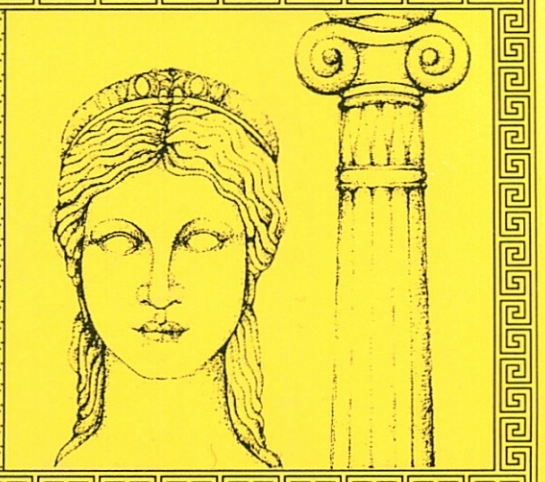
EKTF Líceum Kiadó
Alapítás éve: 1996



Koncz József

A PASCAL programozási nyelv elmélete és gyakorlata

EGER, 1999

Koncz József

A PASCAL programozási nyelv elmélete és gyakorlata

EKTF LÍCEUM KIADÓ, EGER
1999

Készült a KOMA 777. sz. pályázatának támogatásával

Lektorálta:
Papp Zoltán
egyetemi adjunktus

EMTEX—JATEX

A kiadásért felelős:
az Eszterházy Károly Tanárképző Főiskola főigazgatója
Megjelent az EKTF Líceum Kiadó műszaki gondozásában
Kiadóvezető: Rimán János
Felelős szerkesztő: Zimányi Árpád
Műszaki szerkesztő: Rimán Orsolya
Megjelent: 1999. november Példányszám: 500

Készült: Molnár és Társa '2001' Kft. nyomdája, Eger
Ügyvezető igazgató: Molnár György

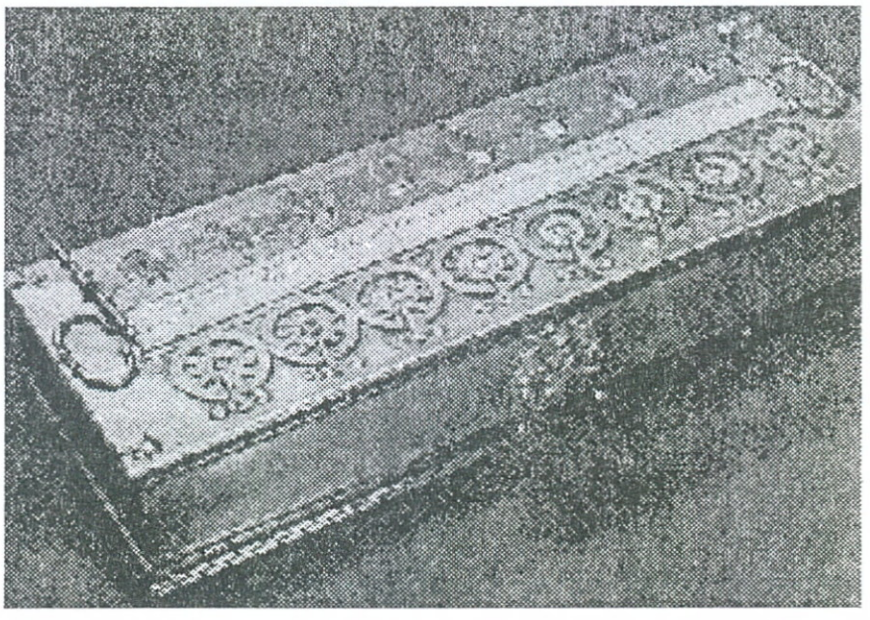
Tartalomjegyzék

Bevezetés

Üdvözöljük az Olvasót a számítástechnika talán egyik legdinamikusabban fejlődő, leginkább újakat felmutatni tudó területén: a programozás világában. Mivel világtendencia, hogy a szoftveres fejlesztések előretörnek a hardveresekkel szemben, nem hiábavaló foglalatosságunk. Könyvünk 16 fejezete a talán egyik legismertebb programozási nyelvvel, a Pascal nyelvvel szeretné megismertetni az érdeklődőt. Mielőtt elkezdenénk a könyv részletes anyagának ismertetését, lássuk, ki is az, akiről a programnyelvet elnevezték.

Blaise Pascal (1623—1692) francia filozófus, matematikus volt. Édesanyja meghalt, amikor Pascal hároméves volt. Iskolába nem járt, apja és magántanítók tanították. Kitűnő szellemi képességei már gyermekkorában megmutatkoztak. Gyermekkora igen nehéz volt. Ennek ellenére saját erejéből, öntevékenyen képezte magát. Saját magától ismerte fel az euklidészi geometria alaptörvényeit 16 éves korában, és komolyan foglalkozott a projektív geometriával is ([16]).

1642-ben számunkra is fontos bizonyítékát adta tehetségének: összeadni és kivonni tudó mechanikus számológépet szerkesztett. Sok új eredményt ért el a fizika területén is. Nevéhez kötődik a binomiális együtthatók elrendezése (Pascal-háromszög) és a teljes indukciós bizonyítás is. Foglalkozott valószínűségszámítással is ([10]).



Nem véletlen, hogy Nikolaus Wirth az 1968-ban definiált nyelvet róla nevezte el. A zürichi egyetem professzoraként olyan magas szintű programnyelvet tervezett, amely főként tudományos műszaki problémák megoldására alkalmazható. A definiált nyelv a Standard Pascal volt: ez az alapja az azóta elkészült implementációknak. Az első implementációk nagygépekre készültek. Az UCSD Pascal megvalósításakor messzemenően figyelembe vették az átvihetőség igényét ([12]). Így a Pascal nyelv főképpen a mikroszámítógépek fejlesztő nyelvévé vált, olyan eszközökkel bővítve, amelyek a mikroszámítógépek sajátos eszközeinek megfelelnek. A nyelv rugalmassága megmutatkozott akkor is, amikor megjelentek az első személyi számítógépek, így Commodore és IBM gépekre is elkészülhettek a fordítók.

A TURBO PASCAL egyike az elsősorban IBM PC-kre elkészített változatoknak. Az amerikai Borland Intézet terméke. 1982 óta foglakoznak e termék fejlesztésével, s azóta több új változat is megjelent már. Az első verzió 1984-ben látott napvilágot. A 2.0-ás változat már 1985-ben megjelent, és főleg az 1.0 hiányosságait igyekezett pótolni. Ez év decemberében készült el a 3.0-ás változat, mely a grafika terén mutatott jelentős fejlődést. Igazán nagy áttörést az 1987-ben megjelent 4.0-ás verzió jelentett, melyben bevezették a unitokat, a különböző képernyők kezelését, és sok új beépített eljárást kapott. Az 5.0-ban megjelent az átlapolási technika (1988). Az 5.5-ös verzió legnagyobb eredménye az objektumorientált programozás lehetősége. 1990-ben a 6.0, 1992-ben a 7.0 verzió jelent meg.

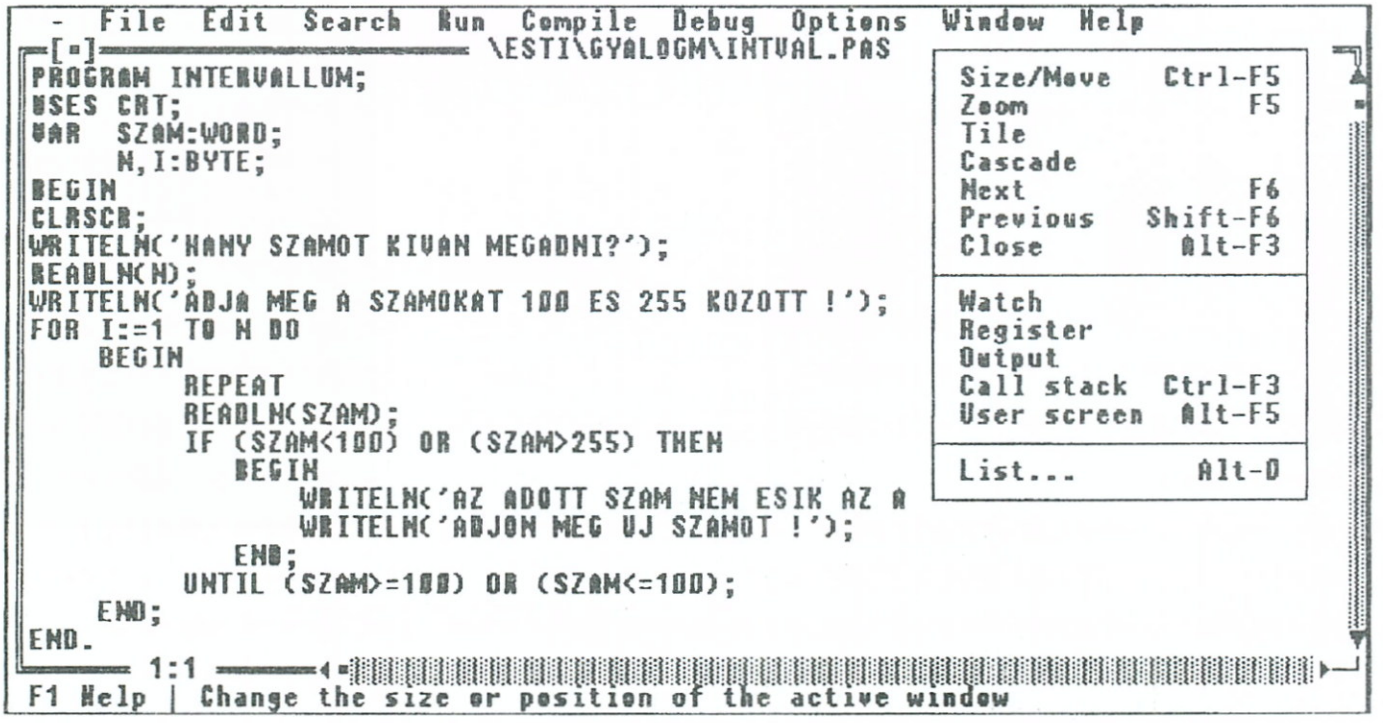
Láthatjuk, hogy hatalmas fejlődés szemtanúja lehetett a világ. Könyvünkben a 6.0-ás verziót szeretnénk a teljesség igénye nélkül bemutatni. A sok meglévő irodalom mellett az a célunk, hogy szemléletesen bemutassuk a Turbo Pascal kezelését és használatát annak érdekében, hogy praktikus algoritmusainkat konkrétan is kipróbálhassuk. Véleményünk szerint az anyag segítségével a nyelv a számítógépes munkával együtt önállóan is tanulható. Nem célunk, hogy profi programozókat képezzünk, de reméljük, hogy egyfajta szemléletet tudunk nyújtani. A fejezetek felépítésében megpróbáljuk azt az utat járni, ami már nagyon sokszor bevált: az elmélet és a gyakorlat sorrendje állandó lesz. Igyekszünk sok kidolgozott feladatot is bemutatni, részletes magyarázattal együtt. Ha a feladatok megkívánják (esetleg máskor is), igyekszünk azt képpel is illusztrálni. Célunk az volt, hogy azokat az érdeklődőket segítsük, akik önállóan, számítógép mellett szeretnék e programnyelvet elsajátítani. A kezdeti nehézségeket és sikertelenségeket a gyakorlással hamarosan áthidalhatják.

