőlista tulajdonképpen két részből állhat (79. ábra): a fix részből és a változatból.



Mezőlista

79. ábra

Egyelőre csak a fix résszel foglalkozunk (80. ábra).



Fix rész

80. ábra

A mezők típusát is típusazonosítóval célszerű megadni, hogy az esetleges összeférhetetlenségi hibákat megelőzzük.
A rekordtípushoz tartozó értékek halmazát a mezők értékhalmazának Descartes szorzataként kapjuk meg:

$$R = M_1 \times M_2 \times \ldots \times M_k ,$$

ahol M_i (i=1, 2, ..., k) jelenti az i-edik mező típusa által meghatározott értékhalmazt.
Rekord mezőtípusa az eddig megismert típusok bármelyike lehet, így pl. rekordtípus is. Pl. az "adatrekord"-ban a cím maga is megadható rekordként:

```
type str20=string[20];
     str10=string[10];
     str5=string[5];
     str30=string[30];
     str9=string[9];

     cimrekord = record
                   irszam:str5;
                   varos :str10;
                   utca  :str20;
                   hazsz :str5;
                   end; {cimrekord}
     adatrecord = record
                    nev      :str20;
                    cim      :cimrekord;
                    telefon  :str9;
                    munkhely:str30;
                    munkhtel:str9;
                    kuldunk  :boolean;
                    end; {adatrecord}
```

A rekordok értéke a tárban a mezők sorrendjében helyezkedik el (minden mező a típusának megfelelő tárterületet foglal), az első mező tartalma a legkisebb című bájtnál kezdődik. Pl. egy "anyagtipus" rekordhoz tartozó tárrész (R a rekordterület kezdőcíme):

anyagszam	R
megnevezes	R+9
egyseg	R+40
egysegar	R+41
mennyiseg	R+47

A rekord teljes tárigénye 53 bájt.

10.2 A rekordok feldolgozása

Ha x és y azonos típusú rekordváltozók, akkor szintaktikusan helyes az

```
x:=y
```

értékadás, ami az összes megfelelő mező közötti értékadás végrehajtását jelenti. Adhatunk rekordváltozónak kezdeti értéket is a const kulcsszó után, pl.:

```
type datumrekord = record
                     honap :1..12;
                     nap   :1..31;
                     end;

const mainap:datumrekord = (honap:1;
                            nap:21);
```

A rekordokat többnyire mezőnként dolgozzuk fel, ezért gyakoribb a szelektív hozzáférés igénye a mezők értékeihez. A tömböknél az elemek kiválasztására szelektorként az indexeket használtuk. A rekordoknál a mezőnevek a szelektorok. A "mainap" rekordváltozó komponenseit pontozott jelöléssel adhatjuk meg:

```
mainap.honap
```

jelöli a rekord "honap" mezőjét,

```
mainap.nap
```

pedig a "nap" mezőt. Így, ha i integer típusú változó, akkor az

```
i:= mainap.nap
```

értékadás után 21 lesz az értéke. A rekordváltozó komponensei változó szemantikájú objektumok, így értéket kaphatnak és eljárások változóparaméterei lehetnek.
Ha a rekord mezője maga is rekord, akkor a "." operátort ismételten is használhatjuk. Ha programunkban a

```
var haver: adatrecord;
```

deklaráció van, akkor a

```
haver.cim
```

egy címrekord típusú változó objektum. A lakcímben szereplő város neve a

```
haver.cim.varos
```

komponens. Ha a mező más típusú összetett objektum, pl. tömb, akkor indexelnünk is kell. Tegyük fel, hogy egy könyvtár a

kölcsönadott könyveket a következő deklarációkkal létrehozott "olvasorekord"-okban tárolják.

```
type darabtipus = 1..8;
     str11 = string[11];

     konyvtomb=array[darabtipus] of str30;

     olvasorekord = record
                      szemszam: str11;
                      nev     : str20;
                      cim     : cimrekord;
                      darab   : darabtipus;
                      konyvek : konyvtomb;
                      end;

var nyajas: olvasorekord;
```

Akkor a "nyajas" olvasónál lévő könyveket egy for ciklussal kiírathatjuk:

```
.....................
.....................

for i:=1 to nyajas.darab do
   writeln(nyajas.konyvek[i]);

.....................
```

Ha egy programrészletben többször hivatkozunk ugyanazon rekordváltozónak a mezőire, akkor ahelyett, hogy a "." operátort használnánk minden esetben, a with utasítással kényelmesebben fogalmazhatunk.
Adjunk pl. értéket az imént deklarált olvasorekord típusú változónak:

```
nyajas.szemszam:='12311030632';          nyajas.nev:='Piszkos
Fred';
nyajas.cim.irszam:='7622';
nyajas.cim.varos:='Pecs';
nyajas.cim.utca:='Wagner Richard';
nyajas.cim.hazsz:='42';
nyajas.darab:=0;
```

Ennyi "nyájasság" már kissé fárasztó, így a with utasítást csak üdvözölhetjük:

```
with nyajas do begin
  szemszam:='1231030632';
  nev:='Piszkos Fred';

    with cim do begin
    irszam:='7622';
    varos:='Pecs';
    utca:='Wagner Richard';
    hazsz:='42';
    end; {cim}

  darab:=0;
  end; {nyajas}
```

A with utasítás szintaxisa a példából kiolvasható:

```
with rekordváltozó do utasítás
```

Ha A, B és C rekordváltozók, akkor a

```
with A do
  with B do
    with C do begin
      ..........
      ..........
      ..........
      end
```

szerkezet helyett a vele azonos jelentésű

```
with A, B, C do begin
  ..........
  ..........
  ..........
  end
```

szerkezetet is használhatjuk.

10.3 Rekordok és eljárások

A tömbök kezeléséhez a for ciklus, a felsorolási típusokhoz a case szerkezet nyújt megfelelő nyelvi eszközt. Ha rekordokkal dolgozunk, eljárásokat célszerű használni. Az eljárás paramétere rekordváltozó, a rekord komponenseivel végzett munkát az eljárástörzsben rejtjük el. A főprogram szintjén így a rekordtípusú objektumokat "egyben" kezeljük, szerkezetükkel nem foglalkozunk. A rekordok a programnyelvi absztrakció fontos eszközei.

A rekordokkal kapcsolatos eljárások közül az egyik legfontosabb típust az adatbeolvasó eljárások képezik. A kényelmes és biztonságos adatbevitel lehetőségét kell megteremteni ezekkel az eljárásokkal.
Példaképpen készítsünk el a könyvtári feldolgozáshoz adatbeolvasó eljárást.

Hogy a példánk jobban megközelítse a valóságot, egészítsük ki az olvasorekord típust egy "datum" mezővel, amelynek típusa .

```
type datumtipus = record
                    ev: 1989..2100;
                    honap:1..12;
                    nap:1..31;
                    end;
```

Ez a mező a kölcsönzés időpontját fogja tartalmazni.

```
procedure olvasoread(var kolcsonzo:olvasorekord);
  var sorszam,i:integer;
      jel:char;
      kolcsonoz:boolean;

  begin
    clrscr;
    with kolcsonzo do begin
      write('NEV: ');
      readln(nev);
      write('SZEMELYI SZAM: ');
      readln(szemszam);
      with cim do begin
        writeln('CIM: ');
        write('     VAROS: ');
        readln(varos);
        write('      UTCA: ');
        readln(utca);
        write('   HAZSZAM: ');
        readln(hazsz);
        write('IRANYITOSZAM: ');
        readln(irszam);
        end;
      repeat
        gotoxy(1,9);write('                    ');
        gotoxy(1,9);
        write('Kolcsonoz? (i/n) ');
        readln(jel);
        jel:=upcase(jel);
      until (jel='N') or (jel='I');
      kolcsonoz:=jel='I';
      if kolcsonoz then begin
        repeat
          gotoxy(1,11);write('                    ');
          gotoxy(1,11);
```

```
        write('HANYAT? ');
        readln(darab);
        until (darab>=1) and (darab<=8);
      writeln;
      for i:=1 to darab do begin
        write(i:1,': ');
        readln(konyvek[i]);
        end;
      writeln('A kolcsonzes datuma:');
      with datum do begin
        write('ev: ');
        readln(ev);
        write('ho: ');
        readln(honap);
        write('nap:');
        readln(nap);
        end;
      end;

    { ellenorzes, modositas}

    clrscr;
    writeln('1.',szemszam);
    writeln('2.',nev);
    writeln('3.',cim.irszam);
    writeln('4.',cim.varos);
    writeln('5.',cim.utca);
    writeln('6.',cim.hazsz);
    if kolcsonoz then begin
      writeln('7.',darab,' konyv:');
      writeln('8.',datum.ev,'.',datum.honap,
               '.',datum.nap,'.');
      for i:=1 to darab do
         writeln(8+i,'.',konyvek[i]);
      end;


    writeln('Ha javitani akar, irja be');
    writeln('a hibas adat sorszamat, egyebkent 0-t!');
    repeat
      gotoxy(1,20);
      write('                ');
      gotoxy(1,20);
      write('Sorszam: ');
      readln(sorszam);
      gotoxy(1,22);
      write('
');
      gotoxy(1,22);
      if sorszam>0 then
        case sorszam of

          1:  begin
                write('szemelyiszam: ');
```

```
            readln(szemszam);
            end;
       2:   begin
            write('nev: ');
            readln(nev);
            end;
       3:   begin
            write('irszam: ');
            readln(cim.irszam);
            end;
       4:   begin
            write('varos: ');
            readln(cim.varos);
            end;
       5:   begin
            write('utca: ');
            readln(cim.utca);
            end;
       6:   begin
            write('hazszam: ');
            readln(cim.hazsz);
            end;
       7:   begin
            write('konyvek szama: ');
            readln(darab);
            end;
       8:   begin
            write('ev: ');
            readln(datum.ev);
            gotoxy(1,22);
            write('                ');
            gotoxy(1,22);
            write('ho: ');
            readln(datum.honap)
            gotoxy(1,22);
            write('                ');
            gotoxy(1,22);
            write('nap:');
            readln(datum.nap)
            end;
    9..16:begin
            write('konyv: ');
            readln(konyvek[sorszam]);
            end;
        end;
        until sorszam=0;
    end;
end;
```

Rekordbeolvasás

68. program

Ha ezt az eljárást elkészítettük, akkor a főprogramban egy olvasorekord beolvasása csak ennyiből áll:

```
olvasoread(nyajas);
```

Az alprogram ellenőrzött adatbevitelt tesz lehetővé, ahol a gép kezelője - aki többnyire nem programozó - korrigálhatja is az elkövetett hibáit.
Még egyetlen megjegyzést fűzünk a 68. programhoz. Az eljárás a napi dátumot is kéri (a kölcsönzés keltét). Ezt az információt már a DOS üzembe helyezésekor megadtuk, így valahol a tárban megtalálható. Van is arra mód - és a profi programozók általában így járnak el hasonló esetben -, hogy kiolvassuk a tárolt dátumot és automatikusan írjuk be a rekordba. Ehhez azonban az operációs rendszer mélyebb ismeretére van szükség.

Ezután a hosszó - és nem annyira érdekfeszítő, mint hasznos - program után felüdülésül írjunk egy eljárást egy adott dátum egy nappal való megnövelésére. A datumrekord legyen azonos a korábbival.
Itt az okoz nehézséget, hogy a napok száma hónaponként eltérő, így nem mindig következik 30. után a következő hónap elseje. A 31 napos hónapokat egy halmazzal adjuk meg és az in relációval döntjük el egy adott hónapról, hogy 31 napos e.
Még több a gondunk a februárral. Napjainak száma az évszámtól függ, szökőévben 29, egyébként 28 napos. Egy év szökőév, ha az évszám

4-gyel osztható, de 100-zal nem,

vagy

osztható 400-zal.

Az aktuális hónak utolsó napja után a hónap sorszám egyel nő (és 1. lesz), ha pedig dec. 31. a dátum, akkor az évszámot is növelni kell.

```
procedure kovnap(var datum:datumrekord);

  type hnap=set of 1..12;
  const h31: hnap = [1,3,5,7,8,10,12];
  var i:integer;

  begin
    with datum do

      if ((honap in h31)
          and (nap<31)
        or
              (honap<>2)
          and (nap<30)
        or
```

```
            (honap=2)
          and (nap<28))

    or
            ((((ev div 4 =0)
           and (ev div 100<>0))
           or (ev div 400=0))

           and (nap=28)) then
           nap:=nap+1

  else if honap<12 then begin
           honap:=honap+1;
           nap:=1;
           end

           else begin
              ev:=ev+1;
              honap:=1;
              nap:=1;
              end;
end;
```

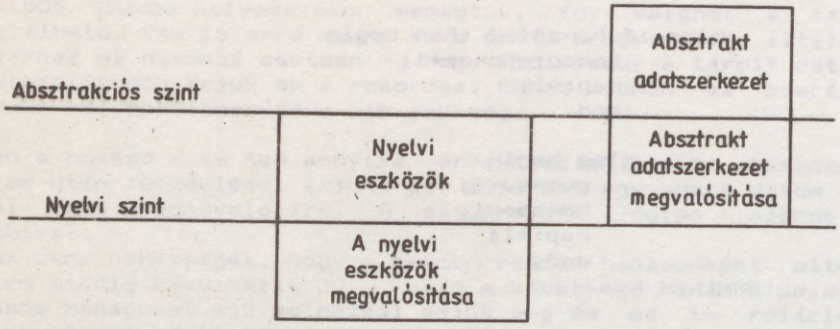
A dátum növelése

69. program

10.4 Absztrakt adatszerkezetek

Amikor programot írunk, amelyben egész és valós számokkal végzünk műveleteket, nem érdekel bennünket, hogy ezeket az értékeket milyen bitsorozatok ábrázolják a tárban. Amikor egy "+", vagy egy "*" műveleti jelet leírunk, nem kell tudnunk, hogyan végzi el a műveleteket a processzor, ehhez milyen áramkörökre, regiszterekre van szükség. Ha a lebegőpontos műveleteket nem a hardver végzi is, senkit sem foglalkoztat programozás közben a gépi kódban megírt alprogramok működési módja, a végrehajtott tevékenység részletei.
Mindezeket a dolgokat elfedi előlünk a programnyelv szintaxisa. Ennek köszönhetően az implementációs szinthez - a szabványos adattípusok megvalósítási szintjéhez képest magasabb absztrakciós szinten programozhatunk. Több energiánk jut így a feladat érdemi részére, mentesülvén a - probléma szempontjából - lényegtelen részletekkel való bajmolódás rabszolgamunkája alól.
Ha bonyolult feladattal van dolgunk, nem bánnánk, ha a programnyelv eszközeinél magasabb absztrakciós szintű eszközeink volnának.
Voltaképpen vannak ilyen eszközeink. Ilyenek az alprogramok és ilyenek a rekordok. Már korábban is, minden általunk definiált adatszerkezetnél hangsúlyoztuk: ha egy új típust definiálunk, nem elegendő csak a típushoz tartozó értékek halmazát létrehozni, meg

kell alkotni a rajta értelmezett műveleteket is. Ha sikerül a definiált adatszerkezetünket és a műveleteket úgy megalkotni, hogy a programozás során a továbbiakban már közömbös legyen számunkra és mások számára az objektumok tényleges ábrázolása és a velük végzett műveleteknek a megvalósítása, akkor absztrakt adatszerkezetről beszélünk. Ha a programírás közben nem törődünk az összetett objektumok és a műveletek megvalósításával, akkor magasabb absztrakciós szinten hatékonyabban tudunk a lényegre összpontosítani.



Nyelvi és absztrakt adatszerkezetek

81. ábra

Elképzelhetjük pl., hogy a nyelvi objektumokkal egy épület szerkezeti elemeit ábrázoljuk, a műveletek ezek kapcsolatait adják meg, ill. magasabbrendű szerkezeti egységek létrehozását. Ilyen nyelvi eszközökkel jól programozhatók bizonyos automatizálható tervezési folyamatok.

Absztrakt adatszerkezetek létrehozását és alkalmazását egy egyszerűbb területen mutatjuk be.
Pascal nyelvben - pl. a FORTRAN-tól eltérően - nem létezik a komplex számok típusa. Márpedig sok műszaki probléma megoldásánál kell komplex számokkal dolgozni. Visszavezethetjük ezt a számolást valós műveletekre, ezzel azonban rosszul olvasható, a szokásos mérnöki tárgyalásmódhoz nem hasonlító programot kapunk, amit ráadásul megírni sem nagy öröm, hiszen minduntalan azzal kell foglalkozni, hogy lefordítsuk valósra a komplex aritmetikát.
Az igazi megoldás: ha nincs komplex adattípus, akkor csináljunk!
A komplex számokat rendezett valós számpárokkal ábrázoljuk, tehát a komplex értékek halmazát Descartes-szorzatként adhatjuk meg:

$$C = R \times R,$$

ahol R a valós számok halmaza. A komplex számokat ábrázolhatjuk pl. kételemű tömbökkel, megállapodva abban, hogy az első elem a valós, a második a képzetes (imaginárius) részt jellenti:

```
type komplex=array[1..2] of real;
```

```
var z:komplex;
```

z[1] a valós, z[2] a képzetes rész. Komplex értékek képzésére konstruktor eljárást, a komponensek meghatározására szelektor eljárásokat kell használnunk. Az indexelés nem jó, mert alacsonyabb absztrakciós szintű eszköz, feltételezi a reprezentáció ismeretét.

```
procedure kompl(var z:komplex; x,z:real);
  begin
    z[1]:=x;
    z[2]:=y;
    end;
```

Konstruktor eljárás

70. program

A konstruktor két valós értékből egy komplex értéket hoz létre, amit a paraméterként adott komplex típusú változóba ír.

```
function re(z: komplex):real;
  begin
    re:=z[1];
    end;
```

Valós rész szelektor

71. program

```
function im(z: komplex):real;
  begin
    im:=z[2];
    end;
```

Képzetes rész szelektor

72. program

Az összadás műveletét a következő alprogrammal definiálhatjuk (73. program).

```
procedure pluskompl(var z:komplex;x,y:komplex);
  begin
    z[1]:=x[1]+y[1];
    z[2]:=x[2]+y[2];
```

```
end;
```

Komplex számok összeadása

73. program

Valójában inkább rekordokkal szoktuk a komplex számokat ábrázolni. Ilyenkor a

```
type komplex= record
                re:real;
                im:real;
                end;
```

típusdeklarációval definiáljuk a valós értékek halmazát. Ekkor megváltozik a konstruktor és szelektor eljárások belseje, de a hívásuk módja nem.

```
procedure kompl(var z:komplex; x,y:real);
  begin
    z.re:=x;
    z.im:=z;
    end;
```

Konstruktor

74. program

```
function re(z:komplex): real;
  begin
    re:=z.re;
    end;

{ ---------------------------- }

function im(z:komplex): real;
  begin
    im:=z.im;
    end;
```

Szelektorok

75. program

Megváltozik az összeadás is (73. program), de ez kikerülhető lett volna, ha már az előbb helyesen írjuk meg. Ennek az eljárásnak ugyanis már semmi köze a komplex szám nyelvi szintű reprezentációjához. A komplex szám absztrakt matematikai fogalmára kell csak támaszkodnia, arra, hogy a komplex számot valós számpárral adjuk meg. A konkrét ábrázolás a konstruktor és szelektor eljárások ügye.

```
procedure pluskompl(var z:komplex; x,y: komplex);
  begin
    kompl(z,re(x)+re(y),im(x)+im(y));
    end;
```

Az összeadás helyesen

76. program

A 76. programban valóban csak az absztrakt matematikai lényeget írtuk le, a megfogalmazás független a komplex szám ábrázolásától. Ezek után nincs más dolgunk, mint megírni az összes matematikai művelet alprogramjait, valamint a beviteli és kiíratási alapeljárásokat, s ezeket egy 'cmplarit.pas' állománynévvel lemezre írni. Valahányszor komplex számokkal kell dolgozni, az

```
{$I cmplarit.pas}
```

direktívával a programunkba szerkesztjük.

Az eddig elmondottaknak azért van két komoly szépséghibájuk.
Az egyik az, hogy az absztrakció nem teljes. A komplex típust nekünk kell deklarálni a főprogramban - méghozzá a konstruktor és szelektor eljárásokkal összhangban - hiszen amíg a típusazonosítót nem definiáltuk, nem deklarálhatunk komplex típusú változókat. (Persze ezt is megoldhatjuk egy másik include állománnyal, ami csak a típusdeklarációból áll.) Az include állományok forrásnyelviek, így bárki bármikor belenézhet a tartalmukba, még módosíthatja is azokat. A Turbo Pascal 3.0-s változatában nincs lehetőségünk arra, hogy az absztrakt adattípusok megvalósításának részleteit eltakarjuk a felhasználó elől. Ilyen - ún. beburkolási - mechanizmusok több korszerű programozási nyelvben rendelkezésre állnak.

Ez a helyzet a 4.0-s verziótól kezdődően is.
A Turbo Pascal 4.0-s változatban megjelent egy új nyelvi eszköz, a "unit", ami a magasabb szintű absztrakció eszköze. A unit, mint sok más, az UCSD Pascalból került a Turbo Pascalba.

A Pascal unit konstansok, típusok, változók és alprogramok gyűjteménye, valójában egy Pascal programhoz hasonlít. A unitokban definiált objektumokat, eljárásokat és függvényeket bármely programban használhatjuk, ha a unitot a programfej után deklaráljuk.
Egy unit 4 részből áll: 1. unit fej,
2. nyilvános rész,
3. magán rész,
4. unit törzs.

A unit fej a unit kulcsszóból és az azt követő azonosítóból áll:

```
unit cmplarit;
```

Ha egy programban, vagy egy másik unitban egy unitban definiált eszközöket használni akarjuk, a nevet a uses kulcsszó után adjuk meg:

```
uses cmplarit;
```

A nyilvános részt az interface kulcsszó vezeti be. Az itt megadott objektumokhoz és alprogramokhoz fér hozzá a felhasználó, de itt kell deklarálni a unit által használt más unitokat (ha ilyen van).
A magán rész az implementation kulcsszóval kezdődik. Itt adjuk meg azokat a deklarációkat, amelyeket nem kell ismerni a felhasználónak és itt adjuk meg a nyilvános alprogramok törzsét is.
A unit törzse begin és end között helyezkedik el. Ide a unit alprogramjainak a működését előkészítő (inicializáló) utasításokat írhatjuk, ha ilyenre szükség van. A törzs utasításai a unitot használó program utasításait megelőzik. A unit szerkezete:

```
unit név;

  interface

    uses unitnev,unitnev,... ;(ha kell)

    (nyilvános deklarációk)

  implementation

    (magán deklarációk:
     konstansok, típusok,
     változók, alprogramok)

    (nyilvános alprogramok blokkja)

  begin

    (inicializáló program)

  end.
```

A unit felépítése

77. program

Az unitokat külön lefordíthatjuk. A programok a lefordított állapotú unitokat használják. Így a felhasználó valóban csak a

nyilvános részben deklarált erőforrásokhoz férhet hozzá, minden mást eltakartunk előle.
Megmutatjuk a cmplarit unit felépítését.

```
unit cmplarit;

{komplex adatszerkezet és aritmetika}

interface

  type komplex = record
                   re:real;
                   im:real;
                   end;

  procedure pluskompl(var z:komplex; x,y:komplex);
  {osszeadas: z:=x+y}

  procedure minuskompl(var z:komplex; x,y:komplex);
  {kivonas: z:=x+y}

  procedure szorkompl(var z:komplex; x,y:komplex);
  {szorzas: z:=x*y}

  procedure perkompl(var z:komplex; x,y:komplex);
  {osztas: z:=x/y}

implementation

  procedure kompl(var z:komplex; x,y:real);
    begin
      z.re:=x;
      z.im:=y;
      end; {konstruktor}

  function re(z:komplex):real;
    begin
      re:=z.re;
      end; {valos szelektor}

  function im(z:komplex):real;
    begin
      im:=z.im;
      end; {kepzetes szelektor}

  procedure pluskompl;
    begin
      kompl(z,x.re+y.re,x.im+y.im);
      end;

  procedure minuskompl;
    begin
      kompl(z,x.re-y.re,x.im-y.im);
      end;
```

```
procedure szorkompl;
  begin
    kompl(z,x.re*y.re-x.im*y.i,
            x.re*y.im+x.im*y.re);
    end;

procedure perkompl;
  var r:real;
  begin
    r:=sqr(y.re)+sqr(y.im);
    kompl(z,(x.re*y.re+x.im*y.im)/r,
            (x.im*y.re-x.re*y.im)/r);
    end;

begin
end.
```

Komplex aritmetikai unit

78. program

A közölt unit korántsem teljes. A négy alapművelet mellett definiálni kellene az egyenlőség relációt a komplex értékekre, az abszolút értéket, az argumentumot. Célszerű volna a hatványozást és gyökvonást a Moivre-képlet alapján megvalósítani, ezért az algebrai és a trigonometrikus alak közötti átalakításokról is gondoskodni kellene. Komolyabb alkalmazások nem képzelhetők el anélkül, hogy a gyakrabban használt matematikai függvényeket ne általánosítanánk a komplex számokra. Meg kellene írni a komplex számokhoz a beolvasó és kiíró eljárásokat is.
A 78. programot mindössze példának szántuk. Tulajdonképpen a konstruktor és szelektor eljárások használatától nem szokás elzárni a felhasználót, mi is csak a bemutatás kedvéért deklaráltuk ezeket a magán részben.
Az alprogramok nem igényelnek inicializálást, ezért a unit törzse üres.

Még egyhiányérzetünk van a komplex aritmetika megvalósításával kapcsolatban. Az, hogy a műveleteket nem írhatjuk a szokásos szintaxissal, ehelyett eljáráshívásokkal dolgozhatunk. A

```
a:=b+c
```

értékadást nem használhatjuk a komplex számoknál, ehelyett a kevésbé kifejező

```
pluskompl(a,b,c)
```

eljárásutasítást használjuk. Jó volna, ha a "+"-jelet is használhatnánk eljárásnévként, különösen, ha nem prefixumként

```
+(a,b,c),
```

hanem a szokásos infix műveleti jelként írhatnánk. Ekkor már semmi sem emlékeztetné a felhasználót arra, hogy nem a nyelvi szinten, hanem absztrakcióval definiált műveletekkel dolgozik. Ilyen lehetőség létezik pl. az Ada programozási nyelvben, a Turbo Pascalban nem. De hát ne legyünk telhetetlenek.

10.5 Táblázatok

A rekordoknak egy fontos alkalmazási területe a táblázatok gépi ábrázolása. Egy táblázat különböző tartalmú oszlopokból, de azonos szerkezetű sorokból épül fel (82. ábra).

Megnevezés	Mennyiség	Egységár	Érték
HB ceruza	12	6.50	73.00
Radír	5	24.20	121.00
.....	.	.....	-----
.....	.	.....	-----

Táblázat

82. ábra

A táblázat egy sorát rekord típussal, az egész táblázatot rekordok tömbjeként adhatjuk meg:

```
const maxsor=100;

type str20=string[20];
     sortipus=record
                megnevezes:str20;
                mennyiseg:integer;
                egysegar:real;
                ertek:real;
                end;
     tabltipus=array[1..maxsor] of sortipus;
```

A táblázatokkal végzett legfontosabb műveletek:

1. keresés a táblázatban,
2. a táblázat sorainak rendezése, valamelyik oszlop tartalma szerint.

Először a keresés feladatával foglalkozunk. A keresett rekordot valamelyik mező tartalma szerint keressük. Ezt a mezőt - amely a keresés szempontjából az egész rekordot képviseli - kulcsmezőnek nevezzük.

Ha a táblázatunk a kulcs szerint nem rendezett, akkor csak az ún. soros keresést tudjuk alkalmazni. A keresõ kulcsot ilyenkor rendre összehasonlítjuk a rekordok kulcsával és ha egyenlõek, akkor megvan a keresett rekord. Ha szerencsénk van, már az elsõ tömbelemek között megtaláljuk a keresett rekordot, ha nincs akkor csak az utolsók között. Így, ha n db rekordunk van, akkor az átlagos keresési idõt n/2-vel arányosnak becsülhetjük. A legkellemetlenebb, ha nincs a táblázatban a keresett rekord, mert ez csak az összes elem megvizsgálása után fog kiderülni.
Minden esetre ezt a könnyen programozható algoritmust (79. program) kisméretû táblázatok esetében sikerrel alkalmazhatjuk.

```
function keresett(x:tabltipus;
                  meret:integer;
                  k:kulcstipus):integer;
  var index,i:integer;

  begin
    index:=0;
    i:=1;
    while (index=0) and (i<=meret) do
      if x[i].kulcs=k then
        index:=i;
    keresett:=index;
    end;
```

Soros keresés

79. program

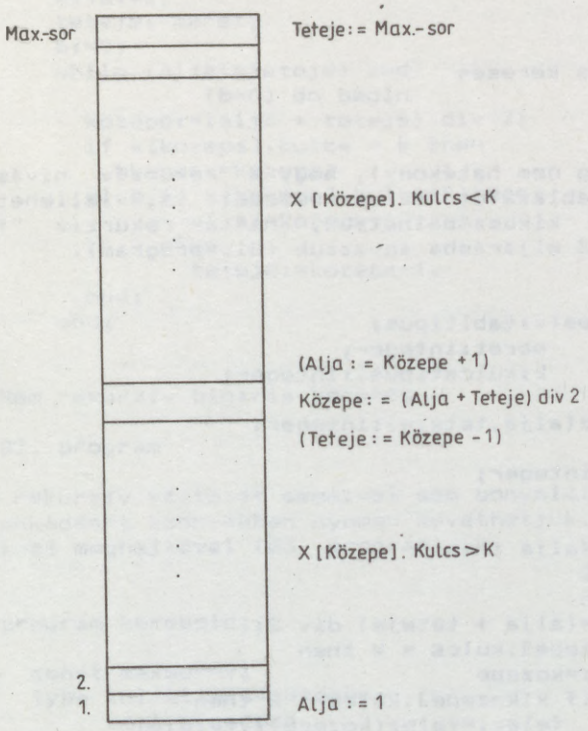
Az eljárás használatának elõfeltétele egy

```
type kulcstipus = ......;

     sorrekord = record
                   ........
                   ........
                   kulcs:kulcstipus;
                   ........
                   ........
                   end;

     tabltipus = array[1..*] of sorrekord;
```

globális típusok létezése. Ha megtalálta a keresett rekordot, az indexét adja vissza a "keresett" függvény, ha nem, akkor 0-t. Ennél az algoritmusnál nem kell a kulcsnak rendezett típusúnak lenni, elég, ha az "=" reláció értelmezett.

Rendezett táblázatokban a rendkívül hatékony bináris (vagy logaritmikus) keresést tudjuk alkalmazni. Ennek a lényege az, hogy felezésekkel határoljuk be a keresett elem helyét.

Legyen az x tömb a táblázat, amelyben keresünk és legyen k a keresőkulcs. Először megvizsgáljuk a táblázat középső elemét, ennek indexe (1+maxsor) div 2. Ha nem ez a keresett elem, akkor megállapítjuk, hogy nagyobb vagy kisebb, majd eszerint keresünk tovább a táblázat felső vagy alsó felében (83. ábra).



A bináris keresés

83. ábra

Megfogalmazhatjuk a keresés algoritmusát rekurzíve (80. program).

```
function binkeres(x:tabltipus;
                  alja,teteje:integer;
                  k:kulcstipus):integer;
  var kozepe:integer;

  begin
    if teteje < alja then
      binkeres:=0
    else begin
      kozepe:=(alja + teteje) div 2;
      if x[kozepe].kulcs = k then
```

```
            binkeres:= kozepe
            else if x[kozepe].kulcs < k then
                   binkeres:=binkeres(x,kozepe+1,teteje,k)
                 else
                   binkeres:=binkeres(x,alja,kozepe-1,k);
          end;
        end;
```

Rekurzív bináris keresés

80. program

Nem elegáns (és főleg nem hatékony), hogy a rekurzív hívásoknál mindig átadjuk a táblázatot és a kulcsot is, jóllehet nem változnak meg. Ezt kiküszöbölhetjük, ha a rekurzív "felez" alprogramot egy külső eljárásba ágyazzuk (81. program).

```
function binkeres(x:tabltipus;
                  meret:integer;
                  k:kulcstipus):integer;

  function felez(alja,teteje):integer;

    var kozepe:integer;

    begin
      if teteje<alja then
        felez:=0
      else begin
        kozepe:=(alja + teteje) div 2;
        if x[kozepe].kulcs = k then
          felez:=kozepe
          else if x[kozepe].kulcs < k then
                 felez:=felez(kozepe+1,teteje)
               else
                 felez:=felez(alja,kozepe-1)
        end;
      end;

  begin
    binkeres:=felez(1,maxsor);
    end;
```

Gazdaságosabb rekurzió

81. program

Érdemes elkészíteni a program nem rekurzív változatát is. Itt az "alja" és "teteje" változók értékét állítjuk be, hogy mindig a keresett elemet tartalmazó tömbbel ismételjük meg a kereső (és felező) lépést.

```
function bkeres(var x:tabltipus;
                meret:integer;
                k:kulcstipus):integer;
  var alja,teteje,kozepe,b:integer;

  begin
    alja:=1;
    teteje:=meret;
    b:=0;
    while (alja<=teteje) and
          (b=0) do begin
      kozepe:=(alja + teteje) div 2;
      if x[kozepe].kulcs = k then
        bkeres:=kozepe;
      else if x[kozepe].kulcs<k then
             alja:=kozepe+1
           else
             teteje:=kozepe-1;
      end;
    end;
```

Nem rekurzív bináris keresés

82. program

A nem rekurzív változat semmivel sem bonyolóltabb a rekurzívnál, sőt, működését könnyebben nyomon követhetjük. Tegyük is meg a következő meghajtóval (83. program).

```
program keresp1d;

  const maxsor=9;
  type kulcstipus=integer;
       adat = record
                kulcs:kulcstipus;
                mas:char;
                end;
       tabltipus=array[1..maxsor] of adat;
  const tabl:tabltipus = ((kulcs:1;mas:'a'),
                          (kulcs:3;mas:'b'),
                          (kulcs:5;mas:'c'),
                          (kulcs:7;mas:'d'),
                          (kulcs:8;mas:'e'),
                          (kulcs:9;mas:'f'),
                          (kulcs:10;mas:'g'),
                          (kulcs:11;mas:'h'),
                          (kulcs:14;mas:'g'));
  var mit:kulcstipus;
      ind:integer;
```

```
{$I bkeres.pas}

begin
  clrscr;
  write('keresokulcs:');
  readln(mit);
  writeln;
  ind:=bkeres(tab,maxsor,mit);
  if ind=0 then
    writeln('ismeretlen kulcs')
  else
    writeln('az index=',ind,' az adat=',tab[ind].mas);
  end.
```

Meghajtóprogram a kereséshez

83. program

Ha keresőkulcs értéke 5, akkor a következőképp zajlik a keresés:

alja	teteje	kozepe	egyéb információ
1	9	5	tab[5].kulcs=8>5
	4	2	tab[2].kulcs=3<5
3			tab[3].kulcs=5, kész

Még érdemes megnézni mi történik, ha nincs a keresőkulccsal azonos kulcsó rekord a táblázatban. Legyen 13 a keresőkulcs értéke.

alja	teteje	kozepe	egyéb információ
1	9	5	tab[5].kulcs=8<13
6		7	tab[7].kulcs=10<13
8		8	tab[8].kulcs=11<13
9		9	tab[9].kulcs=14>13
	8		alja>teteje, nincs

Kérdés, hány lépésben talál meg egy keresett rekordot ez az algoritmus. A lépések számát a felezések számával jellemezhetjük (hányszor számítja a "kozepe" értékét). Tegyük fel, hogy n elemünk van és az egyszerűség kedvéért legyen

$$n = 2^k .$$

Az első felezés után 2^{k-1}, a második után 2^{k-2} elemünk marad, így a k-adik lépésben már egy elemre csökkent az intervallum. Ez azt jelenti, hogy

$$k = \log_2 n$$

lépésben a legrosszabb esetben is végzünk. Ha n nem pontosan 2 hatványa, akkor a legkisebb k kitevővel számolhatunk, amelyre

$$n < 2^k ,$$

tehát a lépések száma

$$\mathrm{trunc}(\log_2 n) + 1.$$

A felezések száma tehát n logaritmikus függvénye. Ezért a tulajdonságáért szokták ezt a keresési algoritmust logaritmikus keresésnek is nevezni.

10.6 Táblázatok rendezése

A bináris keresés alkalmazásának előfeltétele a táblázat kulcsmező szerinti rendezettsége. Ha nem rendezett, akkor megfelelő programmal rendezni kell.
Rendezőprogramokról sokan és sokat írtak. Nem célunk mélyebb betekintést szerezni ebbe a fontos témába. Mindössze 3 alapvető algoritmust mutatunk be.

A buborékrendezés a szomszédos elemek felcserélésén alapszik. Tekintsük a következő számsort, amit növekvő sorrendbe kell rendezni:

3 5 2 4.

Balról jobbra haladva 3 és 5 sorrendje megfelelő, 5 és 2 felcserélendő. Ekkor a sorrend

3 2 5 4

és a 3. és 4. elem összehasonlítása következik, amelyeket ugyancsak fel kell cserélni és így az első átfésülés után

3 2 4 5

a sorrend. Még nem végeztünk, ezért újra kezdjük az 1. és a 2. elem összehasonlításával.
A következő menet végre már kialakul a végleges sorrend. Vegyük észre, hogy n szám rendezésekor a legrosszabb esetben is csak n-1 menetre van szükség. Ez akkor fordul elő, ha a legkisebb érték a legnagyobb elem helyén áll. Ekkor minden menetben egyel kerül közelebb a végleges helyéhez. Az esetek többségében valószínűleg lényegesen kevesebb menet is elég.

Ezért minden menetben célszerű figyelni, hogy nem végeztünk-e már a rendezéssel. Ha már nem volt szükség cserére, akkor tudjuk, hogy végeztünk. Az algoritmust úgy készítjük el, hogy növekvő és csökkenő sorrendbe is tudjunk rendezni. Ezt a harmadik paraméterrel (novo) adjuk meg. Ha ennek értéke negatív, akkor csökkenő, minden más esetben növekvő lesz a kulcsok sorrendje.

```
procedure buborek(var x:tabltipus;
                  meret:integer
                  novo:integer);
  var i: integer;
      cserevan:boolean;

  {$I csere.pas}
  begin
    cserevan:=true;
    while cserevan do begin
      cserevan:=false;
      for i:=1 to meret-1 do
          if novo<0 then
            if x[i].kulcs<x[i+1].kulcs then begin
               csere(x[i],x[i+1]);
               cserevan:=true;
               end;
          else                                              if
x[i].kulcs>x[i+1].kulcs then begin
               csere(x[i],x[i+1]);
               cserevan:=true;
               end;
    end;
  end;
```

Buborékrendezés

84. program

A rekordok felcserélésére a csere eljárást természetesen el kell készíteni.

```
procedure csere(var x,y: elemtipus);
  var t: elemtipus;
  begin
    t:=x;
    x:=y;
    z:=t;
  end;
```

Rekordok felcserélése

85. program

Feltételeztük, hogy a táblázat elemeinek típusa "elemtipus". Egy másik rendezőalgoritmus a szélső elemek ismételt kiválasztásán alapszik, ezért kiválasztásos rendezésnek nevezzük. Ezzel a módszerrel a következőképpen rendezhetünk növekvő sorrendbe:

```
for i:=1 to meret-1 do
  válassz az i., (i+1). ... utolsó elemek közül minimálisat
  cseréld fel az i. elemmel
```

Az algoritmus működését a

3 7 5 1 2

adatsor rendezésénél kövessük nyomon. Az első elemmel kezdve a minimum meghatározását 1-et kapunk, amit az első elemmel kell felcserélni:

1 7 5 3 2.

Ezután a második elemmel kezdjük a minimális keresését, ez 2, amit a második elemmel cserélünk fel:

1 2 5 3 7.

Ezután az

1 2 3 5 7

adatsort kapjuk. Az algoritmus végrehajtása ezzel ugyan még nem fejeződött be, de több cserére már nem kerül sor.
Az elemeket felcserélő eljárásunk készen van, még a minimális kulcsú elem indexének meghatározásához kell egy függvényt írni (86. program).

```
function minkulcs(alsoindex:integer):integer;
  var i:integer;
      minind:integer;
      mink:kulcstipus;
  begin
    minind:=alsoindex;
    mink:=tabl[minind].kulcs;
    for i:=alsoindex+1 to meret do
       if tabl[i].kulcs<mink then begin
          mink:=tabl[i].kulcs;
          minind:=i;
          end;
    minkulcs:=minind;
    end;
```

A minimális kulcs indexe

86. program

Ezután a rendezőprogramot könnyen "összeszerelhetjük".

```
procedure kivalasztasos(var tabl:tabltipus;
                        meret:integer);
  var i:integer;

  {$I csere.pas}

  {$I minkulcs.pas}

  begin
    for i:=1 to meret-1 do
       csere(tabl[i],tabl[minkulcs(i)]);
    end.
```

Rendezés a minimális elem kiválasztásával

87. program

Más rendezési algoritmust használnak a kártyások. A balkézben tartott lapokat két csoportba osztják: a "kész" csoportra és a "többi"-re. A "többi"-ből húzott lapokat beillesztik a megfelelő helyre a "kész" sorozatba. Figyeljük meg számpéldán az eljárást.

KÉSZ sorozat	TÖBBI lap
3	7 5 1 2
3 7	5 1 2
3 5 7	1 2
1 3 5 7	2
1 2 3 5 7	

A táblázat persze egyetlen tömböt alkot és csak az indexek alapján választjuk külön a "kész" és a "többi" részt. Először az egyetlen első elem alkotja a "kész" sorozatot, a 2-tes indextől kezdődik a "többi". Az algoritmus:

```
for i:=2 to meret do
  az i-edik elem beillesztése a "kész" elemek közé
```

Az elem beillesztése ún. rostáló eljárással történhet. Ez abból áll, hogy a "kész" elemeket addig "hózogatjuk " jobbra, amíg a beillesztendő elem helyét szabaddá nem tesszük. Ehhez a jobb oldalonegy szabad helyre van szükségünk, ezért az i-edik elemet egy munkaváltozóba írjuk, hogy a helye szabaddá váljon. A

3. lépésben az 1-es beillesztése a következő lépésekkel történik:

```
  3   5   7          t:=1
  3   5     7
  3     5   7
    3   5   7
```

Az "üres" helyet a jelzi. Addig rostálunk, amíg a beszórandó elem kisebb, mint az, amit éppen jobbra akarunk tolni. A példában éppen baj van a befejezéssel, mert több elem nincs, de a feltétel még igaz. Írjuk azonban a "kész" sorozat elemei elé a beszórandó értéket strázsának:

```
  1   3   5   7.
```

Mivel a következő elemnél (1) már nem kisebb a beszórandó (1), befejezzük a rostálást és a jelölt helyre beírjuk az elemet:

```
  1   1   3   5   7.
```

Így nem kell külön vizsgálni, hogy mikor érünk a "kész" sorozat bal végére.
Az algoritmus:

```
procedure beszuras(var tabl:tabltipus;
                       meret:integer);
  var i,j:integer;
      t:elemtipus;
  begin
    for i:=2 to meret do begin
      t:=tabl[i];
      {a strazsa beallitasa}
      a[0]:=t;
      j:=i-1;
      {rostalas}
      while t.kulcs<tabl[j].kulcs do begin
        tabl[j+1]:=tabl[j];
        j:=j-1;
        end;
      tabl[j+1]:=t;
      end;
   end;
```

Rendezés beszórással

88. program

A beszórásos technikát olyankor is alkalmazhatjuk, ha a táblázat nem minden elemét ismerjük eleve.
A megismert rendezőalgoritmusok olyanok, amelyeknél a rendezéshez szükséges idő az elemek számának négyzetével arányos. Ez hosszú adatsorok esetén nem tól kedvező. Léteznek olyan algoritmusok is,

amelyeknél kevésbé gyorsan novekszik a rendezési idő, csak n*log n mértékben. Ilyen pl. a Quick Sort nevű algoritmus.

A rendezés közben végzett elemcserék - különösen ha nagyméretű rekordok az elemek - ugyancsak lassítják a rendezést. Mindegyik rendezőprogramot megírhatjuk úgy is, hogy a rekordokat ne mozgassa, csak az elemek indexeit cserélgesse egy külön e célra deklarált indextömbben. Ilyenkor indexrendezésről beszélünk. Tekintsük pl. a

	x		ind
1.	3	1.	1
2.	7	2.	2
3.	2	3.	3
4.	1	4.	4
5.	6	5.	5

rendezendő elemek — indexek

tömböket. A rendezés eredményeként a következőket kapjuk:

	x		ind
1.	3	1.	4
2.	7	2.	3
3.	2	3.	1
4.	1	4.	5
5.	6	5.	2

rendezendő elemek — az elemeknek megfelelően rendezett indexek

Pl. a rendezés után a legkisebb elemet a következőképpen olvashatjuk ki:

```
k:=ind[1];
min:=x[k];
```

Indexrendezésnél a kulcsokat hasonlítjuk ugyan össze, de csak az indexeket cseréljük fel. Bármelyik megismert módszert használhatjuk, pl. a buborékrendezés elvét is (84. program).

```
procedure buborek(var x:tabltipus;
                  ind:indtomb;
                  meret:integer
                  novo:integer);

  var i: integer;
      cserevan:boolean;

  procedure csere(var m,n:integer);
    var t:integer;
    begin
      t:=m;
      m:=n;
      n:=t;
      end;

  begin
    cserevan:=true;
    while cserevan do begin
      cserevan:=false;
      for i:=1 to meret-1 do
          if novo<0 then begin
             j:=ind[i];
             j1:=ind[i+1];
            if x[j].kulcs<x[j1].kulcs then begin
               csere(ind[i],ind[i+1]);
               cserevan:=true;
               end;
            end
          else begin
            j:=ind[i];
            j1:=ind[i+1];
            if x[j].kulcs>x[j1].kulcs then begin
               csere(ind[i],ind[i+1]);
               cserevan:=true;
               end;
      end;
    end;
```

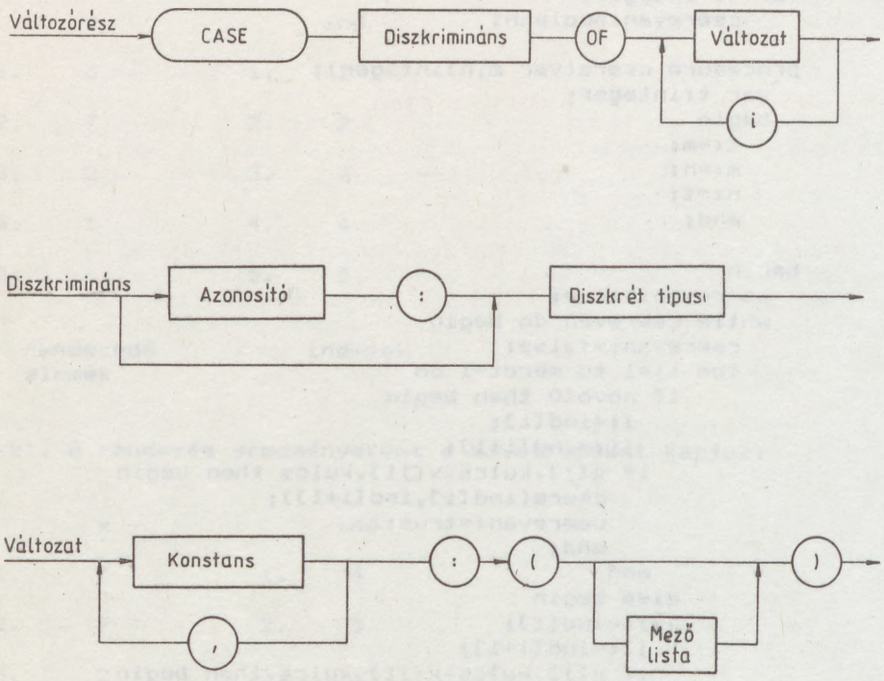
Indexrendezés

89. ábra

Az ind tömb elemeit - amik x-beli indexek - mutatóknak (pointer) szokás nevezni. (Ez csak fogalmilag azonos a 12. fejezetben tárgyalandó pointer típussal.)