Eger, 1999. szeptember

A szerző

1. A Turbo Pascal jellemzői

A Turbo Pascal sorozat a világ egyik legelterjedtebb fordítója. A fejlesztők a 6.0 verzióban teljesen átszervezték a program környezetét, és sokkal használhatóbb, hatékonyabb külsőt adtak neki. Számos új funkciót is beépítettek.



A Turbo Pascal 6.0 munka közben

A Turbo Pascal Integrált Fejlesztői Környezete (Integrated Development Environment, a továbbiakban IDE) a Turbo Pascal programok fejlesztésének ideális színtere. Tartalmaz — többek között — egy szövegszerkesztőt (Editor), fordítót (Compiler), nyomkövetőt (Debugger), különböző beállítási lehetőségeket (Options), online Help rendszert. Ezeknek az eszközöknek a segítségével forrásnyelvű programjainkat kényelmesen tudjuk megírni, lefordítani, és a program futása közben a hibakeresés is könnyebben elvégezhető. A Help menü segítséget — sőt azonnali segítséget — is tud adni a Turbo Pascal rendszerről. Ezt az F1, Ctrl-F1 és az Alt-F1 billentyűk leütésével kérhetjük. A programrendszerből az Alt és az X billentyűk egyidejű lenyomásával léphetünk ki.

Most további billentyűket és billentyűkombinációkat nem kívánunk felsorolni, hiszen azok részletes tárgyalására később sort kerítünk. A Help rendszert is csak az önálló tanulás hangsúlyozása érdekében említettük. Használatáról a munka hevében se feledkezzünk el.

1.1. Hardver- és szoftverigények

A Turbo Pascal 6.0 programrendszer használatához szükségünk van egy IBM PC-vel kompatibilis gépre minimum 640 Kbyte RAM memóriával, 80 karakteroszlopos monitorra, merevlemezes egységre és legalább egy hajlékonylemez-meghajtóra. Az operációs rendszer MS-DOS (PC-DOS) 2.0 vagy magasabb verziószámú legyen. A Turbo Pascal körülbelül 3 Mbyte helyet foglal a merevlemezen (telepítéstől függően). A Turbo Pascal kihasználja az expanded (EMS) memóriát. Használatakor jelentősen nő a fordító sebessége.

Ha van egerünk, akkor a fejlesztő rendszerben az egeret is használhatjuk.

1.2. Telepítés

A fejlesztő rendszernek saját telepítő programja van. Az install lemezekről az INSTALL.EXE programmal telepíthetjük fel a rendszert. (Ez a program az 1. lemezen található.) Elindítás után beállíthatjuk, hogy a program egyes részei milyen alkönyvtárakba kerüljenek. A telepítő rendszer felkínál egy előre kialakított könyvtárstruktúrát, amit elfogadhatunk vagy módosíthatunk. Ha beállítottunk mindent, akkor a START INSTALL menüponttal megkezdődhet a telepítés.

Az installálás után az AUTOEXEC.BAT fájlban meg kell adnunk az elérési utat. Töltsük be ezt a fájlt egy szövegszerkesztőbe, és keressünk egy olyan sort, ami hasonlóan kezdődik:

```
PATH C:\;C:\DOS
```

Írjuk a végére: C:\TP utat, ha az installálás során a c:\tp könyvtárba került a program. Ha nincs PATH-tal kezdődő sor, akkor valahová szúrjuk be:

```
PATH C:\TP
```

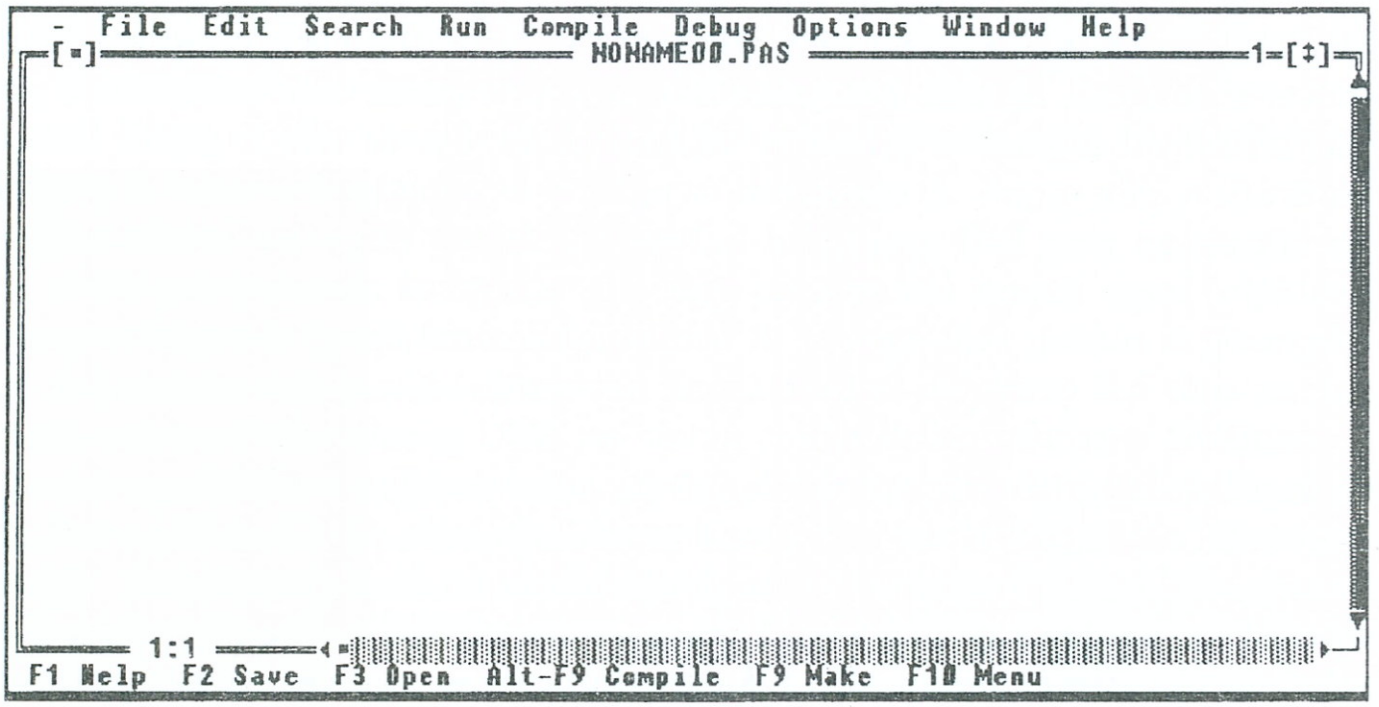
Az AUTOEXEC.BAT fájl elmentése után indítsuk újra a gépet, hogy a változásokat aktualizáljuk.

Ajánlott a TPTOUR program megnézése és kipróbálása, ami bemutatja a fejlesztői környezetet és annak használatát (ablakhasználat, egér stb.).

1.3. Ismerkedés az IDE-vel

A program a TURBO szó begépelésével indítható. A megjelenő ablakban a Turbo Pascal fejlesztői környezetét láthatjuk. A Turbo Pascalban egy programon belül megtalálható a fordító, a szerkesztő, a hibakereső, és még

számos hasznos alkalmazás, így nincs szükségünk külön programok beszerzésére, azok használatának megtanulására (mert az IDE-n belül minden egységes), és sok időt takarítunk meg azzal, hogy nem kell kilépni az IDE-ből, hogy egy másik programot használhassunk.



Az IDE három fő részből épül fel:

— főmenü a képernyő tetején,

— munkaasztal (vagy desktop), ide kerülnek a szerkesztőablakok és a dialógusok,

— státussor a képernyő alján.

A főmenüben található a legtöbb funkció, így ezeket könnyen és gyorsan elérhetjük. A szerkesztőablakból (ahol a legtöbbet fogunk tartózkodni) az F9 gomb lenyomásával juthatunk a főmenübe. Ilyenkor a legutoljára használt menüpont inverzre fog váltani. (Általában a rendszer inverz szöveggel jelzi, hogy hol tartózkodunk.) Itt a jobbra és balra nyilakkal közlekedhetünk. Az Enter vagy a lefelé nyíl megnyomásával „legördíthetjük" a kiválasztott menüt, és egy almenühöz jutunk. Itt a fel és le nyilakkal mozoghatunk, a jobbra és balra nyilakkal átléphetünk egy másik almenübe, és az Enter gombbal aktivizálhatjuk a kiválasztott menüpontot.

Az almenükben a menüpont nevén kívül számos egyéb információt is láthatunk. Ha egy menüpont halványabban látszik, akkor az a pont nem választható ki. A legtöbb menüpontban található egy eltérő színű betű. Ha ezt a gombot lenyomjuk, akkor az olyan, mintha azon a soron egy Entert ütöttünk volna (a kijelölésnek nem szükséges ezen a soron tartózkodni). Ha a menüpont után három pont (...) található, akkor az Enter után egy dialógus jelenik meg, ahol további paramétereket állíthatunk be. A menü-

pont végi kis háromszög további almenüt jelent. A menüpont végén látható billentyűkombináció azt mutatja, hogy a szerkesztőből hogyan juthatunk gyorsan abba a menüpontba. Pl. az F3 ugyanazt jelenti, mintha lenyitnánk a File menüpontot, és itt az Open-t választanánk.

A főmenüben is látunk kitüntetett betűket, ami azt mutatja, hogy mivel tudjuk lenyitni azt a menüt a szerkesztőből, így nincs szükség az F9, nyilak, Enter sorozatra, hanem a lenyomva tartott Alt gomb és a kitüntetett gomb lenyomásával ugyanazt a hatást válthatjuk ki. Pl. az Alt-F gomb hatására legördül a File menü. A menüben az egérrel is barangolhatunk. A megfelelő menüponton meg kell nyomni a bal egér gombot. (Egy kis színes négyszög mutatja, hogy éppen hol van az egér. Ha az egeret mozgatjuk, akkor a kis négyszög is mozog vele együtt. A továbbiakban azt, hogy valahol megnyomjuk az egér bal gombját, kattintásnak nevezzük.) Ha vissza szeretnénk jutni a menüből a szerkesztőablakba, akkor az ESC gombot kell megnyomnunk (a legtöbb helyről vissza tudunk jutni a szerkesztőbe az ESC gombbal). A képernyő legnagyobb részét a munkaasztal foglalja el. Ide különböző ablakok és dialógusdobozok nyílnak. Az ablakok és dialógusdobozok egymást elfedhetik, és mindig a legfelül lévő az éppen aktuális. Ha egy másikat szeretnénk aktuálissá tenni, akkor kattintsunk rá az egérrel, vagy nyomjuk le az Alt gombot és mellé azt a számjegyet, ami a kívánt ablak felső keretében látható. Az ablakokat és dialógusdobozokat keretvonal szegélyezi, ami dupla, ha az ablak aktív, és egyszeres, ha nem. A dialógusok mindig duplák! Ablakból több is nyitva lehet, de dialógusdobozból csak egy, és addig nem is mehetünk át egy ablakba, amíg be nem zártuk. Amelyik ablakot nem használjuk, azt zárjuk is be, mert könnyítjük az átláthatóságot, és memóriát is takaríthatunk meg.

Minden ablaknak és dialógusnak van neve. Ez a felső keretvonal közepén látható. Ha itt fogjuk meg az egérrel (megnyomjuk a gombot rajta, és nyomva tartjuk), akkor az ablakot vagy dialógust áthelyezhetjük a képernyő egy másik részébe. Az ablakok és dialógusok felső keretének a bal oldalán van egy kis tömör négyszög. Ha ide kattintunk, akkor az ablak bezáródik. Bezáráskor az ablak tartalma elvész, ezért ilyenkor az IDE figyelmeztet minket, és megkérdezi, hogy elmentse-e az adatokat. Az ablakok jobb felső sarkában van egy nyíl. Ez, ha csak felfelé mutat, akkor teljes képernyőssé teszi az ablakot, ha fel és le mutat, akkor visszakicsinyíti. A nagyító dobozon kívül a méretváltoztató sarok — minden ablak jobb alsó sarka — használható egy ablak átméretezésére. Fogjuk meg az egérrel a sarkot, és mozgassuk azt! A szövegszerkesztő ablak alján és jobb oldalán egy-egy gördítősáv helyezkedik el. Ezek segítségével gyorsan mozoghatunk a szövegben függőlegesen (jobb gördítősáv) és vízszintesen (alsó gördítősáv). Ha a végeken lévő nyilakra kattintunk, akkor egy sorral vagy karakterrel mozoghatunk. Ilyenkor a lift (kis

négyszög) árnyékosan mozog, mintha a teljes sáv lenne a teljes szöveg. Az egérrel a liftet is megfoghatjuk és mozgathatjuk. Ha a lift mellé kattintunk, akkor oldalanként lapozhatunk (attól függően, hogy a lifttől merre kattintottunk). Az ablak bal alsó részében lévő két szám adja meg a szövegkurzor helyét a szövegben. Ha mellette egy csillag is megjelenik, akkor a szöveget megváltoztattuk, és az ablak bezárásakor rákérdez a mentésre.

Ha a menüben egy parancs után három pont látható (...), akkor ez azt jelenti, hogy a választáskor egy dialógusdoboz jelenik meg. A dialógusdobozban 5 fajta vezérlési típus van.

— opciókiválasztó doboz (check box),
— állítógomb (radio button),
— vezérlőgomb (push button),
— inputdoboz (input box),
— listadoboz (list box).

Az opcióknál (szögletes zárójel) lehetőségünk van opciók közül választani, és egyszerre többet is kijelölhetünk. Ilyenkor a kiválasztott lehetőségeket egy X-szel jelöli a rendszer. Az állítógombokból (kerek zárójel) egyszerre csak egyet lehet kiválasztani. A vezérlőgombok (pl. OK, Cancel, Help) választásakor közvetlen parancsot adhatunk a doboznak. OK esetén elfogadjuk a beállításokat, Cancel esetén nem fogadjuk el, és kimenekülünk (ez megegyezik az ESC billentyűvel), a Help gomb esetén további információkat kérhetünk a dobozra vonatkozóan.

Az inputdobozba valamilyen bevitelt vár a program (pl. egy fájl nevét). Ha az inputdoboz mellett van egy lefelé nyíl, arra kattintva egy HISTORY doboz fog megjelenni. Ebben a dobozban az ebbe az inputdobozba eddig beírt szövegek vannak. Így egy már korábban beírt szöveget újra kérhetünk.

A listadobozban a felsorolt elemek közül választhatunk. Ha nem férnek el a dobozban, akkor a gördítősávokkal lehet lapozni. A listadoboz rendszerint egy inputdobozzal van együtt. A dobozban való mozgásra felhasználhatjuk az egeret vagy a TAB és SHIFT + TAB gombokat, illetve az Alt + kitüntetett billentyű kombinációkat. Próbáljuk ki az eddig leírt lehetőségeket! A munka során az IDE gyors használata feltétlenül szükséges.

1.4. Ismerkedés a szerkesztővel

A Turbo Pascalnak rendkívül jól használható belső szerkesztője van. Annak is érdemes megismerkedni vele, aki nem programot, hanem „csak" szöveget kíván írni. A szerkesztő használata leginkább a WordStar-ra hasonlít. Úgy kell használni, mintha egy írógéppel gépelnénk. A sorok végét Enter-rel jelezhetjük (nem fontos a képernyő szélén Enter-t nyomni, azon túl

is mehetünk). A szerkesztőben, egy sorba maximum 249 karaktert írhatunk, de a fordító csak 126-ig veszi figyelembe. A hosszabb soroknál figyelmeztet. Ha egy sorban javítottunk, akkor nem kell újra Enter-t ütni a sor végén, mert a szerkesztő folyamatosan tárolja az adatokat. A szerkesztőben számos billentyűkombináció is van. Ezeket a billentyűkombinációkat a Help-ből érdemes megnézni, hogy kezdetben gyorsabban tudjunk szerkeszteni ([25]).

Kurzormozgató billentyűk

A kurzor mozgatását a megfelelő nyilak vagy az alábbi Ctrl + karakter-(ek) segítségével is végezhetjük.

Balra egy karakter	Ctrl + S vagy balra nyíl
Jobbra egy karakter	Ctrl + D vagy jobbra nyíl
Egy szónyit balra	Ctrl + A vagy Ctrl + balra nyíl
Egy szónyit jobbra	Ctrl + F vagy Ctrl + jobbra
Egy sortnyit fel	Ctrl + E vagy fel nyíl
Egy sornyit le	Ctrl + X vagy le nyíl
Scroll fel	Ctrl + W
Scroll le	Ctrl + Z
Lapozás fel	Ctrl + R vagy PgUp
Lapozás le	Ctrl + C vagy PgDn
A sor elejére	Home
A sor végére	End
A dokumentum elejére	Ctrl + Home
A dokumentum végére	Ctrl + End

Blokkparancsok

A blokk a szöveg egy kijelölt része. A kijelölést vagy az alábbi kontroll-karakterekkel vagy a SHIFT + nyíl billentyűkkel végezhetjük. A kijelölt rész más színű lesz, mint a többi szöveg. A blokkműveleteknél nagy szerepet játszik a vágólap (clipboard). Ide másolhatunk kijelölt szöveget, amit később be szeretnénk egy másik ablakba másolni. Ennek segítségével a Help-ből kimásolhatjuk a példaprogramokat is egy szerkesztőablakba, amiket aztán futtathatunk.

Kijelölés kezdete	Ctrl + K B
Kijelölés vége	Ctrl + K K
Egy szó kijelölése	Ctrl + K T
Kijelölt rész másolása	Ctrl + K C
Kijelölt rész mozgatása	Ctrl + K V
Kijelölt rész törlése	Ctrl + K Y

Blokk olvasása lemezről	Ctrl + K R
Blokk írása lemezre	Ctrl + K W
Eltüntetés/előhozás	Ctrl + K H
Kijelölt nyomtatása	Ctrl + K P
Kijelölt egy betűvel beljebb	Ctrl + K I
Kijelölt egy betűvel kijjebb	Ctrl + K U
Sor kijelölése	Ctrl + K L

A vágólap kezelése

Másolás a vágólapra	Ctrl + Ins
Kivágás a vágólapra	Shift + Del
Blokk törlése	Ctrl + Del
Vágólapról másolás	Shift + Ins

Beszúrás és törlés

Beszúrás be/ki	Ctrl + V vagy Ins
Sorbeszúrás	Ctrl + N
Sortörlés	Ctrl + Y
Törlés a sor végéig	Ctrl + Q Y
Balra karakter törlése	Ctrl + H vagy Backspace
Karakter törlése	Ctrl + G vagy Del
Jobbra lévő szó törlése	Ctrl + T

Egyéb parancsok

Menü hívása	F10
Program mentése	Ctrl + K vagy F2
Új szerkesztőablak	F3
Ablak bezárása	Alt + F3
Tab beszúrása	Ctrl + I vagy TAB
Tabulátormód	Ctrl + O T
Automatikus bekezdés	Ctrl + O I
Javított sor vissza	Ctrl + Q L
Helyjelölő lerakása	Ctrl + K n ($n = 0, \ldots, 9$)
Helyjelölő keresése	Ctrl + Q n ($n = 0, \ldots, 9$)
Parancs help	Ctrl + F1

Keresések

Keresés	Ctrl + Q F
Keresés és csere	Ctrl + Q A
Következő keresése	Ctrl + L
Művelet megszakítása	Esc

1.5. A menüsor

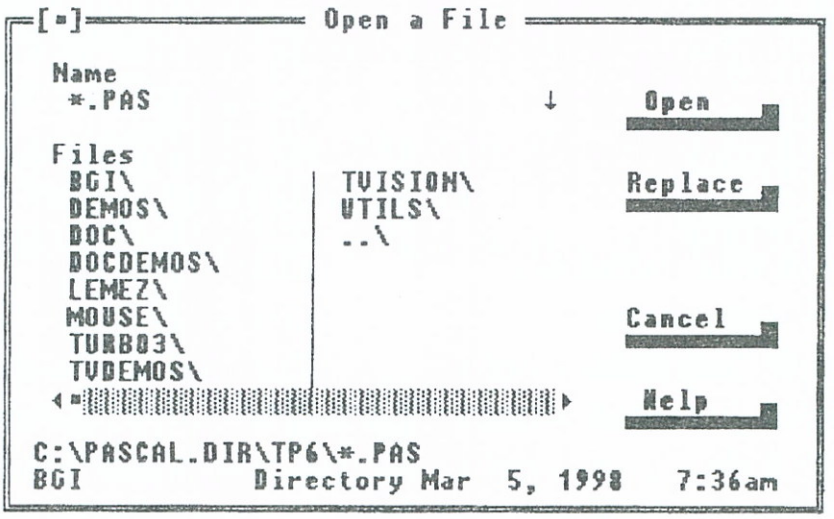
A System menü (–)

Három menüpont kapott itt helyet:
About...: Egy dialógusdobozban közli a Turbo Pascal verziószámát.
Refresh Display: Újra rajzolja a desktopot.
Clear Desktop: Bezárja az összes ablakot.