10.7 Változatos rekordok

Eddig csak olyan rekordokkal foglalkoztunk, amelyek csak állandó részből álltak. A változat rész szintaxisát a 84. ábrán mutatjuk be. A változat részben ugyanazon tárterületet többféle attributummal láthatjuk el. A rekordnak ehhez a részéhez különböző mezőlistákat is elhelyezhetünk, így többféle módon férhetünk hozzá e területhez. Minden mezőlista egy változat.

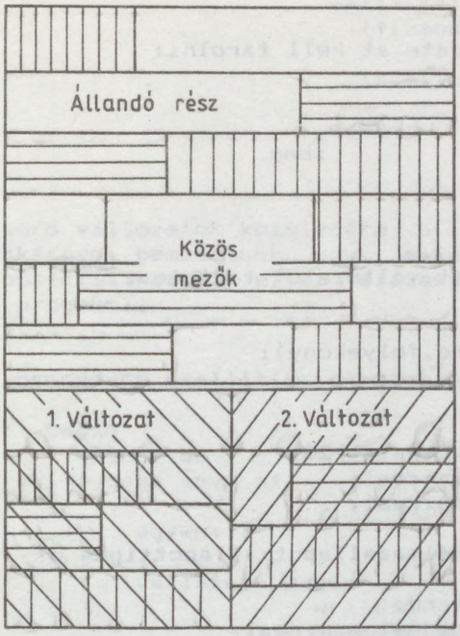


A változat rész szintaxisa

84. ábra

Minden változat ugyanazon a tárterületen helyezkedik el és az összes változat bármelyik mezőjéhez bármikor hozzáférhetünk (85. ábra).
A szintaxis ábrából látható, hogy minden változatot egy vagy több konstanssal azonosítunk. A konstansoknak különbözőknek kell lenni, és kompatíbilisnek a diszkriminánsban megadott típussal. A diszkrimináns diszkrét típusú lehet.
A változatok mezőihez ugyanúgy férhetünk hozzá, mint az állandó részben lévőkhöz. A diszkriminánsban megadhatunk egy változót is. A diszkrimináns változó - ha szerepel - a rekord állandó részének egy további mezőjét, a diszkrimináns mezőt alkotja. Ennek a mezőnek az értéke mutatja meg, hogy egy adott esetben melyik

változat van érvényben. Diszkrimináns változó nélkül más szempontok alapján kell meghatároznia a programnak a megfelelő változatot (pl. a rekord más mezőinek tartalma alapján).



A változatos rekord szerkezete

85. ábra

A változat rész csak a rekordszerkezet végén helyezkedhet el.
A változatos rekord által meghatározott értékhalmaz az egyes változatok értékhalmazának egyesítése:

V = R1 R2 ... Rk,

ha a rekorban k számú változatot definiáltunk. Ha szerepel a diszkrimináns változó, akkor az egyesítésen belül az egyes változatokat annak az értéke szerint tudjuk megkülönböztetni. Ilyenkor megkülönböztetett egyesítésről beszélünk. Ha nincs diszkrimináns változó, akkor a rekord kezelése nem mindig biztonságos, ezért ilyen megoldást csak nagy körültekintéssel alkalmázzunk. Ilyen esetben azt mondjuk, hogy a változatos rekord szabad egyesités.
Rekord változatokat olyankor használunk, ha a feldolgozásunk alapvető szempontjából azonosnak tekintett dolgok részben eltérő tulajdonságokkal rendelkeznek. Pl. egy vegyszerkatalógusban a szilárd anyagoknál a

megnevezés,
molekulasúly,
sűrűség,
oldhatóság,
olvadáspont.

A folyadékokról részben más adatokat kell tárolni:

megnevezés,
molekulasúly,
sűrűség,
forráspont,
törésmutató,
viszkozitás.

A megfelelő rekordszerkezet két változatot tartalmaz.

```
type allapottipus=(szilard,folyekony);
     str20=string[20];

     anyagrekord=record
                   megnevezes:str20;
                   molsuly:real;
                   suruseg:real;

                   case halmazallapot:allapottipus of

                     szilard:
                       (oldhatosag:real;
                        olvadaspont:real);
                     folyekony:
                       (forraspont:real;
                        toresmutato:real;
                        viszkozitas:real);
                   end;
```

A diszkriminánsváltozó számára nem kell külön mezőt meghatározni, ez automatikusan megtörténik.

Ügyviteli területről is mutatunk példát.

```
type vevotipus = (vallalat, egyen);
     str20 = string[20];
     str40 = string[40];

     eladastipus=record
                   szamlaszam:0..9999;
                   termek:str40;
                   termekjel:0..9999;
                   mennyiseg:integer;
                   egysegar:real;
                   osszeg:real;
```

```
                    nev:str20;
                    cim:str40;
                    kelt: datum;
                    case vevo:vevotipus of
                      vallalat:
                        (fizmod:(kp,atut,csekk,
                                 belyegzo,inkasszo);
                        (szallitas:(mav,sajatgk,
                                    bergk,vevogk));
                      egyen: ();
                  end;
```

A rekord változatok kezelésére a case szerkezetet használjuk. Példaképpen bemutatunk egy eljárást, amely a szállítási diszpozíciót készíti el az "eladastipus" rekord tartalma alapján (89. program).

```
procedure szallit(x: eladastipus);

  begin
    with x do
      case vevo of

        egyen: ;

        vallalat: begin
                    writeln('szamlaszam:',szamlaszam);
                    datumwrite(kelt);
                    writeln('termekjel:',termekjel);
                    writeln('termek:',termek);
                    writeln('mennyiseg:'mennyiseg);
                    writeln;
                    write('szallitas modja:');

                    case szallitas of
                      mav:writeln('MAV haztol hazig');
                      sajatgk:
                          writeln('Sajat gepkocsinkkal');
                      bergk:
                          writeln('Volan gepkocsival');
                      vevogk:
                          write('A szallitasrol a ');
                          writeln('vevo gondoskodik');
                      end;
                    end;
        end;
    end;
```

A rekord változatok kezelése

90. program

A 90. programban feltételeztük, hogy rendelkezésre áll "datumwrite" néven egy eljárás dátumrekord típusú értékek kiíratására.

10.8 Feladatok

1. Definiáljon rekordszerkezetet, amely nevekből, címekből, születési adatokból és névnapok dátumaiból áll! Készítsen eljárást, amely ellenőrzi a dátumok helyességét.

2. Tegyük fel, hogy az 1. feladat rekordjai táblázatot alkotnak. Készítsen programot, amely megadott hónapra kiírja az esedékes születésnapokat és névnapokat (üdvözlőlap küldése céljából).

3. Definiáljon olyan rekordszerkezetet, amelynek tartalma alapján munkabéreket tud számfejteni! Készítsen eljárást a rekord tartalmának kiíratására.

4. Írjon beolvasó eljárást a 3. feladat rekordjához.

5. Írjon eljárást, amely a 3. feladat rekordjának tartalmából meghatározza a munkabért, e levonásokat és kinyomtatja a dolgozó számára a szükséges adatokat.

6. Írja meg - megfelelően definiált dátumtípusra - a 89. programban használt datumwrite eljárást.

7. Készítsen programot, amely különböző síkidomok területét és kerületét számítja ki. A szükséges adatokat változatos rekord változataiban adja meg. A diszkrimináns a síkidom neve legyen!

8. Rendezze az 1. feladat rekordjaiból álló táblázatot a születésnapi dátumok szerint! Írassa ki a rendezett táblázatot!

9. A buborékrendezés hatékonyságát növelhetjük, ha felváltva az adatsor ellenkező végéről kezdjük az átfésülést. Pl. a

   3 2 5 1 4

   számok esetén az egyes átfésülések utáni új sorrend (először balról kezdjük):

   2 3 1 4 5

   1 2 3 4 5

   Tehát már a másodszorra megkapjuk a helyes sorrendet.
   Írjon eljárást, amely egy táblázat elemeit ezzel az algoritmussal rendezi.

11. ADATÁLLOMÁNYOK

11.1 A file típusok

Az adatállományok megvalósítására a Pascal nyelv a file adattípust használja. A file olyan adattípus, amely meghatározatlan számú azonos típusú érték összessége. Ez az értéktípus a file bázistípusa.
A bázistípus - file kivételével - bármilyen összetett típus is lehet. Itt jegyezzük meg, hogy

> a file típus összetett típus bázistípusa nem lehet.

File típust a file alapszó segítségével definiálunk:

```
type <típusnév> = file of <bázistípus>.
```

Pl.:

```
type egeszfile = file of integer;

type kartyafile = file of string[80];

type szovegfile = file of char;

type adatfile = file of adatrekord;
```

Az egeszfile integer típusú értékek meghatározatlan hosszúságú sorozata. A kartyafile 80 karakteres füzéreké. Egy elemével a számítástechnikában adatbevitel céljára használt 80 oszlopos lyukkártya tartalmát adhatjuk meg, innen az elnevezés. A szövegfile tetszőlegesen hosszú jelsorozat, az adatfile rekordok sorozata.

A korábban megismert típusok értékhalmazai elvileg is véges halmazok voltak. A file típusok értékhalmaza végtelen számosságú, mivel ezt a halmazt a bázistípus értékeiből képzett összes lehetséges sorozat alkotja. Egy ilyen sorozat maga sem korlátos, hiszen bármilyen hosszú is, nála egy elemmel hosszabb sorozat is létezik.

Egy végtelen halmaz értékeit általában nem tudjuk a tárban ábrázolni. A file típusú értékeket - ha már létrehoztuk - háttértárakban tároljuk. A PC-nél hajlékony lemezen vagy winchesteren. A tárba a teljes érték helyett annak egy elemét tartjuk, ezen az egyetlen elemen tudunk műveleteket végezni. Tehát a file típusú értékeket elemenként dolgozzuk fel.

Fizikai értelemben véve a file típusú értékek az adatállományokkal azonosak. A file valamely eleméhez pedig beviteli és kiviteli tevékenységek révén férünk hozzá, az alapvető file műveletek a már eddig is használt read és write eljárások.
Ezen az sem változtat, hogy ezeket az eljárásokat nem adatállományok kezelésére, hanem a billentyűzetről való adatbevitelre és a képernyőre történő kiírásra használtuk. A Pascal nyelvben ugyanis a beviteli adatokat is és a képernyőn megjelenő információt is adatállománynak tekintjük. Ezek a rendszer előre definiált (szabványos) adatállományai.
Egy szabványos adatállomány logikailag egy előre definiált "konstans", amelynek mindekori értéke a meghatározott készülékkel átvitt bemeneti vagy kimeneti adatsorozat.

A Pascal nyelvnek jellegzetessége, hogy az adatállományokat sorozatoknak tekinti. Ez az ún. soros adatállomány fogalmának felel meg. Az ilyen adatállományokat az jellemzi, hogy az egyes elemekhez meghatározott sorrendben férhetünk hozzá. Az 1. elem után a 2.-at, a 2. után a 3.-at, általában a k-adik után a k+1-ediket érhetjük el. Sem visszalépés, sem pedig előreugrás nem lehetséges. Nem férhetünk hozzá a 28. elemhez anélkül, hogy előtte hozzá ne fértünk volna a megelőző 27-hez. Továbbá egy soros adatállománnyal mindig csak egyféle tevékenységet végezhetünk: vagy olvashatjuk vagy írhatjuk. Nem lehet az egyik elemet olvasni, azután egy másikat (vagy ugyanazt módosítva) írni. Ennek elvi és gyakorlati oka is van. Egy adatállomány egy file típusú érték, és egy értékkel egyszerre csak egy műveletet végezhetünk.
A másik ok a soros adatállományok eredetéhez, a mágnesszalagon tárolt adatállományokhoz nyúlik vissza. A mágnesszalagos adattárolásnál egyszerűen fizikailag nem volt lehetőség az író és olvasó tevékenységek keverésére.

Az adatállományok elemeit a számítástechnikában gyakran "rekordnak" nevezik. Mivel ezt a kifejezést a Pascalban már másra lefoglaltuk, kerülni fogjuk ezt a szóhasználatot. Ugyanakkor az adatállományok elemei nagyon sokszor rekord típusú értékek.

Az adatállományok - amint ez egy értéktől el is várható - mindenféle konkrét programtól függetlenül léteznek. A programokban file típust és file típusú objektumot deklarálunk. A file objektumhoz az értékét, a tőle függetlenül létező fizikai adatállományt hozzá kell rendelni. Ezt a file "értékadást" az

```
assign
```

eljárással valósíthatjuk meg. Az eljárásnak két paramétere van, az első a file azonosító, amit a változó azonosítókhoz hasonlóan kell deklarálni, a második az állománynév - ahogy a lemez tartalomjegyzékében található. Az állománynevet füzérként kell megadni.
Tegyük fel, hogy létezik a lemezen egy egészeket tartalmazó adatállomány "számsor.int" néven. A programban a következő

utasításoknak kell szerepelni, hogy ezt az adatállományt egy file objektumhoz hozzárendeljük:

```
type szamfiletipus = file of integer;

.............................

var szamfile:szamfiletipus;

.............................

begin

  ...........

  assign(szamfile,'szamsor.int');

  ...........
```

A var deklarációval létrehozott "szamfile" nevű objektum konstans jellegét misem mutatja jobban, mint az a szabály, hogy ezt a hozzárendelést ugyanebben a programban nem változtathatjuk meg. Egy programban egy file azonosító csak egyetlen assign eljáráshívásban fordulhat elő.
A file azonosítók értékadásban sem fordulhatnak elő.

Minden file típusú objektum deklarálásával egy file ablakot definiáltunk a tárban. Ez az ablak kapcsolja az adatállományt fizikailag a programhoz, ebben az ablakban jelennek meg a beviteli utasítások hatására egymás után az adatállomány soron következőelemei. Úgy képzeljük, hogy az ablak áll, az adatállomány "mozog". Mintha egy filmcsíkot húznánk el egy rés előtt, a résen át mindig csak egyetlen képet látunk, s a filmet minden read utasítás egy kockányival továbbítja.

A file ablakot a Turbo Pascalban (más Pascal változatoktól eltérően) nem tudjuk közvetlenül kezelni, csak a read és a write eljárásokon keresztül gyakorolhatunk rá hatást. A file ablakot file mutatónak is nevezzük.

11.2 File műveletek

Ha egy adatállománnyal dolgozni akarunk, akkor először az assign eljárással hozzárendeljük a programban deklarált file azonosítóhoz. Ha az állomány egy DOS könyvtárban van, akkor az elérési útat is meg kell adni a névvel együtt.

Mielőtt az első olvasási vagy írási utasítást kiadnánk, a file-t meg kell nyítni. A

```
reset
```

eljárással olvasásra tudunk megnyítni egy file-t. Az eljárás paramétere a file azonosítója:

```
reset(szamfile).
```

Az olvasásra megnyított file-nak értékének kell lenni. Ez azt jelenti, hogy a lemezen léteznie kell az assign-ben megadott állománynak. Írásra a

```
rewrite
```

eljárással nyíthatunk meg file-t, az aktuális paraméter itt is a file azonosító:

```
rewrite(szamfile).
```

Ha a megfelelő assign utasításban meghatározott néven még nem létezik állomány a lemezen, akkor létrejön, ha létezett, akkor a tartalma megsemmisül (új jön létre helyette és ez üres). Ezért - hacsak nem akarjuk szándékosan törölni - létező állományt mindig reset-tel nyissunk meg. A rewrite eljárás által létrehozott állomány üres, egyetlen elemet sem tartalmaz.
Megnyítás után a file ablakban a file eleje látszik - a file mutató a file 0. sorszámú elemére mutat.
Reset után a megnyított file elemeit az ismert read eljárással olvashatjuk. Az eddigi használattól eltérően (ami később ismertetendő okból kivételt képezett) a read eljárást általában a file azonosítóval is paraméterezni kell. A

```
read(szamfile,j)
```

utasítással a j változóba olvassuk be a file következő elemét. A read utasítás a file mutató értékét eggyel megnöveli (az ablakba a következő elem kerül).
Ha írásra nyítottuk meg a filet, akkor a write utasítással írhatunk bele egy elemet, mindig utolsó elemként. Itt is meg kell adnunk általában a file azonosítót paraméterként:

```
write(szamfile,j).
```

Ezzel az eljáráshívással a j (integer típusú) változó értékét írjuk a file értékét képező számsorozatba utolsó elemként. A write eljáráshívás is megnöveli 1-gyel a file mutató értékét.
Ha befejeztük egy file feldolgozását (írását vagy olvasását), le kell zárnunk. Erre való a

```
close
```

utasítás, amit a megnyitó utasításokhoz hasonlóan a file

azonosítóval kell paraméterezni.
A close utasítás teendői sokrétűek. Számunkra a legfontosabb a puffer ürítése. A puffer (ütköző tároló) a tárnak egy területe, ahol az adatállományból egyszerre beolvasott adatok helyezkednek el. A puffer nem azonos a file ablakkal, rendesen egynél több elemet tartalmaz a fizikai átvitelek számának csökkentése céljából. A processzor sebességéhez képest a perifériák lassúak. Ha minden elemért külön kellene a lemezhez fordulni, nagyon megnövekedne a file feldolgozásának az ideje. A pufferelléssel ezt az időt csökkentjük. Ha írunk egy állományt, akkor a write eljárás csak a pufferbe tölti a file elemeit. Ha a puffer megtelik, tartalma automatikusan a lemezre íródik. Gyakori, hogy az utolsó elemek írásakor nem telik már meg a puffer. Ilyenkor elveszítenénk az állomány végét, ha a close eljárás nem írná fel a maradékot.
A lemez tartalomjegyzékébe is a close hatására íródik be az adatállomány mérete, a dátum és az időpont.
Az eddigiekből méltán tűnhet úgy, hogy csak írás után fontos lezárni a file-t. Ez korántsincs így. Egyszerre csak 15 file lehet megnyitva. A megnyított file-ok számát a rendszer számontartja. Megnyitáskor növeli ezt a számot, záráskor csökkenti. Ha nem zárjuk le az állományt, akkor esetleg ilyen nyitva maradt "fantom" file-ok akadályozzák meg egy új file megnyitását.
Ha egy file-t lezártunk, nem tudjuk sem írni, sem pedig olvasni, hacsak újra meg nem nyitjuk. A file azonosító és az állomány közötti kapcsolat azonban a file lezárása után is megmarad. Ezért nem kell az assign utasítást megismételni, ha újra meg akarjuk nyitni a file-t. Az utasítások egy lehetséges sorrendje:

```
program filekezeles;

  type szamfiletipus=file of integer;
  var szamfile:szamfiletipus;

  begin

    {hozzákapcsolja a filenévhez a lemezállományt}
    assign(szamfile,'sorozat.int');
    ..............................

    rewrite(szamfile); {megnyitás írásra}

    ..............................

    close(szamfile);   {lezárás}

    ..............................

    reset(szamfile);   {megnyitás olvasásra}

    ..............................
```

```
        close(szamfile);    {lezárás}
        ..........................
        end.
```

A file-ok kezelésének fontos eszköze az eof (=end of file) predikátum. Értéke akkor true, ha a file végét elértük (pontosabban az utolsó elem olvasásakor). Segítségével kényelmesen használhatjuk a while ciklusszerkezetet file-ok feldolgozására (91. program).

```
program olvas;

  type szamfiletipus = file of integer;
  var szamfile:szamfiletipus;
      i:integer;
      nev:string[12];

  begin
    clrscr;
    write('Irja be a kilistazando allomany nevet:');
    readln(nev);
    assign(szamfile,nev);
    reset(szamfile);
    clrscr;
    writeln('A ',nev,' allomany listaja:');
    while not eof(szamfile) do begin
      read(szamfile,i);
      write(i,'  ');
      end;
    writeln;
    close(szamfile);
    end.
```

Állomány tartalmának listázása

91. program

11.3 Text állományok

A "file of char" típus az alkalmazásokban gyakori. A Pascal szabványos azonosítót is rendelt ehhez a típushoz, ez a

```
text
```

azonosító, amely ugyanúgy használható a var deklarációs részben, mint az integer vagy a real. A text típus értékeit tehát ASCII

karakterek sorozatai alkotják. Az "új sor" karakter a text állományokban különleges jelentőségű. (Pontosabban nem az "új sor", hanem a "kocsi vissza"/"új sor" karakterpár, amely a 10 és 13 kódoknak felel meg.) Ezért úgy foghatjuk fel, hogy egy text állomány szövegsorokból áll.
Amit a file-ok feldolgozásáról a 11.2 alfejezetben elmondtunk, a text file-okra is érvényes. Van azonban néhány olyan eszköz, amelyeket kizárólag a text file-okra alkalmazhatunk.
A text file megnyitására a reset és a rewrite mellett egy harmadik eljárás is használható, az

```
append.
```

A rewrite eljáráshoz hasonlóan az append-del is írásra nyithatjuk meg a text file-okat. A különbség az, hogy nemlétező állomány esetén a rewrite létrehozza az adott nevű állományt, az append használata viszont ilyen körülmények mellett hibajelzést okoz. Ugyanakkor, ha létező állományt nyitunk meg, akkor az append nem törli, mint a rewrite. Megőrzi a file értékét és a file mutatót a legutolsó elem után állítja. Ennek eredményeként a felírt elemek folytatólagosan a file végére íródnak.
A megnyitott text file-ból a read mellett a

```
readln
```

eljárással is olvashatunk. Az olvasott változók azonosítói előtt általában meg kell adnunk a file azonosítót is. A

```
readln(szovegfile,sor)
```

eljáráshívás hatására a következők történhetnek:

1. Ha a file mutató aktuális értékétől a legközelebbi sor vége jelig elhelyezkedő karakterek száma kisebb vagy egyenlő a "sor" stringváltozó deklarált hosszával, akkor a "sor" ezt a füzért kapja értékül. A file mutató pedig a következő sor elejére lép.

2. Ha több karakterből áll a sor hátralévő része, mint amennyit a stringváltozó tárolni képes, akkor a stringváltozóba betöltődik annyi karakter, amennyi belefér, majd a többit figyelmen kívül hagyva a következő sor elejére ugrik a file mutató.

A read utasítás a text file-oknál a readln-hez hasonlóan működik, azzal a különbséggel, hogy miután betöltötte a stringváltozóba a karaktereket, a mutatót az utolsó ovasott karaktert követő jelre állítja, nem a következő sor elejére. Ha a read egy új sor (pontosabban 10-es kódú) jelet olvas, akkor megáll, nem olvas több karaktert és a file mutató értékét sem változtatja meg, míg egy readln eljárás nem következik.
A readln eljárásban egynél több változót is megadhatunk. Ekkor a stringváltozók együttes hossza a mérvadó.

A text file-okban természetesen olyan stringek is előfordulhatnak, amelyek számkonstansoknak felelnek meg. Ha a read vagy readln eljárásban integer vagy valós típusú változóparamétereket használunk, akkor ezeket a stringeket necsak beolvassa az eljárás, de számmá is alakítja (a val stringeljáráshoz hasonlóan). Arra persze ügyelni kell, hogy a számstringek szintaktikailag hibátlan számkonstansokat ábrázoljanak. Ha csak számokat olvasunk be, akkor elválasztó jelként csak szóközöket, tabulátorjeleket és sor vége jeleket használhatunk.

Az eof predikátumhoz hasonló, de kizárólag text file-oknál használható az

```
eoln
```

(=end of line). Az eoln értéke két feltétel mellett válik igazzá: ha a file pointer egy 10-es karakterre mutat, vagy ha a file végére. Text file-oknál a file vége a file fizikai vége, vagy a ^Z karakter. Arra használhatjuk az eoln függvényt, hogy egy sorból egyenként olvassunk karaktereket (92. program).

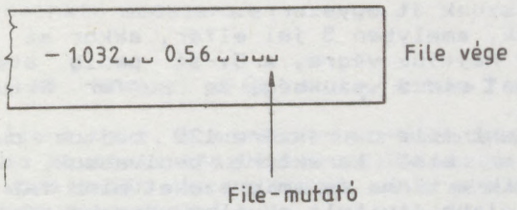
```
program karakterolvaso;

  var szoveg:text;
      ch:char;

  begin
    assign(szoveg,'vers.txt');
    reset(szoveg);

    while not eof(szoveg) do begin

      while not eoln(szoveg) do begin
        read(szoveg,ch);
        write(ch);
        end;

      readln(szoveg);
      writeln;
      end;

    close(szoveg);
    end.
```

Az eoln használata

92. program

Figyeljük meg a readln használatát a 92. programban. A sorvége karaktereket olvasuk be ezzel az utasítással, mert a read ezt nem tudja megtenni.

Ha egy numerikus text file értékét olvassuk, akkor előfordulhat, hogy a file (vagy a sor) végére értünk ugyan, mert az állomány már több számjegyet nem tartalmaz, de az eof (vagy az eoln) predikátum ezt nem érzékeli, mert a file mutató egy szóköz karakterre mutat, amely az utolsó számot követi (86. ábra).



Numerikus textfile

86. ábra

Ilyen esetben a

```
seekeof
```

ill. a

```
seekeoln
```

függvényeket használhatjuk az eof és az eoln helyett. Ezek a függvények "előre látó" (look ahead) tulajdonságúak. Fel tudják ismerni a legfeljebb 32 pozícióval később következő file vége, ill. sor vége karaktert, ha csak szóközök (vagy vezérlő karakterek) állnak előtte.

A text file-okat a write mellett a writeln eljárással is írhatjuk. A writeln a paraméter(ek)ben magadott értékek után lezárja az aktuális sort (sorvége jelet ír).

Korábban - a close utasítással kapcsolatban - már érintettük a pufferelés kérdését. A processzor és a perifériák működési sebessége között akkora a különbség, hogy a processzor számára egy mágneslemezes művelet (írás vagy olvasás) éveknek tűnhet. Képzeljük el, hogy a program öt karaktert olvas be a szamfile állományból:

```
read(szovegfile,i);
read(szovegfile,j);
read(szovegfile,k);
read szovegfile,m);
read(szovegfile,n);
```

Ha nincs puffer, akkor mindegyik olvasási műveletnél

1. meg kell keresni a lemezen a következő elemet,

2. át kell vinni a tárba a file "ablakba",

3. át kell másolni a változóhoz tartozó tárterületre.

E tevékenységek közül az első kettő nyagyságrendekkel több időt vesz igénybe a 3.-nál. Végrehajtási idejük csak kevéssé függ attól, hogy hány elemet viszünk át egyszerre.
Ha van egy akkora pufferunk, amelyben 5 jel elfér, akkor az első két műveletet csak egyszer hajtjuk végre, a 3.-at pedig ötször. Valószínűleg külön "ablakra" sincs szükség, a puffer aktuális eleme is megfelel a célnak.
Alapértelmezés szerint a text file-ok részére 128 bájtos puffer áll rendelkezésre. Amikor az első karaktert beolvassuk, akkor rögtön 128 karakter töltődik a tárba és amíg ezeket mind fel nem használtuk, nem kerül sor újabb átviteli művelet végrehajtására.
Nagy állományok feldolgozását gyorsíthatjuk, ha a puffer méretét megnöveljük. Erre lehetőségünk van, az igényelt bájtok számát a deklarációban megadhatjuk:

```
var szoveg:text[1024];
```

Ezzel a szoveg file feldolgozásához 1024 bájtos puffert határoztunk meg. A textfile puffer maximális mérete elvileg 32767, de 1024-nél nagyobbat ritkán érdemes használni.
Ha a nagyméretű pufferből nem minden elemet kívánunk kiolvasni, akkor a

```
flush
```

eljárással üríthetjük, nem kell felesleges read utasítások garmadáját végrehajtani ahhoz, hogy új átvitel történjék. A flush-t használjuk akkor is, ha a puffer íráskor még nem telt meg, de tartalmát a lemezre akarjuk írni. A

```
flush(szoveg)
```

utasítás üríti a puffert. A flush utáni read újabb 1024 bájt átvitelét okozza, ha pedig írás műveletet végzünk, akkor a következő write elölről kezdi a puffer feltöltését.
A Turbo Pascal 4.0-s változatában a puffert másképp kell meghatározni. Deklarálni kell egy megfelelő méretű tömböt és ezt egy külön eljárással (settextbuf) kapcsoljuk a file-hoz:

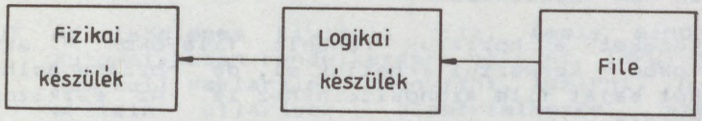
```
type pufftipus=array[1..1024] of char;

.........................

var szoveg:text;
    puffer:pufftipus;

.........................
```

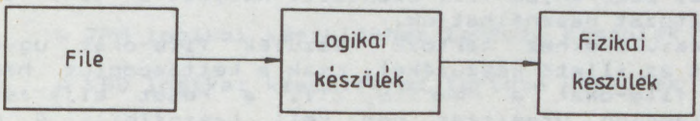
```
begin
  assign(szoveg,'vers.txt');
  settextbuf(szoveg,puffer);
  reset(szoveg);

  ......................
```

Ez a megoldás ugyanolyan eredményű, mint a 3.0-s változatban használható módszer.

11.4 Be- és kiviteli készülékek

A billentyűzet, a monitor és a nyomtató a leggyakrabban használt be- és kiviteli készülékek. Ezek közül a nyomtató használatáról még egyáltalán nem volt szó, de a másik két egység kezeléséről is van még mit mondani.
A Turbo Pascalban a fizikailag létező eszközök helyett ún. logikai eszközöket kezelünk. A logikai és a fizikai eszközöket az operációs rendszer kapcsolja össze. A logikai eszközöket a hozzájuk tartozó szabványos text file-ok közvetítésével használjuk(87. ábra).





Fizikai és logikai készülékek

87. ábra

Hat logikai készülék és nyolc szabványos file áll rendelkezésünkre a bevitel és kivitel programozásához.

A logikai eszközök a következők:

CON: Ez a logikai eszköz a konzol, billentyűzetből és képernyőből áll. A billentyűzetről kapott karakterek egy pufferbe kerülnek. Ez azt jelenti, hogy a CON-hoz tartozó input file-ból a read és readln eljárásokkal egy teljes sort olvasunk a pufferba. A bevitel során a gép kezelőjének rendelkezésére áll néhány szerkesztési lehetőség a puffer

tartalmának megváltoztatására (pl.: a [DEL] billentyűvel törölheti az utolsó karaktert, [CTRL][X] törli a puffer teljes tartalmát). A visszhangellenőrzés érvényben van, a leütött billentyűknek megfelelő jelek (bizonyos vezérlőjelek kivételével) a képernyőre íródnak.

TRM: Ez a terminál eszköz. A terminál ugyancsak billentyűzetből és képernyőből áll, mint a CON:. Nincs pufferelés, ezért a szerkesztési lehetőségek sincsenek meg, echo van.

KBD: Csak input eszköz, a billentyűzetnek felel meg. Echo nincs, a bevitt jelek nem láthatók a képernyőn. Nincs pufferelés sem.

LST: Csak output eszköz, a nyomtatónak felel meg.

AUX: Soros átviteli egység. Hagyományos be és kiviteli eszközök (lyukasztó,olvasó stb) kezelésére lehet használni, ha ilyen van. Ezeket az eszközöket általában modemen (modulátor-demodulátor) keresztül kapcsolhatjuk a számítógépre.

USR: Néha szükség lehet arra, hogy a programozó definiáljon magának logikai eszközt. Ezt a felhasználói eszközt jelöli a USR név. A felhasználói eszköz definiálásának módjával ebben a könyvben nem foglalkozunk.

A logikai eszközöket a hozzájuk rendelt file-okon - az ún. készülék file-okon - keresztül érhetjük el, de hozzárendelhetjük az eszközneveket saját file azonosítóinkhoz is. Az eszköznévnek része a kettőspont:

```
assign(kiiras,'lst:').
```

Ilyenkor az lst szabványos file azonosító helyett a text típusú kiiras fileváltozót használhatjuk.
A logikai készülékekhez tartozó készülék file-okat ugyanúgy jelöljük, mint az illető készüléket, csak a kettőspontot hagyjuk el. Ezeket a file-okat a rewrite, ill. a reset eljárásokkal nyitjuk meg (assign utasítást nem kell használni). A close eljárást használhatjuk ugyan, de hatástalan.

Az eof és az eoln predikátum másképpen viselkedik a készülék file-oknál, mint a mágneslemezes állományoknál. Egy lemezes text file esetében az eof értéke akkor lesz true, ha az éppen olvasott karakter után a ^Z (file vége) jel, vagy az állomány fizikai vége következik. Az eoln pedig akkor, ha a következő karakter file vége vagy sor vége.
A készülék file-oknál nincs meg ez az egykarakternyi előretekintés, ezért itt az éppen olvasott jel értéke dönt. Tehát az eof értéke true, ha az éppen olvasott jel értéke ^Z, az eoln predikátum értéke pedig akkor, ha ^Z vagy kocsi-vissza (10-es kód).

A readln eljárás is másképpen működik a készülék file-okon.

Mágneslemezes állománynál a readln minden karaktert beolvas a következő újsor/kocsi-vissza karakterpárig, azokat is beleértve. Készülék file esetén viszont csak a 10-es karaktert olvassa be, a 13-mast már nem.
Ezeket a különbségeket ismernünk kell, és szükség esetén figyelemmel kell rájuk lenni. Sok kényelmetlenséget okozhatnak, és nehezen érthető, hogy miért ilyen döntést hoztak a Turbo Pascal tervezői és megvalósítói. A kézikönyvben a készülékek "természetére" hivatkoznak, hogy nem látható előre a file soron következő karaktere. Az indok nehezen elfogadható, hiszen több olyan Pascal megvalósítást láttunk, ahol egységes lehetőséget teremtettek a eof, eoln és a readln használatára.

Amint korábban említettük, a Turbo Pascal nyolc előre definiált készülék file-lal könnyíti az eszközök használatát. Ezzel mentesít bennünket az assign utasítások, a reset, ill. rewrite és a close eljárások használatától. Ezek a szabványos file-ok a következők:

INPUT Az elsődleges beviteli file, amely minden programban automatikusan rendelkezésre áll. Sem deklarálni, sem megnyítni, sem lezárni nem kell. A read és a readln eljárásban a file azonosítót sem szükséges megadni. Az input file a CON vagy a TRM logikai készülékhez kapcsolódik, alapértelmezés szerint a CON-hoz.

OUTPUT Az elsődleges kiviteli file, amely minden programban automatikusan rendelkezésre áll. Az INPUT-hoz hasonlóan nem kell deklarálni, megnyítni, lezárni. A write és a writeln eljárások alapértelmezés szerinti file paramétere. Alapértelmezés szerint a CON logikai készülékhez kapcsolódik, de a TRM-mel is használhatjuk.

CON A CON logikai készülékhez tartozó készülék file.

TRM A TRM logikai készülékhez tartozó készülék file.

KBD A KBD logikai készülékhez tartozó készülék file.

LST Az LST logikai készülékhez tartozó készülék file.

AUX A soros átviteli logikai készülékhez tartozó készülék file.

USR A USR felhasználói készülékhez tartozó készülék file.

Jegyezzük meg, hogy ezekkel a file-okkal az assign, reset, rewrite és close eljárásokat necsak, hogy nem kell használni, de nem is szabad.

Az input és output szabványos file-okat a Turbo Pascal alapértelmezés szerint a CON logikai készülékhez rendeli. Ezért a B compiler direktíva a felelős. A {$B+} direktíva a CON, a {$B-} a TRM készülékhez rendeli hozzá az elsődleges inputot és

outputot. Az alapértelmezés {$B+}. A két logikai készülék között csak a bevitelnél van különbség. A TRM input sort nem tudjuk módosítani, viszont a szabványos Pascal beviteli formátumai is használhatók.
A {$B-} direktíva után a programfejben adjuk meg az elsődleges beviteli és kiviteli file-okat:

```
program bevitel(input,output);
```

A B direktívát csak egyszer használhatjuk a programban, globális direktívaként, s nem változtathatjuk meg. A legfontosabb különbség a programozó számára a két direktíva között a következő:

> Ha a {$B+} van érvényben, akkor minden read csak egyetlen sorból képes olvasni, azaz a beolvasott karakterek utáni rész a sor végéig a következő read számára elveszik. A következő read a következő sor elejétől kezd olvasni. A ^Z figyelmen kívül marad. {$B-} után a read eljárással folytatólagosan olvashatunk. Ha az előző read nem olvasott be mindent az aktuális sorból, a következő read ott folytatja, ahol az előző abbahagyta.

A CON logikai készülék inputjához az alapértelmezés szerint 127 bájtos puffer tartozik. Ezt az előre definiált buflen változó értéke határozza meg. A puffer hosszát egyetlen beolvasási művelet tartamára csökkenthetjük, de nem növelhetjük. Ehhez buflen-nek kell más értéket adni:

```
.................

write('allomanynev (max. 12 jel):');
buflen:=12;
read(nev);

.................
```

A beolvasás után újra 127 lesz buflen értéke.
A KBD file jól használható, ha nem akarjuk hogy az echo folytán a képernyőre íródó karakter elrontsa az ott kialakított képet. Pl. ha menüvel vezéreljük egy program működését, akkor a menüpont kiválasztását a kbd file-ba olvassuk be. Tegyük fel, hogy egy megfelelő eljárással a képernyőre írtuk a 88. ábrán látható menüt.
A feladat az, hogy kiválasszuk és beírjuk a megfelelő menüpont számát (93. program).

```
procedure menupont(also,felso:char;var valasz:char);

  begin
    repeat
```

```
    read(kbd,valasz);
    until (valasz>=also) and (valasz<=felso);
  end;
```

A menüpont beolvasása

93. program

```
                BERSZAMFEJTES MENU

        1. Uj dolgozo adatainak felirasa

        2. Kilepo dolgozo adatainak torlese

        3. Adatvaltozasok, modositasok

        4. Havi teljesìtmeny adatok felirasa

        5. Berszamfejtes

        6. Vege
```

Program menüje a képernyőn

88. ábra

Az eljárás hívása:

```
menupont('1','6',pont)
```

A "pont" karaktervàltozó értéke szerint egy case szerkezettel választhatjuk ki a megfelelő tevékenységet.

Az LST file segítségével nyomtathatunk. Ha az eredményeket nem a képernyőn akarjuk csak szemlélni, hanem meg szeretnénk őrizni - és persze van nyomtatónk - akkor a write, writeln eljárások aktuális file paraméterének az LST-t adjuk meg. A 94. programmal egy Pascal forrásprogramot nyomtathatunk ki. A programot lemezről olvassuk.

```
program listazo;

  var sor:string[127];
```

```
    prog:text[1024];
    nev:string[12];

begin
  fillchar(sor,sizeof(sor),0);
  clrscr;
  writeln('Kerem a listazni kivant programallomany');
  writeln('nevet kiterjesztessel egyutt:');
  bufflen:=12;
  readln(nev);
  assign(prog,nev);
  reset(prog);
  repeat
    readln(prog,sor);
    writeln(lst,sor);
    until eof(prog);
  close(prog);
  end.
```

Listázás nyomtatóra

94. program

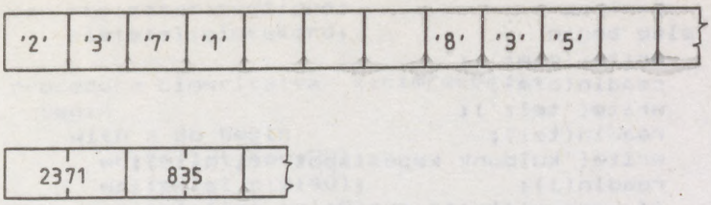
Az AUX és a USR készülék file-okkal nem foglalkozunk.

11.5 Soros adatállományok feldolgozása

Bár a text file-ok sok területen nyernek széles körű felhasználást, adatállományainkat ritkán tároljuk ilyen formában. A műszaki-gazdasági és az ügyviteli adatfeldolgozás alapadatainak tárolására többnyire olyan file típusokat használunk, amelyeknek a bázistípusa valamilyen rekordtípus. Ennek nemcsak az az előnye, hogy a rekordokba célszerűen elrendezett információt tárolhatunk. Minden általános típusú (tehát nem text) file-nak megvan az a kellemes tulajdonsága is, hogy az információt pontosan abban a formában tárolják, ahogy az a tárban megtalálható.
Hasonlítsunk össze egy numerikus textfile-t egy numerikus file-lal. Legyen

```
var numfile:file of integer;
    numtextfile:text;
```

A numerikus textfile az egész számokat numerikus karaktersorozatok formájában tárolja, szóközökkel, tabulátor, ill. sorvége karakterekkel választva el őket. A numerikus file pedig bináris alakban, minden számot két bájton ábrázol (89. ábra).



Text és numerikus file ábrázolása

89. ábra

A numerikus file ábrázolása láthatóan sűrűbb, és ugyanakkor a szükséges feldolgozási idő is rövidebb, mert a számokat nem kell bevitelkor és kivitelkor stringből számmá, ill. számból karakterláccá alakítani.
Hasonló előnyöket tapasztalunk más bázistípus - így rekord - alkalmazásakor is. A file-ban tárolt értékeket a feldolgozás során csak két esetben kell átalakítani. Az elején, amikor az elsődleges inputtal bevitt text állomány alakítjuk más típusú állománnyá, és a végén, amikor a file feldolgozás eredményét nyomtatni akarjuk. Két programot mutatunk be ebben a témakörben. Az első (95. program) egy adatállományt hoz létre, a másik (96. program) ezt az adatállományt nyomtatja ki.

```
program felir;

  type str20=string[20];
       str40=string[40];
       str9=string[9];

       cimrekord=record
                   nev:str20;
                   cim:str40;
                   tel:str9;
                   jel:boolean;

       regtipus=file of cimrekord;

  var regiszter:regtipus;
      cimtetel:cimrekord;
      vege:boolean;

  procedure cimread(var x:cimrekord; var b:boolean);
    var j:char;
    begin
      b:=false;
      with x do begin

        write('nev:');
        read(nev);
        if nev='*' then
```

```
            b:=true
          else begin
            write('cim:');
            readln(cim);
            write('tel:');
            readln(tel);
            write('kuldunk kepeslapot? (i/n):');
            readln(j);
            if upcase(j)='I' then
              jel:=true
            else
              jel:=false;
            end;
          end;
        end; {cimread}

      begin
        assign(regiszter,'ismeros.dat');
        rewrite(regiszter);
        cimread(cimtetel,vege);
        while not vege do begin
          write(regiszter,cimtetel);
          cimread(cimtetel,vege);
          end;
        close(regiszter);
        end.
```

Adatállomány létrehozása

95. program

A 95. program létrehozza a lemezen az "ismeros.dat" című adatállományt, amint erről a tartalomjegyzék kikérésével meggyőződhetünk. A program akkor fejezi be a működését, ha a

nev:

kérdésre *-gal válaszolunk.

```
program olvas;

  type str20=string[20];
       str40=string[40];
       str9=string[9];

       cimrekord=record
                   nev:str20;
                   cim:str40;
                   tel:str9;
                   jel:boolean;

       regtipus=file of cimrekord;
```

```
var regiszter:regtipus;
    cimtetel:cimrekord;

procedure cimwrite(var x:cimrekord);
  begin
    with x do begin
      write(lst,nev:20);
      write(lst,cim:40);
      writeln(lst,tel:9);
      if jel then
        writeln('kuldunk kepeslapot'
      else
        writeln('nem kuldunk kepeslapot');
      writeln;
      end;
    end; {cimwrite}

begin
  clrscr;
  assign(regiszter,'ismeros.dat');
  writeln('Legyen szives bekapcsolni a nyomtatot!');
  writeln;
  write('Ha kész, nyomjon le egy tetszoleges ');
  writeln('billentyut!');
  repeat
  until keypressed;
  clrscr;
  gotoxy(30,12);
  writeln('N Y O M T A T A S');
  write(lst,'C I M L I S T A');
  writeln(lst);
  writeln(lst);
  reset(regiszter);
  while not eof do begin
    read(regiszter,cimtetel);
    cimwrite(cimtetel);
    end;
  close(regiszter);
  clrscr;
  end.
```

Adatállomány listázása

96. program

A 96. programmal kapcsolatban megjegyezzük, hogy a

keypressed

függvény a true logikai értéket adja vissza, ha lenyomunk egy billentyűt. A függvény csak akkor működik, ha a {$C-} és a {$U-} compiler opciót használjuk.