A File menü

Az Alt + F gombokkal a File menübe jutunk, ahol fájlműveleteket végezhetünk.

Open: a File|Open parancs (F3) segítségével új szöveget tölthetünk be egy új vagy meglévő Edit ablakba. Kiválasztásakor egy dialógusdoboz jelenik meg. A doboz tartalmaz egy inputdobozt, fájlok listáját, Open, Replace, Cancel és Help funkciójú gombot, valamint az információs mezőt, ami a kiválasztott fájl adatait jeleníti meg. Ha teljes nevet adunk meg az inputdobozban, és az Open-t választjuk, akkor a fájl egy új ablakba töltődik be. A Replace esetén az éppen aktuálisat fogja felülírni.



Ha a fájlnév joker (wildcard) karaktereket tartalmaz, akkor a fájllista

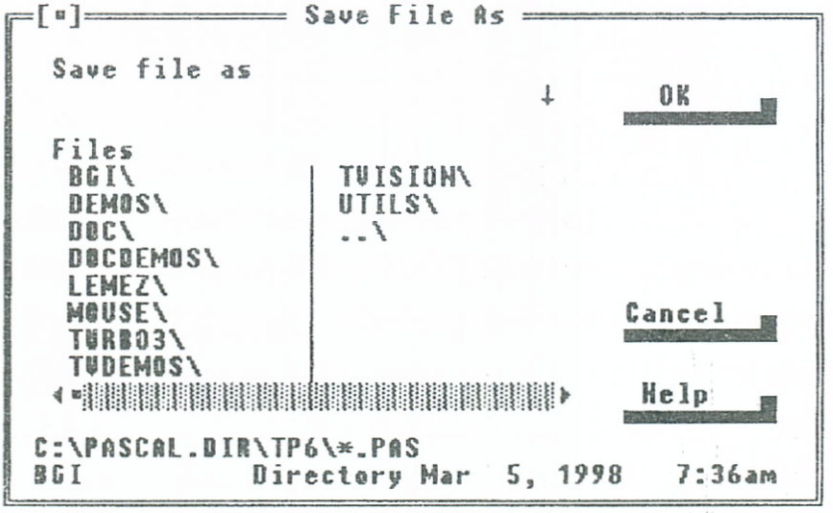
a szűrőnek megfelelően íródik ki. Itt utalhatunk más könyvtárra vagy meghajtóra is, ekkor a lista ennek megfelelően fog változni.

A kurzor le a history listát hozza elő, ahol választhatunk egy korábban betöltött fájlnevet.

File: a File|New parancs egy új ablakot fog nyitni NONAMExx.PAS névvel (ahol az xx 00 és 90 közötti szám, a még névtelen forrásállomány sorszáma). Ha egy ilyen NONAME fájlt akarunk menteni, a rendszer mindig megkérdezi a nevét.

Save: a File|Save parancs (F2) segítségével elmenthetjük az aktuális Edit ablak tartalmát. (Ha az Edit ablak nem aktív, akkor ezt a funkciót nem választhatjuk.

Save As: a File|Save As parancs lehetővé teszi, hogy egy fájlt más néven mentsünk el. (Használata megegyezik az Open parancs alkalmazásával)



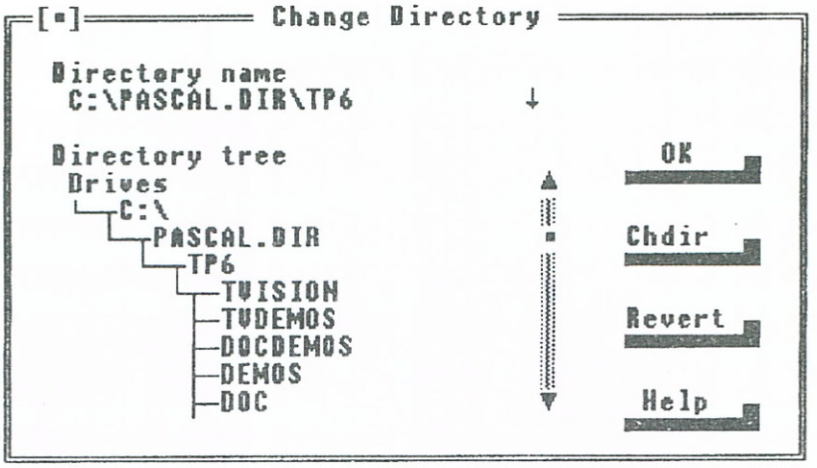
Save All: a File|Save All parancs az összes módosított Edit ablakot elmenti, nem csak az aktuálist.

Change Dir: a File|Change dir paranccsal az aktuális lemezegységet és alkönyvtárat változtathatjuk meg (az ábrát lásd a következő oldalon).

Print: a File|Print parancs az aktív ablak tartalmát kiírja a nyomtatóra. Ha nem aktív egy Edit ablak sem, akkor nem választhatjuk ki. Nyomtatásra használható még a Ctrl + K P billentyűkombináció is.

Get Info: a File|Get info parancs kiválasztásával információt kapunk az IDE állapotáról és a szabad memóriáról.

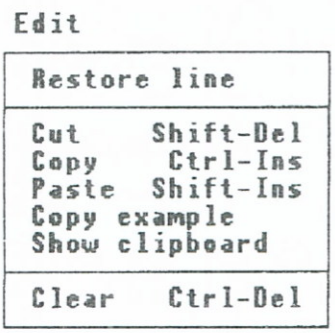
DOS shell: a File|DOS shell parancs lehetővé teszi, hogy ideiglenesen elhagyjuk az IDE-t, és kilépjünk a DOS-ba. Ha vissza szeretnénk jönni az IDE-be, az Exit, DOS parancsot kell használnunk.



Exit: a File|Exit parancs véglegesen elhagyja a rendszert. Ha volt olyan Edit ablakunk, amiben javítottunk, akkor rákérdez a mentésére. A kilépést még az Alt + X-szel is megtehetjük.

Az Edit menü

Az Alt + E az Edit menüre vált. Itt történik a szöveg szerkesztése, kivágása és másolása. A menü legtöbb pontjának használatához szükséges a szöveg egy részének kijelölése (lásd: Ismerkedés a szerkesztővel). A kiválasztott rész más színű lesz, mint a normál szöveg. Szerkesztésnél fontos még a vágólap, ami ideiglenes tárolóként működik.



Restore Line: az Edit|Restore line parancs, az utoljára módosított vagy törölt sort visszaállítja a módosítás előttire.

Cut: az Edit|Cut parancs (Shift + Del) a kijelölt szövegrészt kivágja a szövegből, és a vágólapra teszi.

Copy: az Edit|Copy (Shift + Ins) a kijelölt szövegrészt kimásolja a vágólapra.

Paste: az Edit|Paste parancs (Ctrl + Ins) az utoljára vágólapra került szövegrészt beilleszti a szövegbe, a kurzortól kezdődően.

Copy example: az Edit|Copy Example parancs a Pascal helpjéből a példaprogramot a vágólapra másolja. Ezt úgy is megtehetjük, hogy a példaprogramot kijelöljük, majd a Copy paranccsal kimásoljuk a vágólapra.

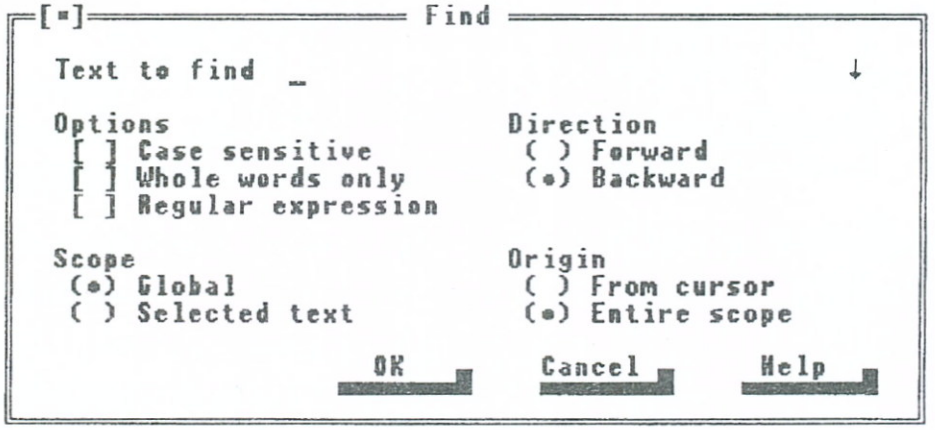
Show clipboard: az Edit|Show clipboard parancsa megnyitja a vágólap ablakot, ahol az eddigi összes kivágás és kimásolás látható. Az aktuális kijelölés más színnel világít.

Clear: az Edit|Clear parancs (Ctrl + Del) törli a kijelölt szövegrészt, de nem helyezi el a vágólapra. Ezzel a vágólapról is törölhetünk.

A Search (keresés) menü

A Search (Alt + S) lehetővé tesz szövegrész-, eljárás- és hibakeresést a forrásfájlban. A megtalált szöveget mással helyettesíthetjük.

Find: a Search|Find (Alt + S F) megjeleníti a Find dialógusdobozt, amely segítségével megadhatjuk a keresési opciókat és a keresendő szöveget.



Az Option részben a következőket állíthatjuk:
Case sensitive: a kis- és nagybetűk között legyen-e különbség?
Whole words only: csak teljes szóazonosság szerint keressen.
Regular expression: grep típusú joker karakterek használhatók.

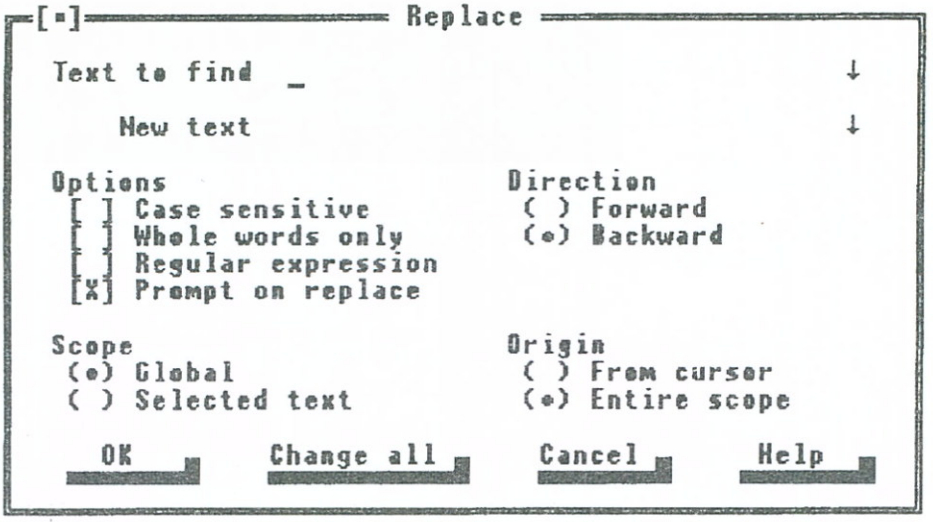
A Direction dobozban megválaszthatjuk a keresés irányát. Forward: előre, Backward: hátra. A Scope dobozban azt állíthatjuk, hogy a teljes szövegben (Global) vagy csak a kijelölt részben keressen (Selected text). Az Origin dobozban a keresés kezdőpontját állíthatjuk. From cursor a kurzortól, Entire Scope a szöveg elejétől indul.

Replace a Search|Replace paranccsal nemcsak kereshetünk, de helyettesíthetünk egy szövegrészt egy másikkal. Felépítése sokban hasonló a Find ablakhoz, csak itt meg kell adnunk, hogy mire szeretnénk lecserélni a keresett szöveget. Az Options ablak egy Prompt on replace sorral bővült, ahol az automatikus cserélés vagy a rákérdezés között választhatunk. A beállítások után az OK vagy a Change all gombbal indíthatjuk el a cserélést. A Change all az összes előfordulót lecseréli, míg az OK csak az elsőre megtaláltat (az ábrát lásd a következő oldalon).

Search Again: a Search|Search Again paranccsal (CTRL + L) az előző keresést vagy cserét folytathatjuk.

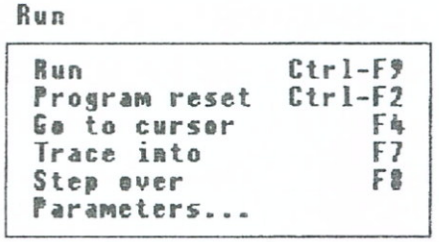
Go to Line Number: a Search|Go To Line Number paranccsal egy megadott sorszámú sorra vihetjük a kurzort a szövegben.

Find Procedure: a Search|Find Procedure parancs megkeresi a megadott eljárást vagy függvényt a forráslistában.



Find Error: a Search|Find Error (Alt + F8) parancs megkeresi a futási idő alatt előforduló hibákat. Ha a kész programot futtatjuk az IDE-n kívül, akkor a hibák helyéről csak egy memóriacímet kapunk. Ilyenkor a forrásprogramot lefordítjuk a fejlesztőben még egyszer, és megadjuk a hiba címét ebben a menüpontban, ekkor az IDE megkeresi a hiba helyét.

A Run (futtatás) menüpont



A Run menüpontban (Alt + R) azok a parancsok találhatók, amelyek a program futtatásához szükségesek.

Run: a Run|Run parancs (Ctrl + F9) elindítja a programot. Ha szükséges (pl. módosult a forrásszöveg), akkor újrafordítja.

Program reset: a Run|Program reset (Ctrl + F2) befejezi a nyomkövetést, felszabadítja a program által lefoglalt memóriaterületet, és lezárja a megnyitott fájlokat.

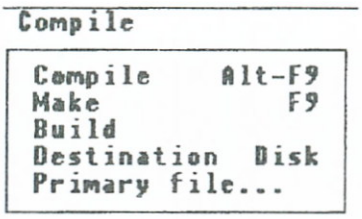
Go to cursor: a Run|Go to cursor (F4) a kurzor sorára egy töréspontot helyez el, és elindítja a programot (a töréspontot lásd később).

Trace Into: a Run|Trace Into (F7) utasításonként hajtja végig a programot, eljáráshoz vagy függvényhez érve azt is soronként hajtja végre.

Step Over: a Run|Step Over (F8) parancs is soronként futtatja a lefordított programot, de az eljárásokat és függvényeket egy lépésben hajtja végre.

Parameters: a Run|Parameters parancshoz tartozó dialógusdobozban a programnak átadandó parancssor paramétereket adhatjuk meg. A változások csak a következő fordításkor lépnek érvénybe.

A Compile menüpont



A Compile menüpont (Alt + C) tartalmazza a fordítással és összeszerkesztéssel kapcsolatos parancsokat.

Compile: a Compile|Compile parancs (Alt + F9) segítségével lefordíthatjuk az aktuális forrásfájlt. A programot csak lefordítás után futtathatjuk.

Make: a Compile|Make parancs (F9) fordítás előtt ellenőrzi a programhoz kapcsolódó összes fájlt, és ha szükséges, akkor lefordítja őket, majd fordítás után összefűzi a részeket.

Build: a Compile|Build parancs újrafordítja az összes fájlt, nem veszi figyelembe a keletkezési idejüket.

Destination: a Compile|Destination paranccsal megadhatjuk, hogy a kész program hova fordítódjon: lemezre vagy a memóriába. A memória gyorsabb fordítást eredményez, de gyakran a program mérete miatt szükséges a lemezre történő fordítás is. Ha a programot az IDE nélkül is szeretnénk futtatni, akkor mindenképpen lemezre kell fordítani. A lemezre fordításkor egy .EXE fájl keletkezik.

Primary file: a Compile|Primary file parancs segítségével megadhatunk egy elsődleges fájlt, így bármelyik ablakban is vagyunk, a fordítás ezzel a fájllal kezdődik.

A Debug menü

A Debug menü (Alt + D) a hibakeresést segíti. A szintaktikai hibákat a fordító megtalálja, de a logikai hibákat nekünk kell megkeresni.

Evaluate/Modify: a Debug|Evaluate/Modify (Alt + F4) hatására megjelenő ablak Expression sorába megadhatjuk a vizsgálni kívánt kifejezést, majd az Enter után a Result sorban megjelenik a kifejezés értéke. A New sorban új értéket adhatunk neki, ha lehetséges. Az ablak jól használható számológépként is.

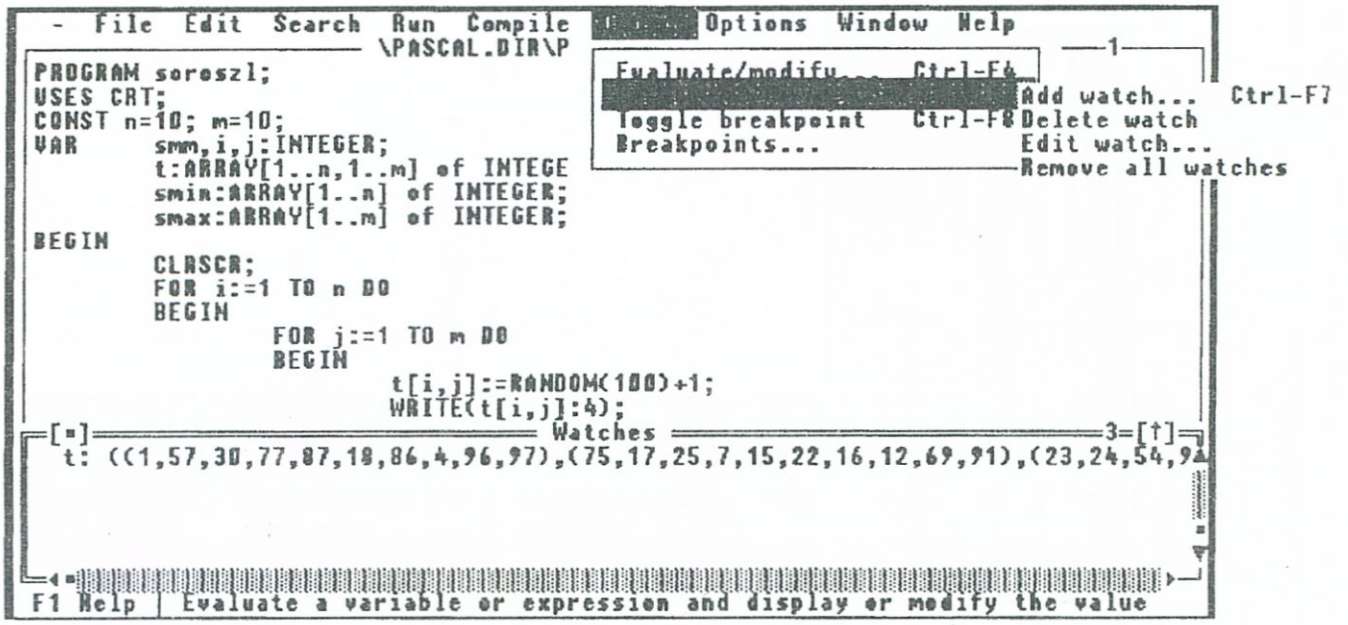
Watches: a Debug|Watches paranccsal a változó figyelő almenüt nyithatjuk meg (az ábra a következő oldalon).

Add watches parancs (Ctrl + F7): új kifejezést vihetünk a megfigyelési ablakba.

Delete watch: a jelenleg kijelölt kifejezést törli a megfigyelési ablakból.

Edit watch: a kifejezést módosíthatjuk.

Remove all watches: az összes bejegyzést töröljük a figyelőablakból.



Toggle breakpoint: a Debug|Toggle (Ctrl + F8) breakpoint menüben a kurzor sorába egy töréspontot tehetünk le vagy szüntethetünk meg. Ha a program futása során egy olyan sorhoz ér, ahol töréspont van, akkor a program futása felfüggesztődik, és a sort látjuk az ablakban.

Breakpoints: Debug|Breakpoints menü egy dialógusdobozt nyit meg, ahol a töréspontjainkat módosíthatjuk. Az OK gombbal folytathatjuk a szerkesztést, Edit gombbal a látható dialógusdobozhoz jutunk:

Ebben a dialógusdobozban állíthatjuk be kézzel a töréspontokat.

Condition: feltételek a töréspontra vonatkozóan.

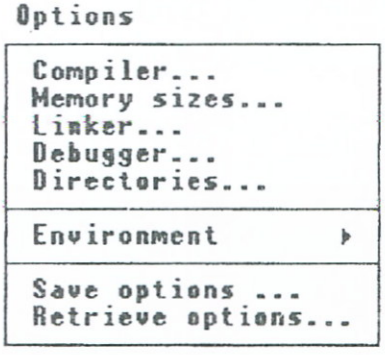
Pass count: hányszor lépje át a program ezt a sort?

File name: melyik forrásfájlban van a töréspont?

Line number: melyik sorban van a töréspont?

A Breakpoints dobozban a Delete gombbal törölhetjük a kiválasztott töréspontot, a View gomb a töréspontra viszi a kurzort a forrásfájlban, a Clear all az összes töréspontot törli.