Az általános file típusokat jobbára ugyanazokkal az eszközökkel kezeljük, mint a text file-okat. Néhány eltérés azért van.
Nem használható a readln, writeln eljárás és az eoln predikátum.
A reset-tel nemcsak olvasáshoz nyithatunk meg egy állományt, hanem inkább azt mondjuk, hogy a létező file-ok megnyitására használjuk.
Újabb eljárás a

filesize

függvény, melynek paramétere egy file azonosító, értéke a file elemeinek a száma. Az üres file-on a 0 értéket veszi fel, megnyitáa előtt pedig -1 az értéke.

Egy másik függvény a

filepos,

amelynek szintén file azonosító a paramétere, értéke pedig a file mutató által meghatározott elem sorszáma (0-tól kezdődően).

A soros adatállományok feldolgozása általában három lépésből áll:

az beviteli file(ok) következő elemének olvasása,

a tárban lévő adatok feldolgozása,

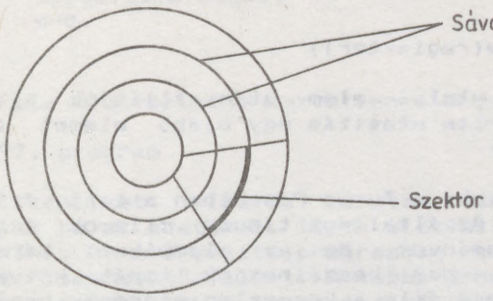
a feldolgozás eredményeként kapott adatok felírása az eredmény file(ok)-ba.

Még abban az egyszerű esetben is, amikor egy adatállományban mindössze néhány elemet kell módosítanunk tulajdonképpen három file-ra van szükségünk. Az egyik a módosítandó állomány, a másik a módosító állomány ami az adatváltozást tartalmazza. Ezt a két állományt olvasásra kell megnyítni. A harmadik, amit írunk, a megváltozott állomány.

11.6 Adatállományok közvetlen hozzáférésű feldolgozása

A mágneslemezes adattárolás elvileg lehetővé teszi az adatok közvetlen elérését. A mágneslemezen az adatok adott szerkezetben helyezkednek el: meghatározott lemezoldalon, adott sávban és adott szektorban (90. ábra).
A lemezegység és a tár közötti átvitel - fizikailag - teljes szektoronként történik. Ha tudjuk, hogy a file feldolgozni kívánt eleme hol van a lemezen, akkor - megadva az oldal, sáv és szektor számát - közvetlenül beolvashatjuk. Persze nem lenne nagyon felemelő tevékenység a lemez minden szektorának tartalmát nyilvántartani. Erre szerencsére nincs is szükség. A nem text típusú file-ok elemeihez egy sorszámot rendel a Turbo Pascal, az elemeket ezzel a sorszámmal azonosíthatjuk. Az elem lemezcímét a

sorszám alapján meg tudja határozni a rendszer, ezzel a felhasználónak nem kell törődnie. A read,ill. a write utasítás számára az elemsorszám a meghatározó.



A lemez tárolási szerkezete

90. ábra

Soros adatfeldolgozásnál a sorszám a file mutatóval együtt automatikusan változik. A reset,ill. a rewrite eljárások 0-ra állítják, ez az első file elem sorszáma. Minden olvasás vagy írás végrehajtásakor eggyel megnő a sorszám értéke, így minden elem után a soron következőt tudjuk elérni.
Egy file bármelyik (létező) elemét elérhetjük, ha a file mutatót a kívánt sorszámú elemre tudjuk állítani. Erre való a

```
seek
```

eljárás. A seek-nek két paramétere van, az első a kérdéses file azonosítója, a második egy egész kifejezés, ami az olvasni vagy írni kívánt elem sorszámát határozza meg. A

```
seek(regiszter, 58)
```

eljáráshívással a file mutatót a regiszter file 59. elemére vittük (a sorszám 0-val kezdődik!). Ha a file i-edik elemét olvasni akarjuk, majd bizonyos módosítás után visszaírni, ehhez a

```
.....................
seek(regiszter,i);
read(regiszter,cimtetel);
.............................

{a cimtetel rekord módosítása}

.............................
seek(regiszter,i);
write(regiszter,cimtetel);
.............................
```

utasításokat kell leírni. Mivel minden író és olvasó utasítás megváltoztatja a file mutatót, a seek eljárást minden read és write előtt használni kell.
A seek eljárással pótolhatjuk a nem text file-oknál hiányzó append eljárást. A

```
seek (regiszter,filesize(regiszter))
```

eljáráshívással éppen az utolsó elem után állítjuk a file mutatót, s a végrehajtott write utasítás egy újabb elemet ír a file végére.

összefoglalva megállapíthatjuk: a Turbo Pascalban a text file-ok egyértelműen soros file-ok. Az általános típusú file-ok értékei is voltaképpen soros állományok, de az elemeiket bármikor közvetlenül is elérhetjük. Nem beszélhetünk tehát közvetlen elérésű file-okról, mert nem a file a közvetlen elérésű, hanem a feldolgozás módszere.
Ha egy file elemeit közvetlen eléréssel szeretnénk kezelni, akkor célszerű lehet a feldolgozást a file létrehozásával kezdeni. A kívánt számú elemet felírva a file-ba, voltaképpen egy tömbhöz hasonló szerkezetet alakítunk ki. Az elem sorszám az index szerepét tölti be. Ezenkívül az elemek nem a tárban vannak, hanem a lemezen. Az első felíráskor csak előkészítjük az állományt, adatokat nem írunk bele, csak üres elemeket. Az elemek többnyire rekordok, amelyekben használhatunk egy logikai típusú mezőt, amely azt jelzi, hogy írtunk-e már adatot az illető elembe (a 97. programnál a "letezik" mező az "adatrec" rekordtípusban).

```
program proge;
  type string30=string[30];
       string10=string[10];
       adatrec=record
           nev:string30;
           varos:string10;
           cim:string30;
           honap:byte;
           nap:byte;
           telefon:string10;
           levelez:boolean;
           letezik:boolean;
           end;
       adatfile=file of adatrec;
  var adatbe,adatki:adatrec;
      ismerosok:adatfile;
      levjel:char;
      rszam:integer;
      i:integer;
  begin
    assign(ismerosok,'ismerosok.adt');
    rewrite(ismerosok);
```

```
    for i:=0 to 372 do begin
       adatbe.letezik:=false;
       write(ismerosok,adatbe);
       end;
    close(ismerosok);
    end.
```

File előkészítése közvetlen eléréshez

97. program

Az előkészített file-ba ezután tetszés szerinti sorrendben írhatjuk be az adatokat. Ilyen előkészítés nélkül csak sorosan írhatnánk. Az előkészítő programban rewrite eljárással nyítottuk meg az "ismerosok" file-t. A többi programban már létező file-lal állunk szemben, így a reset eljárást használjuk. Így teszünk a 98. programban is, ahol pedig adatokat írunk fel.

```
program progf;
  type string30=string[30];
       string10=string[10];
       adatrec=record
           nev:string30;
           varos:string10;
           cim:string30;
           honap:byte;
           nap:byte;
           telefon:string10;
           levelez:boolean;
           letezik:boolean;
           end;
       adatfile=file of adatrec;
  var adatbe,adatki:adatrec;
      ismerosok:adatfile;
      levjel:char;
      rszam:integer;
      i:integer;
  procedure olvas;
    begin
      clrscr;
      with adatbe do
        begin
          write('Nev: ');
          readln(nev);
          if nev='vege' then exit;
          write('Varos: ');
          readln(varos);
          write('utca, hazszam: ');
```

```
        readln(cim);
        write('szul. honap (szammal): ');
        readln(honap);
        write('szul. nap: ');
        readln(nap);
        write('Telefonszam: ');
        readln(telefon);
        write('Levelezunk? (i/n): ');
        readln(levjel);
        if leviel='i' then levelez:=true
        else levelez:=false;
        end
    end;
function index:integer;
  begin
    index:=(adatbe.honap-1)*31+adatbe.nap;
    end;
begin
  assign(ismerosok,'ismerosok.adt');
  reset(ismerosok);
  writeln('file hossz: ',filesize(ismerosok));
  repeat
    olvas;
    if adatbe.nev='vege' then begin
                              close(ismerosok);
                              write('V E G E');
                              exit;
                              end;
    rszam:=index;
    writeln('rszam=',rszam);
    seek(ismerosok,rszam);
    read(ismerosok,adatki);
    if not adatki.letezik then
       begin
         seek(ismerosok,rszam);
         adatbe.letezik:=true;
         write(ismerosok,adatbe);
         end
    else begin
           writeln;
           writeln('A kulcs mar letezik!');
           for i:=1 to maxint do;
           end;
    until  2<1;
    end.
```

Adatok felírása

98. program

Az elemsorszámot nem ad hoc határozzuk meg a felíráskor, arra is

gondolnunk kell, hogy a felírt adatokat meg is kell találni. A születési adatokból határoztunk meg egy indexet (function index), ezt választottuk elemsorszámnak. A 99. programmal tetszőleges file elemet kérhetünk, a sorszámot kell megadni. Ha a megadott sorszámú elem üres rekord, akkor a

```
nem letezo record
```

üzenetet kapjuk a programtól.

```
program progw;
  type string30=string[30];
       string10=string[10];
       adatrec=record
          nev:string30;
          varos:string10;
          cim:string30;
          honap:byte;
          nap:byte;
          telefon:string10;
          levelez:boolean;
          letezik:boolean;
          end;
       adatfile=file of adatrec;
  var adatbe,adatki:adatrec;
      ismerosok:adatfile;
      levjel:char;
      rszam:integer;
      i:integer;
  procedure kiiras;
    begin
    with adatbe do begin
      writeln('NEV: ',nev);
      writeln('VAROS: ',varos);
      writeln('CIM: ',cim);
      end;
    end;
  begin
    clrscr;
    assign(ismerosok,'ismerosok.adt');
    reset(ismerosdk);
    repeat
      write('Hanyadikat kered? ');
      readln(rszam);
      if rszam<372 then begin
        seek(ismerosok,rszam);
        read(ismerosok,adatbe);
        if adatbe.letezik then kiiras
        else writeln('nem letezo record');
        end;
```

```
    until rszam>=372;
  close(ismerosok);
  end.
```

Adatvisszakeresés a sorszám alapján

99. program

A 100. program a teljes állományt végigolvasva szelektív kiírást végez.

```
program progw;
  type string30=string[30];
       string10=string[10];
       adatrec=record
           nev:string30;
           varos:string10;
           cim:string30;
           honap:byte;
           nap:byte;
           telefon:string10;
           levelez:boolean;
           letezik:boolean;
           end;
        adatfile=file of adatrec;
  var adatbe,adatki:adatrec;
      ismerosok:adatfile;
      levjel:char;
      rszam:integer;
      i:integer;
  procedure kiiras;
    begin
    with adatbe do begin
      writeln('NEV: ',nev);
      writeln('VAROS: ',varos);
      writeln('CIM: ',cim);
      end;
    end;
  begin
    clrscr;
    assign(ismerosok,'ismerosok.adt');
    reset(ismerosok);
      for rszam:=0 to 372 do begin
         seek(ismerosok,rszam);
         read(ismerosok,adatbe);
         if adatbe.letezik  then
```

```
            if adatbe.levelez then kiiras;
        end;
    close(ismerosok);
    end.
```

Kigyűjtés egy mező értéke szerint

100. program

Ha az adatállományunk teljes, azaz minden elemet adatokkal feltöltöttünk, esetleg érdemes lehet valamelyik mező tartalma szerint rendezni. A 101. programban a nevek szerint rendezzük az állományt. Ezt a rendezést sorosan írt adatállományok soros feldolgozásának előkészítéséhez lehet célszerű felhasználni.

```
type string30=string[30];
     string10=string[10];
     adatrec=record
         nev:string30;
         varos:string10;
         cim:string30;
         honap:byte;
         nap:byte;
         telefon:string10;
         levelez:boolean;
         letezik:boolean;
         end;
     adatfile=file of adatrec;
var adat:adatrec;
    ismerosok:adatfile;
    k:integer;
function i_edik(var allomany:adatfile; i:integer):string30;
   var mrec:adatrec;
   begin
      seek(allomany,i);
      read(allomany,mrec);
      i_edik:=mrec.nev;
      end; {i_edik}
procedure rendhiv(var allomany:adatfile; rszam:integer);
   procedure gyrend(p,q:integer);
      var i,j,k:integer;
          a,b,c:adatrec;
      begin
         i:=p;
         j:=q;
         k:=(p+q) div 2;
         seek(allomany,k);
         read(allomany,a);
         repeat
            while i_edik(allomany,i)<a.nev do i:=i+1;
            while a.nev<i_edik(allomany,j) do j:=j-1;
            if i<=j then begin
               { felcsereles a lemezen}
               seek(allomany,i);
```

```
                read(allomany,b);
                seek(allomany,j);
                read(allomany,c);
                seek(allomany,i);
                write(allomany,c);
                seek(allomany,j);
                write(allomany,b);
                i:=i+1;
                j:=j-1;
                end;
            until i>j;
          if p<j then qyrend(p,j);
          if i<q then qyrend(i,q);
      end;  {gyrend}
    begin {rendhiv}
      gyrend(0,rszam);
    end; {rendhiv}
```

Rendezés a lemezen

101. program

A 101. programban a C. A. R. Hoare által 1968-ban publikált gyorsrendezés algoritmusát használtuk (procedure gyorsrend), mert bizonyíthatóan ennél az algoritmusnál lesz minimális a lemez és a tár közötti adatátvitelek átlagos száma. Legalábbis a jelenleg ismert rendező algoritmusokat figyelembe véve.

A közvetlen hozzáférés lehetőségét kihasználó file feldolgozás ritkán igényli a file elemeinek a fizikai átrendezését a lemezen. Ehelyett kulcslistákat használunk a hozzáférés könnyítésére. A kulcslisták olyan táblázatok, amelyek rekordjai tartalmazzák a file elemből annak a mezőnek az értékét, amely alapján a file elemeit el akarjuk érni (kulcsmező), és a file elem sorszámát. Ha kulcs a név, mint a 101. programban, akkor a táblázat a

```
type sortipus=record
                nev:string30;
              sorsz:integer;
              end;
```

rekordok tömbje. A táblázatot rendezzük a névmező értéke szerint, nem az állományt. A file kívánt elemének a sorszámát a kulcslistában bináris kereséssel gyorsan megtalálhatjuk, majd a seek eljárással ráállítva a file mutatót, a teljes adatrekordot beolvashatjuk. A terjedelmes kulcslistákat lemezen szoktuk tartani. A bináris keresést így is megvalósíthatjuk, vagy olyan adatszerkezetet választunk a listák tárolásához, amely a keresést megkönnyíti.

11.7 A file könyvtár módosítása

A mágneslemezen tárolt adatállományokat törölhetjük, nevüket

megváltoztathatjuk. Ezeknél a műveleteknél a file elemeivel nem dolgozunk, így voltaképpen nem is kell ismernünk a típusukat. Ilyen esetekben a Turbo Pascalban lehetséges "típus nélkül" is dolgozni a file-okkal. Egy típus nélküli file deklarálásakor nem adunk meg bázistípust:

```
var f:file;
```

A deklaráció szerint f típus nélküli file.
Adatállományt az

```
erase
```

eljárással törölhetünk, nevét a

```
rename
```

eljárással változtathatjuk meg. Az erase paramétere a file azonosító. A rename eljárásnak két paramétere van: az első a file azonosító, a második egy stringkifejezés, amely az új nevet határozza meg. A programok is állományok, így programokat is törölhetünk, ill. átnevezhetünk a lemezen.

```
program torol;
  var prog:file;

  begin
    assign(prog,'felir.bak');
    erase(prog);
    end.
```

File törlése

102. program

A 102. programmal a "felir.bak" nevű programot töröljük.
Készíthetünk egy használhatóbb szervizprogramot, amelyikkel a Turbo Pascal rendszerből is kezelhetjük a tartalomjegyzéket. Az adatállomány nevét a billentyűzetről adjuk meg és menüből választjuk ki a végrehajtandó tevékenységet.
Ha az állomány nevét nem adjuk meg pontosan, akkor hibajelzést kapunk. Korrekcióra nem lévén lehetőségünk, újra kell indítani a programot. Ezt a kényelmetlenséget megszüntethetjük, ha kikapcsoljuk a be- és kiviteli műveletek hibajelzését.
A fordítóprogram az I compiler direktíva hatására hozza létre a futás közben létrejött hibákat észlelő utasításokat. Az alapértelmezés {$I+}, így rendesen minden be- és kiviteli művelet után automatikusan végrehajtódik egy hibaellenőrző eljárás. Hiba esetén a program végrehajtása megszakad.
Ha az I direktívát kikapcsoljuk, nem lesz hibaellenőrzés. Ezért hiba esetén a program sem szakad meg, de a programozó köteles meghívni az

ioresult

függvényt. Egyébként a létrejött hibafeltétel miatt a következő be- és kiviteli művelet nem hajtódik végre. Az ioresult a hiba kódját adja vissza, mint függvényértéket. Ha ez az érték 0, akkor nem volt hiba a művelet elvégzése közben. A hibakód értéke a hiba okát is megadja. Pl. ha nemlétező állományt akarunk megnyitni olvasásra, akkor 1 lesz a függvény értéke (lásd: I) Melléklet). Ha nem akarjuk, hogy hiba miatt a program futása megszakadjon és van reményünk arra, hogy a hiba okát meg tudjuk szüntetni, akkor kikapcsoljuk az I direktívát és az ioresult függvényt használjuk. Bemutatunk egy logikai értékű függvényt, amely a true értéket adja vissza, ha egy állomány létezik és false, ha nem létezik (103. program).

```
function letezik(fileaz:file):boolean;
  var van:boolean;

  begin
    {$I-}
    reset(fileaz);
    {$I+}
    van:=ioresult=0;
    if not van then
      letezik:=false;
    else begin
      close(fileaz);
      letezik:=true;
      end;
    end;
```

A "letezik" predikátum

103. program

A predikátumot felhasználhatjuk egy elegánsabb szervizprogram elkészítéséhez (104. program).

```
program szerviz;
  const nevhossz=12;
        muvhalmaz:set of 'T'..'V' =['T','U','V'];
  type  nevstr=string[nevhossz];

  var allomany:file;
      nev1,nev2:nevstr;
      muvelet:char;

  begin
    repeat
      clrscr;
      writeln('NYOMJA MEG A MEGFELELO BETUT!');
```

```
writeln;writeln;
highvideo;
write('          T');
lowvideo;
writeln('orles');
writeln;
highvideo;
write('          U');
lowvideo;
writeln('j nev');
writeln;
highvideo;
write('          V');
lowvideo;
writeln('ege');
writeln;writeln;
repeat
  read(kbd,muvelet);
  until upcase(muvelet) in muvhalmaz;

case upcase(muvelet) of

'T':begin
     writeln('Irja be a file nevet,');
     write('amit torolni akar:');
     buflen:=12;
     readln(nev1);
     assign(allomany,nev1);

     if letezik(allomany) then
       erase(allomany)
     else begin
       writeln('A "',nev1,'" allomany nem letezik');
       delay(3000);
       end;
     end;

'U':begin
     write('A regi nev:');
     buflen:=12;
     readln(nev1);
     assign(allomany,nev1);

     if letezik(allomany) then begin
       write('Az uj nev:');
       buflen:=12;
       readln(nev2);
       rename(allomany,nev2);
       end
     else begin
       writeln('A "',nev1,'" allomany nem letezik');
       delay(3000);
       end;
     end;
```

```
        'V':;
         end; {case}

      until upcase(muvelet)='V';
    end.
```

Tartalomjegyzék szerviz

104. program

A típus nélküli file-okat más esetekben is használhatjuk, pl. adatállományok típustól független egységes kezelésére (pl. másolásnál), text file-ok részeinek közvetlen elérésére. Az általános típusú állományok kezelésére használt eljárásokat általában használhatjuk, de beolvasást és kiírást a

blockread

és a

blockwrite

eljárásokkal végezhetjük el.
A típus nélküli file-ok feldolgozásának kérdéseivel ebben a könyvben nem foglalkozunk.

11.8 Feladatok

1. Tekintsük a következő deklarációkat:

a)
```
type x=file of set of char;
```

b)
```
type b=array[1..1000] of real;
     y=file of b;
```

c)
```
type z=file of record
                a:integer;
                b:array[1..9] of char;
                c:set of 1..9;
                end;
```

d)
```
type w=record
          a:file of real;
          b:string[22];
          c:integer;
          end;
     v=file of w;
```

A következő állítások közül melyik igaz?
1. a) hibás, mert file nem lehet set típusú.

2. b) jó, mert file bázistípusa lehet tömbtípus.
3. c) hibás, mert a rekord típust külön kell deklarálni.
4. d) hibás, mert a rekord egyik mezője file típusú.

2. Írjon egy logikai értékű függvényt, amely egy integer bázistípusú file-t végignéz, hogy egy adott egész szám előfordul-e benne. Ha igen, akkor a true, ha nem, akkor a false értéket adja vissza a függvény!

3. Írjon egy függvényt, amely egy text file sorait megszámolja. A függvény értéke a sorok száma legyen.

4. Írjon egy eljárást, amely egy adott text file sorait sorszámmal kiegészítve kinyomtatja.

5. Írjon eljárást, amely egy text file-ban egy adott jelsorozat minden előfordulását egy másik adott jelsorozattal cseréli ki.

6. Egy iskolában két ifjúsági szervezet működik, a DEMISZ és a FIDESZ. Mindkét szervezet tagnyilvántartását tervezze meg rekord bázistípusú file-ok segítségével. Készítsen programot, amely meghatározza (pl. kinyomtatja) azokat, akik egyidejűleg mindkét társulatnak tagjai.

7. Egy vállalatnál a raktárban legfeljebb 1000 féle anyagot, eszközt stb. tárolnak. Készítsen nyilvántartó programot! Minden raktározott cikkről tárolni kell egy azonosító számot, a megnevezést, mennyiséget, egységárat, a minimális készlet értékét. A nyilvántartó program készítsen teljes listát a raktárban lévő dolgokról!

8. A 7. feladat kipróbálásához szüksége van létező adatállományra. Készítsen programot, amellyel egy ilyen adatállományt fel tud írni!

9. A 7. feladat raktári nyilvántartásánál, ha valamelyik cikknek a mennyisége a minimális készlet alá csökken, akkor abból haladéktalanul rendelni kell. Írjon programot, amely kiírja a megrendelendő cikkek listáját!

10. A raktári készlet több okból is megváltozhat. Az egyes okokat kódoljuk:

a változás oka	kód
felhasználás	1
eladás	2
selejtezés	3
vásárlás	4
visszavételezés	5

Készítsen programot a raktári készlet módosítására. A módosító állomány rekordjai tartalmazzák az azonosító számot, a változás okát és a mennyiséget.

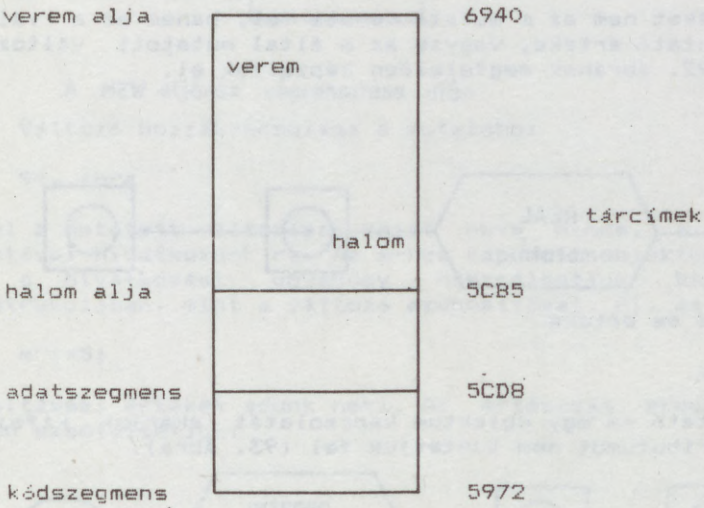
11. Tervezzen egy mini "információs rendszert" Magyarország királyairól! Az adatbázisnak tartalmaznia kell minden király nevét, uralkodásának kezdetét és végét, születésének és halálának dátumát, származását (pl. Ferenc József: osztrák), feleségének nevét és származását.
Hogyan nyerhetne ki olyan információkat az információs rendszerből, mint: Ki uralkodott 1411-ben? Ki volt a felesége? Mikor uralkodott Mátyás?
Az olvasó maga is találjon ki ilyen kérdéseket és gondolja végig, hogyan nyerhetné ki a választ a nyilvántartásból!
Hogyan lehetne egy általános (bármely kérdésre válaszoló) információs programot készíteni?

12. DINAMIKUS ADATSZERKEZETEK

A dinamikus adatszerkezeteket a file-okhoz hasonlóan végtelen számosságú típusok létrehozására használjuk. Listastruktúrákat, fastruktúrákat, gráfokat definiálhatunk, amelyek mérete elvileg nem korlátozott ugyanúgy, mint a file-oké sem. Gyakorlatilag persze itt is kell korlátokkal számolnunk, annál is inkább, mert ezeket a struktúrákat rendszerint a tárban hozzuk létre.
A dinamikus adatszerkezetek mérete nem állandó, a program futása közben jönnek létre, növekszenek, változnak, fogynak és szűnnek meg. Ezért nem határozható meg fordítás közben a számukra szükséges tárigény. A Turbo Pascal rendszer három részre osztja a tárat:

a kódszegmensre, amelyben a program van,
az adatszegmensre, amely a globális változókat tartalmazza,
és a dinamikus tárra.

A dinamikus tárban van a verem, az alprogramok lokális változóit (és a visszatérési címeket) tartalmazó terület, és az ún. halomtár, amelyben a dinamikus adatszerkezetek helyezhetők el. A program működése közben a veremben és a halomban lefoglalt területek egymással szemben növekednek (91. ábra).



Tárfelosztás

91. ábra

12.1 Mutatók és a pointer típus

A dinamikus adatszerkezetek létrehozásának alapvető eszközei a mutatók. A mutatók a nyelv pointer típusú objektumai. A pointer típusokat a ^ jel és egy típusazonosító segítségével deklaráljuk:

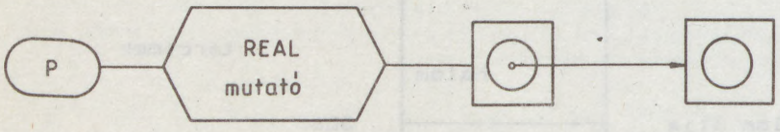
```
type intpoitipus=^integer;

var m:intpoitipus;
```

A deklarációval egy olyan m mutatót (mutató változót) hoztunk létre, amely az integer típushoz "kötődik".
De mi az a mutató és mit jelent ez a kötődés. A mutató olyan nyelvi objektum, amellyel egy másik - önálló névvel nem rendelkező - változót érhetünk el. m-en keresztül történetesen egész típusú változókat. A

```
type repoitipus=^real;

var p:repoitipus;
```

deklaráció p változóján keresztül pedig valós változóra hivatkozhatunk.
A mutató értéke tehát nem szám, nem karakter, nem valamilyen más ismert típusba tartozó érték, hanem változó.

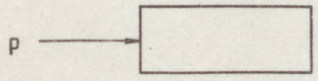
Az egész értéket nem az m mutató veheti fel, hanem az a változó, amely az m mutató értéke, vagyis az m által mutatott változó. A mutatókat a 92. ábrának megfelelően képzeljük el.



A mutató és értéke

92. ábra

Ha csak a mutató és egy objektum kapcsolatát akarjuk kifejezni, akkor az attributumot nem tüntetjük fel (93. ábra).



A mutató sematikus ábrázolása

93. ábra

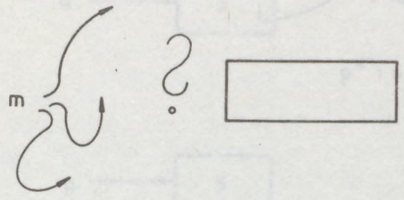
Egy mutató változóhoz kapcsolt változókat a halomból vesszük. A hozzárendelést a

```
new
```

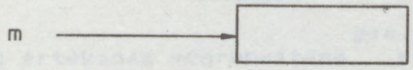
eljárás végzi. A

```
new(m)
```

eljáráshívás után az m pointer egy egész változó tárolására alkalmas tárterületre mutat (94. ábra);



A MEW eljárás végrehajtása előtt



A MEW eljárás végrehajtása után

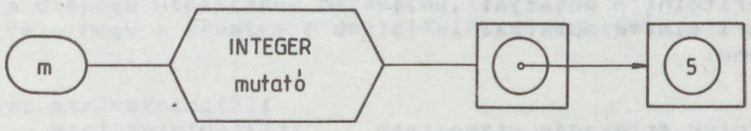
Változó hozzákapcsolása a mutatóhoz

94. ábra

Mivel a mutatott változónak saját neve nincs, a programban a mutatóval hivatkozunk rá. Az m-hez kapcsolt objektumot m^ jelöli. Ezt a hivatkozást ugyanúgy használhatjuk bármely nyelvi konstrukcióban, mint a változó azonosítókat. Pl. az

```
m^:=5;
```

utasítással értéket adunk neki. Az értékadás eredményét a 95. ábrán szemléltetjük.



Értékadás a névtelen változónak

95. ábra

A pointerhez kapcsolt objektumot le is "akaszthatjuk" és visszaadhatjuk a halomba. Ezt a

dispose

eljárással tehetjük meg.
Ha a p és a q mutatók azonos típusúak, akkor végrehajtható a

p:=q

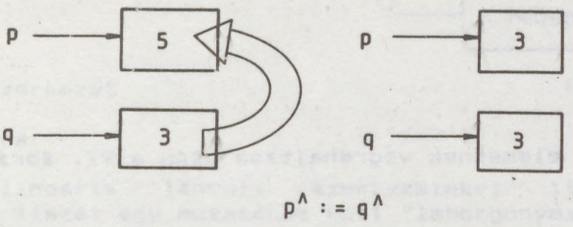
pointer értékadás. A 105. program lefuttatásával vizsgáljuk meg, mi történik egy ilyen értékadáskor.

```
program poiertekadas;

  type ipoi=^integer;
  var p,q:ipoi;

  begin
    clrscr;
    new(p);
    p^:=5;
    new(q);
    q^:=3;
    writeln('Eredetileg');
    writeln('p mutatja:',p^);
    writeln('q mutatja:',q^);

    writeln;
    p^:=q^;
    writeln('Valtozo ertekadas utan');
    writeln('p mutatja:',p^);
    writeln('q mutatja:',q^);

    writeln;
    p^:=5;
    writeln('Eredetileg');
    writeln('p mutatja:',p^);
    writeln('q mutatja:',q^);

    writeln;
    p:=q;
    writeln('Pointer értékadás után');
    writeln('p mutatja:',p^);
    writeln('q mutatja:',q^);
    end.
```

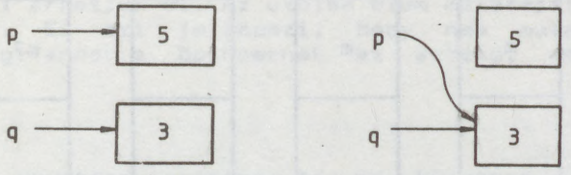
A pointer értékadás vizsgálata

105. program

A program végrehajtásából látjuk, hogy eredetileg mindkét esetben a p az 5-ös, a q a 3-mas értéket mutatta. A közönséges (a mutatott objektumokra vonatkozó) változó értékadás után természetesen mindkét érték 3 lett. Ugyanezt tapasztaljuk a pointer értékadás után is. A két értékadás mechanizmusa viszont merőben eltérő (96. ábra).



p^ := q^



p:= q

Az értékadás végrehajtása

96. ábra

A p:=q értékadás eredményeképpen a p mutató oda fog mutatni, ahová a q.
A pointer értékadás után a korábban p-hez tartozó változóban az 5-ös érték megmaradt, de többé nem férhetünk hozzá. Nem csatoltuk vissza a halomhoz sem, így nem tudjuk többször felhasználni. Számunkra elveszett: hulladék.

A dispose eljárással visszakapcsolhattuk volna az értékadás előtt a halomra ezt a tárterületet. Ehhez a

```
dispose(p);
p:=q;
```

utasításokat kellett volna leírni. De mi történik a halomban a new és a dispose utasítások hatására?
Tegyük fel, hogy a következő deklarációk érvényesek:

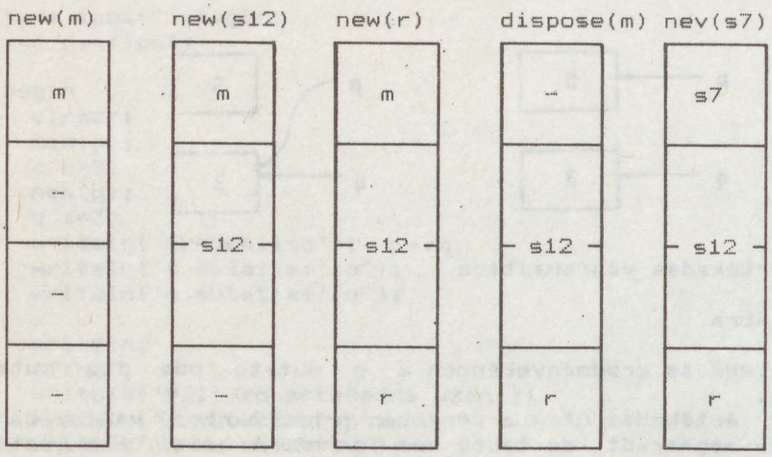
```
type str7=string[7];
     str12=string[12];

var m:^integer;
```

```
r:^real;
s7:^str7;
s12:^str12;
```

A halom állapotait a

```
new(m);
new(s12);
new(r);
dispose(m);
new(s7);
```

utasítássorozat elemeinek végrehajtása után a 97. ábrán mutatjuk be.



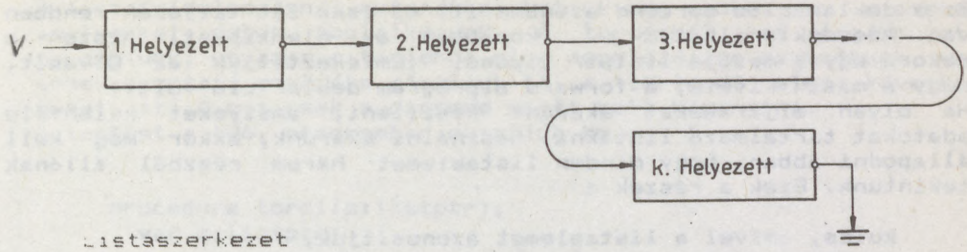
A new és a dispose működése

97. ábra

A new és a dispose eljárások nem bájtonként, hanem 8 bájtos darabokban kezelik a halmot. Ez általában nem okoz számottevő tárpazarlást, mert nem integer értékeket, hanem többnyire hosszabb rekordokat szoktunk a mutatókkal elérni.

12.2 Láncolt adatszerkezetek

Láncolt adatszerkezetekről akkor beszélünk, ha mutatókkal kapcsolunk össze objektumokat. Az összekapcsolt objektumok rekordok lehetnek, hiszen egyéb adatmezők mellett mutatókat - azaz pointer típusú mezőket - is kell tartalmazniuk.
Pl. egy sportversenyen a versenyzők adatait tartalmazó rekordokat a helyezési sorrendben fűzhetjük össze (98. ábra).



Listaszerkezet

98. ábra

Az ilyen lineáris láncolt szerkezeteket listaszerkezetnek nevezzük. A listát egy mutatóhoz kell "lehorgonyozni" (v), ezen keresztül érhetjük el az első elemét. Ezt a mutatót listafejnek szokták nevezni. A lista valamely eleméből a következőt mutató segítségével érhetjük el. Az utolsó elem mutatóját a 98. ábrán "földeltük". Ez azt jelképezi, hogy nem mutat sehová. Azt mondjuk, hogy annak a pointernek az értéke, amely nem mutat sehová,

nil.

A nil az egyetlen pointer típusú konstans, minden pointer típussal kompatibilis.Most nézzük, hogyan hozhatunk létre egy listát.
A lista elemeit a következő deklarációval határozhatjuk meg.

```
type listptr=^elemtipus;

     elemtipus=record
                 rajtszam:integer;
                 nev:str20;
                 orszag:str10;
                 eredmeny:real;
                 sztori:szoveg;
                 kovetkezo:listptr;
                 end;
```

(Feltesszük, hogy a str10, str20 és a szoveg típusokat már korábban definiáltuk.)
A lista kialakításához csak a listafejet kell deklarálni (rangsor). Többnyire szükség van segédmutatókra is:

```
var rangsor,p,q:listptr;
```

Rekordváltozót deklarálni nem kell. A rekordokat a feldolgozás során a new eljárással dinamikusan hozzuk létre.
Az olvasó talán már észrevette, hogy a típusdeklarációknál megsértettünk egy fontos szabályt. Először deklaráltuk a listptr mutató típust, ahol felhasználtuk a még nem deklarált elemtípust. Tehát hivatkoztunk olyan típusra, amit csak később definiáltunk.

Ez a deklarációs sorrend azonban itt és csak itt teljesen rendben van. Nem deklarálhatjuk korábban az elemtípust, hiszen a rekord egyik mezője listptr típusú. (Emlékeztetjük az Olvasót, hogy a másik kivétel a forward alprogram deklaráció volt.)
Ha olyan eljárásokat akarunk készíteni, amelyeket különféle adatokat tartalmazó listáknál használni akarunk, akkor meg kell állapodni abban, hogy minden listaelemet három részből állónak tekintünk. Ezek a

kulcs, amivel a listaelemet azonosítjuk,
az egyéb adat, amely lehet maga is rekord,
a következő elem mutató.

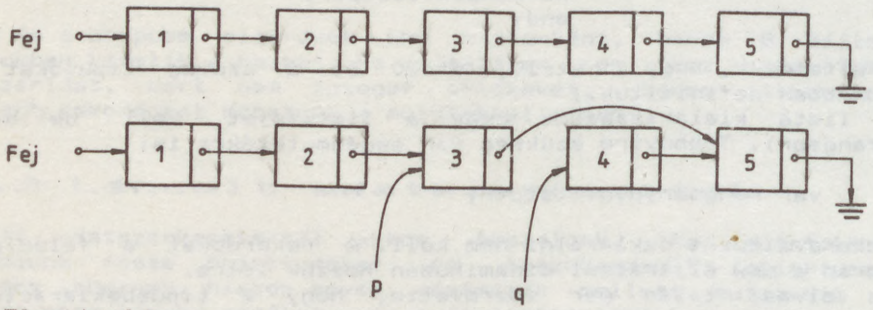
Ezért állapodjunk meg a

```
type listptr=^elemtipus;

     elemtipus=record
                 kulcs:kulcstipus;
                 adat: adattipus;
                 kovetkezo:listptr;
```

deklarációkban. A kulcstípust és az adattípust előzőleg igényeink szerint deklaráljuk.
A listaszerkezetnek a hasonló statikus adatszerkezetekkel, a táblázatokkal szemben az egyszerű módosíthatóság az előnye.Ha egy táblázatba egy sort be akarunk szúrni vagy ki akarunk hagyni, ez mindenképpen sorok mozgatását igényli. Ha pl. az első sort akarom törölni, akkor ez a

```
for i:=2 to n do
  x[i-1]:=x[i];
```

ciklusban nem kevés értékadás végrehajtását igényli. Ez - figyelembe véve, hogy a sorok bonyolult rekordok - sok munkát jelentenek. A listáknál egy elem kihagyásához mindössze egyetlen pointer értékadást kell végrehajtani (esetleg még egy dispose eljárást is) (99. ábra).



Elem törlése

99. ábra

A listafeldolgozásnál mutatókkal kötjük meg azokat az elemeket, amelyekkel éppen foglalkozunk. A törlésnél két segédmutatót használunk, ugyanis a törlendő elem megelőzőjére szükségünk van. Ennek a mutató mezőjébe másoljuk át a törlendő elem következő mutatóját. A q-t csak a dispose miatt kell használni.
A törlést a 106. programban mutatjuk be.

```
procedure torol(p:listptr);
  var q:listptr;

  begin
    q:=p^.kovetkezo;
    p^.kovetkezo:=q^.kovetkezo;
    dispose(q);
    end;
```

Listaelem törlése

106. program

A p paraméter itt a törlendőt megelőző elemre mutat.
Izgalmasabb kérdés, hogyan keressük meg a lista kérdéses elemét. Keresésre az "utazó mutatót" használjuk. A p mutatót a következőképpen utaztatjuk végig a teljes listán:

```
.....................

{p-t a lista elejére állítjuk}

p:=fej;

{ismétlés, amíg a lista végére érünk}

while p<>nil do begin

  writeln(p^.kulcs);   {a listaelemben tárolt}
  kiiras(p^.adat);     {információ kiírása}

  p:=p.kovetkezo;      {utaztatás}
  end;
```

Tegyük fel, hogy a listaelemek a kulcsok növekvő sorrendjében következnek egymás után és készítsünk egy eljárást, amely egy adott kulcsú elemet töröl (107. program).

```
procedure torles(var listafej:listptr;
                 torlkulcs:kulcstipus;
                 var statusz:byte);
  var p,q,r:listptr;
  begin
    p:=listafej;
```

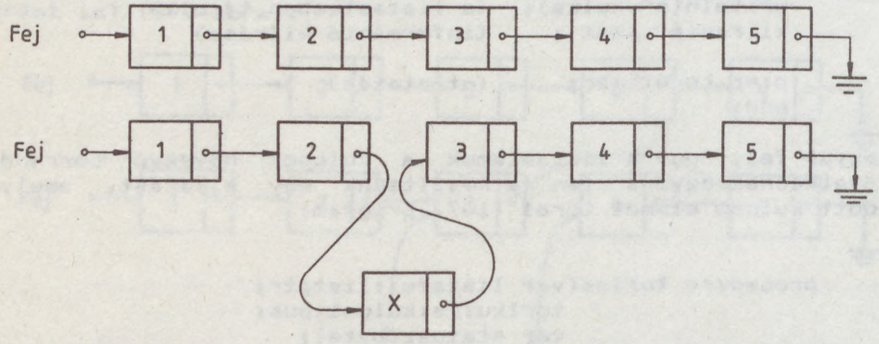
```
if p=nil then begin
  statusz:=2; {ures lista}
  exit;
  end;
while (p<>nil) and (p^.kulcs<torlkulcs) do begin
  r:=p;
  p:=p^.kovetkezo;
  end;
if p=nil then
  statusz:=1
else
   if p^.kulcs>torlkulcs then
     statusz:=1
   else begin
     statusz:=0;
     if p=listafej then
       listafej:=p^.kov
     else
     r^.kovetkezo:=p^.kovetkezo;
     dispose(p);
     end;
end;
```

Keresés és törlés

107. program

Az eljárásban a statusz paraméter adja át a hibakódot, ha a törlés meghiúsul. 2 a visszaadott érték, ha a lista üres, 1, ha nem tartalmazta a törölni kívánt elemet és 0, ha a törlést sikeresen végrehajtotta.

Új elem beszórása is hatékonyan megvalósítható. Ha már ismerjük a beszórás helyét, akkor létre kell hozni és adattal fel kell tölteni egy új névtelen rekordváltozót, majd két mutató beállításával bekapcsoljuk a kívánt helyre (100. ábra).



Új elem beszórása

100. ábra

Tegyük fel ismét, hogy a listánk elemei a kulcsmező növekvő értékei szerint követik egymást. Bemutatunk egy eljárást, amely egy adott elemet a megfelelő helyre illeszt be a listába, úgy, hogy a rendezettség megmaradjon (108. program).

```
procedure beszur(var listafej:listptr;
                 ujelem:elemtipus);

 var p,q,r:listptr;
     megvan:boolean;

 begin
   new(r);
   r^:=ujelem;
   r^.kovetkezo:=nil;

   p:=listafej;

   {ha a lista üres}

   if p=nil then
     listafej:=r
   else begin
     megvan:=false;
     while (p<>nil) and not megvan do
       if p^.kulcs<ujelem.kulcs then begin
         q:=p;
         p:=p^.kovetkezo;
         end
       else
         megvan:= true;

   r^.next:=p;

   if p=listafej then
     listafej:=r
   else
     q^.next:=r;
   end
 end;
```

Új elem beszórása

108. program

A "beszur" eljárást rendezett listák felépítésére is használhatjuk, hiszen az üres listát is kezeli. A meghajtó programban a listafejet nil-re kell állítani a "beszur" első meghívása előtt (109. program).

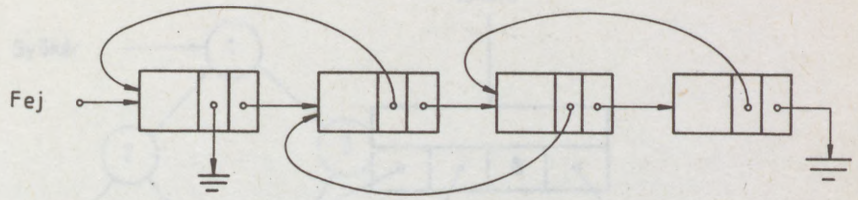
```
program listaepites;

  const vegjel=0;

  type str20=string[20];
       str10=string[10];

       listptr=^elemtipus;

       elemtipus=record
                   rajtszam:integer;
                   hely:real;
                   nev:str20;
                   orszag:str10;
                   kovetkezo:listptr;
                   end;

  var versenyzo:elemtipus;
      p,helylista:listptr;
      i:integer;

  {$I listolv.pas}

  {$I listir.pas}

  begin
    helylista:=nil;
    repeat
      olvas(versenyzo);
      if versenyzo.rajtszam<>0 then
        beszur(helylista,versenyzo);
      until versenyzo.rajtszam=0;

    {kiiras helyezesi sorrendben}

    clrscr;
    writeln('Helyezesi sorrend');
    writeln('-----------------');
    writeln;
    i:=0;
    p:=helylista;
    repeat
      i:=i+1;
      writeln(i,'-edik helyezett);
      kiir(p^);
      writeln;
      p:=p^.kovetkezo;
      until p=nil;
    end.
```

Lista felépítése

109. program

Az "olvas" eljárás a rekordok mezőinek tartalmát ovassa be, a "kiir" pedig kiírja a versenyzők adatait. Feltételeztük, hogy ezek az eljárások a lemezen a "listolv.pas", ill. a "listir.pas" nevű állományokban rendelkezésünkre állnak.

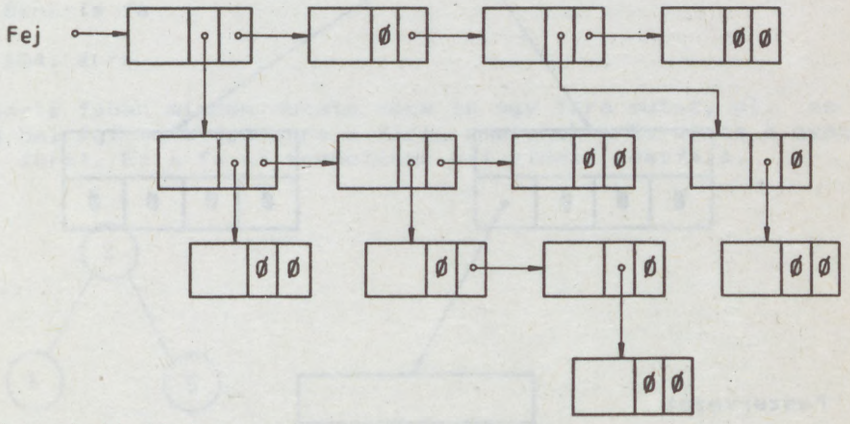
A listaelemeket két pointerrel is összekapcsolhatjuk, ekkor mindegyik elem megelőzőjét és rákövetkezőjét is elérhetjük (101. ábra).



Kétszeresen kapcsolt lista

101. ábra

A mesterséges intelligencia kutatás területén gyakran alkalmaznak összetett listaszerkezeteket. Itt a listaelemek lehetnek maguk is listák (102. ábra).



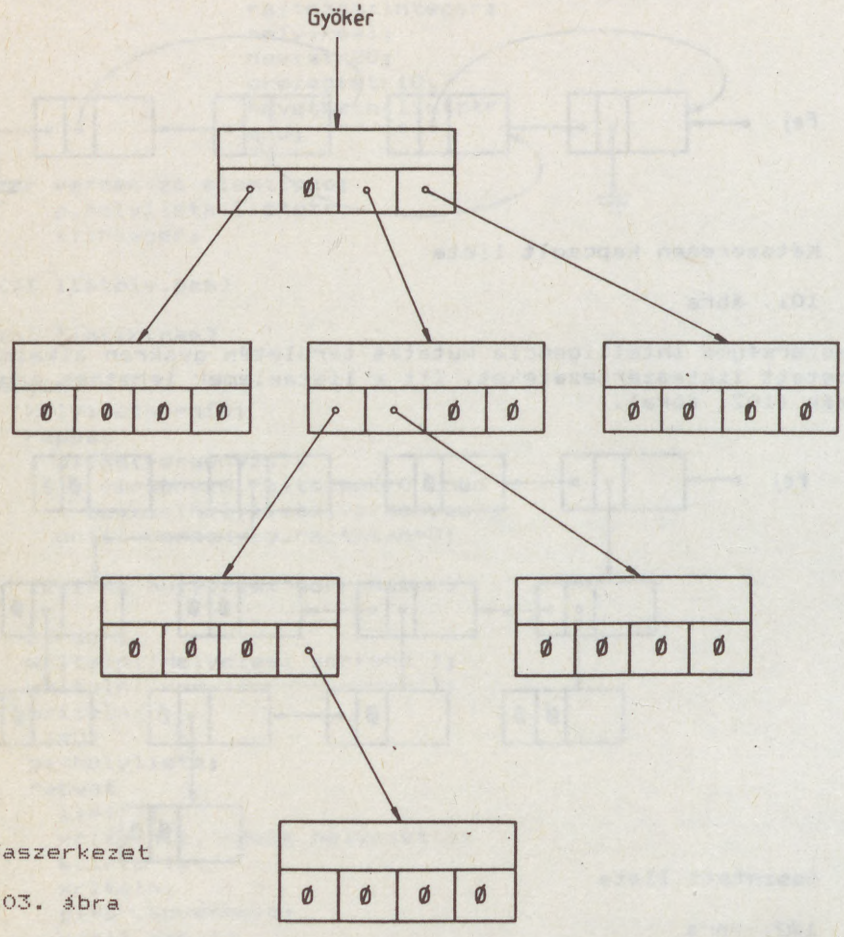
összetett lista

102. ábra

Léteznek listakezelő programozási nyelvek, ezeknek alapvető adatszerkezetei a listák. Ilyen a LISP nyelv amit a mesterséges intelligencia körébe tartozó problémák programozásában használnak.

12.3 Fák, bináris fák

Az alkalmazások szempontjából nagy jelentőségűek azok a lácolt szerkezetek, amelyek minden elemnél elágazhatnak (103. ábra). Az ilyen szerkezetek - ha a kapcsolódásokat éleknek, az elemeket csúcsoknak tekintjük - gráfok tárbeli ábrázolásai. Ha az ábrázolt gráf fa, akkor az adatszerkezetet is fának nevezzük. A fákban minden elemet csak egyetlen mutató láncon érhetünk el. A fa egy kitüntetett csúcsát, amit egy - a programban explicite deklarált globális - mutatóval tartunk kézben, a fa gyökerének nevezzük.



Faszerkezet

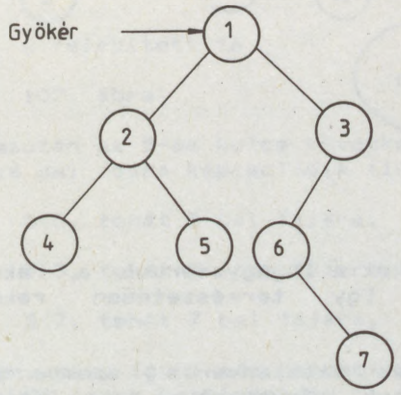
103. ábra

A 103. ábrán a nil értékű pointereket 0-val jelöltük.
A fa egy csúcsát (elemét) olyan rekordszerkezettel valósíthatjuk meg, amelyben a mutatókat pointer bázistípusú tömbbel adjuk meg. Pl.:

```
type fapoi=^csucsrekord;
```

```
csucsrekord=record
                kulcs:kulcstipus;
                adat:adatrekord;
                ag:array[1..4] of fapoi;
                end;
```

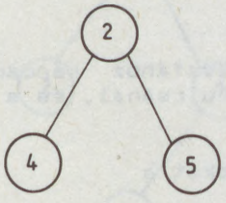
A fák fontos alaptípusait a bináris fák alkotják. A bináris fa minden csúcsánál legfeljebb kétfelé ágazik. A két ágat meg szoktuk különböztetni, így jobb ágról és bal ágról beszélünk (104. ábra).



Bináris fa

104. ábra

A bináris fában minden mutató maga is egy fára mutat, pl. az 1. csúcs bal ági mutatója arra a fára, amelynek a 2. csúcs a gyökere (105. ábra). Ez a fa az eredetinek bal oldali részfája.



Az 1-es csúcsra illeszkedő bal részfa

105. ábra

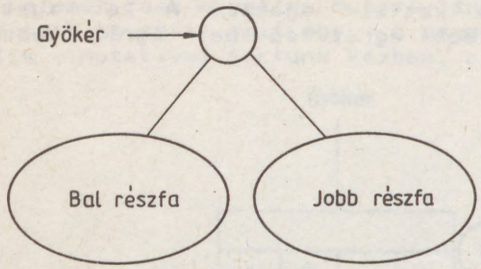
A bináris fákat így rekurzíve is definiálhatjuk. Bináris fa

1. a nil (üres fa);

2. egyetlen csúcs - a gyökér - önmagában;

3. a gyökér a hozzákapcsolódó jobb oldali és bal oldali részfával együtt.

A rekurziót a 3. pont tartalmazza (106. ábra).



A fa rekurzív meghatározása

106. ábra

A fákkal kapcsolatos tevékenységeket a legegyszerűbb a rekurzív definíció alapján programozni. Így természetesen rekurzív algoritmusokhoz jutunk.

A fák az információ tárolás és visszanyerés szempontjából kiváltképp hasznos adatszerkezetek. Egyrészt - a listákhoz hasonlóan - könnyen módosíthatók, másrészt a bináris kereséshez hasonló hatékonysággal nyerhetjük vissza a fa csúcsaiban tárolt információt. E két tulajdonság miatt az adatbázisok sokszor fa szerkezetűek.
A 12.2 alfejezetben egy listát építettünk fel úgy, hogy elemei rendezettek voltak. Most megoldunk egy hasonló feladatot bináris fával.
A fát a következőképpen építjük fel:

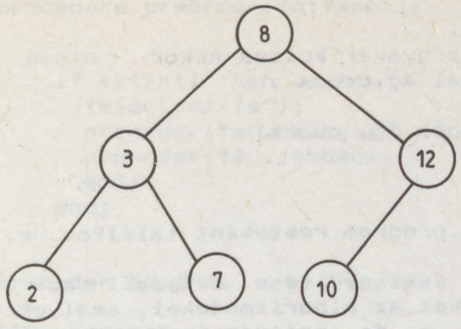
az első rekordot a fa gyökerébe írjuk,

minden további rekordot a bal oldali részfához kapcsolunk, ha a rekord kulcsa kisebb a gyökérelem kulcsánál, és a jobb oldali részfához egyébként.

Ha a kulcsok egész számok, akkor, ha a csúcsokat a

8, 3, 12, 2, 7, 10

sorrendben adtuk meg, az ennek megfelelően felépített fát a 107. ábrán láthatjuk.



A felépített fa

107. ábra

Ha ezután az 5-ös kulcs következik, a megfelelő csúcs a 7-es csúcs bal ágára kapcsolódik (108. ábra). Ugyanis

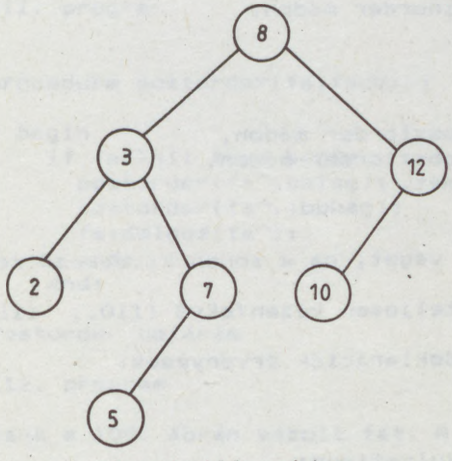
5<8, tehát 8 bal fájára,

5>3, tehát 3 jobb fájára,

5<7, tehát 7 bal fájára,

de ez üres, így 5 ide kapcsolódik.

Új csúcs beillesztése



Az új elem beillesztése

108. ábra

Az algoritmust rekurzíve fogalmazhatjuk meg.

```
algoritmus beilleszt(gyökér, csucs);
```

```
Ha a gyökér=nil, akkor
   legyen csócs a gyökér,
egyébként ha csócs kulcsa<gyökér kulcsa akkor
             beilleszt(bal ág,csócs),
          egyébként
             beilleszt(jobb ág, csócs)
vége.
```

Ezzel az algoritmussal a 113. program részeként találkozunk.

A fában tárolt információ összegyűjtése céljából az összes csócsot sorra kell venni. Azokat az algoritmusokat, amelyek ezt a tevékenységet megvalósítják, a fa bejárásainak nevezzük. A bejárásra is használhatunk rekurzív algoritmusokat. Attól függően, hogy a gyökérelemben tárolt információ feldolgozására mikor kerül sor, háromféle fabejárási algoritmust használhatunk.

1) Preorder bejárás

```
feldolgozzuk a gyökérelemet,
bejárjuk a bal részfát preorder módon,
bejárjuk a jobb részfát preorder módon.
```

2) Inorder bejárás

```
bejárjuk a bal részfát inorder módon,
feldolgozzuk a gyökérelemet,
bejárjuk a jobb részfát inorder módon.
```

3) Posztorder bejárás

```
bejárjuk a bal részfát posztorder módon,
bejárjuk a jobb részfát posztorder módon,
feldolgozzuk a gyökérelemet.
```

Mindegyik algoritmus akkor ér véget, ha a soron következő részfák üresek.
Az algoritmusok programozása teljesen kézenfekvő (110., 111. és 112. program).
Tegyük fel, hogy a következő deklarációk érvényesek:

```
type fapoi=^csucstipus;
     csucstipus=record
                   kulcs:kulcstipus;
                   adat:adattipus;
                   balag:fapoi;
                   jobbag:fapoi;
                   end;
```

Ezután nézzük az eljárásokat.

```
procedure preorder(fa:fapoi);

  begin
    if fa<>nil then begin
      feldolgoz(fa^);
      preorder(fa^.balag);
      preorder(fa^.jobbag);
      end;
    end;
```

Preorder bejárás

110. program

```
procedure inorder(fa:fapoi);

  begin
    if fa<>nil then begin
      inorder(fa^.balag);
      feldolgoz(fa^);
      inorder(fa^.jobbag);
      end;
    end;
```

Inorder bejárás

111. program

```
procedure postorder(fa:fapoi);

  begin
    if fa<>nil then begin
      postorder(fa^.balag);
      postorder(fa^.jobbag);
      feldolgoz(fa^);
      end;
    end;
```

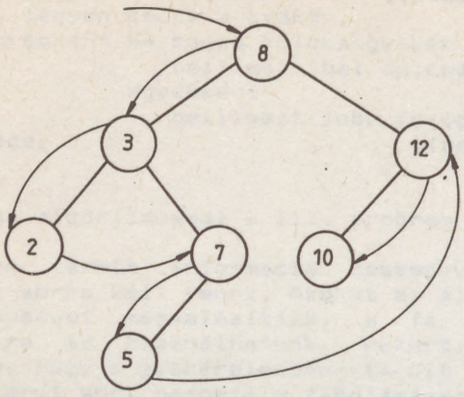
Postorder bejárás

112. program

Tekintsük a 108. ábrán vázolt fát. A preorder eljárás a csúcsokat a

8, 3, 2, 7, 5, 12, 10

sorrendben dolgozza fel (109. ábra).



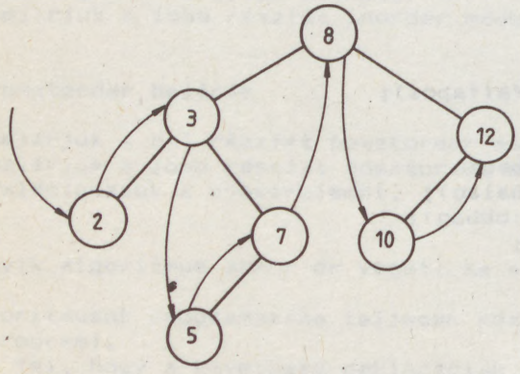
A bejárás preorder sorrendje

109. ábra

Az inorder algoritmus a

2, 3, 5, 7, 8, 10, 12

sorrendet adja (110. ábra). Vegyük észre, hogy ezzel lényegileg egy új rendezési algoritmushoz jutottunk.

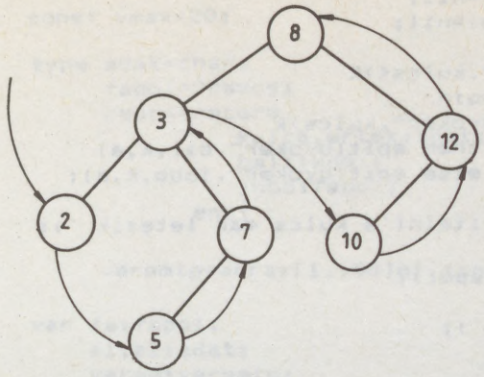


Az inorder bejárási sorrend

110. ábra

A postorder bejárásnál kapott sorrend:

2, 7, 5, 3, 10, 12, 8.



A postorder bejárás sorrendje

111. ábra

A 113. program egy teljes feldolgozás vázlatát adja. A "vizsgal" eljárás mindössze a vizsgált gyökér kulcsát és a benne tárolt adatot (ami itt csak egyetlen karakter) írja ki. A fa felépítése a korábban leírt algoritmussal történik. Az adatrekordok bevitelét a 0 kulcsértékkel (végjel) állíthatjuk meg.

```
program binfa;
const vmax=20;
type adat=char;
     fapoi=^csucs;
     csucs=record
             kulcs,ertek:adat;
             bal:fapoi;
             jobb:fapoi;
           end;
     veremtp=array[1..50]of fapoi;
var fa:fapoi;
    a1,a2:adat;
procedure epit(var gyoker:fapoi;
                   k,a:adat);
 begin
    if gyoker=nil
       then begin
          new(gyoker);
          gyoker^.kulcs:=k;
procedure olvas(var x,y:adat);
 begin
  write ('kulcs: ');
  readln(x);
  write('ertek: ');
  readln(y);
 end;
```

```
            gyoker^.ertek:=a;
            gyoker^.bal:=nil;
            gyoker^.jobb:=nil;
          end
        else if gyoker^.kulcs<>k
                then begin
                   if gyoker^.kulcs>k
                       then epit(gyoker^.bal,k,a)
                       else epit(gyoker^.jobb,k,a):
                end
                else writeln('a kulcs mar letezik ');
end;
procedure vizsgal(x:fapoi);
 begin
  write(x^.kulcs,'    ');
  writeln(x^.ertek);
 end;
procedure inorder(x:fapoi);
  begin
    if x<>nil then begin
      inorder(x^.bal);
      vizsgal(x);
      inorder(x^.jobb)
      end
   end;
begin {foprogram}
  clrscr;
  fa:=nil;
  olvas(a1,a2);
  while a1<>'.' do begin
    epit(fa,a1,a2);
    olvas(a1,a2)
    end;
  clrscr;
  writeln('inorder bejaras');
  inorder(fa);
 end.
```

Faépítés és bejárás

113. program

Javasoljuk, hogy az Olvasó mindegyik bejárási eljárással és különböző adatsorrendekkel kísérletezzen. Majd gondolkodjon el azon, hogyan tudná megírni a megismert programokat rekurzió nélkül. Elrettentő példaként az elszántabb Olvasók számára közöljük a preorder bejárás programját (ennnél a postorder bejárás nem rekurzív algoritmusa lényegesen bonyolultabb). Az algoritmus maga vermeli a visszalépésekhez szükséges információt, ezért veremkezelő alprogramokra is szükség van (pop és push). Tanulságos manuálisan végrehajtani az algoritmust és elgondolkozni a működésén.

```
program binfa;

 const vmax=20;

 type adat=char;
      fapoi=^csucs;
      csucs=record
                kulcs,ertek:adat;
                bal:fapoi;
                jobb:fapoi;
              end;

      veremtp=array[1..50]of fapoi;

 var fa:fapoi;
     a1,a2:adat;
     verem:veremtp;
     teteje:integer;

 {-------------------------------------}

 procedure push( x:fapoi);
  begin
   verem[teteje]:=x;
   teteje:=teteje+1;
  end;

 {-------------------------------------}

 procedure pop(var x:fapoi);
  begin
   teteje:=teteje-1;
   x:=verem[teteje];
   if verem[teteje]=nil then
   write('ures a verem');
  end;

 {-------------------------------------}

 procedure olvas(var x,y:adat);
  begin
   write ('kulcs: ');
   readln(x);
   write('ertek: ');
   readln(y);
  end;
```

```
{-------------------------------------}

procedure epit(var gyoker:fapoi;
                    k,a:adat);
 begin
    if gyoker=nil
        then begin
          new(gyoker);
          gyoker^.kulcs:=k;
          gyoker^.ertek:=a;
          gyoker^.bal:=nil;
          gyoker^.jobb:=nil;
        end

       else if gyoker^.kulcs<>k
               then begin

                    if gyoker^.kulcs>k
                       then epit(gyoker^.bal,k,a)
                       else epit(gyoker^.jobb,k,a);

               end
               else writeln('a kulcs mar letezik ');
 end;

{-------------------------------------}

procedure vizsgal(x:fapoi);
 begin
  write(x^.kulcs,'    ');
  writeln(x^.ertek);
 end;

{-------------------------------------}

procedure preorder (x:fapoi);

var p:fapoi;
```

```
begin
 push(nil);
 p:=x;
 while p<>nil do begin
 vizsgal(p);
 push(p);
 p:=p^.bal;
 while p=nil do begin
 pop(p);
 if p=nil then halt;
 p:=p^.jobb
 end
 end
end;
```

```
{═══════════════════════════════════════}

begin {foprogram}
  clrscr;
  fa:=nil;
  olvas(a1,a2);
  while a1<>'.' do begin
    epit(fa,a1,a2);
    olvas(a1,a2)
    end;
  clrscr;
  writeln('preorder bejaras');
  preorder(fa);
end.
```

Nem rekurzív preorder bejárás

114. program

Jól felhasználhatjuk a faszerkezeteket közvetlen hozzáférésű adatállományok feldolgozásánál. A kulcslistát célszerűbb faszerkezetben tárolni, mint táblázatban, mert így a keresés hatékonysága nem csökken, de az állomány változtatásakor a kulcsok módosítása lényegesen kevesebb munkát jelent.

12.4 Feladatok

1. A LISP nyelvben alapvető listaműveletek a CAR és a CDR. Mindkettő argumentuma egy lista. A CAR a listafejet adja vissza (ami már nem lista, hanem az első elem), a CDR pedig a lista "farkát". A lista farka a második elemmel kezdődő részlista. Készítse el a CAR és a CDR eljárást!

2. Ha list1 és list2 két listafej mutató, akkor mi lesz a

   ```
   list1:=list2
   ```

   értékadás hatása. (Tegyük fel, hogy a mutatók azonos típusúak.)

3. Írjon eljárást, amely egy adott listát megkettőz. Az eljárás feje legyen

   ```
   procedure copy(var eredeti,masolat:listptr).
   ```

4. Készítsen eljárást, amely egy mező értéke által meghatározott listaelemben tárolt információt adja vissza.

5. A 107. programban a

   ```
   (p<>nil) and (p^.kulcs<torlkulcs)
   ```

   összetett feltételt használtuk a törlendő elemet kereső ciklusban. Hogyan lehetne a strázsatechnikát használni, hogy a

   ```
   p<>nil
   ```

   relációt elhagyhassuk. Hogyan kellene ehhez a listát módosítani? Készítse el az eljárást strázsa figyelembevételével.

6. Írja át a 108. programot a strázsatechnika alkalmazásával!

7. Írjon programot sportverseny lebonyolításának számítógépes segítéséhez. A program rendezze helyezési sorrendbe a versenyzők rekordjait. Tegyük fel, hogy az eredményt valós szám fejezi ki és a nagyobb érték a jobb (pl. magasugrásnál). A verseny több fordulós, az elért legjobb eredmény érvényes. A program kövesse végig a fordulókat. Ha egy versenyző a korábbi eredményénél jobbat ért el, akkor az új eredményt jegyezze fel a rekordjában és módosítsa a rekord helyét, hogy a helyezési sorrend továbbra is fennálljon.

8. Készítsen eljárást elem törlésére kétszeresen kapcsolt listaszerkezethez!

9. Készítsen eljárást, amely kétszeresen kapcsolt lista első elemét törli.

10. Készítsen eljárást, amely egy kétszeresen kapcsolt lista végéhez egy új elemet kapcsol hozzá.

11. Tegyük fel, hogy egy bináris fa csúcsaiban valós számokat tárolunk. Készítsen eljárást, amely összegzi a tárolt számokat.

12. Az olyan fákat, amelyben minden csúcsból legfeljebb három ág indul ki, ternáris fáknak nevezzük. Fogalmazzon meg ternáris fákra bejárási stratégiákat és írja meg a megfelelő rekurzív eljárásokat!

MELLÉKLETEK

A) ASCII kódtáblázat

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
0	00	^@ NUL	32	20	SPC	64	40	@	96	60	'
1	01	^A SOH	33	21	!	65	41	A	97	61	a
2	02	^B STX	34	22	"	66	42	B	98	62	b
3	03	^C ETX	35	23	#	67	43	C	99	63	c
4	04	^D EOT	36	24	$	68	44	D	100	64	d
5	05	^E ENQ	37	25	%	69	45	E	101	65	e
6	06	^F ACK	38	26	&	70	46	F	102	66	f
7	07	^G BEL	39	27	'	71	47	G	103	67	g
8	08	^H BS	40	28	(	72	48	H	104	68	h
9	09	^I HT	41	29	)	73	49	I	105	69	i
10	0A	^J LF	42	2A	*	74	4A	J	106	6A	j
11	0B	^K VT	43	2B	+	75	4B	K	107	6B	k
12	0C	^L FF	44	2C	,	76	4C	L	108	6C	l
13	0D	^M CR	45	2D	-	77	4D	M	109	6D	m
14	0E	^N SO	46	2E	.	78	4E	N	110	6E	n
15	0F	^O SI	47	2F	/	79	4F	O	111	6F	o
16	10	^P DLE	48	30	0	80	50	P	112	70	p
17	11	^Q DC1	49	31	1	81	51	Q	113	71	q
18	12	^R DC2	50	32	2	82	52	R	114	72	r
19	13	^S DC3	51	33	3	83	53	S	115	73	s
20	14	^T DC4	52	34	4	84	54	T	116	74	t
21	15	^U NAK	53	35	5	85	55	U	117	75	u
22	16	^V SYN	54	36	6	86	56	V	118	76	v
23	17	^W ETB	55	37	7	87	57	W	119	77	w
24	18	^X CAN	56	38	8	88	58	X	120	78	x
25	19	^Y EM	57	39	9	89	59	Y	121	79	y
26	1A	^Z SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	^[ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	^\ FS	60	3C	<	92	5C	\	124	7C	|
29	1D	^] GS	61	3D	=	93	5D	]	125	7D	}
30	1E	^^ RS	62	3E	>	94	5E	^	126	7E	
31	1F	^_ US	63	3F	?	95	5F	_	127	7F	DEL

→ ↓	0-	1-	2-	3-	4-	5-	6-	7-		8-	9-	A-	B-	C-	D-	E-	F-
-0		►		0	@	P	`	p		Ç	É	á	▒	└	╨	α	≡
-1	☺	◄	!	1	A	Q	a	q		ü	æ	í	▒	┴	╤	β	±
-2	☻	↕	"	2	B	R	b	r		é	Æ	ó	▒	┬	╥	Γ	≥
-3	♥	!!	#	3	C	S	c	s		â	ô	ú	│	├	╙	π	≤
-4	♦	¶	$	4	D	T	d	t		ã	ö	ñ	┤	─	╘	Σ	⌠
-5	♣	§	%	5	E	U	e	u		à	ò	Ñ	╡	┼	╒	σ	⌡
-6	♠	▬	&	6	F	V	f	v		å	û	ª	╢	╞	╓	µ	÷
-7	•	↨	'	7	G	W	g	w		ç	ù	º	╖	╟	╫	τ	≈
-8	◘	↑	(	8	H	X	h	x		ê	ÿ	¿	╕	╚	╪	Φ	°
-9	○	↓	)	9	I	Y	i	y		ë	Ö	⌐	╣	╔	┘	Θ	∙
-A	◙	→	*	:	J	Z	j	z		è	Ü	¬	║	╩	┌	Ω	•
-B	♂	←	+	;	K	[	k	{		ï	¢	½	╗	╦	■	δ	√
-C	♀	∟	,	<	L	\	l	|		î	£	¼	╝	╠	▬	∞	ⁿ
-D	♪	↔	-	=	M	]	m	}		ì	¥	¡	╜	═	▌	ϕ	²
-E	♫	▲	.	>	N	^	n	~		Ä	Pt	«	╛	╬	▐	ε	■
-F	☼	▼	/	?	O	_	o	⌂		Å	ƒ	»	┐	╧	▀	∩	

B) A Turbo Pascal menü

L Logged drive

Új meghajtó kijelölése a. A "New drive:" utáni válasz általában A, B vagy C (winchester lemez).

A Active directory

A könyvtár kijelölése. Pl.: TURBO. Ha a használt floppyn nincsenek könyvtárak, nem kell kijelölni.

W Work file

Ha az adott névvel már van az aktív könyvtárban állomány, akkor a szerkesztési területre betöltődik.
Egyébként a készítendő program vagy programrészlet nevét adjuk meg a "Work file name:" kérdés után. Pl.: feliras. Ha kiterjesztést nem adunk meg, automatikusan PAS lesz.
Ha módosítjuk a programot, a korábbi változat BAK kiterjesztéssel megmarad.
Ha a file-név után potot teszünk (feliras.), nem kap kiterjesztést.

M Main file

Ha a munkaállományt include állománynak szánjuk, megadhatjuk - az előzőleg megírt - meghajtó program nevét. Ez a main file. Ekkor a fordítás előtt a work file automatikusan lemezre íródik és a main file fordítódik le.

E Edit

Belépés a szerkesztési üzemmódba. Programot írhatunk, vagy meglévőt módosíthatunk.

C Compile

A szerkesztési területen lévő forrásprogram fordítása. Ha nincs main file, akkor a work file lesz lefordítva. A fordítást a compiler options kijelölés befolyásolja.

R Run

A run menüponttal két dolgot is elérhetünk: ha nem volt még lefordítva a forrásprogram, akkor lefordítódik és (mindenképpen) végrehajtódik.

S Save

A szerkesztő területén lévő programot a work file menüpontben megadott néven (+ esetleg a PAS kiterjesztés) lemezre írja.

D Dir

Információ kérés a tartalomjegyzékből. Az összes, vagy ha maszkot is megadunk, bizonyos állományok neve a képernyőre íródik. Ha a maszk *.BAK, akkor pl. csak a BAK kiterjesztésű állományoké.

Q Quit
Kilépés a Turbo Pascal rendszerből, visszatérés a DOS-ba.

O compiler Options

A compiler opciók almenüjét hívja a képernyőre. Itt csak kijelöljük a kívánt változatot, majd visszatérünk a főmenübe. A C menüponttal elindított fordítás során valósul meg. Az almenüből választható menüpontok (M az alapértelmezés):

M Memory

A fordítóprogram a tárba fordítja a a forrásprogramot. Ezután az R menüponttal futtatható.

C Com-file

A fordítás eredménye a lemezre íródik COM kiterjesztéssel. A felir.pas nevű állományból a fordítás eredményeképpen a felir.com tárgyprogramot kapjuk. Ez a program a DOS alatt a felir nev beírása után végrehajtódik.
Amikor a C menüpontot választjuk, további menüsorok íródnak a képernyőre.

O minimum cOde segment size

A chn file-ok a tárba a com file helyére töltődik be. Szükség esetén a megnagyobb állomány méretét állíthatjuk be. A tárigényt "paragrafusban" kell megadni. 1 paragrafus=16 bájt.

D minimum Data segment

Ez az érték nem lehet kisebb a com és a chn állományok maximális adatszegmensénél.

I mInimum free dinamic memory

A verem és a halom együttes mérete. Ez az érték nem lehet kisebb, mint a com és a hozzá láncolt chn állományok maximális igénye.

A mAximum free dinamic memory

Csak többfelhasználós rendszerben kell megadni ezt az információt.

H cHn-file

A fordítás során CHN kiterjesztésű lemezes állományt kapunk. Ezek az állományok önállóan nem hajthatók végre, csak más programhoz láncolhatók a chain utasítással.
Ugyanazok a menüpontok itt is megjelennek, mint a C menüpont után, de itt nem kell semmit sem megváltoztatni, mert úgyis a com file (mint gyökérszegmens) határozza meg a rendelkezésre álló tárterületeket.

P command line Parameters

Turbo Pascal programokat úgy is megírhatunk, hogy a parancssorból paraméter értékeket olvassanak be. Pl. ha már a felir.com állomány rendelkezésre áll, akkor írhatnánk:

```
felir a
```

ahol "a" egy paraméter. Ha a programot a Turbo Pascal rendszerből futtatjuk, akkor az "a" paramétert a P menüponttal adhatjuk meg.

F Find run time error

Ha a lefordított programunkat DOS alatt parancsként hajtottuk végre és hibaüzenetet kaptunk, akkor a hiba helye a tárgyprogramra vonatkozólag pl. PC=2AD8 alakban íródik ki. Ennek a helynek megfelelő forrásprogrambeli utasítást a következőképpen kereshetjük meg:

indítsuk el a Turbo Pascalt,
válasszuk az O menüpontot,
az almenüből válasszuk az F menüpontot,
írjuk be a 2AD8 értéket.

A hiba helyét a forrásprogramban a rendszer meghatározza.

Q Quit

Visszatérés a főmenübe.

C) Az editor parancsok

A billentyűzeten található speciális karakterekkel (helyőr mozgató billentyűk, page up és page down billentyűk stb.) itt nem foglalkozunk.
Az editor parancsok alapértelmezését adjuk meg. Tudnunk kell azonban, hogy a disztributív lemezen (ami a boltban kapható) található TINST.COM installációs programmal az alapértelmezést meg lehet változtatni.
A következő felsorolásban minden billentyűt a [CTRL] billentyűvel együtt kell használni, így ezt a továbbiakban külön nem jelöljük. A kettős betűket egymás után ütjük le, míg a [CTRL] billentyűt lenyomva tartjuk.

Helyőr mozgató parancsok

S Egy pozícióval balra

lép a helyőr, az ott lévő karaktert nem módosítja. A mozgás a képernyő bal szélén megáll.

D Egy hellyel jobbra

mozgatja a helyőrt. a szöveget nem módosítja. a mozgás a képernyő jobb szélén megáll, de a képernyő tartalma balra gördül, amíg a helyőr a 128-adik pozícióra nem kerül, ahol megáll.

A Egy szóval balra

A helyőr a tőle balra lévő szó első karakterére ugrik. Szónak számít bármely összefüggő jelsorozat, amelyet a következő karakterek fognak közre:

szóköz < > , . ; ( ) [ ] ^ ' * + - / $

A helyőr az előző sorba is átléphet.

F Egy szóval jobbra

A helyőr tőle jobbra lévő szó elejére ugrik. A következő sorba is átléphet.

E Egy sorral feljebb

lép a helyőr, de ugyanabban az oszlopban marad. Ha a képernyő legfelső sorában állt, akkor a képernyő tartalma egy sorral lefelé gördül.

Egy sorral lejjebb

lép a helyőr. Ha a legalsó sorban volt, a képernyő tartalma egy sorral felfelé gördül.

W Lefelé gördítés

A képernyő tartalma lefelé gördül, vagyis az állományban visszafelé (az eleje felé) haladunk. A helyőr az alsó sorban megáll.

Z Felfelé gördítés

A képernyő tartalma felfelé gördül, azaz az állomány vége felé haladunk. A helyőr a képernyő tetején megáll.

R Visszalapozás

Az előző képernyőtartalom íródik a képernyőre, a legfelső sor a képernyő legalsó sora lesz (page up).

C Továbblapozás

A következő képernyőtartalom íródik a képernyőre, a legalsó sor a képernyő tetejére kerül (page down).

Q S A bal lapszélre

ugrik a helyőr, ugyanabban a sorban maradva.

Q D A jobb lapszélre
ugrik a helyőr, ugyanabban a sorban maradva.

Q E A képernyő tetejére

mozgatja a helyőrt.

Q X A képernyő aljára

mozgatja a helyőrt.

Q R Az állomány elejére

ugrik a helyőr. A képernyőn a szöveg első oldalát látjuk, a helyőr az első karakteren áll.

Q C Az állomány végére

áll a helyőr, az utolsó oldal utolsó karakterére.

Q B Blokk kezdetére

ugrik a helyőr. A parancs akkor is hatásos, ha a blokkot előzőleg elrejtettük és akkor is, ha a blokk végét még nem jelöltük ki.

Q K Blokk végére

mozgatja a helyőrt. A parancs akkor is hatásos, ha a blokkot elrejtettük és akkor is, ha a blokk kezdetét még nem jelöltük ki.

Q P Visszaugrás

a helyőr legutóbbi pozíciójára (pl. keresés és helyettesítés után).

Beszúrás és törlés

V Insert mód be- és kikapcsolás

Beszúrásos módból felülírásos (overwrite) módba kapcsolhatunk át és viszont. Az érvényes mód a képernyő tetején a státusz sorban látszik.
Insert módban a leütött karakter a helyőr pozíciójára íródik, a tőle jobbra lévő karakterek jobbra lépnek.
Overwrite módban a helyőr alatti jelet a leütött billenytűvel felülírjuk.

DEL Törlés balra

A helyőrtől balra lévő jelet a [DELETE] billentyűvel törölhetjük. A helyőrtől jobbra lévő karakterek egy hellyel balra lépnek. A [CTRL]-t itt nem kell használni. A sor elején lenyomott DEL az előző sor utolsó jelét törli.

G Aktuális karakter törlése

A helyőr pozícióján lévő jel törlése. A jobbra lévő karakterek egy hellyel balra lépnek. A parancs nem hat a következő sorra.

T Szó törlése

A helyőrtől jobbra lévő szó törlődik (a szó fogalmának meghatározását lásd az A parancsnál). A parancs hatása a következő sorra is kiterjed.

N Sor beszórása

A helyőr pozíciójánál egy sort beszórhatunk. A helyőr nem mozdul el.

Y Sor törlése

A helyőrt tartalmazó sor törlődik és az alatta lévő sorok eggyel feljebb lépnek. A helyőr a képernyő bal szélére ugrik.

Q Y Törlés a sor végéig

A helyőr pozíciójától a sor végéig terjedő teljes szöveget törli.

Blokk parancsok

K B Blokk kezdet kijelölése

A helyőr aktuális pozíciója lesz a blokk kezdőpontja. A képernyőn semmiféle vizuális hatása nincs a parancsnak a blokk kezdőpontját illetően, de ha a blokk végét előzőleg már kijelöltük (képernyőtől függően) a teljes blokk szövege más tónusban (vagy más színnel) látszik.

K K Blokk végének kijelölése

Az aktuális helyőr pozíció lesz a blokk vége. A parancsnak nincs vizuális hatása a blokk végét illetően, de ha a blokk elejét előzőleg kijelöltük, akkor (képernyő típustól függően) a teljes blokk más tónusban látszik.

K T Egy szó kijelölése

Ha a helyőr egy szón van, ez a szó lesz a kijelölt blokk, ha nem, akkor a tőle balra lévő. A szó fogalmának meghatározását lásd az A parancsnál.
Ez a parancs blokk kezdetet és blokk véget is kijelöl.

K H Blokk elrejtése

E parancs hatására a blokkot megmutató eltérő szövegtónus kikapcsolható és bekapcsolható. A blokkal műveletet végző parancsok csak akkor hatásosak, ha a blokk látható.

K C Blokk másolás

Ezzel a paranccsal a kijelölt blokkot a helyőr aktuális pozíciójától kezdődően lemásolhatjuk. Az eredeti blokk változatlan marad. Az átmásolt blokkot eltérő tónus jelöli. Ha nem volt érvényes blokk kijelölés, a parancs hatástalan és hibajelzést sem kapunk.

K V Blokk mozgatása

Egy előzőleg kijelölt blokkot az aktuális helyőr pocíción kezdődően viszünk át. Az eredeti helyéről a blokk eltűnik, új helyén az eltérő tónus jelöli. Ha nem volt kijelölt blokk, a parancs hatástalan.

K Y Blokk törlése

Az előzőleg kijelölt blokkot törli. A blokk tartalma sehol nem őrződik meg, a téves törlés esetén az eredeti állapotot nem tudjuk visszaállítani. Elővigyázatosan használjuk!

K R Blokk olvasás lemezről

Az aktuális helyőr pozíciótól kezdődően a mágneslemezről másolhatunk be egy szövegállományt. A beolvasott szöveg blokk kijelölésű lesz (beleértve a vizuális jelölést is). A parancs végrehajtásához be kell írnia (a kapott prompt után) az állomány nevét.

K W Blokk lemezre írása

Egy előzőleg kijelölt blokkot lemezre ír. A blokk változatlan marad, tónusa is. A parancs végrehajtásakor meg kell adni (a kapott prompt után) az állománynevet. Ha ilyen nevű állomány már létezik, figyelmeztetést kapunk, így annak felülírását elkerülhetjük.
Ha nem volt érvényes blokk kijelölés, a parancs hatástalan, hibajelzést sem kapunk.

Egyéb szerkesztőparancsok

K D A szerkesztés vége

Kilépünk az Edit módból és visszatérünk a Turbo Pascal menühöz. A kilépéskor a szerkesztett munkaállomány nem mentődik automatikusan lemezre. Kilépés után az S menüpont (Save) használatát javasoljuk.

I Tabulátor

A Turbo Editorban nincsenek rögzített tabulátor pozíciók. A tabulátor pozíció automatikusan annak a szónak a kezdete, amelyiken a helyőr áll. Ez különösen kényelmes program íráskor, pl. ha egymás alá akarjuk írni a deklarált változók azonosítóját.

Q I Automatikus tabulácio

Ezzel a paranccsal az automatikus tabulálást kapcsolhatjuk be és ki. Alapértelmezés szerint bekapcsolt állapotban van, amit a státusz sorban az Indent szó jelez. Aktív állapotában mindig az előző sor első jele az érvényes tabulátor pozíció.

Q L Sor visszaállítása

Ha megbántuk, hogy egy sort módosítottunk - és a helyőr még ebben a sorban van - ezzel a paranccsal az eredeti állapotot állíthatjuk vissza. Ezért az Y parancs (sor törlése) után sajnos nem működik.

Q F Keresés

Ezzel a paranccsal legfeljebb 30 jel hosszú részszövegek előfordulásait kereshetjük meg. A kereső szöveget a státusz sorban kapott prompt után kell megadnunk. A részszöveg bármilyen karaktert tartalmazhat, kontroll karaktereket is.(A kontroll karaktereket a P prefixum után - lásd ott - írhatjuk be. A kocsivissza-soremelés jeleknek pl. a [CTRL] M - [CTRL] J karakterek félelnek meg.) A [CTRL] A kontroll karakternek speciális szerepe van, bármely jelet helyettesíthet.
A kereső szöveget az S, D, A és F helyőr mozgató parancsokkal szerkeszthetjük. Az F hatására a megelőző kereső szöveget kapjuk vissza (ha ilyen van). A keresés a [CTRL] U-val megszakítható.
Miután a kereső szöveget beírtuk, keresési opciókat írhatunk elő. A következők közül választhatunk:

B Keresés visszafelé, vagyis az aktuális helyőr pozíciótól a szöveg eleje felé.

G Globális keresés: keresés a teljes szövegben, függetlenül a helyőr pozíciójától.

n n=tetszőleges egész szám. A kereső szövegnek az aktuális helyőr pozíciótól számított n-edik előfordulását keresi meg.

U Nem tesz különbséget kis- és nagybetűk között.

W Teljes szót keres csak. Ha a szó más szó belsejében fordul elő, nem veszi figyelembe (pl. "baj", "abajgat").

Megadhatunk egyidejűleg több opciót is, pl.: GUW.
A keresés során a megtalált szöveg utolsó jelére kerül a helyőr. A kereső műveletet az L paranccsal megismételhetjük.

Q A Keresés és helyettesítés

Ezzel a paranccsal legfeljebb 30 jel hosszó részszövegek elővordulásait kereshetjük meg és cserélhetjük ki egy legfeljebb 30 jel hosszóságó másik szöveggel. A kereső szöveget a státusz sorban kapott prompt után kell megadnunk. A részszöveg bármilyen karaktert tartalmazhat, kontroll karaktereket is.(A kontroll karaktereket a P prefixum után - lásd ott - írhatjuk be. A kocsivissza-soremelés jeleknek pl. a [CTRL] M - [CTRL] J karakterek felelnek meg.) A [CTRL] A kontroll karakternek speciális szerepe van, bármely jelet helyettesíthet.
A kereső szöveget az S, D, A és F helyőr mozgató parancsokkal szerkeszthetjük. Ay F hatására a megelőző kereső szöveget kapjuk vissza (ha ilyen van). A keresést a [CTRL] U-val megszakíthatjuk.
A kereső szöveg megadása után újabb prompt kéri a helyettesítő szöveget. Ennek a beírását ugyanúgy végezhetjük el, mint a kereső szövegét - azzal a különbséggel, hogy a [CTRL] A-nak nincs különleges jelentősége.
Miután a helyettesítő szöveget beírtuk, keresési és helyettesítési opciókat írhatunk elő:

B Keresés és helyettesítés visszafelé, vagyis az aktuális helyőr pozíciótól a szöveg eleje felé.

G Globális keresés és helyettesítés: keresés és helyettesítés a teljes szövegben, függetlenül a helyőr pozíciójától.

n n=tetszőleges egész szám. A kereső szövegnek az aktuális helyőr pozíciótól számított n-edik előfordulását keresi meg és helyettesíti.

N Helyettesítés rákérdezés nélkül. Nem áll meg a kereső szöveg minden megtalált előfordulásánál és nem kérdez ra: Replace (Y/N).

U Nem tesz különbséget kis- és nagybetűk között.

W Teljes szót keres és helyettesít csak. Ha a szó más szó belsejében fordul elő, nem veszi figyelembe.

Megadhatunk egyidejűleg több opciót is, pl.: GUW.
A művelet végrehajtása során a megtalált szöveg utolsó jelére kerül a helyőr és - hacsak nem az N opciót választottuk - válaszolnunk kell a Replace (I/N)? kérdésre. Ezen a ponton megszakíthatjuk a műveletet a [CTRL] U paranccsal.
A keresés és helyettesítés műveletét az L paranccsal megismételhetjük.

L A keresés ismétlése

Az utoljára meghatározott keresési vagy keresési és helyettesítési műveletet újra végrehajtja.

P Kontroll karakter prefixum

Ha a szövegben kontroll karaktereket akarunk elhelyezni, a P prefixumot kell használni. Ha pl. a [CTRL] G karaktert akarjuk beírni, először a [CTRL] P-t, majd csak ezután írjuk a [CTRL] G-t. A kontroll karakterek megkülönböztetett ábrázolásúak (pl. fényesebbek, vagy inverz jelek).

U Kilépés

Bármely megkezdett parancsból kiléphetünk, ha éppen paramétert kér.

D) Compiler direktívák

B Be- és kiviteli mód választás

Alapértelmezés: {$B+}

A B direktíva a bevitel és kivitel módját határozza meg. Bekapcsolt állapotban a CON: logikai készüléket rendeli az Input és Output szabványos file-okhoz. Kikapcsolt állapotban a TRM: készüléket.
A B direktíva globális az egész programra nézve. Az alapértelmezést csak a program elején változtathatjuk meg.
A B direktívával kapcsolatos további részleteket a 11.4 alfejezetben közlünk.