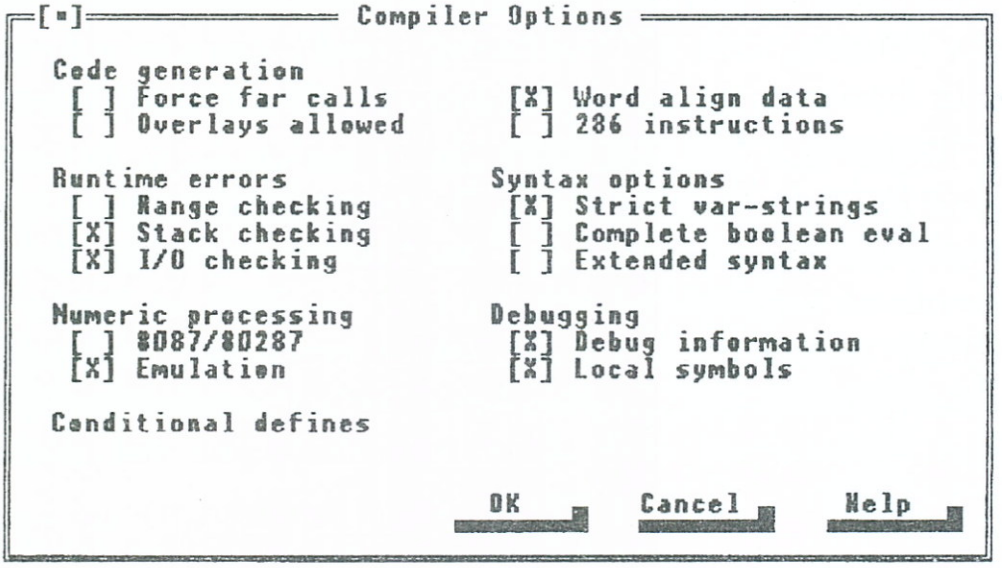
Az Options menüpont



Az Options menüben (Alt + O) mindenféle beállítások vannak mind az IDE-re, mind a program fordítására.

Compiler: az Options|Compiler menüben a fordításra vonatkozó beállítások vannak (az ábrát lásd a következő oldalon).

Force far call: a fordítás során minden cím két szón tárolódik, így lehetővé válik a távoli címzés is. A program hosszabb lesz, de néha szükséges a használata (pl. megszakítások írásakor). (Megegyezik a **$F** direktívával.)



Overlays allowed: átfedéses technika engedélyezése csak akkor szükséges, ha használjuk is az átfedéses technikát (Ilyen esetben csak a program azon része van a memóriában, ami szükséges, így lehetőségünk van nagyon nagy programok írására is) (Megegyezik a **$O** direktívával).

Word align data: szóhatárra igazított változó használat. Bekapcsolva, a program hoszszabb lesz, de néhány gépen gyorsabban fog futni (80186 és 80286 processzorok előnyben részesítik azokat az adatokat, amelyek szóhatárra kerülnek) (Megegyezik a **$A** direktívával).

286 instructions: a kész programban lesznek olyan utasítások, amelyek a 80286-os processzorban benne vannak, de a 8088-as processzorokban nincsenek. Ezek a programok gyorsabban futnak a 80286-os gépeken, de a korábbiakon egyáltalán nem fognak futni. (Megegyezik a **$G** direktívával.)

Range checking: a program a futás során figyeli a túlcímzéseket és a változók értékhatárát. Ha valahol túlcsordulás van, akkor hibajelzéssel leáll. Kikapcsolt állapotban nem fog megállni. (Programtesztelés során érdemes bekapcsolni, kivéve, ha célunk, hogy a túlcímzést megengedjük, pl. dinamikus tömböknél.) (Megegyezik a **$R** direktívával.)

Stack checking: a program a futás során figyeli a vermet, és ha megtelik, akkor hibaüzenettel leáll. Kikapcsoláskor nem fog megállni. (Megegyezik a **$S** direktívával.)

I/O checking: a program figyeli a fájlműveletek során fellépő hibákat. Ha hiba fordul elő, akkor a program hibaüzenettel leáll. Kikapcsoláskor nem fog megállni. (Megegyezik a **$I** direktívával.)

Strict var-strings: bekapcsolt állapotban a fordító figyeli a cím szerint

átadott paraméterek hosszát, és ha eltérő, akkor fordítási hibával leáll. Előfordulhat azonban, hogy az átadott string hossza nem egyezik meg az eljárásban vagy függvény fejrészében megadottal, de a visszaadott string biztosan el fog férni. Ilyenkor ki kell kapcsolni ezt az opciót, hogy a fordító engedélyezze. (Megegyezik a **$V** direktívával.)

Complete boolean: a logikai kifejezések előrelátását kapcsolhatjuk be vagy ki. (Megegyezik a **$B** direktívával.)

Extended syntax: ha bekapcsoljuk, akkor függvényt is hívhatunk úgy meg, mint egy eljárást. A visszatérési érték elvész. (Megegyezik a **$X** direktívával.)

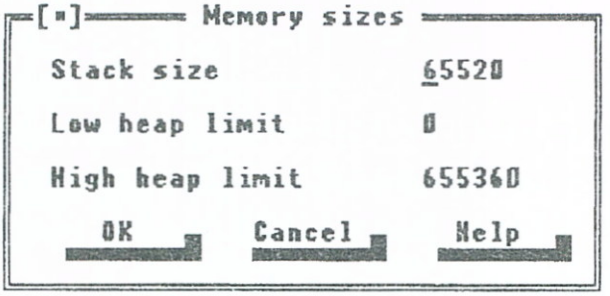
8087/80287: engedélyezi a matematikai processzor használatát. Bekapcsolásakor bizonyos műveletek gyorsabbak lesznek, és használhatók speciális adattípusok is. A program csak akkor futtatható, ha van matematika processzor. (Megegyezik a **$N** direktívával.)

Emulation: ha nincs matematikai processzor, akkor szoftveresen emulálja azt. (Megegyezik a **$E** direktívával.)

Debug information: ha bekapcsoljuk, akkor lehetséges a nyomkövetés. Bekapcsolt állapotban a kész program hosszabb lesz, és futása is lassabb, ezért ha a programot készre fordítjuk, akkor kapcsoljuk ki. (Megegyezik a **$D** direktívával.)

Local symbols: lokális változókról információt helyez el a programban. A Debug opcióval együtt használjuk. (Megegyezik a **$L** direktívával.)

Memory sizes: az Options|Memory sizes paranccsal a felhasználható memória méretét határozhatjuk meg. Stack size: a verem mérete, Low heap limit: minimális szabad memória a program futtatásához, High heap limit: maximálisan felhasznált memória mennyisége.



Linker: az Options|Linker dialógus doboza, az egyes részek összefűzéséhez kapcsolódó opciókat állíthatjuk be. A map file meghatározza, hogy milyen információs fájl keletkezzen az összefűzésről, a Link buffer pedig azt határozza meg, hogy az összefűzés hol történjen, lemezen vagy memóriában. Természetesen a memóriában gyorsabban történik, de nem mindig van elegendő szabad hely.

Debugger: az Options|Debugger dialógusban a nyomkövetést állíthatjuk be. Az Integrated a belső nyomkövetőhöz készít információkat, a Standalone a külső nyomkövetőkhöz (pl. TURBO DEBUGGER). A Display swaping azt határozza meg, hogy nyomkövetéskor mikor kell a szerkesztőképernyőről a programképernyőre váltani. None: semmikor, Smart: csak ha van a képernyőre írás, Always: minden sor végrehajtásakor.

Directories: az Options|Directories dialógusban azt adjuk meg, hogy mit hol kell keresnie a rendszernek. Egy-egy dobozban több könyvtárat is megadhatunk, ilyenkor pontosvesszővel (;) kell elválasztani. Ha az EXE & TPU sor üres, akkor az EXE és TPU kiterjesztésű fájlok az aktuális könyvtárban keletkeznek.

Az Environment almenüben az IDE-re vonatkozó beállítások szerepelnek.

Preferences: az Options|Environment|Preferences dobozban a következőket állíthatjuk:

Screen sizes: a képernyőméretet 25 vagy 43/50 sorosra állítja. Ez azonban függ a videokártyától.

Source Tracking: az új fájl betöltésének helye. New file: új ablak keletkezik, Current window: a jelenlegi ablakba töltődik be.

Auto save: az automatikus mentést állíthatjuk be; Editor files: szerkesztendő fájl, Environment: környezeti beállítások, Desktop: munkaasztal. Ez azt jelenti, hogy amikor kilépünk az IDE-ből, akkor a beállított részek elmentődnek, és a legközelebbi indításnál ez lesz az érvényes.

Desktop file: A desktop file hova kerüljön? None: sehova, Current directory: jelenlegi könyvtár, Config file directory: ahova a config fájlok kerülnek. Ez rendszerint a \TP.

Editor: az Options|Environment|Editor dialógusdobozban a szövegszerkesztővel kapcsolatos beállítások vannak.

Create backup files: készüljön-e biztonsági másolat a javított fájlokról?

Insert mode: a szerkesztő beszúrásos vagy felülírásos módban induljon?

Autoindent mode: automatikus bekezdések.

Use tab characters: a Tab gomb hatására a Tab (a szóközök) kerüljön a szövegbe.

Optimal fill: a sorok elején Tab-ok (szóközök) optimálisan lesznek.

Backspace unindent: a törlőgomb is kövesse a bekezdéseket.

Cursor through tabs: bekapcsoláskor a kurzorral a tabon belül is tudunk mozogni, míg kikapcsolt állapotban a kurzor átugorja.

Mouse: Options|Environment|Mouse dialógusdobozban az egér paramétereit állíthatjuk be.

Right mouse button: a jobb egérgombnak adhatunk funkciót.

Mouse double click: a dupla kattintás közötti időt állíthatjuk be.

Reverse mouse buttons: a két egérgomb felcserélését jelenti.

Startup: Options|Environment|Startup dialógusdobozban az IDE indításakori konfigurálása lehetséges. Az itt beállítottak csak a következő indításnál lépnek életbe.

Dual monitor support: ha a gépünkben két videokártya van (hercules + egyéb), akkor lehetőség van mind a kettő használatára. Ilyenkor a herculesen az IDE-t látjuk, míg a másikon a program képernyőjét.

Graphics screen save: grafikus képernyő elmentése. Az IDE több memóriát foglal, ha be van kapcsolva, de a VGA grafikus képernyő nem borul össze a nyomkövetéskor.

EGA/VGA palette save: EGA/VGA videokártyánál elmenti a paletta színeket. Általában a Graphics screen save választással használjuk együtt.

CGA snow checkin: régebbi CGA videokártyáknál a gyors képre írás „havazást" eredményezett a képernyőn. Ennek kiküszöbölésére használjuk ezt az opciót.

LCD color set: az LCD kijelzőhöz állítja be a színeket.