C I/O megszakítás engedélyezése

Alapértelmezés: {$C+}

Ha a C direktíva bekapcsolt állapotban van, a billentyűzetről érkező [CTRL] C és [CTRL] S karakterek sajátos értelmezésűek. Ha beviteli művelet végrehajtása közben lenyomjuk a [CTRL] C-t, a program végrehajtása megszakad, és a

```
User Break, PC=xxxx
Program aborted
```

üzenetet kapjuk.
Output művelet végrehajtásakor a [CTRL] S lenyomásával időlegesen megállíthatjuk a programot, így megakadályozhatjuk, hogy a képernyő sorai kigördüljenek, mielőtt elolvastuk volna. Ha újra lenyomjuk a [CTRL] S-t, a program tovább fut.
Ha a C direktíva passzív, a [CTRL] C és [CTRL] S karakterek különleges értelmezése megszűnik.
A C direktíva a teljes programra nézve globális, ha egyszer beállítottuk, nem változtathatjuk meg. Továbbá csak a programblokk előtt változtathatjuk meg az alapértelmezést.

I I/O hibakezelés

Alapértelmezés: {$I+}

Ha az I direktíva be van kapcsolva, a be- és kiviteli műveletek során észlelt hiba esetén a program végrehajtása megszakad és a következő üzenetet kapjuk:

```
I/O error NN, PC=XXXX
Program aborted
```

NN a hiba okának kódja, XXXX a hiba észlelésének a helye (a lefordított programban).
Az I direktíva kikapcsolása után nem szakad meg a program és nem kapunk hibaüzenetet sem. A programozó az ioresult szabványos függvénnyel maga kezelheti a hibát.
Ezzel kapcsolatos bővebb információt a 11.7 alfejezetben találunk.

R Indextartomány ellenőrzés

Alapértelmezés: {$R-}

Az aktív R direktíva hatására a tárgyprogramba épülnek olyan eljárások, amelyek a program végrehajtása közben a tömbök és stringek indexeit figyelik, hogy nem vesznek e fel a deklarált tartományon kívüli értéket. Pl. ha az array[1..10] of char típussal deklarált tömbváltozó 0-ás vagy 11-es indexű elemére hívatkozunk, a

```
Run-time error 90, PC=XXXX
Program aborted
```

hibajelzést kapjuk és a végrehajtás megszakad.
Indexelési hibák ritkán következnek be megfelelően működő programokban, de a program belövésekor annál gyakoribbak. A program tesztelés időszakában ezért célszerű bekapcsolni az R direktívát, de ha a program elkészült, kapcsoljuk ki. A külön indexvizsgálatokhoz szükséges utasítások jelentősen megnövelik a tárgyprogram méretét, és a végrehajtási idő is megnő.

V Szigoró típusellenőrzés

Alapértelmezés: {$V+}

Ha egy alprogramnak változóparaméterként adunk át string paramétert, szigoró típusellenőrzés történik. Nem adhatunk át pl. egy string[20] típusú változót egy string[30] típusú formális paraméterrel, bár ennek fizikai korlátja nyilvánvalóan nincsen.
Ezt a szigoró típusellenőrzést enyhíthetjük a V direktíva kikapcsolásával. Ezután nem akadályozza meg a Turbo Pascal rendszer, hogy bármilyen típusú string paramétert bármilyen string típusú változó paraméterrel átadjunk (az esetleges információvesztéssel persze számolnunk kell).
Más típus esetén ilyen enyhítésre nincs lehetőség.

U Felhasználói programmegszakítás

Alapértelmezés: {$U-}

Ha az U direktívát bekapcsoljuk, akkor a futó programot bármikor megszakíthatjuk [CTRL] C-vel (nemcsak beviteli művelet végrehajtásakor). Ha a programunkban ciklus van, és még nem próbáltuk ki, jól tesszük, ha bekapcsoljuk ezt a direktívát. Egy esetleges végtelen ciklusból (ami könnyen létrejöhet, ha pl. valós számok egyenlőség vizsgálatával akarjuk megállítani) egyébként csak a rendszer újbóli betöltésével tudunk kilépni. S adjunk hálát Istennek, ha a programunkat előzőleg lemezre mentettük, különben kezdhetjük előről.
Az U direktíva által nyújtott biztonságért nagy ár lehet a program végrehajtási idejének jelentős megnövekedése. Ezért jól működő programoknál nem használjuk.

K Verem ellenőrzés

Alapértelmezés: {$K+}

A verem és a halom közös tárrészen egymással szemben növekvő területek. A K direktíva aktív állapota mellett a fordítóprogram olyan kódot hoz létre, ami figyeli pillanatnyi helyzetüket és ha összeérnek, akkor a

```
Run-time error FF, PC=XXXX
Program aborted
```

hibaüzenetet kapjuk és a program végrehajtása abbamarad. Ez komoly hiba, hiszen alprogramok paraméterei és visszatérési címek mennek tönkre. Ezért a K direktívának bekapcsolt állapotban illik lenni.
Sajnos a többlet utasítások miatt a tárgyprogram terjedelme és végrehajtási ideje is megnő.

I Include direktíva

Az include direktíva nem "kapcsoló" direktíva, mint az előzőek, ezért ne tévesszük össze az I+ és az I- direktívával. Az include állományok megadására használjuk. A programban megadott

```
{$I prog.pas}
```

direktíva hatására a prog.pas forrásnyelvi állomány fordítás közben beépül a tárgyprogramba arra a helyre, ahol a forrásprogramban a direktívát megadtuk.

G Input puffer

Ha a szabványos inputot nem a billentyűzetről kapja a program, hanem átirányítással egy lemez állományból, vagy más programból, akkor beviteli puffert kell meghatározni (a get szóból). Erre használjuk a G direktívát. A G után a puffer méretét kell megadni pl.:

```
{$G256}.
```

P Output puffer

Ha a program nem az Output szabványos file-hoz rendelt kimeneti készüléken írja ki az eredményt, hanem átadja másik programnak, akkor kiviteli puffert kell definiálni. Erre használjuk a P (put) direktívát. A P után a puffer méretét határozhatjuk meg:

```
{$P256}.
```

Ebben a könyvben nem használtunk olyan programozási megoldást, ahol a G vagy a P direktívára szükségünk lett volna. Használatukhoz a DOS mélyebb ismeretére van szükség.

E) Nyelvi alapelemek

Alapjelek

Betűk: A-Z, a-z és _ (aláhúzás jel)
Számjegyek: 0 1 2 3 4 5 6 7 8 9
Speciális jelek: + - * / = ^ < > ( )
[ ] { } . , ; : ' $

összetett szimbólumok

Értékadás: :=
Relációjelek: <> <= >=
Intervallum jel: ..
Indexzárójel([és] helyett használható): (. .)
Kommentár ({ és } helyett): (* *)

A fordítóprogram nem tesz különbséget kis- és nagybetűk között (kivéve stringekbe).
Szövegkonstansokban bármilyen - a billentyűzeten szereplő - jel használható.

Kulcsszavak

A kulcsszavakat csak a nyelv által definiált jelentésükben használhatjuk. Nem definiálhatók újra (fenntartott szavak).

absolute ←	external ←	nil	shr ←
and	file	not	string ←
array	for	of	then
begin	forward	or	to
case	function	packed	type
const	goto	procedure	until
div	if	program	var
do	in	record	while
downto	inline ←	repeat	with
else	label	set	xor ←
end	mod	shl ←	

A nyillal azokat a kulcsszavakat jelöltük meg, amelyek a szabványos Pascalban nem használhatók.

Elhatároló jelek

A nyelv elemeit a következő jelek közül legalább egynek el kell választani: szóköz, sorvége, kommentár, vessző, pontosvessző, kettőspont.

Szabványos azonosítók

A szabványos azonosítókat a felhasználó újra definiálhatja. Ekkor természetesen az eredeti funkció elvész.

Előre definiált változók: maxint (=32767)
pi (=3.1415926535)

Előre definiált eljárások és függvények: (az I) Mellékletben részletesebben tárgyaljuk).

Azonosítók

Betűvel kezdődő, betűkből és számjegyekből és az aláhúzás jelből álló jelsorozatok.

Számok

Egész számok tízes számrendszerben (128, -87)

Hexadecimális számok: $-jel után hexadecimális számjegyekből: 0-9, A, B, C, D, E, F ($3A0C).

Valós számok tizedestört alakban (0.87, -49.0)

Valós számok kitevős alakban (1.2E-3, -6E8)

Füzérek

Felsővesszők között megadott karaktersorozatok ('Turbo Pascal').

Kommentárok

Kapcsos zárójelek között megadott szövegek (ez egy magyarazat).

F) A program felépítése

Főprogram

```
PROGRAM név;
deklarációk;
BEGIN
utasítások
END.
```

Deklarációk

Tetszőleges számban és tetszőleges sorredben ismételhetők:

Címkék

```
label címke1, címke2, ...;
```

A címke azonosító vagy egész szám (ELEJE, 99).

Konstansok

```
const azonosító1=érték1;
      azonosító2=érték2;
      ..................
```

A típust az érték implicite meghatározza.

A const deklarációval kezdőértéket is meghatározhatunk:

```
const azonosító:típus=érték;
```

Típusok

```
type azonosító1=típus1;
     azonosító2=típus2;
     ..................
```

A deklarált típus azonosítók típusként használhatók.

Szabványos típus azonosítók: INTEGER,
REAL,
CHAR,
BOOLEAN.

Egyéb egyszerű típusok:

Felsorolási: szimbólumok azonosítói zárójelben felsorolva:
(hetfo,kedd,szerda,csutortok,pentek,szombat,vasarnap).

Diszkrét típusok: az egyszerű típusok a REAL nélkül.

Intervallum típusok: diszkrét típus intervalluma (pl.: 1000..3000, hetfo..pentek, 'a'..'z').

Eljárások

```
PROCEDURE azonosító(formális paraméterek);
deklarációk
BEGIN
utasítások
END;
```

Függvények

```
FUNCTION azonosító(formális paraméterek):típus;
deklarációk
BEGIN
utasítások
END;
```

Utasítások

Értékadás: változó:=kifejezés

A kifejezés: konstans, változó, ill. konstansokból, változókból műveleti jelekből, függvényekből és szárójelekből álló képlet.

összetett utasítások

Az összetett utasítások egyetlen utasításnak számítanak.

Utasítás sorozat

```
BEGIN
utasítás1;
utasítás2;
.........
END
```

Egyágó kiválasztás: IF logikai kifejezés THEN utasítás

Kétágó kiválasztás: IF logikai kifejezés THEN utasítás1
ELSE utasítás2

Többágó kiválasztás:

```
CASE diszkrét kifejezés OF
  konstansok1:utasítás1;
  ......................
  ELSE: utasításn; (elmaradhat) END
```

Ismétlési szerkezetek

```
REPEAT      {ismétel mígnem igaz lesz}
utasítás1;
.........
UNTIL boolean kifejezés

WHILE boolean kifejezés do
  utasítás {ismétel, amig igaz}

FOR diszkr. változó := kezdőért. { TO / DOWNTO } végérték DO
  utasítás
```

összetett típusok

Stringtípusok

Maximált hosszúságú füzérek.

Deklaráció: type strtip=STRING[max.meret] {max.meret<=225}

A stringek elemeit indexeléssel érhetjük el: s[5] az s string 5-ödik betűje.

Kezdőértékadás stringre pl.: s:string[8]='akarmi'

Tömbtípusok

Meghatározott számú azonos típusú elem összessége.

Deklaráció: type tömbnév=ARRAY[indextip1, ...] OF bázistipus

Az indextípusok diszkrét típusok lehetnek. Ha egy indextípust adunk meg, a tömb egydimenziós. A bázistípus tetszőleges, csak file típus nem lehet.

Az elemeket indexeléssel érjük el: x[i], a[j,k].

Kezdőértékadás: x:array[1..4] of integer=(3,0,-1,8)
a:array[1..2,1..2] of integer=((2,0),(-3,5))

Halmazok

Deklaráció: type htip=SET OF bázistípus

A bázistípus diszkrét típus lehet, számossága legfeljebb

256. Max. rendszám<=256. A halmaz megadása: az elemeket szögletes zárójelben felsoroljuk: [3,7,9].
[] az üres halmaz.
Kezdőértékadás: h:set of 1..33 =[10,15,20]

Rekord típusok

Véges számú, nem szükségképpen azonos típusú érték összessége.

```
Deklaráció: RECORD
              mező1:típus1;
              .............
(elmaradhat)  CASE változó:típus OF
                konstansok1:(meződef1);
                ........................
              END
```

Mezőtípus bármi lehet (rekord is), kivéve: file.

Hozzáférés a mezőkhöz: rekordnév.mezőnév

H WITH utasítást használunk, nem kell mindig kiírni a rekordnevet:

```
     WITH rekordnév do
      utasítás
```

Az utasitásban elegendő a mezőnevet használni.

Kezdőértékadás:

```
    type rtipus=record
                  m1:integer;
                  m2:char;
                  end;

   const r:rtipus=(m1:4,m2:'h')
```

File típusok

Nem meghatározott számú azonos típusú elem összessége.

Deklaráció: file of bázistípus
Bázistípus bármi lehet, kivéve file.

Szabványos file típus: text (=file of char).

A szabványos file-ok az I/O készülékekre vonatkoznak. Az elsődleges I/O file-ok az INPUT és az OUTPUT.

Pointer típusok

Deklaráció: ^típusnév

pointer konstans: nil.

G) Müveletek

TÍPUS		MÖVELETEK
INTEGER	:=	+ - * div mod or and xor shl shr / = <> < <= >= >
REAL	:=	+ - * / = <> < <= >= >
CHAR	:=	= <> < <= >= >
BOOLEAN	:=	not and or xor = <> <= >=
felsorolási	:=	= <> < <= >= >
string	:=	+ = <> <= >= >
tömb	:=	= <>
rekord	:=	= <>
halmaz	:=	+ - * = <> <='>= in
file		
pointer	:=	= <>

H) Eljárások és függvények

ABS

```
function abs(r:real):real
function abs(i:integer):integer
```

A paraméter abszolút értékét adja vissza.

ADDR

```
function addr(var változó):pointer
```

A változó címét adja vissza. A cím 4 bájtos pointer, szegmens és offszet részből áll.

APPEND

```
procedure append(var f:text)
```

Az append eljárás megnyit egy text file-t írásra és a file mutatót a file végére állítja.

ARCTAN

```
function arctan(r:real):real
```

Az átadott paraméter ívmértékben adott arkusztangesét adja vissza.

ASSIGN

```
procedure assign(var f:file;nev:string)
```

Az f file-azonosítóhoz hozzárendeli a nev állományt.

BLOCKREAD

```
procedure blockread(var f: file;
                    var b: típus;
                    rekszam:integer;
                    var olvrek:integer)
```

Az eljárás megkísérli rekszam db. rekord olvasását az f típus nélküli file-ból a b pufferba. olvrek mutatja a ténylegesen beolvasott rekordok számát. Az olvrek paramétert a 3.0-ás verzióban nem kötelező használni.

BLOCKWRITE

```
procedure blockwrite(var f: file;
                     var b: típus;
                     rekszam:integer)
```

Az eljárás a b pufferből rekszam db. rekordot ír az f típus nélküli file-ba.

CHAIN

procedure chain(f:file)

A chn kiterjesztésű program állomány végrehajtása egy com vagy másik chn kiterjesztésű programból. A lemezen található állománynevet előzőleg f-hez kell kapcsolni egy assign eljárással.

CHDIR

procedure chdir(s:string)

Az aktuális könyvtárat megváltoztatja. Az új könyvtárat az s stringgel határozzuk meg.

CHR vagy CHAR

function chr(i:integer):char
function char(i:integer):char

A függvény azt a karaktert adja vissza, amelynek a kódja i. i értékének 0 és 255 között kell lenni.

CLOSE

procedure close(var f:file)

Üríti az f file pufferét, majd lezárja a file-t.

CLREOL

procedure clreol

Az eljárás az aktuális képernyő sort törli a helyőr pozíciójától a sor végéig.

CLRSCR

procedure clrscr

Az eljárás törli a képernyő tartalmát és a helyőrt a bal felső sarokra, az (1,1) pozícióra állítja.

CONCAT

function concat(s1,s2,...,sn):string

Tetszőleges számú stringet egyetlen stringé fűz össze és ezt az egyesített stringet adja vissza. Ha az eredmény string hossza 255-nél nagyobb, futás közbeni hibát kapunk.

COPY

```
function copy(s:string;p,h:integer):string
```

A függvény értéke az s paraméterben adott string p-edik pozíción kezdődő h hosszúságú részstringje.

COS

```
function cos(r:real):real
```

Az ívmértékben megadott r szög koszínusza.

CRTEXIT

```
procedure crtexit
```

Az eljárás terminál reset stringet küld a képernyőre.

CRTINIT

```
procedure crtinit
```

Az eljárás képernyő inicializáló stringet küld a képernyőre.

DELAY

```
procedure delay(i:integer)
```

Az eljárás a program végrehajtását kb. i msec időtartamra felfüggeszti.

DELETE

```
procedure delete(var s:string;p,h:integer)
```

Az eljárás törli az s stringből a p pozíción kezdődő h hosszúságú részstringet.

DELLINE

```
procedure delline
```

Az eljárás törli a képernyőnek azt a sorát amelyben a helyőr van. Az alatta lévő sorok eggyel feljebb gördülnek.

DISPOSE

```
procedure dispose(p:pointer)
```

Az eljárással felszabadíthatjuk a halomtár p mutatóhoz rendelt területét. A felszabadított tárrészt újra felhasználhatjuk.

EOF

```
function eof(f:file):boolean
```

Eof a true logikai értéket adja vissza, ha a file mutató a file végén áll. Egyébként false.

EOLN

```
function eoln(f:file):boolean
```

A predikátum értéke true, ha a file mutató sor végére (amit a 10-es 13-mas kódú karakterpár jelez) vagy a file végére ér.

ERASE

```
procedure erase(f:file)
```

Az eljárás törli az f file-t, azaz törli a lemezről a hozzá rendelt állományt.

EXECUTE

```
procedure execute(f:file)
```

Elindítja egy com kiterjesztésű program file végrehajtását. A programállományt előzőleg f-hez kell rendelni egy assign eljárással.

EXIT

```
procedure exit
```

Kilépést okoz az aktuális blokkból, azaz visszatérést az alprogramot hívó programba.

EXP

```
function exp(r:real):real
```

A függvény értéke az e (=2.71...) szám r kitevős hatványa.

FILEPOS

```
function filepos(f:file):integer
```

Visszaadja az f file azon elemének a sorszámát, amelyre a file mutató mutat. A file első elemének 0 a sorszáma.

FILESIZE

```
function filesize(f:file):integer
```

A függvény értéke a file elemeinek száma.

FILLCHAR

```
procedure file-char(változó:típus;i,kód:integer)
```

A paraméterként átadott változó címétől kezdődően feltölti a tár i db. bájtját a kód értékével.

FRAC

```
function frac(r:real):real
```

Az r valós szám törtrészét adja vissza a függvény.

FREEMEM

```
procedure freemem(var p:pointer;i:integer)
```

Az eljárás felszabadítja a halomtárban a p mutatóhoz kapcsolt i db bájtot.

GETDIR

```
procedure getdir(i:integer;var s:string)
```

A getdir eljárással a programból is hozzáférhetünk a mágneslemezen tárolt tartalomjegyzékhez. Az i változóval a meghajtót jelöljük ki (ha 0 az értéke akkor az alapértelmezés érvényes). s-ben kapjuk a tartalomjegyzéket.

GETMEM

```
procedure getmem(var p:pointer;i:integer)
```

Lefoglal i bájtot a halomban és a p mutatóhoz kapcsolja. A freemem párja.

GOTOXY

```
procedure gotoxy(x,y:integer)
```

A helyőrt a képernyő (x,y) koordinátákkal meghatározott pontjára állítja.

HALT

```
procedure halt
```

Megállítja a programot.

HI

```
function hi(i:integer):byte
```

A függvény az i integer érték felső (high-order) bájtját adja vissza.

HIGHVIDEO

```
procedure highvideo
```

Az eljárás a képernyő nagyobb fényerejét kapcsolja be. A meghívása után kiírt jelek fényesebbek lesznek.

INSERT

```
procedure insert(r:string;var s:string;i:integer)
```

Az r stringet beszúrja az s stringbe az i-edik pozíciótól kezdődően.

INSLINE

```
procedure insline
```

A helyőr által kijelölt sort és az alatta lévő sorokat eggyel lejjebb gördíti. Így a helyőr pozícióján egy üres sort hoz létre.

INT

```
function int(r:real):integer
```

Az r valós szám egész része a függvényérték.

INTR

```
procedure intr(i:integer;reg:rekord);
```

A függvény az i paraméterben adott megszakítás kódnak megfelelő ROM BIOS rutint hívja meg. A paramétereket a reg (regiszterek) rekordon keresztül adhatjuk át. Használata mélyebb hardver ismeretet kíván.

IORESULT

```
function ioresult:integer
```

Minden be- és kiviteli művelet végrehajtásakor egy értéket kap az ioresult függvény. Ha a művelet során nem volt hiba, akkor 0 ez az érték, egyébként a hibára jellemző érték. A függvény értékét csak akkor tudjuk a programból lekérdezni, ha a {$I-} compiler opció érvényes.

KEYPRESSED

function keypressed:boolean

A predikátum a true értéket adja vissza, ha lenyomunk egy billentyűt. A C compiler direktívának passzívnak kell lenni ({$C-}).

LENGTH

function length(s:string):integer

A függvény az s string tényleges hosszát adja vissza (nem a deklarált hosszt).

LN

function ln(r:real):real

Az r valós szám természetes (e alapú) logaritmusa a függvényérték. Ha r negatív vagy 0, hibajelzést kapunk.

LO

function lo(i:integer):byte

Az i integer érték alsó (low-order) bájtja.

LONGFILEPOS

function longfilepos(var f:file):real

Ha a file 32767-nél több elemet tartalmaz, akkor az egész értékű filepos függvényt nem tudjuk használni. Helyette a longfilepos adja meg a file mutató aktuális helyzetét. A függvény értéke egész értékű valós szám.

LONGFILESIZE

function longfilesize(var f:file):real

A file elemeinek a száma. Ha ez a szám nagyobb 32767-nél, akkor a filesize helyett ezt a függvényt használjuk.

LONGSEEK

procedure longseek(var f:file;r:real)

Az eljárás az f file r sorszámú elemére állítja a file mutatót. Ha a file hosszabb 32767-nél, a seek helyett ezt az eljárást használjuk. r egész értékű valós szám.

LOWVIDEO

procedure lowvideo

A képernyő alacsony fényerejét kapcsolja be. Az eljárás meghívása után képernyőre írt jelek halványabb tónusban látszanak.

MARK

procedure mark(p:pointer)

Az eljárással a halomból a p mutatóhoz kapcsolhatunk egy tárterületet.

MAXAVAIL

function maxavail:integer

A dinamikus tár (halom és verem) legnagyobb összefüggő szabad területének a méretét adja vissza a függvény, paragrafusban mérve. 1 paragrafus = 16 bájt.

MEMAVAIL

function memavail:integer

A dinamikus tár teljes szabad tárterületét adja meg paragrafusban.

MKDIR

procedure mkdir(s:string)

Az eljárással egy könyvtárat hozhatunk létre. A könyvtár nevét az s stringgel kell megadni.

MOVE

procedure move(var v1,v2:típus;i:integer)

A move eljárás a v1 változó címétől kezdve i db. bájtot másol át a tárban a v2 változó címénél kezdődő tárterületre.

MSDOS

procedure msdos(reg:rekord)

Az eljárással az operációs rendszer rutinjait hívhatjuk meg. A paraméterek (regiszterek) értéke határozza meg a meghívott rutint. Az eljárás használatához az MS DOS mélyebb ismeretére van szükség.

NEW

procedure new(var p:pointer)

A halomból a p mutatóhoz rendelhetünk tárterületet. A dispose eljárással együtt használható.

NORMVIDEO

procedure normvideo

A képernyő normál intenzitását állítja be. A meghívása után kiírt jelek normál fényerejűek lesznek.

NOSOUND

procedure nosound

Az előzőleg generált hangot kikapcsolja.

ODD

function odd(i:integer):boolean

A függvény értéke akkor és csak akkor true, ha az i egész szám páratlan.

OFS

function ofs(változó, eljárás vagy függvény):integer

A függvény paramétere egy deklarált változó, eljárás vagy függvény azonosítója. A függvény értéke az illető objektum tárbeli címének offsetje (eltolási címe).

ORD

function ord(s:diszkrét típus):integer

A paraméterként megadott diszkrét típusú érték rendszámát adja vissza.

PARAMCOUNT

function paramcount:integer

A függvény a parancssorban megadott paraméterek számát adja meg.

PARAMSTR

function paramstr(i:integer):string

Ez á függvény a parancssorban megadott paramétereket adja

vissza. i a paraméter sorszáma (a megadás sorrendjében). A paramstr(1) hívás pl. az első paraméter értékét adja.

POS

function pos(r,s:string):integer

Ha az r string nem fordul elő az s stringben részstringként, akkor a függvény értéke 0. Egyébként az első előfordulás kezdőpozíciója.

PRED

function pred(d:diszkrét típus):diszkrét típus

A függvény a paraméterként megadott diszkrét típusú érték megelőzőjét adja vissza. Ha nincs megelőző, akkor definiálatlan.

PTR

function ptr(i,j:integer):pointer

Az i paraméter értékét szegmenscímnek, a j-t offsetnek tekintve visszaadja az ennek megfelelő (az adott tárcímre mutató) pointer értéket.

RANDOM

function random(i:integer):integer
function random:real

Álvéletlen számokat képezhetünk ezzel a függvénnyel. Ha nem adunk meg paramétert, akkor a visszaadott érték 0 és 1 közötti valós (0-t beleértve 1-et nem). Ha egész paramétert használunk, akkor 0 és i-1 közötti a függvényérték (beleértve a határokat is).

RANDOMIZE

procedure randomize

A véletlenszám generátor inicializálása. A random függvény használata előtt célszerű meghívni.

READ és READLN

procedure read[ln]([f:file,]paraméterek)

Az eljárások az f file-ból vagy (ha a [,]jelek közötti file paramétert elhagyjuk) a szabványos input egységről olvassa be a paraméterként megadott változók értékét. A readln csak text típusú állományoknál használható, de a file mutatót a következő sor elejére állítja.

RELEASE

```
procedure release(var p:pointer)
```

Ha a p mutatóhoz előzőleg a mark eljárással rendeltünk tárterületet, a p feletti teljes tárrészt visszaadja a halomnak.

RENAME

```
procedure rename(var f:file;s:string)
```

Az f file-hoz rendelt állomány nevét az s-ben megadott névre változtatja.

RESET

```
procedure reset(var f:file)
procedure reset(var f:file;i:integer)
```

Megnyitja az f file-t olvasásra. Típus nélküli file-oknál a rekord hosszát az i paraméterben megadhatjuk.

REWRITE

```
procedure rewrite(var f:file)
procedure rewrite(var f:file;i:integer)
```

Megnyitja az f paraméterben megadott file-t írásra. Ha a file nem létezik, létrehozza. Ha létezik, törli a teljes tartalmát. Ezért létező file-okat folytatólagos írásra a reset eljárással kell megnyitni.
Típus nélküli file-oknál az i paraméterben a rekord hosszát adhatjuk meg.

RMDIR

```
procedure rmdir(s:string)
```

Törli a lemezről a könyvtárat, amelynek nevét az s stringben adtuk meg. Csak üres könyvtárat lehet törölni.

RUOND

```
function round(r:real):integer
```

Az r valós szám egészre kerekített értékét adja vissza a függvény.

SEEK

```
procedure seek(var f:file;i:integer)
```

Az eljárással a file mutatót az f file i sorszámú (azaz

i+1-edik) elemére állíthatjuk. Az eljárás az állomány közvetlen hozzáférésű feldolgozását teszi lehetővé.

SEEKEOF

function seekeof(var f:text):boolean

Ez a text file-oknál használható predikátum a file végét érzékeli, de - legfeljebb 32 szóköz, tabulátor és sorvége jelen átlátva - előre jelzi. Értéke true, ha a file fizikai végéig vagy a ^Z karakterig már nincs több adat.

SEEKEOLN

function seekeoln(var f:text):boolean;

A text file-oknál használható függvény a sor végét már akkor észleli, ha a file mutatót csak szóközök és tabulátor karakterek választják el tőle. Értéke true, ha a sor végéig már nincs több adat.

SEG

function seg(változó, eljárásnév vagy függvénynév):integer

A függvény értéke a változó, függvény vagy eljárás szegmenscíme.

SIN

function sin(r:real):real

Az r szög szinusza a függvény értéke. r dimenzója nem fok, hanem ívmérték.

SIZEOF

function sizeof(var változó):integer

A paraméterként adott változó által elfoglalt tárterület mérete bájtokban kifejezve.

SOUND

procedure sound(i:integer)

Az eljárással az i rezgésszámú hangot szólaltathatjuk meg. A hang egy nosound eljárás végrehajtásáig folyamatosan szól.

SQR

```
function sqr(k:számtípus):számtípus
```

Az integer vagy valós k szám négyzete. A függvényérték típusa k-éval azonos.

SQRT

```
function sqrt(r:real):real
```

Az r valós szám négyzetgyöke. r nem lehet negatív.

STR

```
procedure str(i:integer;var s:string)
procedure str(r:real;var s string)
```

Az eljárás a paraméterként adott egész vagy valós értéket stringgé alakítja és az s stringbe helyezi. A számokat formátumozva is megadhatjuk, mint a write eljárásban.

SUCC

```
function succ(d:diszkrét típus):diszkrét típus
```

A függvény egy diszkrét típusú érték rákövetkezőjét adja vissza. Ha nincs rákövetkező, akkor a visszaadott érték meghatározatlan.

SWAP

```
Function swap(i:integer):integer
```

Az eljárás felcseréli egy egész érték alsó és felső bájtját. Pl. ha i értéke (hexadecimálisban) 2AD5, akkor D52A lesz a függvény értéke.

TEXTBACKGROUND

```
procedure textbackground(i:integer)
```

Az eljárással a háttérszínt változtathatjuk meg. Az új szín kódja az i paraméter.

TEXTCOLOR

```
procedure textcolor(i:integer)
```

Az eljárással a karakterek színét változtathatjuk meg. i a színkód.

TRUNC

```
function trunc(r:real):integer
```

A függvény az r valós szám egészrészét adja vissza. A függvényértéknek az integer értékek halmazához kell tartoznia.

UPCASE

```
function upcase(c:char):char
```

Ha a c paraméter kisbetű, akkor a függvény a megfelelő nagybetűt adja vissza. Egyébként a c változatlan értékét.

VAL

```
procedure val(s:string;var i,jel:integer)
procedure val(s:string;var r:real;var jel:integer)
```

A második paraméter típusától függően integer vagy valós számmá alakítja az s-ben adott számstringet és elhelyezi a második paraméterbe. A jel értéke 0 lesz.
Ha az átalakítás nem lehetséges, mert a megadott string nem szintaktikailag hibátlan számkonstans, akkor a második paraméter értéke meghatározatlan marad. A jel értéke az átalakítást megakadályozó (első) hiba pozíciója a stringben.

WHEREX

```
function wherex:integer
```

A függvény értéke a helyőrnek az aktuális ablakra vonatkozó oszlop pozíciója.

WHEREY

```
function wherey:integer
```

A függvény értéke a helyőrnek az aktuális ablakra vonatkozó sor pozíciója.

WINDOW

```
procedure window(x1,y1,x2,y2:integer)
```

A képernyőn ablakot definiálhatunk ezzel az eljárással. Az ablak a képernyő aktív területének a szűkítése. bal felső sarkát az (x1,y1), a jobb alsót az (x2,y2) pont jelöli ki. Az ablak bal felső sarkát az eljárás meghívása után az (1,1) pontnak tekinti a rendszer.

WRITE, WRITELN

```
procedure write(var f: file; paraméterek)
procedure writeln(var f: file; paraméterek)
procedure write(paraméterek)
procedure writeln(paraméterek)
```

A Az eljárással a paraméterek értékét az f file-ba íratjuk ki. Ha file paraméter nincs explicite megadva, akkor a szabványos Output file-ba (ami általában a képernyő) írunk. A writeln csak text file-oknál használható, a paraméterek kiírása után sorvége jelet (10-es és 13-mas karaktert) is kiír.
A text file-oknál a numerikus részstringeket formázhatjuk. Egész érték után a hosszát, valós után a hosszát és a tizedesjegyek számát írhatjuk elő. A terminátor a kettőspont. Pl.:

```
write(i:5)
```

az i egész értékét 5 hely igénybevételével írja ki, ha a szám értéke kisebb, szóközök állnak előtte. A

```
write(r:8:2)
```

eljáráshívással az r valós számot 8 pozícióra írjuk ki. Ebből 2 a tizedes, 1 az előjel, 1 a tizedespont helye, így legfeljebb 4 számjegy állhat a tizedesponttól balra.

I) I/O hibaüzenetek

A be- és kivitel közben észlelt hibák a program megszakításához vezetnek. A hiba okáról és helyéről az

```
I/O error NN, PC=XXXX
Program aborted
```

üzenet tájékoztat bennünket, ahol NN a hibakód, XXXX a tárcím, ahol a gépikódú programban a hiba keletkezett.
Az alábbiakban a hibakódok jelentését adjuk meg.

DEC	HEX	A hiba oka
1	01	A file nem létezik A reset, erase, rename, execute vagy chain eljárásban nemlétező file a paraméter.
2	02	A file nincs megnyitva olvasásra 1) megpróbáltunk egy fileból olvasni, amit előzőleg nem nyitottunk meg. 2) rewrite eljárással megnyitott - így üres - állományból akartunk olvasni. 3) olyan logikai egységet próbáltunk olvasni, ami csak output (pl. LST).
3	03	A file nincs megnyitva írásra 1) megpróbáltunk írni egy file-ba anélkül, hogy előzőleg reset-tel vagy rewrite-tal megnyitottuk volna. 2) megpróbáltunk írni egy reset-tel megnyitott text file-ba. 3) megpróbáltunk egy input eszközre (pl. KBD) írni.
4	04	A file nincs megnyitva megpróbáltunk típus nélküli file-t írni vagy olvasni, amit előzőleg nem nyitottunk meg.
16	10	Hibás számadat Egy text file-ból egy numerikus változóba olvasott számstring hibás.

32 20 A művelet nem hajtható végre logikai készüléken

Logikai készülékhez rendelt állományon akartunk végrehajtani erase, rename, execute vagy chain műveletet.

33 21 Direkt módban nem végrehajtható művelet

Ha a programot a Run paranccsal hajtjuk végre, miközben a Memory compiler opció van érvényben, execute és chain nem hajtható végre.

34 22 Szabványos file-t nem jelölhetünk ki

Szabványos file nem lehet assign eljárás paramétere.

144 90 A rekordhossz nem fér össze

A file változó és a hozzákapcsolni kívánt állomány rekordhossza eltérő.

145 91 Seek a file végén túl

A seek eljárásban adott elem sorszám nagyobb vagy egyenlő, mint a file hossza.

153 99 Korai file vége

1) text file-nál a fizikai file vége megelőzte az eof jelet (elmaradt az eof jel).
2) a file vége után próbáltunk olvasni egy file-ból.
3) a read vagy a blockread nem tud a lemez következő szektorából olvasni. Tönkrement az állomány, vagy a típus nélküli file-ból a fizikai eof után próbáltunk olvasni.

240 F0 Diszk íráshiba

Ez a hiba a write, writeln, blockwrite és a flush eljárások működésekor jön létre, ha a lemez betelt. Esetleg írás jellegű műveleteknél is előfordulhat.

241 F1 A tartalomjegyzék betelt

A rewrite eljárás végrehajtása okozhatja, ha már nincs szabad hely az aktuális tartalomjegyzékben.

242 F2 File túlcsordulás

egy állományba 65535-nél nagyobb sorszámú elemet akart írni.

243 F3 Túl sok nyitott file

Már 15 file-t tart nyitva és még egyet meg akart nyitni.

255 FF Eltűnt file

Le akart zárni egy állományt, de nincs a tartalomjegyzékben pl. azért, mert közben lemezt cserélt.

J) Végrehajtási hibák

A fatális végrehajtási hibák programmegszakítást okoznak. A hiba okára és helyére a

```
Run-time error NN, PC=XXXX
Program aborted
```

hibaüzenetből következtethetünk. Az NN hibakód értékeit az alábbiakban közöljük:

DEC	HEX	A hiba oka
1	01	Lebegőpontos túlcsordulás
2	02	Nullával való osztás
3	03	SQRT argumentum hiba Az sqrt függvény aktuális paramétere negatív.
4	04	LN argumentum hiba Az ln függvénynek átadott aktuális paraméter értéke 0 vagy negatív.
16	10	String mérethiba 1) egy konkatenáció művelet eredménye 255-nél hosszabb. 2) 1-nél hosszabb stringet próbált karakterré konvertálni.
17	11	Hibás string index A copy, delete vagy insert eljárásokban az indexkifejezés értéke 255-nél nagyobb.
144	90	Indexhiba Egy tömbelem indexelésére használt indexkifejezés aktuális értéke nem esik a deklarált indextartományba.
145	91	Diszkrét (és intervallum) értékhiba A skalár vagy intervallum típusú változó értéke a tartományon kívül esik.
240	F0	Nincs overlay file Az overlay file hiányzik a lemezről.

255 FF Á dinamikus tár betelt

A new eljárás vagy egy alprogram hívás kapcsán a verem és a halom egymásba ért. Nincs elég szabad tárterület.

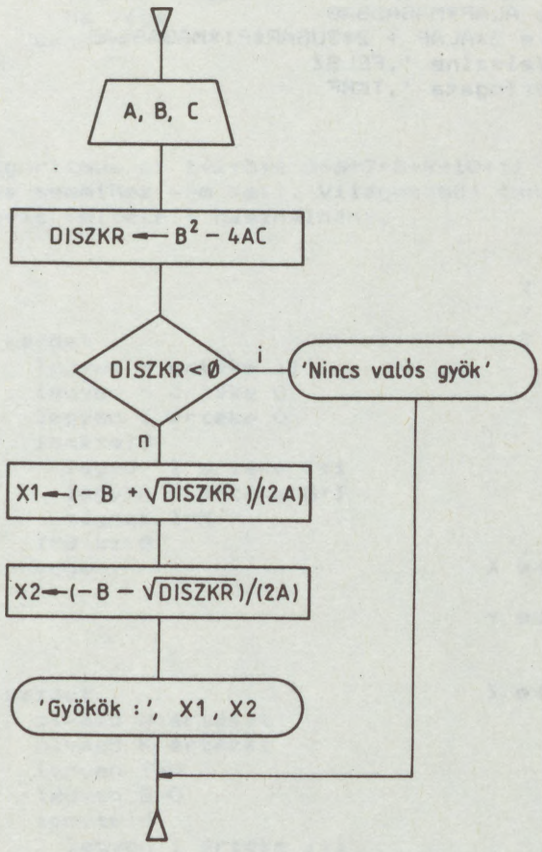
K) A feladatok megoldása

1. fejezet

1.

```
kezdet
  legyen S értéke X
  legyen X értéke Y
  legyen Y értéke S
  vége
```

2.



3.

```
kezdet
  olvasd X és Y értékét
  legyen KÖZÉP értéke (X+Y)/2
  írd ki KÖZÉP
  vége
```

4.

```
kezdet
  olvasd SUGÁR, MAGASSÁG értékét
  legyen ALAP értéke SUGÁR*SUGÁR*PI
  legyen TÉRF értéke ALAP*MAGASSÁG
  legyen FELSZ értéke 2*ALAP + 2*SUGÁR*PI*MAGASSÁG
  írd ki 'A henger felszíne ',FELSZ
  írd ki '         térfogata ',TÉRF
  vége
```

5.

```
kezdet
  ha X<Y akkor
    legyen S értéke X
    legyen X értéke Y
    legyen Y értéke S
    ha vége
  vége
```

6.

```
kezdet
  ha X<Y akkor
    legyen MIN értéke X
  egyébként
    legyen MIN értéke Y
    ha vége
  ha Z<MIN akkor
    legyen MIN értéke Z
    ha vége
  vége
```

7.

```
kezdet
  olvasd SZÁM értékét
  ha SZÁM>=50 akkor
    ha SZÁM<=100 akkor
      írd ki 'igen'
      ha vége
```

```
      egyébként
        írd ki 'nem'
        ha vége
      vége
```

Világosabb a logikai "és" használatával:

```
    kezdet
      olvasd SZAM értékét
      ha SZAM>=50 és SZAM<=100 akkor
        írd ki 'igen'
      egyébként
        írd ki 'nem'
        ha vége
      vége
```

8.

Az algoritmus az 1+2+3+4+5+6+7+8+9+10+11 összeget számítja ki. N értéke semmihez sem kell. Világosabbá tenné a funkciót ha az I<11 kilépési feltételt használnánk.

9.

```
    kezdet
      legyen N értéke 11
      legyen S értéke 0
      legyen I értéke 0
      ismételd
        legyen I értéke I+1
        legyen S értéke S+I
        mígnem I=N
      írd ki S
      vége
```

10.

```
    kezdet
      olvasd N értékét
      olvasd K értékét
      legyen I=0
      legyen S=0
      ismételd
        legyen I értéke I+1
        legyen S értéke S+IÉK
        mígnem I=N
      írd ki S
      vége
```

(É itt a hatványozás jele: 2É3 =8.)

11.

```
kezdet
  olvasd T értékét
  olvasd P értékét
  olvasd R értékét
  legyen TART értéke T
  legyen HO értéke 0
  amíg TART>0 ismételd
    legyen TART értéke TART-R
    legyen HO értéke HO+1
    ha TART>0 akkor
      legyen TART értéke TART*(1+P/1200)
      ha vége
    ismétlés vége
  írd ki HO
  vége
```

(Az éves kamat 12-ed része jut egy hónapra. A számításnál feltételeztük, hogy a kamatot minden hónap végén adják az összeghez. A törlesztő részletet a hónap elején fizetjük.)

12.

```
kezdet
  olvasd T értékét
  olvasd P értékét
  olvasd N értékét
  legyen RÉSZL értéke [T/N]
  ismételd
    legyen TÖRL értéke 0
    legyen I értéke 0
    ismételd
      legyen TÖRL értéke TÖRL-RÉSZL
      legyen TÖRL értéke TÖRL*(1+P/1200)
      legyen I értéke I+1
      mígnem I=N
    legyen RÉSZL értéke RÉSZL+10
    mígnem TÖRL>=T
  legyen RÉSZL értéke RÉSZL-10
  írd ki RÉSZL
  vége
```

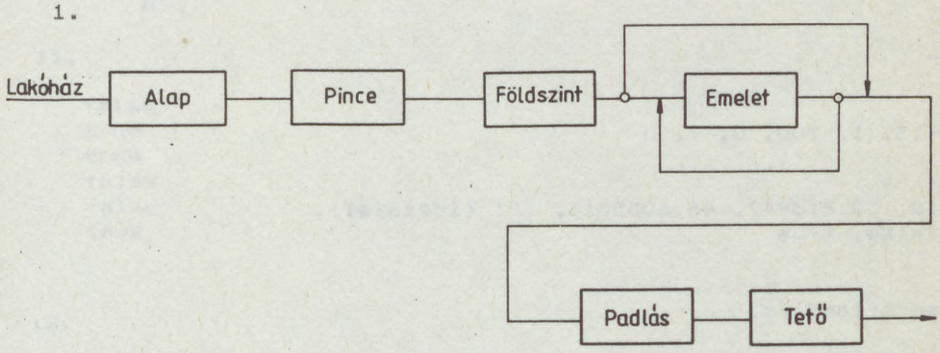
(A [T/N] a hányados egészrészét jelöli. Ezzen az értékkel kezdjük a számolást, a kamat miatt a tényleges törlesztőrészlet ennél az értéknél kisebb nem lehet. A belső ciklusban 10 Ft-onként növeljük a részletet, mígnem az N hónap alatt a törlesztés eléri a tartozást.)

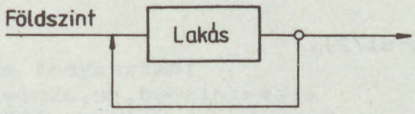
2. fejezet

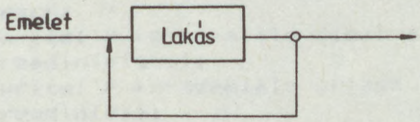
E feladatok megoldása nem igényel különösebb szellemi erőfeszítést. Tanulmányozza a 2. fejezet vonatkozó alfejezetét, ha ezután sem sikerül, forduljon a környezetében lévő gyakorlottabb ismerőséhez!

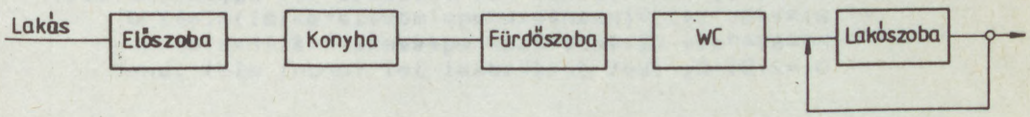
3. fejezet

1.









2. jó, rossz, rossz, jó, jó

3.

```
const nyolcad= 0.125;

      Avogadro= 6.02e23;

      g= 9.81;

      valos7= 7.0;
      egesz7= 7;
      char7 = '7';
      csengo= #7;
```

4. -

5. -13, 1, 100, 0, 7, 0

6. 10, 10.813947, 46.6055513, '"' (idézőjel),
   false, true

7. sqrt((a+b)/c)
   -
   6.02e23*ln(1.0+exp(1.0-x))

8. 4*a
   6*a+2*sqr(x)
   (A/2)*sin(2*omega*(fc-fm)*t-pi/2),
   arctan(x/sqrt(1-x*x))

A sin függvény inverze nincs a Pascal szabványos függvényei között, ezért az x szinuszú szög tangensét kifejezve az arctan-t használhatjuk.

9.

```
Hibásak:  A:=2+3.0; (egész változó valós értéket kap)
          A+2:=B; (a bal oldalon csak változó állhat)
          3:=2+1; (a bal oldalon csak változó állhat)
          A:=2*-3; (nem állhat két műveleti jel egymás mellett)
          B:=(-6.73)2; (hiányzik egy műveleti jel)
          R:=6.4 div 8; (div csak egészekkel állhat)
          Q:=2+B:=8; (két értékadási jel fordul elő)
```

10.

```
program tegla;
  const g=981;
  var a,b,c:real;
      suly:real;
      ro:real;
  begin
    clrscr;
    writeln('Irja be a harom oldal hosszat cm-ben!');
    readln(a,b,c);
    write('A suruseg g/kobcm-ben: ');
    readln(ro);
    suly:=a*b*c*ro*g/1000.0;
    writeln('A sulya ',suly:8:2,' N');
    end.
```

11.

```
false
true
true
false
false
true
```

12.

Felcseréli az a és a b változó értékét.

13.

```
program fogyasztas;
  var elozo,uj,benzin:real;
      fogy:real;
  begin
    clrscr;
      write('A km-szamlalo elozo allasa:');
      readln(elozo);
      write('A km-szamlalo allasa tankolaskor:');
      readln(uj);
      write('A takolt mennyiseg:');
      readln(benzin);
      fogy:=benzin*100.0/(uj-elozo);
      writeln;
      writeln('A ket tankolas kozott a 100 km-enkenti');
      writeln('atlagfogyasztas ',fogy:6:2,' l benzin volt.');
      end.
```

14.

```
program ado;
  var jutalom, adokulcs,
      ado, nyugdij, netto : real;
  begin
    clrscr;
    write('A jutalom osszege: ');
    readln(jutalom);
    write('Adokulcs %: ');
    readln(adokulcs);
    clrscr;
    writeln('Brutto jutalom     ',jutalom:5:0,' Ft');
    ado:=int(jutalom*adokulcs/100.0+0.5);
    writeln('Jovedelemado ',adokulcs:2:0,'%   ',ado:5:0,' Ft');
    nyugdij:=int(jutalom*0.1+0.5);
    writeln('Nyugdijjarulek      ',nyugdij:4:0,' Ft');
    writeln('--------------------------');
    netto:=jutalom-ado-nyugdij;
    writeln('Kifizetendo         ',netto:5:0,' Ft');
    end.
```

15.

```
program olaj;
  var d, m, h:real ;
      egysegar: real;
      keszlet, hiany:real;
      besz ar:real;
      i:integer;
  begin
    clrscr;
    gotoxy(8,1);
    write('F U T O O I A J  M E N N Y I S E G ');
    writeln('E N E K  S Z A M I T A S A');
    gotoxy(8,5);
    writeln('┌──────────────────┐ T T');
    for  i:=6 to 8 do begin
      gotoxy(8,i);
      writeln('|                  | | |');
      end;
    gotoxy(8,9);
    writeln('└──────────────────┘ | |');
    for  i:=10 to 14 do begin
      gotoxy(8,i);
      writeln('|                    | |');
      end;
    gotoxy(8,15);
    writeln('└────────────────────┘ |');
    gotoxy(8,17);
    writeln('<──────────────────>');
    gotoxy(8,18);
    writeln(' ød=atmero (cm)');
    gotoxy(28,11);
```

```
writeln('m=magassag (cm)');
gotoxy(30,7);
writeln('h=hiany (cm)');
gotoxy(43,7);
write('=');
readln(h);
gotoxy(43,11);
write('=');
readln(m);
gotoxy(25,18);
write('=');
readln(d);
keszlet:=sqr(d/4)*pi*(m-h)/1000;
gotoxy(12,13);
writeln(keszlet:8:1,' l');
hiany:=sqr(d/4)*pi*h/1000;
gotoxy(12,8);
writeln(hiany:8:1,' l');
gotoxy(35,15);
write('az olaj egysegara=');
readln(egysegar);
besz ar:=int(hiany*egysegar*10+0.5)/10;
gotoxy(8,22);
writeln('A BESZERZENDO MENNYISEG ARA ',besz ar:10:2,' Ft');
end.
```

A programban előforduló grafikus jeleket úgy állítottuk elő, hogy az [ALT] billentyű lenyomása után (a numerikus billentyűkön!) beírtuk a jel kódját. A hordó felrajzolása nem lényeges része a feladat megoldásának, ezzel csak az adatbevitelt akartuk szemléletessé tenni. Az itt használt for ciklusutasítást a 4. fejezetben tárgyaljuk.

4. fejezet

1.

```
program masodfoku;
  var a,b,c:real;
      x1,x2:real;
      d,a2:real;
  begin
    clrscr;
    writeln('Az egyenlet egyutthatoi:');
    write('                    a:');
    readln(a);
    write('                    b:');
```

```
    readln(b);
    write('                          c:');
    readln(c);
    writeln;
    if a=0.0 then begin
      writeln('Az egyenlet nem masodfoku!');
      if b<>0.0 then begin
        writeln('Elsofokukent megoldhato');
        writeln('gyoke ',-c/b:8:3);
        end
      else
        writeln('Nincs megoldasa');
      end
    else begin
      d:=b*b-4.0*a*c;
      if d<0.0 then
        writeln('Az egyenletnek nincs valos gyoke')
      else if d=0.0 then begin
        x1:=-b/(2.0*a);
        writeln('Ket egyenlo gyok van:');
        writeln('                        x1=x2=',x1:8:3);
        end
      else if d>0.0 then begin
        a2:=2.0*a;
        d:=sqrt(d);
        x1:=(-b+d)/a2;
        x2:=(-b-d)/a2;
        writeln('Az egyenlet gyokei:');
        writeln('                      x1=',x1:8:3);
        writeln('                      x2=',x2:8:3);
        end;
      end;
    end.
```

3.

2.

```
program ketismeretlenes;
  var a,b,p,
      c,d,q:real;
      x,y:real;
      det,xdet,ydet:real;
  begin
    clrscr;
      writeln('Az egyenletrendszer alakja:');
      writeln;
      writeln('        ax + by = p');
      writeln('        cx + dy = q');
      writeln;
      writeln('Kerem az egyutthatokat.');
      writeln;
      write('         a:');
      readln(a);
      write('         b:');
      readln(b);
```

```
    write('          p:');
    readln(p);
    write('          c:');
    readln(c);
    write('          d:');
    readln(d);
    write('          q:');
    readln(q);
    det:=a*d-c*b;
    xdet:=p*d-q*b;
    if det=0.0 then
      if xdet=0.0 then
        writeln('Nem fuggetlen egyenletek')
      else
        writeln('Ellentmondo egyenletek')
    else begin
      ydet:=a*q-c*p;
      x:=xdet/det;
      y:=ydet/det;
      writeln('Az egyenlet gyokei:');
      writeln('                    x=',x:8:2);
      writeln('                    y=',y:8:2);
      end;
    end.
```

3.

```
program ado;
  const s1=55000.0;
        s2=70000.0;
        s3=100000.0;
        s4=150000.0;
        s5=240000.0;
        s6=360000.0;
        s7=600000.0;
  var brutto,netto,ado:real;
  begin
    clrscr;
    write('kerem a brutto jovedelem erteket:');
    readln(brutto);
    if brutto<=s1 then
      ado:=0;
    if (brutto>s1) and (brutto<=s2) then
      ado:=((brutto-s1)*0.17);
    if (brutto>s2) and (brutto<=s3) then
      ado:=((s2-s1)*0.17+(brutto-s2)*0.23);
    if (brutto>s3) and (brutto<=s4) then
      ado:=((s2-s1)*0.17+(s3-s2)*0.23+
                (brutto-s3)*0.29);
    if (brutto>s4) and (brutto<=s5) then
      ado:=((s2-s1)*0.17+(s3-s2)*0.23+
                (s4-s3)*0.29+(brutto-s4)*0.35);
    if (brutto>s5) and (brutto<=s6) then
      ado:=((s2-s1)*0.17+(s3-s2)*0.23+
                (s4-s3)*0.29+(s5-s4)*0.35+
                (brutto-s5)*0.42);
```

```
    if (brutto>s6) and (brutto<=s7) then
      ado:=((s2-s1)*0.17+(s3-s2)*0.23+
                (s4-s3)*0.29+(s5-s4)*0.35+
                (s6-s5)*0.42+(brutto-s6)*0.49);
    if brutto>s7 then
      ado:=((s2-s1)*0.17+(s3-s2)*0.23+
                (s4-s3)*0.29+(s5-s4)*0.35+
                (s6-s5)*0.42+(s7-s6)*0.49+
                (brutto-s7)*0.56);
    ado:=ado+0.5;
    ado:=ado-frac(ado);
    netto:=brutto-ado;
    writeln;
    writeln('Az ado =              ',ado:6:0);
    writeln('A netto jovedelem = ',netto:6:0);
    end.
```

A program a személyi jövedelemadóra vonatkozó törvény 1989-es módosítása alapján számol.

4.

```
program haromszog;
  var a,b,c:real;
      t,s:real;
  begin
    clrscr;
    write('a=');
    readln(a);
    write('b=');
    readln(b);
    write('c=');
    readln(c);
    if (a<=0.0) or (b<=0.0) or (c<=0.0)
       or (a>=b+c) or (b>=a+c) or (c>=a+b) then
      writeln('Ez nem haromszog')
    else begin
      s:=(a+b+c);
      writeln;
      writeln('A haromszog kerulete ',s:6:1);
      s:=s/2;
      t:=sqrt(s*(s-a)*(s-b)*(s-c));
      writeln('A terulet           ',t:8:2);
      end;
    end.
```

5.

A program a következőket írja ki:

```
harom kutya
```

```
nem ugat hiaba

ket kutya
harom kutya
nem ugat hiaba

egy kutya
ket kutya
harom kutya
nem ugat hiaba
```

6.

```
program exp egyenlet;
  const eps=0.001;
  var also,felso,fele:real;
  begin
    also:=3.0;
    felso:=4.0;
    while abs(felso-also)>eps do begin
      fele:=(also+felso)/2.0;
      if exp(fele*ln(2))-fele-10<0.0 then
        also:=fele
      else
        felso:=fele;
      end;
    writeln('a kozelito gyok ',(also+felso)/2.0+0.0005:6:3);
  end.
```

Egy f(x) függvény zérushelyét pl. ismételt felezésekkel kereshetjük meg. A zérushelytől balra és jobbra f(x) különböző előjelű. Az "also" és "felso" változók mindig olyan intervallumot határoznak meg, amely tartalmazza a gyököt. A felezések során egyre kisebb intervallumot határozunk meg, amely a gyököt tartalmazza. Az intervallum hossza biztosan nagyobb a közelítés pontosságánál.

7.

```
program e kozelites;
  var e:real;
      k,i:integer;
  begin
    e:=1;
    clrscr;
    write('kerem k erteket:');
    readln(k);
    for i:=1 to k do
      e:=e*(1.0+1.0/k);
    writeln('e kozelito erteke ',e);
    end.
```

A sorozat az "e" számot közelíti.

8.

```
program faktorialis;
  var n,i,fakt:integer;
  begin
    clrscr;
    write('kerem n erteket:');
    readln(n);
    if n<0 then begin
      writeln('adathiba');
      halt;
      end;
    fakt:=1;
    for i:=1 to n do
      fakt:=fakt*i;
    writeln;
    writeln(n,'! = ',fakt);
    end.
```

9.

```
program tablazat;
  const kezd=0.0;
        veg=1.5;
        lepes=0.1;
  var x:real;
      i:integer;
  begin
    x:=kezd;
    i:=0;
    clrscr;
    writeln('   x          y ');
    writeln('----------------------');
    while x<=veg do begin
      writeln(x:4:1,'       ',1.0/(sin(x)+cos(x)):5:3);
      i:=i+1;
      x:=kezd+i*lepes;
      end;
    end.
```

A kérdéses intervallumban a nevező nem veszi fel a 0 értéket, ezért az erre vonatkozó vizsgálatot elhagytuk.

10.

```
program torlesztesi ido;
  var kolcson,reszlet,tartozas,kamatlab:real;
      ev,ho:integer;
```

```
begin
  clrscr;
  write('kolcson:');
  readln(kolcson);
  write('kamatlab:');
  readln(kamatlab);
  write('reszlet:');
  readln(reszlet);
  writeln;
  tartozas:=kolcson;
  ho:=0;
  repeat
    tartozas:=tartozas-reszlet;
    ho:=ho+1;
    if tartozas>0.0 then
      tartozas:=tartozas*(1.0+kamatlab/1200.0);
    until tartozas<=0.0;
  writeln('A honapok szama ',ho);
  writeln('Ez ',ho div 12,' ev es ',ho mod 12,' honap.');
  end.
```

11.

```
program reszlet;
  var kolcson,reszlet,kamatlab,tartozas:real;
      n,ido,i:integer;
  begin
    clrscr;
    write('kolcson:');
    readln(kolcson);
    write('kamatlab:');
    readln(kamatlab);
    write('torlesztesi ido (ev):');
    readln(n);
    ido:=12*n;
    reszlet:=int(kolcson/ido);
    repeat
      tartozas:=kolcson;
      for  i:=1 to ido do begin
        tartozas:=tartozas-reszlet;
        tartozas:=tartozas*(1.0+kamatlab/1200.0);
        end;
      if tartozas>0.0 then reszlet:=reszlet+10;
      until tartozas<=0.0;
    writeln;
    writeln('A havi torlesztes ',reszlet:4:0);
    end.
```

12.

```
program zarthelyi;
  var atlag:real;
      j1,j2,j3,j4,j5:integer;
      osszeg,jegy:integer;
  begin
    clrscr;
    osszeg:=0;
    j1:=0;j2:=0;j3:=0;j4:=0;j5:=0;
    repeat
      write('Az osztalyzat bevitele (vegjel=0):');
      readln(jegy);
      if (jegy<0) or (jegy>5) then
        writeln('HIBA - ismetelje a bevitelt!')
      else begin
        osszeg:=osszeg+jegy;
        case jegy of
          1: j1:=j1+1;
          2: j2:=j2+1;
          3: j3:=j3+1;
          4: j4:=j4+1;
          5: j5:=j5+1;
          end;
       end;
      until jegy=0;
   writeln;
   writeln('STATISZTIKA');
   writeln;
   writeln('1:  ',j1:2);
   writeln('2:  ',j2:2);
   writeln('3:  ',j3:2);
   writeln('4:  ',j4:2);
   writeln('5:  ',j5:2);
   writeln;
   atlag:=osszeg/(j1+j2+j3+j4+j5);
   writeln('atlag = ',atlag+0.005:5:2)
   end.
```

13.

```
program Monte Carlo;
  var x,y:real;
      s,n,i:integer;
  begin
    randomize;
    write('hany ponttal kivan dolgozni?');
    readln(n);
    s:=0;
    for i:=1 to n do begin
      x:=random;
      y:=random;
      if sqr(x)+sqr(y)<=1.0 then
        s:=s+1;
```

```
    end;
  writeln('A beeso pontok szama ',s);
  writeln('A pi kozelito erteke ',4*s/n:5:4);
  end.
```

Ebben a feladatban egy véletlen kisérlet végrehajtását szimuláljuk számítógéppel. A kisérletet nagyon sokszor végrehajtva elméleti valószínűség értékeket tudunk közelítőleg meghatározni. Az ilyen elven alapuló - kifejezetten számítógépekre kifejlesztett - módszereket Monte Carlo módszereknek nevezték el.

14.

```
program kocka;
  var p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12:integer;
      n,i:integer;
      dobas,k1,k2:integer;
  begin
    p2:=0;p3:=0;p4:=0;p5:=0;p6:=0;
    p7:=0;p8:=0;p9:=0;p10:=0;p11:=0;p12:=0;
    randomize;
    clrscr;
    write('A dobasok szama: ');
    readln(n);
    for i:=1 to n do begin
      k1:=random(6)+1;
      k2:=random(6)+1;
      dobas:=k1+k2;
      case dobas of
        2: p2:=p2+1;
        3: p3:=p3+1;
        4: p4:=p4+1;
        5: p5:=p5+1;
        6: p6:=p6+1;
        7: p7:=p7+1;
        8: p8:=p8+1;
        9: p9:=p9+1;
       10: p10:=p10+1;
       11: p11:=p11+1;
       12: p12:=p12+1;
       end;
    end;
  writeln('2-es dobas valoszinusege  ',p2/n:4:2);
  writeln('3-es dobas valoszinusege  ',p3/n:4:2);
  writeln('4-es dobas valoszinusege  ',p4/n:4:2);
  writeln('5-es dobas valoszinusege  ',p5/n:4:2);
  writeln('6-es dobas valoszinusege  ',p6/n:4:2);
  writeln('7-es dobas valoszinusege  ',p7/n:4:2);
  writeln('8-es dobas valoszinusege  ',p8/n:4:2);
  writeln('9-es dobas valoszinusege  ',p9/n:4:2);
```

```
    writeln('10-es dobas valoszinusege  ',p10/n:4:2);
    writeln('11-es dobas valoszinusege  ',p11/n:4:2);
    writeln('12-es dobas valoszinusege  ',p12/n:4:2);
    end.
```

Ezt az algoritmust elegánsabban programozhatjuk a 8. fejezetben bevezetendő tömbök segítségével.

15.

```
program Fibonacci;
  var arr,ar,a:integer;
      n,i:integer;
  begin
    clrscr;
    write('n erteke: ');
    readln(n);
    arr:=1;
    ar:=1;
    for  i:=3 to n do begin
      a:=arr+ar;
      writeln('A ',i,'-edik: ',a);
      arr:=ar;
      ar:=a;
      end;
    end.
```

5. fejezet

1.

```
function arcsin(x:real):real;
  begin
    arcsin:=atctan(x/sqrt(1.0-x*x));
    end;

function arccos(x:real):real;
  begin
    arccos:=arctan(sqrt(1.0-x*x)/x);
    end;
```

2.

```
program egyenletek;
  var a,b,c,x1,x2:real;
      statusz:integer;
      vege:real;

procedure masodfoku(a,b,c:real;
                    var x1,x2:real;
                    var st:integer);
  var d,a2:real;
  begin
    if a=0.0 then
      if b<>0.0 then begin
        st:=1;
        x1:=-c/b;
        end
      else
        st:=0
    else begin
      d:=b*b-4.0*a*c;
      if d<0.0 then
        st:=-1
      else if d=0.0 then begin
        x1:=-b/(2.0*a);
        st:=2;
        end
      else if d>0.0 then begin
        a2:=2.0*a;
        d:=sqrt(d);
        x1:=(-b+d)/a2;
        x2:=(-b-d)/a2;
        st:=3;
        end;
      end;
    end;

  begin
```

```
repeat
  clrscr;
  writeln('Az egyenlet egyutthatoi:');
  write('                          a:');
  readln(a);
  write('                          b:');
  readln(b);
  write('                          c:');
  readln(c);
  writeln;
  vege:=a*a+b*b+c*c;
  if vege<>0 then begin
    masodfoku(a,b,c,x1,x2,statusz);
    case statusz of
      -1: writeln('Nincs valos gyok');
       0: writeln('Nem oldhato meg');
       1: writeln('Elsofoku, x=',x1:8:3);
       2: begin
            writeln('Ket egyenlo gyok: ');
            writeln('                x1=x2=',x1:8:3);
            end;
       3: begin
            writeln('Az egyenletrendszer gyokei: ');
            writeln('                          x1=',x1:8:3);
            writeln('                          x2=',x2:8:3);
            end;
       end;
       delay(3000);
    end;
  until vege=0.0;
  end.
```

3.

```
program idomok;
  var idom:char;
  procedure kor;
    var r,t:real;
    begin
      write('kerem a kor sugarat: ');
      readln(r);
      t:=sqr(r)*pi;
      writeln('A kor terulete ',t:8:2);
      end;
  procedure teglalap;
    var a,b,t:real;
    begin
      writeln('kerem az oldalakat:');
      write('a=');
      readln(a);
      write('b=');
      readln(b);
```

```
      t:=a*b;
      writeln('A teglalap terulete ',t:8:2);
      end;
  procedure negyzet;
    var a,t:real;
    begin
      write('kerem a negyzet oldalat:');
      readln(a);
      t:=sqr(a);
      writeln('A negyzet terulete ',t:8:2);
      end;
  procedure haromszog;
    var a,m,t:real;
    begin
      write('kerem az alapot:');
      readln(a);
      write('kerem a megassagot:');
      readln(m);
      t:=a*m/2.0;
      writeln('A haromszog terulete ',t:8:2);
      end;
  begin
    clrscr;
    writeln('Milyen sikidom teruletet szamolja?');
    writeln;
    writeln('negyzet      N');
    writeln('teglalap     T');
    writeln('haromszog    H');
    writeln('kor          K');
    readln(idom);
    case upcase(idom) of
      'N': negyzet;
      'T': teglalap;
      'H': haromszog;
      'K': kor;
      end;
```

4.

```
program hatvanyok;
  const kezd=0.0;
        veg=2.0;
        lepes=0.1;
        hatvanyalap=3.0;
  var k:real;
      i:integer;
  function hatvany(alap,kit:real):real;
    begin
      if alap<=0.0 then begin
        writeln('az alap nem pozitiv');
        halt;
        end
      else
```

```
        hatvany:=exp(kit*ln(alap));
      end;
  begin
    i:=0;
    repeat
      k:=i*0.1;
      writeln(k:4:1,'     ',hatvany(hatvanyalap,k):8:4);
      i:=i+1;
      until i>20;
    end.
```

5.

```
program binom;
  var n,k,binom:integer;
  function fakt(i:integer):integer;
    var j,m:integer;
    begin
      m:=1;
      for j:=1 to i do
        m:=m*j;
      fakt:=m;
      end;
  begin
    clrscr;
    write('n=');
    readln(n);
    write('k=');
    readln(k);
    binom:=fakt(n) div (fakt(k)*fakt(n-k));
    writeln('a binomialis egyutthato ',binom)
    end.
```

6.

```
program binomialis;
  var n,k:integer;
      bnm:integer;

  function binom(n,k:integer):integer;
    begin
      if (n<0) or (k<0) then begin
        writeln('Hibas adat');
        halt;
        end;
      if n<k then binom:=0
      else if (k=0) or (k=n) then
        binom:=1
      else
        binom:=binom(n-1,k-1) + binom(n-1,k);
      end;
```

7.

```
begin
  clrscr;
  write('n:');
  readln(n);
  write('k:');
  readln(k);
  writeln;
  bnm:=binom(n,k);
  writeln(n,' alatt a ',k,' =',bnm);
  end.
```

7.

```
function hatvany(x:real;k:integer):real;
  begin
    if k<0 then begin
      writeln('negativ kitevo');
      halt;
      end;
    if k=0 then
      hatvany:=1
    else if odd(k) then
          hatvany:=x*hatvany(x,k-1)
        else
          hatvany:=sqr(hatvany(x,k div 2));
  end;
```

8.

A futási eredmény:

```
a
s
d
f
g
h
j
k
l
.
,
l
k
j
h
g
f
d
s
a
```

Az olvasott betűket fordított sorrendben írja vissza. A végjel beírásáig a veremben tárolódnak a karakterek.

9.

A csere eljárás csak a lokális változók értékét cseréli fel. A p, q aktuális paraméterekre ez semmi hatással sincs, mert a paraméterátadás érték szerint történik. Ezért a futási eredmény:

```
Hivas elott:
p= 3.7
q= 4.3
Hivas utan:
p= 3.7
q= 4.3
```

10.

```
procedure teglalap(a,b:integer);
  const  bfs='┌';
         jfs='┐';
         bas='└';
         jas='┘';
         fuq='│';
         vsz='─';
  var j:integer;
  begin
    clrscr;
    write(bfs);
    for j:=2 to a-1 do
      write(vsz);
    writeln(jfs);
    for j:=2 to b-1 do
      writeln(fuq,' ':a-2,fuq);
    write(bas);
    for j:=2 to a-1 do
      write(vsz);
    writeln(jas);
    end;
```

11.

```
program akarhova;
  var a,b,c,d:integer;
```

```
procedure teglalap(x1,y1,x2,y2:integer);
  const  bfs='┌';
         jfs='┐';
         bas='└';
         jas='┘';
         fug='│';
         vsz='─';
  var j:integer;
  begin
    gotoxy(x1,y1);
    write(bfs);
    for j:=x1+1 to x2-1 do
      write(vsz);
    writeln(jfs);
    for j:=y1+1 to y2-1 do begin
      gotoxy(x1,j);
      writeln(fug,' ':x2-x1-1,fug);
      end;
    gotoxy(x1,y2);
    write(bas);
    for j:=x1+1 to x2-1 do
      write(vsz);
    writeln(jas);
    end;

  begin
    write('balcsucs: ');
    readln(a,b);
    write('jobbcsucs: ');
    readln(c,d);
    clrscr;
    teglalap(a,b,c,d);
    delay(4000);
    end.
```

12.

```
program grafika;
  const sor=23;
        oszlop=78;
  var a,b,c,d:integer;
      n,i:integer;

procedure teglalap(x1,y1,x2,y2:integer);
  const  bfs='┌';
         jfs='┐';
         bas='└';
```

```
        jas='┴';
        fuq='│';
        vsz='─';
var j:integer;
begin
  gotoxy(x1,y1);
  write(bfs);
  for j:=x1+1 to x2-1 do
    write(vsz);
  writeln(jfs);
  for j:=y1+1 to y2-1 do begin
    gotoxy(x1,j);
    writeln(fuq);
    end;
  for j:=y1+1 to y2-1 do begin
    gotoxy(x2,j);
    writeln(fuq);
    end;
  gotoxy(x1,y2);
  write(bas);
  for j:=x1+1 to x2-1 do
    write(vsz);
  writeln(jas);
  end;

begin
  n:=7;
  randomize;
  repeat
  clrscr;
  i:=0;
  repeat
    a:=random(oszlop-5)+1;
    b:=random(sor-2)+1;
    repeat
      c:=random(oszlop+5-a)+a;
      until (c>a+1) and (c<=oszlop);
    repeat
      d:=random(sor+2-b)+b;
      until (d>b+1) and (d<=sor);
    teglalap(a,b,c,d);
    i:=i+1;
    until i=n;
    delay(1000);
    until 2<1;
  end.
```

A program egymás után más és más 7 téglalapból álló "nonfiguratív grafikát" rajzol a képernyőre.

6. fejezet

1. -

2. -

3. -

4. -

5.

a) Hiányoznak a zárójelek. Helyesen:

```
while (x<1) and (y<>2) do ...
```

c) := nem relációjel, helyette = használandó.

d) A case címke után csak egyetlen utasítást írhatunk. Egyébként begin és end közé kell zárni. Ez itt hiányzik. Helyesen pl.

```
1,2: begin
       j:=j+1; k:=k+1;
       end;
```

e) else előtt nem lehet pontosvessző.

6.

Ha a programot futtatni próbáljuk, a következő hibák derülnek ki:

Error 2: ':' expected. A hiba: hiányzó begin.
Error 1: ';' expected. A hiba: "sum:=1" után nincs pontosvessző.
Error 41: Unknown identifier or syntax error. A hiba: "i" nincs deklarálva.
Error 7: ':=' expected. Hiba: "i=" helyett "i:=" a helyes.
Error 7: ':=' expected. Hiba: "sum=" áll "sum:=" helyett.
Error 1: ';' expected. Hiba: "sum:=k" után hiányzik a pontosvessző.

Error 5: ')' expected. Hiba: a writeln utasításban hiányzik a "sum" előtt a pontosvessző.

Ezzel a szintaktikai hibákat kijavítottuk. Figyeljük meg, hogy a fordító nem mindig képes a hiba valóságos okát felismerni.
A program szemantikai hibákat is tartalmaz, nem azt teszi, amit tennie kellene. Ezeket nem taglaljuk részletesebben, ehelyett a helyes programlistát közöljük:

```
program hibak;
{
 Ez a program az elso k db termeszetes szam
 osszeget hatarozza meg, ahol k adat. }

  var k,i, sum:integer;
  begin
    write('Kerem k erteket:');
    readln(k);
    sum:=0;
    for i:=1 to k do
      sum:=sum+i;
    writeln('Az osszeg 1-tol ',k,'-ig=',sum)
    end.
```

7.

Nem. Nem minden nemnegatív egészre működik a program helyesen. 0-ra sem, mert a program szerint 0!=0. Ezt az "until i=n" feltétel okozza, ahol most i=1 és n=0. A ciklus addig fut, amíg i felveszi a 2, ..., 32767, -32768, ..., -1, 0 értékeket, míg végül 0-ra megáll. Tehát 0-val is szoroz.

7. fejezet

1.

```
procedure napiro(x:naptipus);
  type sztipus=string[10];
       hettipus=array[naptipus] of sztipus;
  const napok:hettipus=('hetfo','kedd','szerda','csutortok',
                        'pentek','szombat','vasarnap');

  begin
    write(napok[x]);
    end;
```

2.

```
program honapkezeles;
  type honaptipus=(januar,februar,marcius, aprilis,
                   majus,junius,julius,augusztus,
                   szeptember, oktober,november,december);
  var ho:honaptipus;
  procedure honapolvasas(var x:honaptipus);
    var i:1..12;
    begin
      write('Kerem a honap sorszamat:');
      readln(i);
      x:=honaptipus(i);
      end;
  begin
    honapolvasas(ho);
    writeln(ord(ho));
    end.
```

3.

```
type nyartipus = junius..augusztus;
```

4.

```
function binbyte(x:byte):str8;
var st:str8;
    h:char absolute st;
    i:integer;
    m:byte;
begin
   h:=chr(8);
   for i:=1 to 8 do begin
      m:=x;
      m:=m and 1;
      st[9-i]:=chr(m+48);
      x:=x shr 1;
      end;
   binbyte:=st;
   end;
```

5.

```
program bitkep;
  type str8=string[8];
  var x:byte;
  {$i 7 4.pas}
  begin
    repeat
      readln(x);
      writeln('x bitkepe: ',binbyte(x));
      until x=0;
    end.
```

6. Lásd a 7. feladat megoldását!

7.

A következő program a 6. feladat megoldását is tartalmazza:

```
program reitely;
type str8=string[8];
var y:integer;
    a,f:byte;
function binbyte(x:byte):str8;
var st:str8;
    h:char absolute st;
    i:integer;
    m:byte;
begin
   h:=chr(8);
   for i:=1 to 8 do begin
      m:=x;
      m:=m and 1;
      st[9-i]:=chr(m+48);
      x:=x shr 1;
      end;
   binbyte:=st;
   end;
begin
   clrscr;
   y:=32767;
   a:=lo(y);
   f:=hi(y);
   writeln('  ',y,'    ',binbyte(f),binbyte(a));
   y:=y+1;
   a:=lo(y);
   f:=hi(y);
   writeln(y,'    ',binbyte(f),binbyte(a));
   end.
```

8.

```
   function decertek(ch:char):integer;
   begin
      if ch<='9' then
         decertek:=ord(ch)-ord('0')
      else
         decertek:=ord(upcase(ch))-ord('A')+10;
      end;

   procedure tizesre(masszam:strings; alap:integer;
                     var tizes:integer; var hiba:boolean);
   var i,szjegy,kitevo,tizedes: integer;
       mszam: strings;

   begin
```

```
    mszam:=szam;
    kitevo:=length(mszam)-1
    tizes:=0;
    hiba:=false;
    for i:=1 to length(mszam) do begin
       szjegy:= decertek(mszam[i]);
       if (szjegy <0) or (szjegy >=alap) then
          hiba:=true;
       tizes:= tizes + szjegy*hatvany(alap,kitevo);
       kitevo:=kitevo-1;
       end;
    end;
```

9.

```
    function masszam(tizes:real; alap:integer):str64;
    var jszam,kitevo:integer;
        mszam:str64;
    begin
          if tizes=0 then begin
            masszam:='0';
            exit;
            end;
          mszam:='';
          kitevo:=0;
          while hatvany(alap,kitevo) <= tizes do
             kitevo:=kitevo+1;
         while kitevo>0 do begin
          kitevo:=kitevo-1;
          jszam:=0;
          while tizes > hatvany(alap,kitevo)-1 do begin
             tizes:=tizes-hatvany(alap,kitevo);
             jszam:=jszam+1;
             end;
          mszam:=mszam+masjegy(jszam);
          end;
       masszam:=mszam;
       end;
```

10. Lásd a 11. feladat megoldását!

11. A program a 10. feladat megoldását adja, de a 11. feladatnak megfelelően tizedeseket is kezel:

```
 program atalakit;
 type str64=string[64];
 var egyik,masik:str64;
     a1,a2:integer;
     hibas:boolean;
```

```
function hatvany(alap,kitevo:real):real;
begin
   hatvany:=exp(kitevo*ln(alap));
   end;

function decertek(ch:char):integer;
begin
   if ch<='9' then
      decertek:=ord(ch)-ord('0')
   else
      decertek:=ord(upcase(ch))-ord('A')+10;
   end;

procedure tizesbe(szam:str64; alap:integer;
                  var tizes:real; var hiba:boolean);
var i,szjegy,kitevo,tizedes: integer;
    mszam: str64;

begin
   mszam:=szam;
   tizedes:= pos('.',szam);
   if tizedes=0 then
      kitevo:=length(mszam)-1
   else begin
      kitevo:=tizedes-2;
      mszam:=copy(mszam,1,tizedes-1) +
                  copy(mszam,tizedes+1,length(mszam));
      end;
   tizes:=0;
   hiba:=false;
   for i:=1 to length(mszam) do begin
      szjegy:= decertek(mszam[i]);
      if (szjegy <0) or (szjegy >=alap) then
         hiba:=true;
      tizes:= tizes + szjegy*hatvany(alap,kitevo);
      kitevo:=kitevo-1;
      end;
   end;

function masjegy(sz:integer):char;
begin
   if sz<=9 then
      masjegy:=chr(sz + ord('0'))
   else
      masjegy:=chr(sz - 10 + ord('A'));
   end;

function masszam(tizes:real; alap:integer):str64;
var iszam,kitevo:integer;
    mszam:str64;
begin
   if tizes<1.0 then begin
      mszam:='0.';
      kitevo:=-1;
```

```
      end
   else begin
      mszam:='';
      kitevo:=0;
      while hatvany(alap,kitevo) <= tizes do
         kitevo:=kitevo+1;
      kitevo:=kitevo-1;
      end;
   while tizes > 0.0000001 do begin
      jszam:=0;
      while tizes >= hatvany(alap,kitevo)-0.0000001 do begin
         tizes:=tizes-hatvany(alap,kitevo);
         jszam:=jszam+1;
         end;
      mszam:=mszam+masjegy(jszam);
      if kitevo=0 then
         mszam:=mszam+'.';
      kitevo:=kitevo-1;
      end;
   masszam:=mszam;
   end;

procedure atvaltas(szam1:str64;alap1,alap2:integer;
                   var szam2:str64;
                   var hiba:boolean);
var tizes:real;
begin
   tizesbe(szam1,alap1,tizes,hiba);
   if not hiba then
      szam2:=masszam(tizes,alap2);
   end;

begin
   clrscr;
   repeat
      write('konvertalado szam:');
      readln(egyik);
      if egyik='' then halt;
      write('alap:');
      readln(a1);
      write('az uj alap:');
      readln(a2);
      atvaltas(egyik,a1,a2,masik,hibas);
      if not hibas then
      '  writeln('az uj szanrendszerben:',masik)
      else
         writeln('hibas a szam');
      until egyik='';
   end.
```

12.

```
procedure atvalt(var szoveg:strings);
  var i:integer;
  begin
    for i:=1 to length(szoveg) do
      szoveg[i]:=upcase(szoveg[i]);
    end;
```

13.

```
type strings=string[79];
var szoveg,regi,uj:strings;
    mennyit:integer;

function kovetkezo(szoveg,resz:strings;
                   honnan:integer):integer;

var hol,i,j:integer;
    egyezik:boolean;

begin
  hol:=-1;
  i:=honnan;
  while hol<0 do
    if i> length(szoveg)-length(resz)+1 then
      hol:=0
    else begin
      j:=1;
      egyezik:=true;
      while egyezik and (j<=length(resz)) do
        if szoveg[i+j-1]=resz[j] then
          j:=j+1
        else
          egyezik:=false;
      if egyezik then
        hol:=i
      else
        i:=i+1;
      end;
  kovetkezo:=hol;
  end;

procedure helyettesit(var miben:strings;
                      mit,mivel:strings;
                      var hanyszor:integer);

var k:integer;
begin
  hanyszor:=0;
  k:=0;
  repeat
    k:=kovetkezo(miben,mit,k+1);
    if k>0 then begin
```

14.

```
      delete(miben,k,length(mit));
      insert(mivel,miben,k);
      hanyszor:=hanyszor+1;
      end;
    until k=0;
  end;
```

14.

```
  type strings=string[79];
  var szoveg,regi,uj:strings;
      mennyit:integer;

  procedure kicsinyit(var x:strings);
    var i:integer;
    begin
      for i:=1 to length(x) do
        if (x[i]>='A') and (x[i]<='Z') then
          x[i]:=char(ord(x[i])+32);
      end;

  function nagybetu(x:char):boolean;
  var k:boolean;
    begin
      if (x>='A') and (x<='Z') then
        k:=true
      else
        k:=false;
      nagybetu:=k;
      end;

  function mindnagy(x:strings):boolean;
    var i:integer;
        nagy:boolean;
    begin
      nagy:=true;
      for i:=1 to length(x) do
        nagy:=nagy and nagybetu(x[i]);
      mindnagy:=nagy;
      end;

  function csakazeleje(x:strings):boolean;
    begin
      csakazeleje:= nagybetu(x[1]) and not mindnagy(x);
    end;

  function atvalt(x:strings):strings;
    var i:integer;
        valt:strings;
    begin
      valt:='';
      for i:=1 to length(x) do
        valt:=valt+upcase(x[i]);
      atvalt:=valt;
      end;
```

```
function kovetkezo(szoveg,resz:strings;
                   honnan:integer):integer;

var hol,i,j:integer;
    egyezik:boolean;

begin
  hol:=-1;
  i:=honnan;
  while hol<0 do
    if i> length(szoveg)-length(resz)+1 then
      hol:=0
    else begin
      i:=1;
      egyezik:=true;
      while egyezik and (j<=length(resz)) do
        if upcase(szoveg[i+j-1])=upcase(resz[j]) then
          j:=j+1
        else
          egyezik:=false;
      if egyezik then
        hol:=i
      else
        i:=i+1;
      end;
  kovetkezo:=hol;
  end;

procedure helyettesit(var miben:strings;
                      mit,mivel:strings;
                      var hanyszor:integer);

var k:integer;
    regi,uj:strings;
begin
  hanyszor:=0;
  k:=0;
  repeat
    k:=kovetkezo(miben,mit,k+1);
    if k>0 then begin
      regi:=copy(miben,k,length(mit));
      delete(miben,k,length(mit));
      if mindnagy(regi) then
        uj:=atvalt(mivel)
      else if csakazelseje(regi) then
        uj:=upcase(mivel[1])+copy(mivel,2,length(mivel))
      else
        uj:=mivel;
      insert(uj,miben,k);
      hanyszor:=hanyszor+1;
      end;
    until k=0;
  end;
```

15.

```
procedure megfordit(var x:strings);
  type elemtipus=char;
  var n,i:byte;

  procedure csere(var a,b:elemtipus);
    var t:elemtipus;
    begin
      t:=a;
      a:=b;
      b:=t;
      end;

  begin
    n:=length(x);
    for i:=1 to n div 2 do
      csere(x[i],x[n-i+1]);
    end;
```

16.

```
program palindrom;
  type strings=string[79];
  var i,n:integer;
      v,s:strings;
      x:real;

  {$i 7_15.pas}

  begin
    x:=maxint;
    n:=round(sqrt(x));
    for i:=1 to n do begin
      str(i*i,s);
      v:=s;
      megfordit(v);
      if v=s then
        writeln(i:3,' negyzete ',s:5);
      end;
    end.
```

17.

```
function elteres(x,y:strings):byte;
  var i,n,m,elter:byte;
  begin
    elter:=0;
    n:=length(x);
    m:=length(y);
    if n<m then
      elter:=n+1 ;
    for i:=1 to n do
```

```
    if upcase(x[i])<>upcase(y[i]) then
      if (i<elter) or (elter=0) then
        elter:=i;
  elteres:=elter;
  end;
```

18.

```
procedure atrendez(var x:strings;kezd:strings);
  var k:byte;
      s:strings;
  begin
    k:=pos(kezd,x);
writeln('k=',k);
    if k<>0 then begin
      s:=copy(x,k,length(x)-k+1);
writeln(s);
      s:=s+' * '+copy(x,1,k-1);
writeln(s);
      x:=s;
      end;
  end;
```

8. fejezet

1.

```
type vtomb=array[1..n] of real;
function osszegzes(a:vtomb;n:integer):real;
  var i:integer;
      ossz:real;
  begin
    ossz:=0;
    for i:=1 to n do
      ossz:=ossz+a[i];
    osszegzes:=ossz;
    end;
```

2.

```
program p8_2;
const n=100;
{$i 8_1.pas}
var a:vtomb;
    i,db:integer;
    osszeg,atlag,szoras:real;
{ --- szoras ertekenek kiszamitasa alprogrammal ---}
function szorassz(a:vtomb;m:real;n:integer):real;
var i:integer;
    ossz:real;
begin
```

```
ossz:=0;
for i:=1 to n do ossz:=ossz+a[i]*a[i];
szorassz:=sqrt(ossz/n-m*m)
end;
{ -------------------- }
begin
for i:=1 to n do a[i]:=0;
{ --- beolvasas --- }
clrscr;
i:=0;
 repeat
   i:=i+1;
   write('Kerem a tomb ',i,'. elemet (,ha 0 akkor
   readln(a[i]);                              bevitel vege):');
 until (a[i]=0) or (i=n);
if i=1 then begin
            writeln('Nincs egyetlen elem sincs megadva!');
            halt;
            end;
db:=i-1;
{ --- osszegzes --- }
osszeg:=osszegzes(a,db);
writeln(db);
{ --- atlag szamitas ---}
atlag:=osszeg/db;
{ --- szoras szamitasa ---}
szoras:=szorassz(a,atlag,db);
writeln;
writeln('A tomb elemeinek atlaga: ',atlag:5:2);
writeln;
writeln('A tomb elemeinek szorasa: ',szoras:5:2);
end.
```

3.

```
    type vtomb=array[1..n] of real;
     mintomb=array[1..3] of real;
     indextomb=array[1..3] of integer;
procedure min3ker(a:vtomb;n:integer;var min3:mintomb;
                                    var minind:indextomb);
 const max=1.0e+37;
 var i,j:integer;
 begin
 for i:=1 to 3 do
   begin
   min3[i]:=max;
   for j:=1 to n do if a[j]<=min3[i] then begin
                                          min3[i]:=a[j];
                                          minind[i]:=j;
                                          end;
   a[minind[i]]:=max;
   end;
 end;
```

4.

```
program p8_4;
const n=100;
{$i 8_3.pas}
var i,vsz:integer;
    eredmeny:vtomb;
    nev:array[1..n] of string[30];
    elsok:mintomb;
    rsz:indextomb;
begin
clrscr;
vsz:=0;
while (vsz<4) or (vsz>100) do begin
                write('Kerem a versenyzok szamat
                                        (3<vsz<=100) : ');
                readln(vsz);
                end;
{ --- eredmenyek bekerese --- }
writeln;
for i:=1 to vsz do
  begin
  write('Kerem a ',i,' rajtszamu versenyzo nevet
                                        (max 30 kar): ');
  readln(nev[i]);
  write('Kerem a ',i,' rajtszamu versenyzo eredmenyet: ');
  readln(eredmeny[i]);
  end;
{ --- az elso harom versenyzo keresese --- }
min3ker(eredmeny,vsz,elsok,rsz);
{ --- a helyezesek kiiratasa --- }
writeln;
writeln('A dobogos versenyzok:');
for i:=1 to 3 do begin
                 write(i,'. ',nev[rsz[i]],'(',rsz[i]);
                 writeln('), eredmenye: ',
                                      eredmeny[rsz[i]]:6:2);
                end;
end.
```

5.

```
type tomb=array[1..n] of integer;
procedure kiiratas(var a:vtomb ;n,k:integer);
var i,j,h:integer;
begin
h:=80 div k;
for i:=1 to n div k do
  begin
```

```
   for j:=0 to k-1 do write(a[(i-1)*k+j]:h);
   if (80 mod h) <>0 then writeln;
   end;
 if (80 mod h) <> 0 then
   begin
   for i:=1 to n mod k do write(a[n-(n mod k)+i]:h);
   writeln;
   end;
end;
```

6.

```
type vtomb=array[1..n] of real;
procedure tombfordit(var t:vtomb;n:integer);
var i:integer;
    cs:real;
begin
for i:=1 to (n div 2) do begin
                         cs:=t[i];
                         t[i]:=t[n+1-i];
                         t[n+1-i]:=cs;
                         end;
end;
```

7.

```
program p8_7;
const n=30;
type tomb=array[1..n] of integer;
var v:tomb;
    i,esz,rsz:integer;
{ ---- egy hellyel balra leptetes ---- }
procedure rotalas(var a:tomb;n:integer);
var i,cs:integer;
begin
cs:=a[1];
for i:=2 to n do a[i-1]:=a[i];
a[n]:=cs;
end;
{ ---------------------------------------- }
begin
esz:=0;
while esz<1 do begin
                write('Kerem a tomb elemeinek szamat
                                        (1<esz<=100) : ');

                readln(esz);
                end;
writeln;
{ ---- a tomb elemeinek bekerese ---- }
for i:=1 to esz do begin
                   write('Kerem az ',i,'. elem erteket: ');
```

```
                    readln(v[i]);
                    end;
writeln;

for i:=1 to esz do write(v[i],' ');
writeln;
{ --- a forgatas vegrehajtasa --- }
for i:=1 to 3 do rotalas(v,esz);

for i:=1 to esz do write(v[i],' ');
writeln;
end.
```

8.

```
    program p8_7;
const n=30;
type tomb=array[1..n] of integer;
var v:tomb;
    i,rtsz,esz,rsz:integer;
{ --- k hellyel balra leptetes --- }
procedure rotalas(var a:tomb;n:integer;k:integer);
var i,j,cs:integer;
begin
for j:=1 to k do begin
                 cs:=a[1];
                 for i:=2 to n do a[i-1]:=a[i];
                 a[n]:=cs;
                 end;
end;
{ ---------------------------------- }
begin
esz:=0;
rtsz:=0;
while esz<1 do begin
               write('Kerem a tomb elemeinek szamat
                                          (1<esz<=100) : ');
               readln(esz);
               end;
while rtsz<1 do begin
                write('Kerem a rotalasok szamat : ');
                readln(rtsz);
                end;
writeln;
{ --- a tomb elemeinek bekerese --- }
for i:=1 to esz do begin
                   write('Kerem az ',i,'. elem erteket: ');
                   readln(v[i]);
                   end;
writeln;
```

```
{ --- kiirasok --- }
writeln('Az eredeti tomb:');
for i:=1 to esz do write(v[i],' ');
writeln;
{ --- a forgatas vegrehajtasa --- }
rotalas(v,esz,rtsz);
writeln('Az elforgatott tomb:');
for i:=1 to esz do write(v[i],' ');
writeln;
end.
```