Use expanded memory: ha bekapcsolt állapotban van, akkor az IDE megpróbálja egy részét felrakni az expanded (EMS) memóriába. Így több helyet biztosít a készülő programnak.

Load TURBO.TPL: betöltse az alapértelmezésbeli Unitokat, vagy ne?

Window heap size/Editor heap size/Overlay heap size: az IDE mire mennyi memóriát tartson fent magának?

Swap file directory: az IDE bizonyos részét a merevlemezre teszi ki. Itt adhatjuk meg, hogy hol legyen ez a hely. (Ha van ramdiskünk, érdemes azt megadni.)

Colors: itt az IDE színeit állíthatjuk be. Az első ablakban csoportok vannak, a másodikban a csoporton belüli részek. Ezeknek a részeknek a színét határozza meg a Foreground és a Background. A Foreground az írás színét, a Background a háttérszínt határozza meg. Alattuk az éppen beállított színeket látjuk.

Save options: Options|Save options a beállításokat menthetők el. Ha \TP\TURBO.TP néven mentjük el, akkor a következő IDE indítástól a mostani lesz az alapbeállítás.

Retrieve options: Options|Retrieve options segítségével egy elmentett beállítást tölthetünk be.

A Window menüpont

A Window menüben az IDE ablakait állíthatjuk át, hívhatjuk be, vagy éppen becsukhatjuk.

Size/Move: a Window|Size/Move (Ctrl + F5) paranccsal az aktuális ablakot mozgathatjuk vagy átméretezhetjük. A mozgatás a nyíl gombokkal történik, az átméretezés a SHIFT és a nyíl gombok együttes lenyomásával. A befejezést az Enter-rel jelezzük.

Zoom: a Window|Zoom (F5) paranccsal egy ablakot tehetünk gyorsan teljes méretűvé, majd visszaállíthatjuk az előző méretet.

Tile: a Window|Tile parancs úgy rendezi el az ablakokat, hogy mindegyik látszódjon.

Cascade: a Window|Cascade parancs úgy rendezi el az ablakokat, hogy azok egymás mögött legyenek.

Next: a Window|Next (F6) parancs a következő ablakot teszi aktuálissá.

Previous: a Window|Previous (Shift + F6) parancs az előző ablakot teszi aktuális ablaknak.

Close: a Window|Close (Alt + F3) parancs bezárja az aktuális ablakot.

Watch: a Window|Watch parancs megnyitja a változó figyelőablakot.

Register: a Window|Register parancs megnyitja a processzor regisztereit kijelző ablakot.

Output: a Window|Output parancs nyit egy ablakot, amiben a futó program képernyője látszik.

Call stack: a Window|Call stack (Ctrl + F3) parancs megnyitja a verem ablakot.

User screen: a Window|User screen (Alt + F5) átkapcsol a futó programra.

List: a Window|List (Alt + 0) dialógusban mutatja az összes nyitott ablakot. Itt kiválaszthatjuk az aktuális ablakot, és a Delete gombbal bezárhatunk ablakokat.

A Help menü

A Help menüben segítséget kereshetünk a Turbo Pascal szabályairól, elemeiről és magáról az IDE-ről is ([25]). Ha megnyitunk egy Help ablakot, akkor abban vannak fényesebb szavak is. Ha ezekre a szavakra klikkelünk, vagy a kurzort ráállítva megnyomjuk az Enter gombot, akkor egy újabb help-et kapunk a fényes szóval kapcsolatban. Az egérrel vagy a SHIFT és a nyilakkal kijelölhetünk egy tetszőleges részt, és azt a vágólapra másolhatjuk. Így a példaprogramokat is kimásolhatjuk, és egy szerkesztőablakba bemásolva azonnal ki is próbálhatjuk.

Contents: a Help|Contents parancs a Help tartalomjegyzéke. Egy Help ablakot nyit, amelyben a fontosabb fejezetcímeket olvashatjuk.

Index: a Help|Index (Shift + F1) parancsra a Help lapok tartalomjegyzékét kapjuk abc-sorrendben.

Topic search: a Help|Topic search (Ctrl-F1) azt a Help lapot fogja megadni, amelyet az a szó azonosít, amin a kurzor áll. Ha nincs ilyen Help lap, akkor az indexbe kerülünk. Így gyorsan kérhetünk információt egy-egy Pascal-parancsról.

Previous topic: a Help|Previous topic (Alt + F1) visszafelé lapoz a Helpben. Az IDE a Helpben való barangolásunk során megjegyzi a megnézett lapokat, és így tud visszafelé is haladni.

Help on help: a Help|Help on help parancs egy rövid információt ad a Help használatáról. Ez egyébként a Contents-ben is megtalálható.

2. A Pascal program

Javasoljuk, hogy az előzőekben felsorolt és taglalt fejlesztői környezeti lehetőségeket, a programírás közben alkalmazzuk rendszeresen. Most, hogy már ismerjük a programfejlesztő környezet lehetőségeit, foglalkozzunk részletesebben a Pascal programnyelv szabályaival, és a program felépítésével. Azt előre el kell mondani, hogy meglehetősen szigorú a fordítóprogram. Pontos, de jól tanulható szabályokat kell fejünkbe vésni. Ez a fejezet éppen ezért eléggé „száraz" lesz, de kárpótolja az olvasót az, hogy ha ezen a részen túljutunk, és sikeresen elsajátítottuk az anyagot, akkor nem kell örökösen a fordító hibaüzeneteit olvasgatni azért, hogy már megint nyelvtani hibát ejtettünk, ami egy idő után eléggé bosszantó.

A legelső téma, amit át kell tekintenünk: a Pascal nyelv jelölései: szintaktikai elemei. A PC-ken található jelrendszerek alapja az ASCII karakterkészlet. Mint tudjuk, ez 256 féle karaktert jelent. Ezt a jelkészletet használja a Turbo Pascal rendszer is. Egyes jeleknek azonban lényegesen nagyobb szerepük van a nyelv szempontjából, mint másoknak.

2.1. A Pascal nyelv jelölései: szintaktikai elemek

A Pascal programok szimbólumokból és szimbólumelválasztókból állnak ([14]). Ez azt jelenti, hogy a program kódolásakor meghatározott jeleket, jelcsoportokat használhatunk annak érdekében, hogy a program szintaktikailag helyes legyen, és a fordító egyértelműen értelmezhesse.

A programban használható jeleket, jelsorozatokat a következő csoportokba soroljuk:

1. Betűk. Az angol ábécé 26 nagy- és 26 kisbetűjét használhatjuk a program kódolása során. A magyar ékezetes betűk szintaktikai elemként nem használhatók. Azokat csak karakterfüzérek elemeiként vagy megjegyzésekben alkalmazhatjuk.

2. Számjegyek. Számjegyként a tízes számrendszerben használatos (decimális) számjegyeket (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) használhatjuk.

Ezekből a karakterekből épülnek fel a további nyelvi elemek, amelyek tulajdonképpen karaktersorozatok, de egyértelmű jelentéssel rendelkeznek.

3. Speciális szimbólumok. A kötött jelentésű karaktereket, karaktersorozatokat hívjuk speciális szimbólumoknak. Ezeket a nyelv szintaktikai szerkezeteinek megadásához használjuk, és meghatározott funkciók kötődnek hozzájuk.

Különleges karakterek

<, >, +,	Használatuk ugyanaz, mint a matematikában.
–, *, /, =	(A * a szorzásjel.)
_	Aláhúzójel. Névben (azonosítóban) szerepelhet „betűként". Valójában nem aláhúzójel, mert két karakter egy pozíción a Turbo Pascal programban nem lehet.
@	Mutató típusú változók címzésében használjuk.
,	Szerepelhet felsorolásokban elválasztójelként.
.	Tizedespontként és programvége-jel funkciójára is használjuk.
;	Utasításokat elválasztó szimbólum.
#	Hess jel (hash marker). Karakterek ASCII kódjainak jelölésénél írható.
$	Hexadecimális alakban megadott számok előtt kell megadni.
^	Mutató típusú változókat jelölünk vele.
(,)	A matematikában megszokott jelentésük van, de használjuk őket különböző felsorolásoknál is.
[,]	Főleg tömbváltozók indexelésére és halmazkonstansok jelölésénél használjuk. Nincs a matematikában megszokott jelentésük.
{, }	A benne foglaltak csak magyarázó szöveget jelentenek.
'	Felülvessző. Kezdő és záró idézőjelként is funkcionál.

Kétkarakteres speciális szimbólumok

A kétkarakteres szimbólumok két karakterből állnak. Azok logikailag és formailag is összetartoznak. Így van a szimbólumnak meghatározott jelentése. Karaktereik között nincs (nem lehet) elválasztó, mert akkor két egykarakteres szimbólummá válnak.

:=	Az értékadó utasítás szimbóluma.
..	Az intervallumtípus deklarációjában alkalmazzuk.
<=	Relációjel. Jelentése: kisebb vagy egyenlő ($\leq$), halmazoknál $\subseteq$.
>=	Relációjel. Jelentése: nagyobb vagy egyenlő ($\geq$), halmazoknál $\supseteq$.
<>	Relációjel. Jelentése: nem egyenlő ($\neq$)
(*	Magyarázó szöveg kezdő zárójele ({)
*)	Magyarázó szöveg vége (})

Bizonyos értelemben a különböző zárójelpárokat is ide sorolhatjuk, mert azok csak együtt szerepelhetnek a programban.

Alapszavak

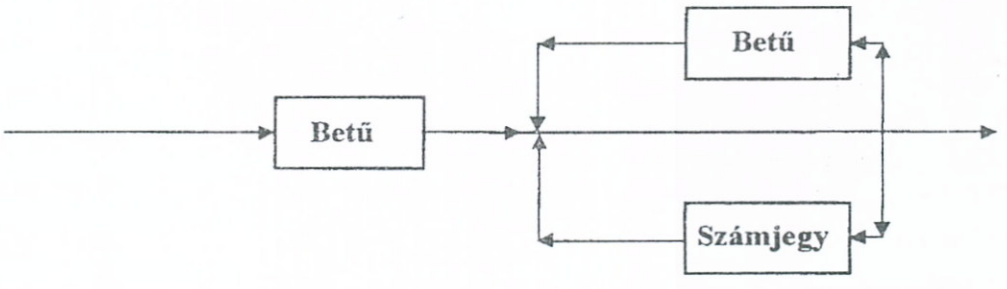
Az alapszavak karakterei rögzített jelentésű szimbólumokat alkotnak ([14]). Ezek a szavak foglalt szavak. Csak a Pascal nyelv definíciójában lefektetett szabályok szerint használhatók. Alapszó soha sem lehet név (azonosító). Az alapszavak kis- és nagybetűk sorozatai, és a fordító nem tesz különbséget a kis- és nagybetűk között. Kezdetüket vagy végüket nem jelöljük speciális karakterekkel.