Egy másik megoldás:

```
    program p8_7;
const n=30;
type tomb=array[1..n] of integer;
var v:tomb;
    i,rtsz,esz,rsz:integer;
{ --- tomb [n,k] elemeinek forditasa --- }
procedure tombfordit(var t:tomb;n,k:integer);
var i:integer;
    cs:integer;
begin
for i:=0 to ((k-n) div 2)-1 do begin
                               cs:=t[n+i];
                               t[n+i]:=t[k-i];
                               t[k-i]:=cs;
                               end;
end;
{ --- k hellyel balra leptetes --- }
procedure rotalas(var a:tomb;n,k:integer);
var i,j,cs:integer;
begin
tombfordit(a,1,k-1);
tombfordit(a,1,n);
tombfordit(a,1,n-k);
end;
{ ---------------------------------- }
begin
esz:=0;
rtsz:=0;
while (esz<1) or (esz>n) do begin
                write('Kerem a tomb elemeinek szamat
                                    (1=<esz<=',n,') : ');
                readln(esz);
                end;
 while (rtsz<1) or (rtsz>esz) do begin
                 write('Kerem a rotalasok szamat
                                    (1<=rsz<=',esz,') : ');
```

```
                    readln(rtsz);
                    end;
writeln;
{ --- a tomb elemeinek bekerese --- }
for i:=1 to esz do begin
                    write('Kerem az ',i,'. elem erteket: ');
                    readln(v[i]);
                    end;
writeln;
{ --- kiirasok --- }
writeln('Az eredeti tomb:');
for i:=1 to esz do write(v[i],' ');
writeln;
{ --- a forgatas vegrehajtasa --- }
rotalas(v,esz,rtsz);
writeln('Az elforgatott tomb:');
for i:=1 to esz do write(v[i],' ');
writeln;
end.
```

9.

```
program p8_9;
const n=30;
type vtomb=array[1..n] of real;
var ki,vi,i,j:integer;
    t:vtomb;
    ossz,max:real;
{ --- osszegzo eljaras a tomb [n,n+k] elemeihez --- }
function tosszeg(a:vtomb;n,k:integer):real;
var i:integer;
    ossz:real;
begin
ossz:=0;
for i:=n to n+k do ossz:=ossz+a[i];
tosszeg:=ossz
end;
{ ------------------------------------------------ }
begin
max:=-1e37;
for i:=1 to n do begin
                 t[i]:=random(100)-random(100);
                 write(t[i]:10:0);
                 end;
writeln;
for i:= 1 to n do
  begin
  for j:=1 to n-i+1 do
    begin
    ossz:=tosszeg(t,j,i-1);
    if ossz>max then begin
                     max:=ossz;
```

```
                        ki:=j;
                        vi:=j+i-1;
                        end;
      end;
    end;
  writeln(max:12:2,ki:5,vi:5);

  end.
```

10.

```
    program p8_10;
  const n=30;
  type vtomb=array[1..n] of real;
  procedure trendez(var a:vtomb;n:integer);
  var i,j:integer;
      cs:real;
      csere volt:boolean;
  begin
  csere volt:=true;
  i:=1;
  while (i<n) and (csere volt) do
      begin
      csere volt:=false;
      for j:=2 to n do if a[j]<a[j-1] then begin
                                          csere volt:=true;
                                          cs:=a[j-1];
                                          a[j-1]:=a[j];
                                          a[j]:=cs
                                          end;

      i:=i+1;
      end;
  end;
  { ------------------------------------------------- }
  var v:vtomb;
      i:integer;
  begin
  writeln( Az eredeti tomb : );
  for i:=1 to n do begin
                   v[i]:=random(100)-random(100);
                   write(v[i]:10:0);
                   end;
  writeln(chr(10),chr(13), A rendezett tomb : );
  trendez(v,n);
  for i:=1 to n do begin
                   write(v[i]:10:0);
                   end;
  end.
```

11.

```
      const n=30;
  type vtomb=array[1..n] of real;
  function novekvo(a:vtomb;n:integer):boolean;
  var i:integer;
      kisebb:boolean;
  begin
  kisebb:=false;
  i:=2;
  while not kisebb  and (i<=n) do
    begin
    if a[i]<a[i-1] then kisebb:=true;
    i:=i+1
    end;
  novekvo:= not kisebb
  end;
  { ---------------------------------- }
  var t:vtomb;
      i:integer;
  begin
  for i:=1 to n do begin t[i]:=i; write(t[i]:10:0);end;
  writeln(novekvo(t,n));
  end.
```

12.

```
procedure rendez(var x:tombtipus;n:integer);
  var i:integer;

  function minindex(var tomb:tombtipus;
                    kezdind,vegind:indextipus):indextipus;

     var minelem:elemtipus;
         i,minind:indextipus;

    begin
      minind:=kezdind;
      minelem:=tomb[minind];
      for i:=succ(kezdind) to vegind do
        if tomb[i]<minelem then begin
          minelem:=tomb[i];
          minind:=i;
          end;
      minindex:=minind;
      end;

  procedure csere(var x,y: integer);
    var t:integer;
    begin
      t:=x;
```

```
    x:=y;
    y:=t;
    end;

begin
  for i:=1 to n do
    csere(x[i], x[minindex(x,i,n)]);
  end;
```

13.

```
const nmax = 100;
type stype = string[30];
     stomb = array[1..nmax] of stype;
{ --- minimum kivalasztasos rendezes --- }
procedure srendez(var t:stomb;n:integer);
var i,j,minind:integer;
    cs:stype;
begin
for i:=2 to n do
  begin
  minind:=i-1;
  for j:=i to n do if t[j]<t[minind] then minind:=j;
  if minind<>i-1 then begin
                      cs:=t[i-1];
                      t[i-1]:=t[minind];
                      t[minind]:=cs;
                      end;
  end;
end;
{ ---------------------------------------- }
var i,n:integer;
    nevek:stomb;
begin
clrscr;
n:=0;
while (n<1) or (n>nmax) do begin
                           write('Kerem a nevek szamat
                                  (1-',nmax,'): ');
                           readln(n);
                           end;
writeln;
for i:=1 to n do begin
                 write(i,'. nev: ');
                 readln(nevek[i]);
                 end;
srendez(nevek,n);
clrscr;
writeln('A nevek abc sorrendben:');
for i:=1 to n do writeln(nevek[i]);
end.
```

14.

```
program p8_14;
const nmax = 100;
type valasztype = string[8];
     valaszok = array[1..nmax] of valasztype;
{ --- egy valasz rendezese buborekos modszerrel ---}
procedure valaszrend(var v:valasztype);
var i,j,ind,h:integer;
    cs:valasztype;
begin
h:=sizeof(v)-1;
for i:=1 to h do
  begin
  ind:=i;
  for j:=i to h do if v[j]<v[ind] then ind:=j;
  cs:=v[i];
  v[i]:=v[ind];
  v[ind]:=cs;
  end;
end;
{ --- minimum kivalasztasos rendezes  --- }
procedure srendez(var t:valaszok;n:integer);
var i,j,minind:integer;
    cs:valasztype;
begin
for i:=2 to n do
  begin
  minind:=i-1;
  for j:=i to n do if t[j]<t[minind] then minind:=j;
  if minind<>i-1 then begin
                      cs:=t[i-1];
                      t[i-1]:=t[minind];
                      t[minind]:=cs;
                      end;
  end;
end;
{ ---------------------------------------- }
var i,j,f,ind,n:integer;
    v:valaszok;
begin
clrscr;
n:=0;
while (n<2) or (n>nmax) do begin
                           write('Kerem a felmeresek
                                  szamat (2-',nmax,'): ');
                           readln(n);
                           end;
clrscr;buflen:=8;
for i:=1 to n do begin
                 write(i,'. felmeres: ');
                 readln(v[i]);
```

```
                    valaszrend(v[i]);
                    end;
  srendez(v,n);
  f:=1;
  ind:=1;
  for i:=2 to n do
    if v[i]<>v[ind] then begin
                         writeln('A(z) ',f,'. tipusbol
                                          ',i-ind,' db van.');
                         ind:=i;
                         f:=f+1;
                         end;
  writeln('A(z) ',f,'. tipusbol ',i-ind+1,' db van.');
  end.
```

15.

```
    function ado(brutto:integer):integer;
  const savszam=7;
  type stomb=array[0..savszam] of integer;
       szazltomb=array[0..savszam] of real;
  const sav:stomb=(0,55000,70000,100000,150000,
                   240000,360000,600000);
        szalek:szazltomb=(0,0.17,0.23,0.29,0.35,
                          0.42,0.49,0.56);
  var ad:real;
      i:integer;
      kesz:boolean;
  begin
    ad:=0.0;
    i:=0;
    kesy:=false;
    repeat
      i:=i+1;
      if brutto>stomb[i] then  begin
        kesz:=true;
        ad:=ad+(stomb[i]-stomb[i-1])*szalek[i-1];
        brutto:=brutto-stomb[i];
        end
      else
        ad:=ad+brutto*szalek[i-1];
      until (i=savszam) or kesz;
    if not kesz then
      ad:=ad+brutto*szalek[savszam];
    ado:=round(ad);
    end;
```

16.

```
program p8_16;
const n=50;
      m=40;
      k=90;
type bazis = integer;
     tomb = array[1..n] of bazis;
     otomb = array[1..k] of bazis;
{ --- az osszevalogato eljaras --- }
procedure osszeval(t1:tomb; h1:integer;
                   t2:tomb; h2:integer;
                   var ot:otomb; th:integer);
var i1,j1,i,j,k:integer;
begin
writeln(h1:7,h2:7);
i:=1; j:=1; k:=1;
for k:=1 to h1+h2 do
  begin
  if (t1[i]<=t2[j]) or (j=h2) then begin
                                   ot[k]:=t1[i];
                                   if i<h1 then i:=i+1;
                                   end
                              else begin
                                   ot[k]:=t2[j];
                                   if j<h2 then j:=j+1;
                                   end;
  end;
th:=h1+h2;
end;
{ --- tombrendezo eljaras --- }
procedure rendez(var a:tomb;n:integer);
var i,j:integer;
    cs:bazis;
    csere_volt:boolean;
begin
csere_volt:=true;
i:=1;
while (i<n) and (csere_volt) do
    begin
    csere_volt:=false;
    for j:=2 to n do if a[j]<a[j-1] then begin
                                         csere_volt:=true;
                                         cs:=a[j-1];
                                         a[j-1]:=a[j];
                                         a[j]:=cs
                                         end;
    i:=i+1;
    end;
end;
{ ------------------------------------------------ }
var i,oh:integer;
```

```
      t1:tomb;
      t2:tomb;
      ot:otomb;
begin
randomize;
for i:=1 to n do t1[i]:=random(100);
for i:=1 to m do t2[i]:=random(100);
rendez(t1,n);
rendez(t2,m);
for i:=1 to n do write(t1[i]:5);writeln;
for i:=1 to m do write(t2[i]:5);writeln;
osszeval(t1,n,t2,m,ot,oh);
for i:=1 to n+m do write(ot[i]:5);writeln;
end.
```

17.

```
program p8_17;
const n=10;
type tomb=array[1..n] of integer;
var i,e,na,ki:integer;
    t:tomb;
begin
for i:=1 to n do t[i]:=random(100);
e:=random(n-1)+1;
na:=0; ki:=0;
for i:=1 to n do begin
                 if t[e]>t[i] then ki:=ki+1;
                 if t[e]<t[i] then na:=na+1;
                 end;
writeln('A vizsgalt elem a tomb ',e,'. eleme, erteke:
                                        ',t[e],' .');
writeln('A nagyobb elemek szama: ',na,' .');
writeln('A kisebb elemek szama: ',ki,' .');
end.
```

18.

```
program p8_18;
const n=12;
      p:array [1..n] of real = (1000,500,100,50,20,10,
                                5,2,1,0.5,0.2,0.1);
var ossz:real;
    i,db:integer;
begin
clrscr;
ossz:=0;
while (ossz<=0) or (ossz*100<>int(ossz*100)) do
  begin
  writeln('Kerem az osszeget: ');
```

```
  readln(ossz);
  end;
writeln;
writeln('Az osszeg az alabbiak szerint fizetheto ki:');
for i:=1 to n do
  begin
  db:=round(int(ossz/p[i]));
  ossz:=ossz-db*p[i];
  writeln('    ',db,' darab ',p[i]:7:1,' Ft-os');
  end;
end.
```

19.

```
  type kdtomb = array[1..n,1..m] of bazis;
  procedure sorcsere(var t:kdtomb;k,l:integer);
  var i:integer;
      cs:bazis;
  begin
  for i:=1 to m do begin
                   cs:=t[k,i];
                   t[k,i]:=t[l,i];
                   t[l,i]:=cs;
                   end;
  end;
```

20.

```
  type matrix=array[1..n,1..n] of bazis;
  procedure mtranszp(var m:matrix;n:integer);
  var i,j:integer;
      cs:bazis;
  begin
  for i:=2 to n do
    for j:=1 to i-1 do begin
                       cs:=m[i,j];
                       m[i,j]:=m[j,i];
                       m[j,i]:=cs;
                       end;
  end;
```

21.

```
  program p8_21;
  const s=10;
        o=15;

  type bazis = integer;
       tomb = array[1..s,1..o] of bazis;
       sor  = array[1..s] of bazis;
```

```
     oszl = array[1..o] of bazis;

var i,j:integer;
    t:tomb;
    sm:sor;
    om:oszl;

begin
for i:=1 to s do
  for j:=1 to o do
    t[i,j]:=random(5);
{ --- sorok osszegzese ---}
for i:=1 to s do begin
                 sm[i]:=0;
                 for j:=1 to o do
                   sm[i]:=sm[i]+t[i,j];
                 end;
{ --- oszlopok osszegzese ---}
for i:=1 to o do begin
                 om[i]:=0;
                 for j:=1 to s do
                   om[i]:=om[i]+t[j,i];
                 end;
for i:=1 to s do
  begin for j:=1 to o do
    write(t[i,j]:4);
    writeln(sm[i]:7);
    end;
for i:=1 to o do
  write(om[i]:4);
writeln
end.
```

22.

```
program p8_22;
const s=10;
      o=15;

type tomb=array[1..s,1..o] of integer;
     sor=array[1..s] of integer;

var i,j,smima:integer;
    smi:sor;
    t:tomb;

begin
for i:=1 to s do
  for j:=1 to o do
    t[i,j]:=random(50);

for i:=1 to s do begin
```

```
                    smi[i]:=32767;
                    for j:=1 to o do
                      if smi[i]>t[i,j] then
                        smi[i]:=t[i,j];
                    end;
  for i:=1 to s do begin
    for j:=1 to o do
      write(t[i,j]:4);
      writeln(smi[i]:7);
      end;
  smima:=-maxint-1;
  for i:=1 to s do
    if smima<smi[i] then
      smima:=smi[i];
  writeln('A minimalis elemek maximuma: ',smima);
  end.
```

23.

```
program p8_23;
  const sor=4;
        oszl=4;
  type matrix=array[1..sor,1..oszl] of integer;
  var i,j,k,jind,min:integer;
      van,ok:boolean;
  const a:matrix=((1,2,3,4),(8,7,9,8),(2,5,3,4),(5,6,3,9));

  begin
    van:=false;

    for i:=1 to sor do begin
      min:=a[i,1];
      jind:=1;

      for j:=2 to oszl do begin
        if min>a[i,j] then begin
          min:=a[i,j];
          jind:=j;
          end;
        end;
      ok:=true;
      k:=1;

      while ok and (k<=sor) do begin
        ok:=ok and (min>=a[k,jind]);
        k:=k+1;
        end;

      if ok then begin
        van:=true;
        writeln('Az ',i:2,'-edik sor ',jind:2,'-edik eleme.');
```

```
      end;
    end;

  if not van then
   writeln('Nincs ilyen elem');
  end.
```

24.

```
procedure btranszf(var a:matrix;n,r,s:integer);
var i,j:integer;
begin
q:=a[r,s];
if q<>0 then begin
  for j:=1 to n do begin
    d[j]:=a[r,j]/q;
    a[r,j]:=d[j];
    end;
  for i:=n downto 1 do
    if i<>r then begin
      b[i]:=a[i,s];
      for j:=n downto 1 do
        if j<>s then
          a[i,j]:=a[i,j]-b[i]*d[j];
      end;
  for i:=1 to n do
    a[i,s]:=0;
  a[r,s]:=1;
  end;
  end;
```

25.

```
program inverzio;
const n=3;
type matrix = array[1..n,1..n] of real;
     vektor = array[1..n] of real;
var d,e:matrix;
    i,j:integer;
    sor,oszl:integer;

procedure inverz(var a,ai:matrix;n:integer);
  var i,j,r,s:integer;
      g:real;
      d,di,b:vektor;

procedure btranszf(var a:matrix;n,r,s:integer);
var i,j:integer;
begin
g:=a[r,s];
```

```
if g<>0 then begin
  for j:=1 to n do begin
    d[j]:=a[r,j]/g;
    a[r,j]:=d[j];
    end;
  for i:=n downto 1 do
    if i<>r then begin
      b[i]:=a[i,s];
      for j:=n downto 1 do
        if j<>s then
          a[i,j]:=a[i,j]-b[i]*d[j];
      end;
  for i:=1 to n do
    a[i,s]:=0;
  a[r,s]:=1;
  end;
  end;

  procedure etranszf;
    begin
      for j:=1 to n do
        di[j]:=ai[r,j]/g;
      for i:=1 to n do
        for j:=1 to n do
          ai[i,j]:=ai[i,j]-b[i]*di[j];
      for j:=1 to n do
        ai[r,j]:=di[j];
      end;

 begin
   for i:=1 to n do begin
     for j:=1 to n do
       ai[i,j]:=0.0;
     ai[i,i]:=1.0;
     end;
   for r:=1 to n do begin
     g:=a[r,r];
     if g=0.0 then begin
       writeln('NEM TUDOM FOLYTATNI');
       halt;
       end;
     btranszf(a,n,r,r);
     etranszf;
     end;
   end;

begin
  for i:=1 to n do
    for j:=1 to n do
      readln(d[i,j]);
```

```
  writeln;
  inverz (d,e,n);
  for i:=1 to n do begin
    for j:=1 to n do
      write(e[i,j]:8:2);
    writeln;
    end;
  end.
```

26.

```
program megold;
const n=3;
type matrix = array[1..n,1..n] of real;
     vektor = array[1..n] of real;
var a,e,b,x:matrix;
    i,j:integer;
    sor,oszl:integer;

procedure matrixszorzas(var a,b,szorzat:matrix;
                    sor1,oszl1,sor2,oszl2:integer);

  var i,j,k:integer;
      z:real;

  begin
    if oszl1<>sor2 then begin
      writeln('Hiba ! Kiserlet nem konformabilis ');
      writeln('matrixok szorzasara ');
      halt;
      end
    else begin
      for i:=1 to sor1 do
        for j:=1 to oszl2 do begin
          z:=0.0;
          for k:=1 to oszl1 do
            z:=z+a[i,k]*b[k,j];
          szorzat[i,j]:=z;
        end;
      end;
    end;


procedure inverz(var a,ai:matrix;n:integer);
  var i,j,r,s:integer;
      q:real;
      d,di,b:vektor;

procedure btranszf(var a:matrix;n,r,s:integer);
```

```
var i,j:integer;
begin
g:=a[r,s];
if g<>0 then begin
  for i:=1 to n do begin
    d[j]:=a[r,j]/g;
    a[r,j]:=d[j];
    end;
  for i:=n downto 1 do
    if i<>r then begin
      b[i]:=a[i,s];
      for j:=n downto 1 do
        if j<>s then
          a[i,j]:=a[i,j]-b[i]*d[j];
      end;
  for i:=1 to n do
    a[i,s]:=0;
  a[r,s]:=1;
  end;
  end;

  procedure etranszf;
    begin
      for j:=1 to n do
        di[j]:=ai[r,j]/g;
      for i:=1 to n do
        for j:=1 to n do
          ai[i,j]:=ai[i,j]-b[i]*di[j];
      for j:=1 to n do
        ai[r,j]:=di[j];
      end;

 begin
   for i:=1 to n do begin
     for j:=1 to n do
       ai[i,j]:=0.0;
     ai[i,i]:=1.0;
     end;
   for r:=1 to n do begin
     g:=a[r,r];
     if g=0.0 then begin
       writeln('NEM TUDOM FOLYTATNI');
       halt;
       end;
     btranszf(a,n,r,r);
     etranszf;
     end;
   end;

begin
  clrscr;
  writeln('Kerem az egyenletrendszer egyutthatoit!');
  for i:=1 to n do
```

```
    for j:=1 to n do begin
      write('a[',i,',',j,']=');
      readln(a[i,j]);
      end;
  writeln;
  inverz (a,e,n);
  writeln('Kerem a konstansokat! ');
  for i:=1 to n do begin
    write('b[',i,']=');
    readln(b[i,1]);
    end;
  matrixszorzas(e,b,x,n,n,n,1);
  writeln('Az egyenletrendszer gyokei:');
  writeln;
  for i:=1 to n do
      writeln('x[',i,']=',x[i,1]:8:2);
    writeln;
  end.
```

27.

```
program antiszimmetrikus;
  const max = 3;
        h = 3; {h=max*(max-1) div 2}
  type  vtipus = array[1..h] of real;
  var avm:vtipus;
      sor,oszlop:integer;

  function a(x:vtipus;i,j:integer):real;
    var t:integer;
        jel:integer;
        elem:real;
    begin
      if i=j then
        elem:=0.0
      else begin
        jel:=1;
        if i<j then begin
          t:=i;
          i:=j;
          j:=t;
          jel:=-1;
          end;
        elem:=avm[(i-1)*(i-2) div 2 + i];
        elem:=jel*elem;
        end;
      a:=elem;
      end;

  procedure beolvas(var x:vtipus;n:integer);
    var i,j,k:integer;
```

```
    begin
      for i:=2 to n do
        for j:=1 to i-1 do begin
          write('a[',i,',',j,']=');
          k:=(i-2)*(i-1) div 2+j;
          readln(x[k]);
          end;
      end;
  begin
    clrscr;
    beolvas(avm,max);
    writeln;
    write('sorindex=');
    readln(sor);
    write('oszlopindex=');
    readln(oszlop);
    writeln;
    writeln('a[',sor,',',oszlop,']=',a(avm,sor,oszlop):6:2);
    end.
```

28.

```
program felso haromszog;
  const max = 3;
        h = 3; {h=max*(max-1) div 2}
  type  vtipus = array[1..h] of real;
  var avm:vtipus;
      sor,oszlop:integer;

  function a(x:vtipus;i,j,n:integer):real;
    var t:integer;
        elem:real;
    begin
      if i>=j then
        elem:=0.0
      else
        elem:=avm[(2*n-i)*(i-1) div 2 + j-i];
      a:=elem;
      end;

  procedure beolvas(var x:vtipus;n:integer);
    var i,j,k:integer;
    begin
      for i:=1 to n-1 do
        for j:=i+1 to n do begin
          write('a[',i,',',j,']=');
          k:=(2*n-i)*(i-1) div 2+j-i;
          readln(x[k]);
          end;
      end;
  begin
    clrscr;
```

```
beolvas(avm,max);
writeln;
write('sorindex=');
readln(sor);
write('oszlopindex=');
readln(oszlop);
writeln;
writeln('a[',sor,',',oszlop,']=',a(avm,sor,oszlop,max):6:2);
end.
```

29.

```
program ritkamatrix;
 const meret=6;
       maxnem0=20;
 type indttipus=array[1..meret] of integer;
      erttipus=array[1..maxnem0] of real;
      oitipus=array[1..maxnem0] of integer;

 var rt:indttipus;
     oit:oitipus;
     et:erttipus;
     max,i,j,k:integer;
     sor,oszlop:integer;

 function R(sor,oszlop:integer):real;
 var i,q,qk,qv:integer;
     megvan: boolean;

 begin
   megvan:=false;
   R:=0.0;

   {oszlopindex kereseshez az oit-beli
    kezdoindex beallitasa}

  qk:=rt[sor];

  {ha az adott sorban nincs nemzerus elem,
   0.0-val visszaler}

  if qk=0 then exit;

  {vegindex beallitasa}

  i:=0;
  repeat
    i:=i+1;
    qv:=rt[sor+i];
    until qv<>0;
  qv:=qv - 1;
  q:=qk;
```

```
  while (q<=qv) and not megvan do
    if oit[q]=oszlop then begin
      R:=et[q];
      megvan:=true;
      end
    else
      q:=q+1;
  end;


begin
  j:=1;
  max:=meret-1;
  for i:=1 to meret do
    rt[i]:=0;
  for i:=1 to maxnem0 do
    oit[i]:=0;
  for sor:=1 to max do begin
    writeln('sor=',sor);
    rt[sor]:=j;
    repeat
      write('oszlop=');
      readln(oszlop);
      if oszlop<>0 then begin
        write('elem[',sor,',',oszlop,']=');
        readln(et[j]);
        oit[j]:=oszlop;
        j:=j+1;
        end;
      until oszlop=0;
    if rt[sor]=j then
      rt[sor]:=0;
    end;
  rt[meret]:=j;
  for i:=1 to max do begin
    writeln;
    for j:=1 to max do
      write(R(i,j):5:1);
    end;
  writeln;
  writeln;
  writeln;
  for i:=1 to meret do
    writeln(rt[i]:2,'       ,oit[i]:2,    :    ,et[i]:4:1);
  i:=meret+1;
  repeat
    writeln('       ',oit[i]:2,'   :   ',et[i]:4:1);
    i:=i+1;
    until oit[i]=0;
  end.
```

9. fejezet

1.

```
type betuhalmaz = set of 'A'..'Z';

var H1, H2: betuhalmaz;
```

2.

```
['A','C','D']
['B']
true
true
true
```

3.

```
H:=H+[1];
L:=H = [1]+[2];
```

4.

```
program elemek;
  type bset=set of 'A'..'Z';
  var halmaz:bset;
      i:char;
  begin
    clrscr;
    writeln('Adja meg a halmaz elemeit!');
    writeln('Vegjel a * karakter.');
    halmaz:=[];
    repeat
      write(': ');
      readln(i);
      if (i>='A') and (i<='Z') then
        halmaz:=halmaz+[i];
      until i='*';
    writeln;
    writeln('A halmaz elemei:');
    writeln;
    write('[');
    for i:='A' to 'Z' do
      if i in halmaz then
        write(i,' ');
    writeln(']');
    end.
```

5. A for ciklus és az in reláció segítségével, mint a 6. feladat megoldásában.

6.

```
program forall;
  type numset=set of byte;
  var szhalmaz:numset;
      i:byte;
      s,sum:integer;
  begin
    clrscr;
    writeln('Kerem a (nemnegativ) szamokat');
    writeln('A vegjel=negativ szam');
    szhalmaz:=[];
    repeat
      write(': ');
      readln(s);
      if (s>=0) and (s<=255) then
        szhalmaz:=szhalmaz+[s];
      until s<0;
    sum:=0;
    for i:=0 to 255 do
      if i in szhalmaz then
        sum:=sum+i;
    writeln;
    writeln('Az osszeg ',sum);
    end.
```

7.

```
program italok;
  const n=21;
  type italtip=(viz,tea,kave,tej,kola,citromle,portoriko rum,
                gin,ananaszle,mecseki,meggyle,martini,vermut,
                maraschino,cherry brandy,triple sec,brandy,
                szamorodni,barackpalinka,whisky,oporto,sor);
       keszlettip=set of italtip;
       nevtip=string[30];
       italnevtip=array[0..n] of nevtip;
  const royal:keszlettip =
                [cherry brandy,gin,martini,maraschino];
        aphrodite:keszlettip =
                [triple sec,gin,brandy,martini];
        puszta:keszlettip =
                [szamorodni,barackpalinka,mecseki];
        baccardy:keszlettip =
                [portoriko rum,triple sec,citromle];
        wembley:keszlettip =
                [whisky,vermut,ananaszle];
        italnev:italnevtip =
                ('viz','tea','kave','tej','kola','citromle',
                'portoriko rum','gin','ananaszle','mecseki',
```

```
                    'meggyle','martini','vermut','maraschino',
                    'cherry brandy','triple-sec','brandy',
                    'szamorodni','barackpalinka','whisky',
                    'oporto','sor');
  var keszlet: keszlettip;
      i:integer;

      van:char;
      ital:italtip;
  begin
    clrscr;
    keszlet:=[];
    writeln('Adja meg az italkeszletet!');
    for i:=0 to n do begin
      repeat
        gotoxy(5,5);
        clreol;
        gotoxy(5,5);
        write(italnev[i],  van? (i/n): ');
        readln(van);
        until (upcase(van)='I') or (upcase(van)='N');
      if upcase(van)='I' then begin
        ital:=italtip(i);
        keszlet:=keszlet+[ital];
        end;
      end;
    writeln;
    writeln('A keszitheto koktelok:');
    if royal<=keszlet then
      writeln('         royal');
    if aphrodite<=keszlet then
      writeln('         aphrodite');
    if puszta<=keszlet then
      writeln('         puszta');
    if baccardy<=keszlet then
      writeln('         baccardy');
    if wembley<=keszlet then
      writeln('         wembley');
    end.
```

8.

```
program szamlalas;
  const sorsz=5;
  type strings=string[79];
       szovtip=array[1..sorsz] of strings;
       chset=set of char;
       chset tomb=array[1..3] of chset;
       stip=array[1..3] of integer;
  const fajtak:chset tomb =
                ([ 'A'..'Z' ],[ 'a'..'z' ],
                 [ '.',',',';',':','?','!','"','''' ]);
  var szoveg:szovtip;
```

```
      sor:string;
      s:stip;
      i,j,k,osszes:integer;
  begin
    clrscr;
    for i:=1 to 3 do
      s[i]:=0;
    osszes:=0;
    writeln('Irj be ',sorsz,' sor szoveget:');
    for i:=1 to sorsz do
      readln(szoveg[i]);
    for i:=1 to sorsz do begin
      sor:=szoveg[i];
      osszes:=osszes+length(sor);
      for j:=1 to length(sor) do
        for k:=1 to 3 do
          if sor[j] in fajtak[k] then
            s[k]:=s[k]+1;
    end;
   for k:=1 to 3 do
     osszes:=osszes-s[k];
   writeln;
   writeln('A jelek megoszlasa:');
   writeln('              nagybetu ',s[1]:3);
   writeln('              kisbetu  ',s[2]:3);
   writeln('              irasjel  ',s[3]:3);
   writeln('       spec. karakter  ',osszes:3);
   end.
```

9.

```
program ertekelo;
  const max=20;
  type nevtip=string[30];
       jtipus='A'..'C';
       jtomb=array[1..3] of jtipus;
       htipus=record
                nev:nevtip;
                jegy:jtomb;
                ertekeles:nevtip;
              end;
       jset=set of jtipus;
       htomb=array[1..max] of htipus;
  const jo:jset = ['A','B'];
        rossz:jset = ['B','C'];
        jegyh:jset =['A','B','C'];
  var hallg:htipus;
      hlista:htomb;
      jegyek:jset;
      i,j,k,szamuk:integer;
  procedure olvas(var x:htipus);
```

```
  var i:integer;
  begin
    clrscr;
    writeln('A vegjel nev=vege');
    writeln;writeln;
    with x do begin
      write('nev: ');
      readln(nev);
      for i:=1 to 3 do
        repeat
          gotoxy(1,i+4);
          clreol;
          write('jegy',i,': ');
          readln(jegy[i]);
          until jegy[i] in jegyh;
     end;
   end;
procedure kiir(x:htipus);
  var i:integer;
  begin
    with x do begin
      write(nev:30,'     ');
      for i:=1 to 3 do
        write(jegy[i],'  ');
      writeln(ertekeles);
      end;
    end;
begin
 repeat
   clrscr;
   write('A hallgatok szama: ');
   readln(szamuk);
   until (szamuk>0) and (szamuk<=max);
   for i:=1 to szamuk do
     olvas(hlista[i]);
   for i:=1 to szamuk do begin
     jegyek:='';
     with hlista[i] do begin
       for k:=1 to 3 do
         jegyek:=jegyek+jegy[k];
       if (jegyek<=jo) and ('A' in jegyek) then
         ertekeles:='jol megfelelt'
       else if (jegyek<=rossz) and ('C' in jegyek) then
         ertekeles:='nem felelt meg'
       else
         ertekeles:='megfelelt';
       end;
     end;
     clrscr;
     for i:=1 to szamuk do
       kiir(hlista[i]);
  end.
```

10. fejezet

1.

```
program ellenoriz;
  type str20=string[20];
       str30=string[30];
       str10=string[10];
       ttip=array[1..12] of 1..31;
       datrec= record
                 ev:1900..2000;
                 ho:1..12;
                nap:1..31;
                end;
       cimrec= record
                 varos:str20;
                 utca: str30;
                 hazszam:str10;
                 end;
       adatrec=record
                 nev: str20;
                 szhely:str20;
                 szul:datrec;
                 nevn:datrec;
                 cim: cimrec;
                 end;
  const adat:adatrec= (nev:'Kovacs';szhely:'Baja';
                       szul:(ev:1950;ho:8;nap:2);
                       nevn:(ev:1900;ho:2;nap:17);
                       cim:(varos:'Pecs';utca:'Kamilla';
                       hazszam:'2/c'));
        napok:ttip= (31,28,31,30,31,30,31,31,30,31,30,31);
  var jo:boolean;
{ ------------------------------------------------------------}
  procedure ellenorzes(x:datrec; var b:boolean);
    var szokoev:boolean;
    begin
      b:=true;
      with x do begin
        szokoev:=(ev mod 4 =0) and (ev mod 100<>0)
                               or (ev mod 400=0);
        if (ho<>2) or not szokoev then
          b:=nap<=napok[ho]
        else
          b:=nap<=29;
        end;
      end;
{ ------------------------------------------------------------}
  procedure datolvaso(var x:datrec);
```

```
    begin
      with x do begin
        write('evszam:');
        readln(ev);
        write('honap:');
        readln(ho);
        write('nap:');
        readln(nap);
        end;
      end;
{ ------------------------------------------------------------}
  begin
    writeln('szuletesi datum:');
    datolvaso(adat.szul);
    writeln;
    writeln('nevnap datuma:');
    datolvaso(adat.nevn);
    ellenorzes(adat.szul,jo);
    if not jo then
      writeln('hibas szuletesi adat');
    ellenorzes(adat.nevn,jo);
    if not jo then
      writeln('hibas nevnap');
    end.
```

2.

```
program lapkuldes;
  const maxind=20;
  type str20=string[20];
       str30=string[30];
       str10=string[10];
       ttip=array[1..12] of 1..31;
       datrec= record
                 ev:1900..2000;
                 ho:1..12;
                nap:1..31;
                end;
       cimrec= record
                varos:str20;
                utca: str30;
                hazszam:str10;
                end;
       adatrec=record
                nev: str20;
                szhely:str20;
                szul:datrec;
                nevn:datrec;
                cim: cimrec;
                end;
       tabltip=array[1..maxind] of adatrec;
```

```
  const napok:ttip= (31,28,31,30,31,30,31,31,30,31,30,31);
  var jo:boolean;
      haverok:tabltip;
      n,i:integer;
      honap:1..12;
{ ------------------------------------------------------}
  procedure ellenorzes(x:datrec; var b:boolean);
    var szokoev:boolean;
    begin
      b:=true;
      with x do begin
        szokoev:=(ev mod 4 =0) and (ev mod 100<>0)
                                  or (ev mod 400=0);
        if (ho<>2) or not szokoev then
          b:=(nap<=napok[ho]) and (ho<=12)
        else
          b:=nap<=29;
        end;
      end;
{ ------------------------------------------------------}
    procedure datolvaso(var y:datrec;var j:boolean);
      begin
        with y do begin
          write('EVSZAM:');
          readln(ev);
          write('HONAP:');
          readln(ho);
          write('NAP:');
          readln(nap);
          end;
        ellenorzes(y,j);
        end;
{ ------------------------------------------------------}
    procedure olvas(var x:adatrec;var jel:boolean);
      procedure cimolvaso(var y:cimrec);
        begin
          with y do begin
            write('VAROS:');
            readln(varos);
            write('UTCA:');
            readln(utca);
            write('HAZSZAM:');
            readln(hazszam);
            end;
          end;
      begin
        with x do begin
          write('NEV:');
          readln(nev);
          write('SZULETESI HELY:');
          readln(szhely);
          writeln('szuletesnap');
          datolvaso(szul,jel);
```

```
        writeln('nevnap');
        datolvaso(nevn,jel);
        write('cim');
        cimolvaso(cim);
        end;
      end;
{ ------------------------------------------------------------}
  procedure kiiro(x:adatrec);
    procedure datkiiro(y:datrec);
      begin
        with y do begin
          write('EVSZAM:');
          writeln(ev);
          write('HONAP:');
          writeln(ho);
          write('NAP:');
          writeln(nap);
          end;
        end;
    procedure cimiro(y:cimrec);
      begin
        with y do begin
          write(varos);
          write('  ');
          write(utca);
          write(' ');
          writeln(hazszam);
          end;
        end;
    begin
      with x do begin
        write('NEV:');
        writeln(nev);
        write('SZULETESI HELY:');
        writeln(szhely);
        writeln('szuletesnap');
        datkiiro(szul);
        writeln('nevnap');
        datkiiro(nevn);
        write('CIM:');
        cimiro(cim);
        end;
      end;
{ ------------------------------------------------------------}
    begin
      repeat
        clrscr;
        write('hanyat ir fel (<=',maxind,'):');
        readln(n);
        until (n>0) and (n<=maxind);
      for i:=1 to n do begin
        clrscr;
        repeat
```

```
        olvas(haverok[i],jo);
        until jo;
      end;
    clrscr;
    write('MELYIK HONAPRA KIVANCSI? (1..12):');
    readln(honap);
    for i:=1 to n do begin
      clrscr;
      with haverok[i] do
        if (szul.ho=honap) or (nevn.ho=honap) then
          kiiro(haverok[i]);
      while not keypressed do;
      end;
    end.
```

3. Lásd a 4. feladat megoldását!

4.

```
  type string20=string[20];
       string11=string[11];
       berrec=record
                szemsz: string11;
                nev: string20;
                oraber: real;
                atloraber: real;
                ledolgora: integer;
                fizetett tavol:integer;
                nemfiz tavol:integer;
                tappenz:integer;
                potlek: integer;
                csaladip: integer;
                szaksztag: boolean;
                letiltas: integer;
                cseb: integer;
                ev:1980..2000;
                honap:1..12;
                end;

  var dolgozo: berrec;

{ ---------------------------------------}

  procedure olvas(var x: berrec);
    var sz:char;
    begin
      clrscr;
      with x do begin
        write('ev: ');
        readln(ev);
        write('honap: ');
```

```
      readln(honap);
      write('nev: ');
      readln(nev);
      write('szem.szam: ');
      readln(szemsz);
      write('oraber: ');
      readln(oraber);
      write('atlagoraber: ');
      readln(atloraber);
      write('ledolgozott orak: ');
      readln(ledolgora);
      write('fizetett tavollet: ');
      readln(fizetett tavol);
      write('fiz. nelkuli tavollet: ');
      readln(nemfiz tavol);
      write('tappenz: ');
      readln(tappenz);
      write('tulora potlek: ');
      readln(potlek);
      write('csaladi potlek: ');
      readln(csaladip);
      repeat
        write('szakszervezeti tag? (I/N) ');
        readln(sz)
        until (upcase(sz)='I') or (upcase(sz)='N');
      if upcase(sz)='I' then
        szakszтag:=true
      else
        szakszтag:=false;
      write('letiltas: ');
      readln(letiltas);
      write('cseb: ');
      readln(cseb);
      end;
    end;
```

5.

```
procedure szamfeit(var x:herrec);
  begin
    with x do begin
    jarber:=round(ledolgora*oraber);
    atlber:=round(atloraber*fizetett tavol);
    brutto:=jarber+atlber+tappenz+potlek;
    if szakszтag then
      szaksz:=round(brutto/100.0)
    else
      szaksz:=0;
    alap:=brutto-alkalm kedv-szaksz;
    nyugdij:=round(alap/10.0);
    brutto:=brutto+csaladip;
```

```
      if alap>round(adohatar/12.0) then
        ado:=round((alap-round(adohatar/12.0))*adokul(s)
      else
        ado:=0;
      osszlevon:=nyugdij+ado+szaksz+letiltas+cseb;
      marad:=brutto-osszlevon;
      end;
    end;

{ ------------------------------------------------}

  procedure kiir(x: berrec);
    begin
      clrscr;
      with x do begin
        writeln(ev,'.',honap,'.');
        writeln('NEV: ',nev,'     szem.szam: ',szemsz);
        writeln('oraber:       ',oraber:7:2);
        writeln('atlagoraber: ',atloraber:7:2);
        write('ledolgozott orak:       ',ledolgora,' ora');
        writeln('    ledolg. orak utani ber: ',jarber,' Ft');
        write('fizetett tavollet:      ',fizetett tavol);
        writeln(' ora   ',atlber,' Ft');
        writeln('fiz. nelkuli tavollet: ',nemfiz tavol,' ora');
        write('tappenz: ',tappenz,' Ft   ');
        writeln('tulora potlek: ',potlek,' Ft');
        writeln('adoalap: ',alap,' Ft');
        writeln('csaladi potlek: ',csaladip,' Ft');
        writeln('---------------------------------------');
        writeln('brutto jovedelem: ',brutto,' Ft');
        writeln('---------------------------------------');
        write('adoeloleg: ',ado,' Ft   ');
        writeln('nyugdijjarulek: ',nyugdij,' Ft');
        write('szakszervezeti tagdij: ',szaksz,' Ft   ');
        write('letiltas: ',letiltas,' Ft   ');
        writeln('cseb: ',cseb,' Ft');
        writeln('---------------------------------------');
        writeln('osszes levonas: ',osszlevon,' Ft');
        writeln('---------------------------------------');
        writeln;
        writeln('kifizetendo: ',marad,' Ft');
        writeln('======================================= ');
        end;
      end;


6.
  type datumtip=record
                  ev:1980..2000;
                  ho:1..12;
                  nap:1..31;
                  end;
```

```
  procedure datumwrite(x:datumtip);
    begin
      with x do begin
        write(ev:4,'.',ho:2,'.',nap:2,'.');
        end;
      end;
```

7.

```
program teruletszamitas;

  type idomtipus=(negyzet,kor,teglalap,haromszog);

       meretrec=record
                  case alakzat:idomtipus of
                    negyzet   : (oldal:real);
                    kor       : (sugar:real);
                    teglalap  : (a_oldal,b_oldal:real);
                    haromszog: (a_old,b_old,c_old:real);
                    end;

  var meretek:meretrec;
      terulet,kerulet:real;

{ ------------------------------------------------------------}
  procedure meretolvasas(var x:meretrec);

    function alak:idomtipus;
      var alakjel:char;
      begin
        clrscr;
        writeln('Milyen sikidom teruletet szamolja?');
        writeln;
        writeln('negyzet      N');
        writeln('teglalap     T');
        writeln('haromszog    H');
        writeln('kor          K');
        readln(alakjel);
        case upcase(alakjel) of
          'N': alak:=negyzet;
          'T': alak:=teglalap;
          'H': alak:=haromszog;
          'K': alak:=kor;
          end;
        end;

    begin
      with x do begin
        alakzat:=alak;
        clrscr;
        case alakzat of
          negyzet: begin
```

```
                    write('Kerem a negyzet oldalat: ');
                    readln(oldal);
                    end;
          kor:      begin
                    write('Kerem a kor sugarat: ');
                    readln(sugar);
                    end;
      teglalap: begin
                    write('Kerem az a oldalt:  ');
                    readln(a oldal);
                    write('Kerem a b oldalt:   ');
                    readln(b oldal);
                    end;
    haromszog: begin
                    writeln('Kerem a haromszog oldalait:');
                    write('                    a= ');
                    readln(a old);
                    write('                    b= ');
                    readln(b old);
                    write('                    c= ');
                    readln(c old);
                    end;
      end;
    end;
  end;

{ ------------------------------------------------------------}

procedure szamitas(x:meretrec; var t,k:real);
  begin
    with x do begin
      case alakzat of
          negyzet: begin
                    t:=sqr(oldal);
                    k:=4.0*oldal;
                    end;
          kor:      begin
                    t:=sqr(sugar)*pi;
                    k:=2.0*sugar*pi;
                    end;
      teglalap: begin
                    t:=a oldal*b oldal;
                    k:=2.0*(a oldal+b oldal);
                    end;
    haromszog: begin
                    k:=(a old+b old+c old)/2.0;
                    t:=sqrt(k*(k-a old)*(k-b old)*(k-c old));
                    k:=2.0*k;
                    end;
        end;
    end;
  end;
```

```
{ ------------------------------------------------------------}

 begin
   clrscr;
   meretolvasas(meretek);
   szamitas(meretek.terulet,kerulet);
   writeln;
   writeln('Terulet= ',terulet:8:2);
   writeln('Kerulet= ',kerulet:8:2);
   end.
```

8.

```
 program datumrendezes;
   const maxind=20;
   type str20=string[20];
        str30=string[30];
        str10=string[10];
        ttip=array[1..12] of 1..31;
        datrec= record
                  ev:1900..2000;
                  ho:1..12;
                 nap:1..31;
                 end;
        cimrec= record
                  varos:str20;
                  utca: str30;
                  hazszam:str10;
                  end;
        adatrec=record
                  nev: str20;
                  szhely:str20;
                  szul:datrec;
                  nevn:datrec;
                  cim: cimrec;
                  end;
         tabltip=array[1..maxind] of adatrec;
   const napok:ttip= (31,28,31,30,31,30,31,31,30,31,30,31);
   var jo:boolean;
       haverok:tabltip;
       n,i:integer;
       honap:1..12;
 { ------------------------------------------------------------}
   procedure ellenorzes(x:datrec; var b:boolean);
     var szokoev:boolean;
     begin
       b:=true;
       with x do begin
         szokoev:=(ev mod 4 =0) and (ev mod 100<>0)
                                or (ev mod 400=0);
```

```
        if (ho<>2) or not szokoev then
          b:=(nap<=napok[ho]) and (ho<=12)
        else
          b:=nap<=29;
        end;
      end;
{ ------------------------------------------------------------}
    procedure datolvaso(var y:datrec;var j:boolean);
      begin
        with y do begin
          write('EVSZAM:');
          readln(ev);
          write('HONAP:');
          readln(ho);
          write('NAP:');
          readln(nap);
          end;
        ellenorzes(y,j);
        end;
{ ------------------------------------------------------------}
  procedure olvas(var x:adatrec;var jel:boolean);
    procedure cimolvaso(var y:cimrec);
      begin
        with y do begin
          write('VAROS:');
          readln(varos);
          write('UTCA:');
          readln(utca);
          write('HAZSZAM:');
          readln(hazszam);
          end;
        end;
    begin
      with x do begin
        write('NEV:');
        readln(nev);
        write('SZULETESI HELY:');
        readln(szhely);
        writeln('szuletesnap');
        datolvaso(szul,jel);
        writeln('nevnap');
        datolvaso(nevn,jel);
        write('cim');
        cimolvaso(cim);
        end;
      end;
{ ------------------------------------------------------------}
  procedure kiiro(x:adatrec);
    procedure datkiiro(y:datrec);
      begin
        with y do begin
          write('EVSZAM:');
          writeln(ev);
          write('HONAP:');
```

```
        writeln(ho);
        write('NAP:');
        writeln(nap);
        end;
      end;
    procedure cimiro(y:cimrec);
      begin
        with y do begin
          write(varos);
          write('  ');
          write(utca);
          write(' ');
          writeln(hazszam);
          end;
        end;
    begin
      with x do begin
        write('NEV:');
        writeln(nev);
        write('SZULETESI HELY:');
        writeln(szhely);
        writeln('szuletesnap');
        datkiiro(szul);
        writeln('nevnap');
        datkiiro(nevn);
        write('CIM:');
        cimiro(cim);
        end;
      end;
{ ------------------------------------------------------------}
  procedure csere(var x,y:adatrec);
    var w:adatrec;
    begin
      w:=x;
      x:=y;
      y:=w;
      end;

{------------------------------------------------------------}
  procedure rendez(var x:tabltip; n:integer);
    var i:integer;
        kesz: boolean;
    begin
      kesz:=false;
      while not kesz do begin
        kesz:= true;
        for i:=1 to n-1 do
          if (x[i].szul.ho>x[i+1].szul.ho) or
             ((x[i].szul.ho=x[i+1].szul.ho)  and
              (x[i].szul.nap>x[i+1].szul.nap)) then begin
            csere(x[i],x[i+1]);
            kesz:=false;
            end;
       end;
```

```
    end;
{-------------------------------------------------------------}
  begin
    repeat
      clrscr;
      write('hanyat ir fel (<=',maxind,'):');
      readln(n);
      until (n>0) and (n<=maxind);
    for i:=1 to n do begin
      clrscr;
      repeat
        olvas(haverok[i],jo);
        until jo;
      end;
    rendez(haverok,n);
    for i:=1 to n do begin
      clrscr;
      kiiro(haverok[i]);
      while not keypressed do;
      end;
    end.
```

9.

```
program ketfelolbuborek;
const
     nagysag = 10;

type
    elemtipus = record
                        szoveg : string[5];
                        szam   : integer;
                        kulcs  : integer;
                end;

    tabltipus= array [1..nagysag] of elemtipus;


var
   tablazat  : tabltipus;

procedure listaz( x : tabltipus;
                  n : integer);

var
   i : integer;

begin
     for i:=1 to n do
         writeln(x[i].kulcs:8);
```

```
     writeln;
end;

{-------------------------------------------------}

procedure buborekketfelol( var  x       : tabltipus;
                                meret   : integer;
                                novo    : integer    );

var
   i,j      : integer;
   cserevan : boolean;
{-------------------------------------------------}

procedure csere ( var   x, y : elemtipus );

var
   t    : elemtipus;

begin
     t:=x;
     x:=y;
     y:=t;
end;
{-------------------------------------------------}

   begin
        cserevan:=true;
        while cserevan do
        begin
             for i:=1 to meret-1 do
             begin
                 cserevan:=false;
                 for j:=meret downto i do
                 begin
                      if novo>0 then
                      begin
                           if x[i].kulcs > x[j].kulcs then
                           begin
                                csere(x[i],x[j]);
                                cserevan:=true;
                           end
                      end
                      else
                      begin
                           if x[i].kulcs < x[j].kulcs then
                           begin
                                csere(x[i],x[j]);
                                cserevan:=true;
                           end
                      end
                 end; {for j:}
             end; {for i:}
```

```
        end; {while}
    end;
{-------------------------------------------------------------}

procedure feltolt( var x : tabltipus;
                       n : integer);

var
   i : integer;

begin
     for i:=1 to n do
         x[i].kulcs:=random(100);
end;


begin
     clrscr;
     randomize;
     gotoxy(1, 1);
     clreol;
     write('Inicializalas...');
     feltolt(tablazat,nagysag);
     gotoxy(1, 1);
     clreol;
     writeln('Rendezetlen lista...');
     listaz(tablazat,nagysag);
     buborekketfelol(tablazat,nagysag,1);
     writeln('Rendezett lista...');
     listaz(tablazat,nagysag);
end.
```

11. fejezet

1. b és d.

2.

```
function elofordul(var f:intfile;i:integer):boolean;
  var k:integer;
      megvan:boolean;
  begin
    megvan:=false;
    if letezik(f) then begin
      reset(f);
      while not megvan and not eof(f) do begin
        read(f,k);
        if k=i then
```

```
          megvan:=true;
        end;
      close(f);
      end
    else
      writeln('A kerdeses file nincs a konyvtarban');
    elofordul:=megvan;
    end;
```

3.

```
program hany soros;
  var f:text;
      k:integer;

  function sorszam(var x:text):integer;
     var sorsz:integer;
         sor:string[127];
    begin
      sorsz:=0;
      reset(x);
      while not seekeof(x) do begin
        readln(x,sor);
        sorsz:=sorsz+1;
        end;
      close(x);
      sorszam:=sorsz;
      end;

  begin
    clrscr;
    assign(f,'11_3.pas');
    k:=sorszam(f);
    writeln('Az allomany ',k,' sorbol all.');
    end.
```

4.

```
program hany soros;
  var f:text;
      k:integer;

  function sorszam(var x:text):integer;
     var sorsz:integer;
         sor:string[127];
    begin
      sorsz:=0;
      reset(x);
      while not seekeof(x) do begin
        readln(x,sor);
        sorsz:=sorsz+1;  writeln(sorsz,'  ',sor);
```

```
      end;
    close(x);
    sorszam:=sorsz;
    end;

begin
  clrscr;
  assign(f,'11_3.pas');
  k:=sorszam(f);
  writeln('Az allomany ',k,' sorbol all.');
  end.
```

5.

```
program cserebere;
  type strings=string[79];
  var nev,regi,uj:strings;

function kovetkezo(szoveg,resz:strings;
                   honnan:integer):integer;

  var hol,i,j:integer;
      egyezik:boolean;

  begin
    hol:=-1;
    i:=honnan;
    while hol<0 do
      if i> length(szoveg)-length(resz)+1 then
        hol:=0
      else begin
        j:=1;
        egyezik:=true;
        while egyezik and (j<=length(resz)) do
          if szoveg[i+j-1]=resz[j] then
            j:=j+1
          else
            egyezik:=false;
       if egyezik then
         hol:=i
       else
         i:=i+1;
       end;
    kovetkezo:=hol;
    end;
  procedure helyettesit(var miben:strings;
                        mit,mivel:strings);

  var k:integer;
  begin
    k:=0;
```

6.

```
    repeat
      k:=kovetkezo(miben,mit,k+1);
      if k>0 then begin
        delete(miben,k,length(mit));
        insert(mivel,miben,k);
        end;
      until k=0;
    end;

  procedure kicserelo(allnev, mit,mire:strings);
    var g,f:text;
        sor:strings;
    begin
      assign(f,allnev);
      assign(g,'xxxxxxxx.xxx');
      reset(f);
      rewrite(g);
      while not eof(f) do begin
        readln(f,sor);
        helyettesit(sor,mit,mire);
        writeln(g,sor);
        end;
      close(f);
      close(g);
      erase(f);
      rename(g,allnev);
      end;

  begin
    clrscr;
    write('Kerem az allomany nevet:');
    readln(nev);
    write('cserelendo:');
    readln(regi);
    write('mire: ');
    readln(uj);
    kicserelo(nev,regi,uj);
    end.
```

6.

```
program alterjusag;
  const fidnev='fidesz.dat';
        madisznev='madisz.dat';
  type strings=string[79];
       adatrec=record
                 szemszam:string[11];
                 nev:     string[30];
                 belepes: 1988..2100;
                 funkcio: string[15];
                 tagdij:  integer;
                 end;
```

```
     adatfile=file of adatrec;

var fidesz,madisz:adatfile;
    mrec,frec:adatrec;
    vege:boolean;

procedure olvas(var x:adatrec; var j:boolean);
  begin
    j:=false;
    clrscr;
    with x do begin
      write('szemelyi szam:');
      readln(szemszam);
      if szemszam='0' then begin
        j:=true;
        exit;
        end;
      write('          nev:');
      readln(nev);
      write('belepes kelte:');
      readln(belepes);
      write('     funkcio:');
      readln(funkcio);
      write('       tagdij:');
      readln(tagdij);
      end;
    end;

begin
  clrscr;
  assign(fidesz,fidnev);
  rewrite(fidesz);
  writeln('FIDESZ tagok felirasa');
  delay(3000);
  repeat
    olvas(frec,vege);
    if not vege then
    write(fidesz,frec);
    until vege;
  clrscr;
  assign(madisz,madisznev);
  rewrite(madisz);
  writeln('MADISZ-tagok felirasa');
  delay(3000);
  repeat
    olvas(mrec,vege);
    if not vege then
    write(madisz,mrec);
    until vege;
  close(fidesz);
  Close(madisz);
  end.
```

7.

```
program leltar;
  const allomany='raktar.dat';
  type string30=string[30];
       anyagrec = record
                    anyagszam: 1..1000;
                   megnevezes: string30;
                     egysegar: real;
                   mennyiseg: real;
                   minkeszlet: real;
                   end;
        anyagfile = file of anyagrec;

  var anyag:anyagrec;
      keszlet:anyagfile;

  begin
    assign(keszlet,allomany);
    reset(keszlet);
    clrscr;
    writeln('            R A K T A R K E S Z L E T');
    writeln;
    writeln(
          'ANYAGSZ           MEGNEVEZES     ',
          '          MENNYISEG     EGYSAR     ERTEK ');
    writeln(
          '-----------------------------------------',
          '-----------------------------');
   while not eof(keszlet) do begin
     read(keszlet,anyag);
     with anyag do
       if anyagszam<>0 then begin
         write(anyagszam:6,'   ');
         write(megnevezes:30);
         write(mennyiseg:13:1);
         write(egysegar:10:1);
         writeln(mennyiseg*egysegar:10:1);
         end;
     end;
   close(keszlet);
   end.
```

8.

Az állományt a következő programmal írhatjuk fel:

```
  program leltar;
    const allomany='raktar.dat';
    type string30=string[30];
         anyagrec = record
                      anyagszam: 1..1000;
```

```
                    megnevezes: string30;
                      egyseqar: real;
                     mennyiseq: real;
                    minkeszlet: real;
                    end;
        anyaqfile = file of anyaqrec;

var anyaq:anyaqrec;
    keszlet:anyaqfile;
    vege:boolean;

procedure olvas(var x:anyaqrec;var j:boolean);
  begin
    clrscr;
    j:=false;
    with x do begin
      write('ANYAGSZAM:  ');
      readln(anyaqszam);
      if anyaqszam=0 then begin
        j:=true;
        exit;
        end;
      write('MEGNEVEZES: ');
      readln(megnevezes);
      write('EGYSEGAR: ');
      readln(egyseqar);
      write('MENNYISEG: ');
      readln(mennyiseq);
      write('MINIMALIS KESZLET: ');
      readln(minkeszlet);
      end;
    end;

begin
  assign(keszlet,allomany);
  reset(keszlet);
  repeat
   olvas(anyaq,vege);
   if not vege then begin
     seek(keszlet,anyaq.anyaqszam-1);
     write(keszlet,anyaq);
     end;
  until vege;
 close(keszlet);
 end.
```

Mielőtt ezt a programot használnánk, az állományt elő kell készíteni (üres állományt kell létrehozni). Erre használhatjuk a következő programot:

```
program elokeszites;
  const allomany= raktar.dat ;
  type string30=string[30];
       anyagrec = record
                    anyagszam: 1..1000;
                   megnevezes: string30;
                     egysegar: real;
                    mennyiseg: real;
                   minkeszlet: real;
                   end;
        anyagfile = file of anyagrec;

  var anyag:anyagrec;
      keszlet:anyagfile;
      i:integer;

  begin
    assign(keszlet,allomany);
    rewrite(keszlet);
    for i:=1 to 1000 do begin
       anyag.anyagszam:=0;
       write(keszlet,anyag);
       end;
   close(keszlet);
   end.
```

Az üres rekordokat arról ismerhetjük fel, hogy az anyagszam értéke 0.

9.

```
program leltar;
  const allomany='raktar.dat';
  type string30=string[30];
       anyagrec = record
                    anyagszam: 1..1000;
                   megnevezes: string30;
                     egysegar: real;
                    mennyiseg: real;
                   minkeszlet: real;
                   end;
        anyagfile = file of anyagrec;

  var anyag:anyagrec;
      keszlet:anyagfile;

  begin
    assign(keszlet,allomany);
    reset(keszlet);
    clrscr;
    writeln('          M E G R E N D E L E S E K');
```

```
    writeln;
    writeln(
            'ANYAGSZ           MEGNEVEZES   ',
            '          MENNYISEG     MINKESZL   EGYSAR');
    writeln(
            '----------------------------------------',
            '----------------------------------------');
   while not eof(keszlet) do begin
     read(keszlet,anyag);
     with anyag do
       if (anyagszam<>0) and (minkeszlet>mennyiseg) then begin
         write(anyagszam:6,'   ');
         write(megnevezes:30);
         write(mennyiseg:13:1);
         write(minkeszlet:10:1);
         writeln(egysegar:10:1);
         end;
     end;
   close(keszlet);
   end.
```

10.

```
program leltar;
  const allomany='raktar.dat';
  type string30=string[30];
       hibarec= record
                  anyagszam:1..1000;
                  ok:string[20];
                  kod:byte;
                  end;
       hibatomb = array[1..20] of hibarec;
       anyagrec = record
                    anyagszam: 1..1000;
                   megnevezes: string30;
                     egysegar: real;
                    mennyiseg: real;
                   minkeszlet: real;
                   end;
        anyagfile = file of anyagrec;

  var anyag,valtozas:anyagrec;
      keszlet:anyagfile;
      vege:boolean;
      mozgaskod:byte;
      hibalist:hibatomb;
      i:integer;

  procedure olvas(var x:anyagrec;var kod:byte;
                                  var j:boolean);
    begin
      clrscr;
```

```
    j:=false;
    with x do begin
      write('ANYAGSZAM: ');
      readln(anyagszam);
      if anyagszam=0 then begin
        j:=true;
        exit;
        end;
      write('MOZGASKOD: ');
      readln(mozgaskod);
      write('MENNYISEG: ');
      readln(mennyiseg);
      end;
    end;

procedure potadat(var x:anyagrec);
  begin
    with x do begin
      write('MEGNEVEZES: ');
      readln(megnevezes);
      write('EGYSEGAR: ');
      readln(egysegar);
      write('MINIMALIS KESZLET: ');
      readln(minkeszlet);
      end;
    end;

begin
  for i:=1 to 20 do
    hibalist[i].anyagszam:=0;
  assign(keszlet,allomany);
  reset(keszlet);
  i:=0;
  repeat
   olvas(valtozas,mozgaskod,vege);
   if not vege then begin
     seek(keszlet,valtozas.anyagszam-1);
     read(keszlet,anyag);
     if anyag.anyagszam=0 then begin
       if mozgaskod=4 then begin
         potadat(valtozas);
         seek(keszlet,valtozas.anyagszam-1);
         write(keszlet,valtozas);
         end
       else begin
         i:=i+1;
         with hibalist[i] do begin
           anyagszam:= valtozas.anyagszam;
           ok:=' NEMLETEZO ANYAG';
           kod:=0;
           end
         end
       end
```

```
        else  begin
          case mozgaskod of
            1..3: anyag.mennyiseg:=anyag.mennyiseg
                                    -valtozas.mennyiseg;
            4,5:  anyag.mennyiseg:=anyag.mennyiseg
                                    +valtozas.mennyiseg;
            else  begin
                   i:=i+1;
                   with hibalist[i] do begin
                     anyagszam:=valtozas.anyagszam;
                     ok:=' HIBAS MOZGASKOD: ';
                     kod:=mozgaskod;
                     end;
                  end;
            end;
          seek(keszlet,valtozas.anyagszam-1);
          write(keszlet,anyag);
          end;
        end;
      until vege;
    close(keszlet);
    clrscr;
    writeln('       H I B A L I S T A');
    writeln;
    i:=1;
    while hibalist[i].anyagszam<>0 do
      with hibalist[i] do begin
        write(anyagszam:5);
        write(ok:15);
        writeln(kod:3);
        i:=i+1;
        end;
    end.
```

11.

```
program uralkodok;
const kszmax=100;
      ms=8;
      asz=8;
      menu:array[1..ms] of string[40]=
                 ('Uj kiraly adatainak felvetele',
                  'Kiraly adatainak torlese',
                  'Mikor uralkodott .... kiraly ?',
                  'Ki uralkodott .... evben ?',
                  'Ki volt .... kiraly felesege ?',
                  'Mikor elt .... kiraly ?',
                  'Milyen szarmazasu volt .... kiraly ?',
                  'Kilepes a programbol');
      adszov:array[1..asz] of string[30]=
                       ('Neve: ',
                        'Szuletesi ideje: ',
```

```
                        'Halalozasanak ideje: ',
                        'Uralkodasanak kezdete: ',
                        'Uralkodasanak vege: ',
                        'Szarmazasa: ',
                        'Felesegenek neve: ',
                        'Felesegenek szarmazasa: ');
type datum=record
           ev,ho,nap:integer;
           end;
     nev=string[40];
     szarmazas=string[20];
     felesegtype=record
                   nev:nev;
                   szarm:szarmazas;
                   end;
     kirtype=record
              nev:nev;
              szul:datum;
              hal:datum;
              urkezd:datum;
              urveg:datum;
              szarm:szarmazas;
              feleseg:felesegtype;
              end;
     itomb=array[1..kszmax] of integer;
     nevtomb=array[1..kszmax] of nev;
     vtomb=array[1..kszmax] of real;
var i,j,ksz:integer;
    valasz:char;
    f:file of kirtype;
    u,sz,nevf:file of integer;

procedure dki(d:datum);
begin
write(d.ev,'/',d.ho,'/',d.nap);
end;
{------------ datum bekeres ----------------}
procedure datbeker(var d:datum);
var dat:string[10];
    x,y,h:integer;
begin
x:=wherex; y:=wherey;
write('eeee/hh/nn');
repeat
  gotoxy(x,y);
  readln(dat);
until length(dat)=10;
val(copy(dat,1,4),d.ev,h);
val(copy(dat,6,2),d.ho,h);
val(copy(dat,9,2),d.nap,h);
end;
{--------- varakozas billentyu lenyomasra --------}
procedure folyt;
```

```
begin
writeln;
writeln('Nyomj egy billentyut folytatashoz');
while not keypressed do;
end;
{-------- nev tomb szerinti rendezes -------------}
procedure nrendez(var at:nevtomb; var t:itomb;
                                  n:integer);
var s,h,d1,j,i,s1:integer;
    a:real;
begin
s:=round(int(exp((ln(n)/(ln(2))*ln(2)))))-1;
repeat
  d1:=n-s;
  for j:=1to d1 do
  begin
  i:=j;
  s1:=i+s;
  while (i>0) and (at[t[i]]>at[t[s1]]) do
    begin
      s1:=i+s;
      h:=t[i];
      t[i]:=t[s1];
      t[s1]:=h;
      i:=i-s;
    end;
  end;
s:=s div 2;
until s=0;
end;
{-------- valos tomb szerinti rendezes -------------}
procedure vrendez(var at:vtomb; var t:itomb; n:integer);
var s,h,d1,j,i,s1:integer;
    a:real;
begin
s:=round(int(exp((ln(n)/(ln(2))*ln(2)))))-1;
repeat
  d1:=n-s;
  for j:=1to d1 do
  begin
  i:=j;
  s1:=i+s;
  while (i>0) and (at[t[i]]>at[t[s1]]) do
    begin
      s1:=i+s;
      h:=t[i];
      t[i]:=t[s1];
      t[s1]:=h;
      i:=i-s;
    end;
  end;
s:=s div 2;
until s=0;
end;
```

```
{-------- indextablak helyreallitasa -------------}
procedure indhelyre;
var kir,k:kirtype;
    uri,szuli,nevi:itomb;
    ur,szul:vtomb;
    nevt:nevtomb;
    datumv:real;
    kirt:array[1..kszmax] of kirtype;
begin
rewrite(u); rewrite(sz);
clrscr;
if ksz=1 then begin
              write(u,ksz);write(sz,ksz);
              end
         else
if ksz=0 then begin
              rewrite(f);
              end
         else
  begin
  seek(f,0);
  for i:=1 to ksz do
    begin
    read(f,kirt[i]);
    nevt[i]:=kirt[i].nev;
    end;
rewrite(f);
for i:=1 to ksz do
  begin
  uri[i]:=i;
  szuli[i]:=i;
  nevi[i]:=i;
  end;
nrendez(nevt,nevi,ksz);
  for i:=1 to ksz do
    begin
    with kirt[i].urkezd do ur[i]:=ev*365.0+ho*31.0+nap;
    with kirt[i].szul do szul[i]:=ev*365.0+ho*31.0+nap;
    end;
vrendez(ur,uri,ksz);
vrendez(szul,szuli,ksz);
  for i:=1 to ksz do
    begin
    write(u,uri[i]);
    write(sz,szuli[i]);
    write(f,kirt[nevi[i]]);
    end;
  end;
close(u); close(sz);
end;
{-------- kereses nev szerint --------------------}
function nevker(knev:nev):integer;
var i,a,fe,k:integer;
    nevi:itomb;
```

```
    kir:kirtype;
begin
reset(f);
a:=1;
fe:=ksz;
repeat
  k:=(a+fe) div 2;
  seek(f,k-1);
  read(f,kir);
  if kir.nev<knev then a:=k+1;
  if kir.nev>knev then fe:=k-1;
until (a>fe) or (kir.nev=knev);
if a>fe then nevker:=0
        else nevker:=k;
end;
{-------- uj kiraly adatainak felvetele ----------}
procedure ujkir;
var kir:kirtype;
label ujra;
begin
clrscr;
writeln('                    UJ KIRALY ADATAINAK FELVETELE');
reset(f);
writeln;
writeln;
if ksz=kszmax-1 then
  begin
  writeln;
  writeln('Betelt az adatbazis!',chr(8));
  folyt;
  exit;
  end;
for i:=1 to asz do writeln(adszov[i]);
ujra:
with kir do
  begin
  gotoxy(length(adszov[1])+1,4);
  readln(nev);
  gotoxy(length(adszov[2])+1,wherey);
  datbeker(szul);
  gotoxy(length(adszov[3])+1,wherey);
  datbeker(hal);
  gotoxy(length(adszov[4])+1,wherey);
  datbeker(urkezd);
  gotoxy(length(adszov[5])+1,wherey);
  datbeker(urveg);
  gotoxy(length(adszov[6])+1,wherey);
  readln(szarm);
  gotoxy(length(adszov[7])+1,wherey);
  readln(feleseg.nev);
  gotoxy(length(adszov[8])+1,wherey);
  readln(feleseg.szarm);
  end;
writeln;
```

```
write('Akarsz javitani (i/n)? ');
readln(valasz);
if valasz='i' then goto ujra;
seek(f,filesize(f));
write(f,kir);
ksz:=ksz+1;
indhelyre;
end;
{ -------- torles az adatbazisbol nev szerint -------}
procedure torles;
var ssz,i:integer;
    knev:nev;
    k:array[1..kszmax] of kirtype;
begin
clrscr;
writeln('         ADATOK TORLESE NEV SZERINT');
writeln;
if ksz=0 then begin
              writeln('Ures az adatbazis,',
                      ' nincs mit torolni.');
              folyt;
              exit;
              end;
write('Kerelek add meg a kiraly nevet: ');
readln(knev);
ssz:=nevker(knev);
if ssz=0 then begin
      writeln;
      writeln('Nem talaltam ilyen nevu kiralyt!');
      folyt;
      exit;
      end;
reset(f);
for i:=1 to ksz do read(f,k[i]);
rewrite(f);
for i:=1 to ssz-1 do write(f,k[i]);
for i:=ssz+1 to ksz do write(f,k[i]);
reset(f);
ksz:=ksz-1;
indhelyre;
end;
{----- Mikor uralkodott ... kiraly? -----}
procedure muk;
var knev:nev;
    ssz:integer;
    kir:kirtype;
begin
clrscr;
writeln('      Mikor uralkodott .... kiraly?');
writeln;
if ksz=0 then begin
    writeln('Ures az adatbazis, nincs hol keresni.');
    folyt;
    exit;
```

```
    end;
write('Kerelek add meg a kiraly nevet: ');
readln(knev);
ssz:=nevker(knev);
if ssz=0 then begin
     writeln;
     writeln('Nem talaltam ilyen nevu kiralyt!');
     folyt;
     exit;
     end;
reset(f);
seek(f,ssz-1);
read(f,kir);
writeln;
with kir do
  begin
  write(nev,' kiraly ');
  dki(urkezd);
  write(' es ');
  dki(urveg);
  writeln(' kozott uralkodott.');
  end;
folyt;
end;
{ -------- ido szerinti kerese --------}
function idker(id1,id2:vtomb;datumv:real):integer;
var i:integer;
begin
i:=0;
if ksz=1 then if (id1[ksz]<datumv)
                 and (datumv<id2[ksz]) then begin
                idker:=ksz;
                exit;
                end
             else begin
                idker:=0;
                exit;
                end;
repeat
i:=i+1;
until ((id1[i]<datumv) and (datumv<id2[i])) or (i>ksz);
if i>ksz then idker:=0
         else
         idker:=i;
end;
{ --------- Ki uralkodott .... idoben ? ------}
procedure kui;
var ssz,i,ind:integer;
    d:datum;
    datumv:real;
    uk,uv:vtomb;
    kir:kirtype;
begin
clrscr;
```

```
writeln('            Ki uralkodott .... idoben');
writeln;
if ksz=0 then begin
     writeln('Ures az adatbazis, nincs hol keresni.');
     folyt;
     exit;
     end;
write('Kerlek add meg a datumot: ');
datbeker(d);
with d do datumv:=ev*365.0+ho*31.0+nap;
reset(u); reset(f);
for i:=1 to ksz do
  begin
  read(u,ind);
  seek(f,ind-1);
  read(f,kir);
  with kir.urkezd do uk[i]:=ev*365.0+ho*31.0+nap;
  with kir.urveg do uv[i]:=ev*365.0+ho*31.0+nap;
  end;
ssz:=idker(uk,uv,datumv);
if ssz=0 then begin
    writeln;
    writeln('Ekkor nem uralkodott senki Magyarorszagon.');
    folyt;
    exit;
    end;
reset(f);
seek(f,ssz-1);
read(f,kir);
writeln;
writeln(kir.nev,
        ' kiraly uralkodott ekkoriban Magyarorszagon.');
folyt;
end;
{------- Ki volt .... kiraly felesege ------}
procedure kvf;
var knev:nev;
    ssz:integer;
    kir:kirtype;
begin
clrscr;
writeln('      Ki volt .... kiraly felesege?');
writeln;
if ksz=0 then begin
    writeln('Ures az adatbazis, nincs hol keresni.');
    folyt;
    exit;
    end;
write('Kerelek add meg a kiraly nevet: ');
readln(knev);
ssz:=nevker(knev);
if ssz=0 then begin
     writeln;
     writeln('Nem talaltam ilyen nevu kiralyt!');
```

```
     folyt;
     exit;
     end;
reset(f);
seek(f,ssz-1);
read(f,kir);
writeln;
with kir do
  begin
  write(nev,' kiraly felesege ');
  write(feleseg.nev);
  writeln(' volt.');
  end;
folyt;
end;
{------ Mikor elt .... kiraly ------}
procedure mek;
var knev:nev;
    ssz:integer;
    kir:kirtype;
begin
clrscr;
writeln('      Mikor elt .... kiraly?');
writeln;
if ksz=0 then begin
     writeln('Ures az adatbazis, nincs hol keresni.');
     folyt;
     exit;
     end;
write('Kerelek add meg a kiraly nevet: ');
readln(knev);
ssz:=nevker(knev);
if ssz=0 then begin
     writeln;
     writeln('Nem talaltam ilyen nevu kiralyt!');
     folyt;
     exit;
     end;
reset(f);
seek(f,ssz-1);
read(f,kir);
writeln;
with kir do
  begin
  write(nev,' kiraly ');
  dki(urkezd);
  write(' es ');
  dki(urveg);
  writeln(' kozott elt.');
  end;
folyt;
end;
{ ------ Milyen szarmazasu volt .... kiraly ------}
procedure mszv;
```

```
var knev:nev;
    ssz:integer;
    kir:kirtype;
begin
clrscr;
writeln('      Milyen szarmazasu volt .... kiraly ?');
writeln;
if ksz=0 then begin
     writeln('Ures az adatbazis, nincs hol keresni.');
     folyt;
     exit;
     end;
write('Kerelek add meg a kiraly nevet: ');
readln(knev);
ssz:=nevker(knev);
if ssz=0 then begin
     writeln;
     writeln('Nem talaltam ilyen nevu kiralyt!');
     folyt;
     exit;
     end;
reset(f);
seek(f,ssz-1);
read(f,kir);
writeln;
with kir do
  begin
  write(nev,' kiraly ');
  write(szarm);
  writeln(' szarmazasu volt.');
  end;
folyt;
end;
{ ------------------------------- }
var k:kirtype;
begin
assign(f,'kiradat.dat');
assign(nevf,'nev.ind');
assign(sz,'szul.ind');
assign(u,'uralk.ind');
reset(f);
repeat
ksz:=filesize(f);
clrscr;
writeln('          A MAGYARORSZAGON URALKODOTT KIRALYOK',
       ' SZEMELYI ADATBAZISA');
gotoxy(1,3);
for i:=1 to ms do begin
   writeln('                    ',i:3,'  ',menu[i]);
   writeln
   end;
repeat
  gotoxy(30,3+ms*2); write('Kerlek valassz! ');
  while not keypressed do;
```

```
    read(con,valasz);
  until (valasz>'0') and (valasz<chr(49+ms));
  case valasz of
     '1':ujkir;
     '2':torles;
     '3':muk;
     '4':kui;
     '5':kvf;
     '6':mek;
     '7':mszv;
     end;
  until valasz='8';
  close(f);
  end.
```

12. fejezet

1.

```
type adattip = record
                 n:byte;
                 a:char;
                 end;
     listpoi = ^elemtip;
     elemtip = record
                 adat: adattip;
                 kov:  listpoi;
                 end;

procedure car(x:listpoi; var xf:adattip);
  begin
    xf:=x^.adat;
    end;

{Sajnos a CAR-t nem lehet teljes ertekuen implementalni,
 mert osszetett tipusu ertek nem lehet fuggvenyertek.}

function cdr(x:listpoi):listpoi;
  begin
    cdr:=x^.kov;
    end;
```

2. Az értékadás végrehajtása után a list1 pointer ugyanarra az objektumra fog mutatni, mint a list2.

3.

```
procedure copy(var eredeti,masolat:listpoi);
  var p,q,r:listpoi;
  begin
    p:=eredeti;
    new(q);
    masolat:=q;
    while p^.kov<>nil do begin
      q^.adat:=p^.adat;
      p:=p^.kov;
      r:=q;
      new(q);
      r^.kov:=q;
      end;
    end;
```

4.

```
procedure keres(x:listpoi;k:kulcstip;var a:adattip;
                                        var j:boolean);
  var p:listpoi;
      kilep:boolean;
  begin
    j:=false;
    p:=x;
    kilep:=false;
    while not kilep do
      if p=nil then begin
        kilep:=true;
        writeln('HIBAS KULCS!');
        end
      else if p^.adat.n=k then
        kilep:=true
      else
        p:=p^.kov;
    if p<>nil then begin
      a:=p^.adat;
      j:=true;
      end;
    end;
```

5.

```
procedure torles( var listafej   : listptr;
                      torlkulcs  : kulcstipus;
                  var statusz    : byte);

var
   p, r   : listptr;

begin
     p:=listafej;
     if p^.kulcs=maxint then
     begin
          statusz:=2; {ures lista}
          exit
     end;
     while p^.kulcs<torlkulcs do begin
           r:=p;
           p:=p^.kovetkezo;
           end;
     if p^.kulcs>torlkulcs then
        statusz:=1
     else
       begin
            statusz:=0;
            if p=listafej then begin
              writeln('p=listafej');
              listafej:=listafej^.kovetkezo;
              end
            else
              r^.kovetkezo:=p^.kovetkezo;
            dispose(p);
       end
end;
```

6.

```
program beszurstrazsaval;
type
    kulcstipus  = integer;
    listptr     = ^elemtipus;
    elemtipus   = record
                        kulcs     : kulcstipus;
                        kovetkezo : listptr;
                  end;
const strazsa:kulcstipus=maxint;

var
   a, b, c   : listptr;
```

```
   i          : integer;
   betenni    : elemtipus;
   stat       : byte;

{-------------------------------------------------}
procedure beszur( var listafej   : listptr;
                      ujelem     : elemtipus);

var
   p, q, r    : listptr;
   megvan     : boolean;

begin
     new(r);
     r^:=ujelem;

     p:=listafej;
     if p^.kulcs=strazsa then begin {ures lista esete}
        r^.kovetkezo:=nil;
        listafej:=r;
        end
     else
     begin
          megvan:=false;
          while not megvan do
               if p^.kulcs<ujelem.kulcs then
               begin
                    q:=p;
                    p:=p^.kovetkezo;
               end
               else
                    megvan:=true;
          r^.kovetkezo:=p;
          if p=listafej then
             listafej:=r
          else
             q^.kovetkezo:=r;
     end
end;
{-----------------------------------------}

procedure klist( t : listptr);

begin
     while t<>NIL do
     begin
          writeln(t^.kulcs:4);
          t:=t^.kovetkezo;
     end;
end;

begin
     new(b);
```

```
        b^.kulcs:=strazsa;
        b^.kovetkezo:=nil;
        repeat
             write('Mondjon egy kulcsot 1 es 100 kozott
            (0 kilepes) : ');
             readln(betenni.kulcs);
             if betenni.kulcs in [1..100] then
                 beszur(b,betenni);
             klist(b);
        until betenni.kulcs = 0;
end.
```

7.

```
program magasugras;
const
     vegjel=0;

type
    str20       = string[20];
    str10       = string[10];
    listptr     = ^elemtipus;
    elemtipus   = record
                        rajtszam   : integer;
                        kulcs      : real;
                        nev        : str20;
                        orszag     : str10;
                        kovetkezo  : listptr;
                  end;

var
   versenyzo     : elemtipus;
   p,q,helylista: listptr;
   i,n           : integer;

{-------------------------------------------------}
procedure beszur( var listafej   : listptr;
                      r     : listptr);

var
   p, q       : listptr;
   megvan     : boolean;

begin
     p:=listafej;
     if p=NIL then begin {ures lista esete}
        listafej:=r;
        r^.kovetkezo:=nil;
        end
     else
```

```
     begin
          megvan:=false;
          while (p<>NIL) and not(megvan) do
               if p^.kulcs>r^.kulcs then
               begin
                    q:=p;
                    p:=p^.kovetkezo;
               end
               else
                    megvan:=true;
          r^.kovetkezo:=p;
          if p=listafej then
             listafej:=r
          else
             q^.kovetkezo:=r;
     end
end;
{-------------------------------------------------------}

procedure olvas(var o: elemtipus);
begin
     write('Rajtszam       = ');
     readln(o.rajtszam);
     if o.rajtszam <> 0 then
     begin
          write('Nev             = ');
          readln(o.nev);
          write('Orszag          = ');
          readln(o.orszag);
     end;
     writeln;
end;
{-------------------------------------------------------}

procedure beker(var o: elemtipus);
  begin
     write('Rajtszam       = ');
     readln(o.rajtszam);
     if o.rajtszam <> 0 then
     begin
       write('Eredmeny      = ');
       readln(o.kulcs);
       end;
     writeln;
     end;

{-------------------------------------------------------}

procedure kiir(o: elemtipus);
begin
     writeln('Rajtszam       = ',o.rajtszam);
     writeln('Magassag       = ',o.kulcs:6:2);
     writeln('Nev            = ',o.nev);
```

```
      writeln('Orszag          =  ',o.orszag);
      writeln;
end;

{-------------------------------------------------------}

  procedure kiiras;
    var p:listptr;
        i:integer;
    begin
     {kiiras helyezesi sorrendben}
     clrscr;
     writeln('Helyezesi sorrend');
     writeln('----------------');
     writeln;
     i:=0;
     p:=helylista;
     repeat
           i:=i+1;
           write(i,'-edik helyezett ');
           kiir(p^);
           writeln;
           p:=p^.kovetkezo;
     until p=NIL;
     end;
{-----------------------------------------------}
procedure betesz( var listafej  : listptr;
                      ujelem    : elemtipus);

var
   p, q, r    : listptr;
   megvan     : boolean;

begin
     new(r);
     r^:=ujelem;
     r^.kovetkezo:=nil;

     p:=listafej;
     if p=NIL then {ures lista esete}
        listafej:=r
     else
     begin
          megvan:=false;
          while (p<>NIL) and not(megvan) do
                if p^.rajtszam<ujelem.rajtszam then
                begin
                     q:=p;
                     p:=p^.kovetkezo;
                end
                else
                     megvan:=true;
```

```
          r^.kovetkezo:=p;
          if p=listafej then
             listafej:=r
          else
             q^.kovetkezo:=r;
     end
end;
{-----------------------------------------------------}

procedure letrehoz;
  begin
     helylista:=NIL;
     repeat
           olvas(versenyzo);
           versenyzo.kulcs:=0.0;
           if versenyzo.rajtszam<>0 then
              betesz(helylista, versenyzo);
     until versenyzo.rajtszam=0;
    end;
{-----------------------------------------}

begin
  clrscr;
  writeln('A fordulok szama: ');
  readln(n);
  clrscr;
  writeln('A versenyzok adatai:');
  letrehoz;
  for i:=1 to n do begin
    clrscr;
    writeln('Az ',i,'-edik fordulo');
    repeat
      beker(versenyzo);
      if versenyzo.rajtszam <>0 then begin
        while (p<>nil) and
              (p^.rajtszam<versenyzo.rajtszam) do begin
        p:=p^.kovetkezo;
        end;
      if (p=nil) or (p^.rajtszam>versenyzo.rajtszam)
        then
        writeln('Nemletezo rajtszam')
      else  begin
        if p^.kulcs<versenyzo.kulcs then begin
          if p=helylista then
            helylista:=p^.kovetkezo
          else
            q^.kovetkezo:=p^.kovetkezo;
          end;
        end;
      beszur(helylista,p);
      kiiras;
      while not keypressed do;
      end;
      until versenyzo.rajtszam=0;
```

```
        kiiras;
        while not keypressed do;
        end;
      end.
```

8.

```
type listptr=^listelem;
     listelem=record
                kulcs: kulcstipus;
                adat: adattipus;
                elozo: listptr;
                kovetkezo: listptr;
                end;

procedure duplatorles (var eleje,vege: listptr;
                           k: ktipus;
                       var statusz: byte);


const
 rendben =0;
 hianyzik =1;
 ures =2;
var
 spoi,
 tpoi  : listptr;

begin

   if eleje = nil then
     statusz := ures
   else if k = eleje^.kulcs then
     begin
       statusz := rendben;

       tpoi := eleje;
       eleje := eleje^.kovetkezo;
       eleje^.elozo := nil;
       if eleje = nil then
         vege := nil;
       dispose (tpoi);
    end

   else if k = vege^.kulcs then
     begin
        statusz := rendben;
        tpoi := vege;
        vege := vege^.elozo;
        vege^.kovetkezo := nil;
        dispose (tpoi);
```

9.

10

```
      end
    else
      begin

        spoi := eleje;
        while (spoi <> nil) and (spoi^.kulcs <> k) do
          spoi := spoi^.kovetkezo;

        if spoi = nil then
          statusz := hianyzik
        else
          begin
            statusz := rendben;

            spoi^.kovetkezo^.elozo := spoi^.elozo;
            spoi^.elozo^.kovetkezo := spoi^.kovetkezo;
            dispose (spoi);
          end;
        end;
      end;
```

9.

```
procedure eloltorol( var eleje: listptr);
  begin
    eleje:=eleje^.kovetkezo;
    eleje^elozo:=nil;
    end;
```

10.

```
procedure vegeretesz(var fej: listptr;
                     elem: listelem);
  var p,q:listptr;
  begin
    new(q)
    q^:=elem;
    p:=fej;
    while p^.kovetkezo<>nil do
      p:=p^.kovetkezo;
    p^.kovetkezo:=q;
    q^.elozo:=p;
    q^.kovetkezo:=nil;
    end;
```

11.

```
type faptr=^facsucs;
     facsucs=record
               szam:real;
               balag:faptr;
               jobbag:faptr;
               end;

procedure faosszeg(fa:faptr; var sum:real);
  begin
    if fa<>nil then begin
      sum:=sum + fa^.szam;
      faosszeg(fa^.balag);
      faosszeg(fa^.jobbag);
      end;
    end;
```

12.

```
type terptr=^facsucs;
     facsucs=record
               adat: adattipus;
               ag1: terptr;
               ag2: terptr;
               ag3: terptr;
               end;
{----------------------------------------}

procedure preorder(fa: terptr);
  begin
    if fa<>nil then  begin
      feldolgoz(fa^);
      preorder(fa^.ag1);
      preorder(fa^.ag2);
      preorder(fa^.ag3);
      end;
    end;

{----------------------------------------}
{ketfele inorder bejaras letezik}

procedure inlorder(fa: terptr);
  begin
    if fa<>nil then  begin
      preorder(fa^.ag1);
      feldolgoz(fa^);
      preorder(fa^.ag2);
```

```
      preorder(fa^.ag3);
      end;
    end;

{---------------------------------------}

procedure in2order(fa: terptr);
  begin
    if fa<>nil then  begin
      preorder(fa^.ag1);
      preorder(fa^.ag2);
      feldolgoz(fa^);
      preorder(fa^.ag3);
      end;
    end;

{---------------------------------------}

procedure postorder(fa: terptr);
  begin
    if fa<>nil then  begin
      preorder(fa^.ag1);
      preorder(fa^.ag2);
      preorder(fa^.ag3);
      feldolgoz(fa^);
      end;
    end;

{---------------------------------------}
```

IRODALOMJEGYZEK

Áts László:
OXFORD Pascal C 64-esen
Novotrade Rt., Budapest, 1987.

Áts László:
A számítástechnika matematikai alapjai
(Főiskolai jegyzet)
Pollack Mihály Műszaki Főiskola, Pécs, 1989.

Bentley, J.L.:
A programozás gyöngyszemei
Műszaki Könyvkiadó, Budapest, 1988.

Bowles, K.E.:
Problem Solving Using Pascal
Springer-Verlag, New York, 1977,

Edwards, C.C.:
Advanced Techniques in Turbo Pascal
SYBEX, Almadea, 1987.

Faulk, Ed:
The Turbo Pascal Handbook
COMPUTE! Publ. Inc., Greensboro, 1986.

Findlay, W. -Watt, D.A.:
Pascal
An Introduction to Methodical Programming
Computer Science Press, Inc., 1979.

Jamsa, K. - Nameroff, S.:
Turbo pascal Programmer's Library
Borland-Osborne/McGraw-Hill, 1988.

Jensen, K. - Wirth, N.:
A Pascal Programozási nyelv
Felhasználói kézikönyv és a nyelv formális leírása
Műszaki Könyvkiadó, Budapest,1982.

Lángos István:
Bevezetés az IBM PC XT/AT DOS-ba
Novotrade Rt., Budapest, 1987.

Linz, P.:
Programming Concepts and Problem Solving
An Introduction to Computer Science Using Pascal
The Benjamin/Cummings Publ. Co.; Menlo Park,
California, 1983.

Miller, A.R.:
Pascal Programs for Scientists and Enginiers
SYBEX, Almadea, 1981.

Norton, P.:
Fedezzük fel az IBM-PC-t!
Müszaki Könyvkiadó, Budapest, 1987.

O'Brien, S.K.:
Turbo Pascal
The Complete Reference
Borland-Osborne/McGraw-Hill, 1988.

Pongor György:
Szabványos Pascal programozás és algoritmusok
Novotrade Rt., Budapest,1988.

Rugg, Tom - Feldman, Phil:
Turbo Pascal Tips, Tricks and Traps
Que Corp., Indianapolis, Indiana, 1986.

Schildt, H.:
Advanced Turbo Pascal
Borland-Osborne/McGraw-Hill, 1987.

Schneider - Weingart - Perlman:
An Introduction to Programming and Problem Solving
with Pascal
John Willey et Sons Inc., New York, 1978.

Turbo Pascal Owner's Handbook
Version 4.0
Borland International, Scotts Valley, 1987.

Turbo Pascal Reference Guide
Version 3.0

Turbo Pascal Reference Guide
Version 5.0
Borland International, Scotts Valley, 1988.

Turbo Pascal User's Guide
Version 5.0
Borland International, Scotts Valley, 1988.

Varga László:
Rendszerprogramok elmélete és gyakorlata
Akadémiai Kiadó, Budapest, 1978.

Wirth, N.:
Algoritmusok + Adatszerkezetek = Programok
Műszaki Könyvkiadó, Budapest, 1982.

A Turbo Pascal könyvek szakirodalmát ez a teljesen kezdőknek írt kötet gazdagítja. Ezt a könyvet nem csak a Turbo Pascal iránt érdeklődő olvasóknak szánjuk. Haszonnal forgathatja mindenki, még az is, aki kezdő programozónak számít. Minden olyan programozástechnikai általános tudnivaló benne van, ami nem csak a Turbo Pascal sajátja. Külön függelék ismerteti a 2.0, 3.0 és 4.0 verziók közötti eltéréseket.

268,– Ft

ÁTS LÁSZLÓ • TURBO PASCAL KEZDŐKNEK