A Turbo Pascal alapszavai a következők ([24]):

absolute	end	inline	procedure	type
and	external	interface	program	unit
array	file	interrupt	record	until
begin	for	label	repeat	uses
case	forward	mod	set	var
const	function	nil	shl	while
div	goto	not	shr	with
do	if	of	string	xor
downto	implementation	or	then	
else	in	packed	to	

Név (azonosító)

A programokat és alprogramokat (eljárásokat és függvényeket), az egységeket (Unitokat), konstansokat, típusokat, változókat a nevükkel jelöljük. A név (azonosító): betűvel kezdődő, betűvel vagy számmal folytatódó, tetszőleges hosszúságú karaktersorozat lehet, de a fordító csak az első 63 karaktert azonosítja, és a kis- és a nagybetűk között nem tesz különbséget.



A nevekben előforduló, egymásnak megfelelő kis- és nagybetűket azonosnak tekinti a rendszer. A betűk halmazába most az angol ABC 26 kis- és nagybetűje és az aláhúzójel tartozik. Az azonosítás — az első 63 karakterig — karakterenként történik, így két név már akkor is különböző, ha a hosszuk nem egyezik. Ha csak a 64. karaktertől tér el két név egymástól,

akkor azok azonos neveket jelentenek.

A rendszer a program minden területén egyértelműen tudja azonosítani a különböző blokkokban deklarált azonos neveket a minősítő azonosító segítségével ([24]). A minősítő azonosító is egy név (Program neve, Unit neve, Record neve), amit pont (.) követ, és ezután következik az a név, amely a minősítés nélkül nem lenne egyértelmű.

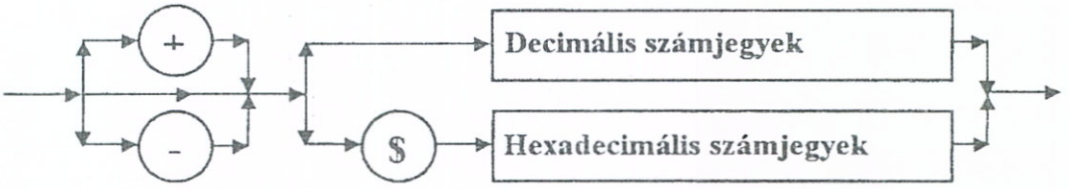
```
program nevek;
uses crt;
var red :byte;
rekord :record
red:string;
end;
begin red:=2;
      rekord.red:='alma';
end.
```

```
=[■]================================ Watches
  red: 2
  crt.red: 4
  rekord.red: 'alma'
```

Számkonstans

A Turbo Pascal programban a számkonstansok (számok) lehetnek egészek vagy valósak ([3]).

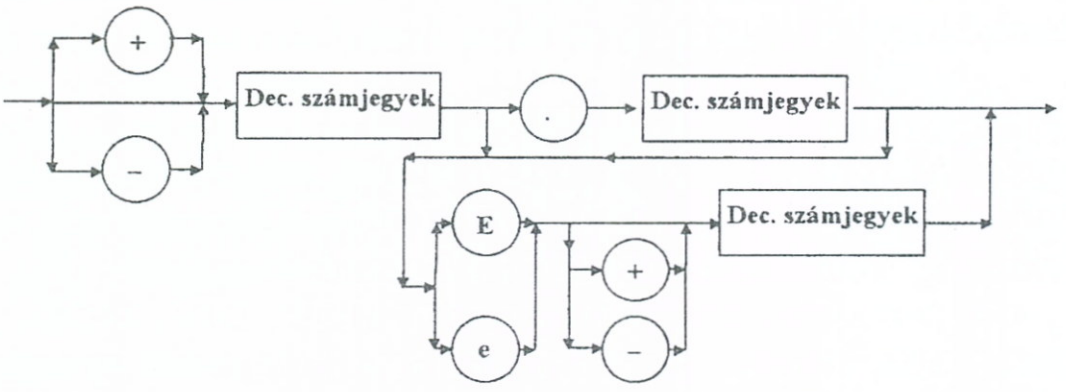
Az egész számkonstansok lehetnek decimális vagy hexadecimális alakúak.



A decimális számkonstansok írásakor — a már korábban felsorolt és a számolásnál is szokásos — 10 számjegyet és a + vagy − jeleket mint előjeleket használhatjuk. A hexadecimális számnak is lehet előjele. A hexadecimális számjegyeket a decimális számjegyek és az A, B, C, D, E, F vagy az a, b, c, d, e, f betűk alkothatják. Az első számjegy előtt a $ jellel kell jelölni azt, hogy hexadecimális szám következik. Ezután legfeljebb 8 hexadecimális számjegy következhet. Az egész számkonstansok értékei a $[-2147483648, 2147483647]$ intervallumban lehetnek.

A valós típusú számkonstansok a tizedestörteket jelentik a Pascal programban. Ezeket a számokat hasonlóan írhatjuk, mint ahogy azokat a számolásnál megszoktuk. Számjegyei csak decimális számjegyek lehetnek. A számnak lehet előjele is. A tizedespontot kell használni a szám egész és tört részének elválasztására, pl. −322.5. A pont határolójel, tehát a pont után lennie kell számjegynek. A 32. írásmód hibás. Helyette, például 32 vagy 32.0

írandó. 32 írásakor egész típusúnak, 32.0 esetén valós típusúként ábrázolja a számot a fordító. A valós számkonstansok írásakor 10 hatványaival is megszorozhatjuk a számot, de a 10 helyett az E vagy az e betűt kell alkalmazni. Az E (e) betűk az exponens szóra utalnak. Természetesen a hatványkitevőnek is lehet előjele. Pl.: −322.5 helyett −3.225E2 vagy −3225e−1 is írható.



SZAMOK.PAS

```
PROGRAM SZAMOK;
USES CRT;
CONST C=3.14E3;
CONST B=$12;
VAR A:REAL;
BEGIN READLN(A);
WRITELN(A,' ',B,' ',C);
READLN;
END.
```

Output

```
2345.123E-2
2.3451230000E+01 18 3.1400000000E+03
```

Szövegkonstans

A szövegkonstans — más néven karakterlánc vagy string — karakterek sorozatát jelenti. A szöveg úgy adható meg, hogy a karakterek sorozatát felülvesszők (') közé zárjuk ([17]). A szövegkonstans karaktereinek száma a $[0, 255]$ intervallumban lehet. Karakterként minden ASCII kódú karakter szerepelhet, kivéve a felülvessző és az Enter karaktereket. Pl.: 'Üdvözöllek dicső lovag.', 'Üss le egy tetszőleges billentyűt!'. Az üres szövegkonstanst a '' jelenti. Ha ' karaktert szeretnénk látni a szövegben, akkor két felülvessző karaktert kell tenni a szövegkonstansba, pl. ''''Turbo Pascal''''.

A Turbo Pascal programban a szövegkonstans úgy is megadható, hogy az egyes karakterek ASCII kódja előtt a # jelet kell írni. Ezzel a lehetőség-

gel különböző vezérlőkaraktereket is elhelyezhetünk a szövegkonstansban. Ennek igen nagy jelentősége van, ha a különleges karakter nem írható le egyszerűen a billentyűzetről, vagy a karakter valamilyen vezérlő karakter, pl. #10, #13, #7. Ez egy Enter (kocsi vissza, soremelés) és egy hangjelzés. A következő példa idézőjelben helyezi el a Turbo Pascal szöveget: #39Turbo Pascal#$27. Mint látható, az ASCII kód megadható decimálisan és hexadecimális alakban is.

Operátorok

Az operátor szó mindazokat a műveleteket, relációkat jelenti, amelyeket az operandusokkal végzünk. Közülük már többet felsoroltunk a különleges karaktereknél és a kétkarakteres speciális szimbólumoknál. Pl.: +, −, /, *, =, <, >, <=, >=, <>. De ide tartoznak a NOT, DIV, MOD, AND, OR, XOR, SHL, SHR, IN stb. szimbólumok is. A különböző operátorokat és a kifejezéseket még a későbbiekben részletesen elemezzük.

Címke

A címke egy helymegjelölő szimbólum. Utasítás helyét szimbolizálja. A címkéket is kell a Pascal programban deklarálni. Címke lehet név vagy előjel nélküli decimális egész szám, pl. Cimke1, ide2, 21, 0, 9999. Az egész számnak a $[0, 9999]$ intervallumban kell lennie. Számjegyekkel megadott esetben a vezető 0 számjegyek nem jelentenek más címkét ([24]). A 21, 0021 vagy a 0000021 címkék ugyanazt a címkét jelentik. Ha egy utasítást címke előz meg, akkor GoTo Címke utasítással hivatkozhatunk az utasításra.

Fordító direktívák

A fordítóprogram számára kiadott speciális utasításokat nevezzük fordító direktíváknak. Ezek megváltoztathatják a lefordítandó program néhány tulajdonságát. A direktívákat kapcsos zárójelek közé kell tenni. A direktíva előtt $ jel van, pl. {$N+} vagy {$R−}. Néhánnyal az adott fejezetben részletesen foglalkozunk.

Elválasztó szintaktikai elemek

A szintaktikai elemek speciális osztályát alkotják az elválasztó szintaktikai elemek. A szóközt (a space karaktert, a billentyűzeten rendszerint hosszú billentyű), a sorvége jelet (kocsi vissza, soremelés jelet; az Enter billentyű), a kommenteket elválasztóként is használhatjuk a Pascal szimbólumok között.

— A Pascal szimbólumok belsejében elválasztó és annak része sem lehet.
— Bármely két név, szám vagy alapszó között legalább egy elválasztó elem kell!
— Tetszőleges számú elválasztó elem állhat két egymást követő szintaktikai elem között.

— Ha egy elválasztó elemet kicserélünk egy másikra, akkor a program értelme nem változik.

A komment magyarázó szöveg. Kapcsos zárójelek vagy a (*, *) kétkarakteres szimbólumpár között helyezzük el a szöveget. Tetszőleges, karakterekből és sorvégekből álló jelsorozat lehet, de { vagy *) nem lehet benne. Hossza nincs korlátozva. A fordítóprogram nem veszi figyelembe. A program készítőjének vagy felhasználójának szóló tájékoztató, magyarázó szöveg. Segít eligazodni a programban.

A ; is elválasztó jel, abban az értelemben, hogy az utasításokat a ; karakterrel választhatjuk el egymástól. A ; kitétele nem szükséges az End előtt. Az Else elé pontosvesszőt tenni tilos! A program végét nem pontosvessző (;), hanem pont (.) jelzi.

A program sorai

Itt jegyezzük meg azt, hogy a program sorai legfeljebb 126 karakter hosszúak lehetnek. Ennek ellenére a program írásakor nem javasoljuk a hosszú sorok írását, mert ekkor a program áttekinthetetlenné válik. A Turbo Pascal szerkesztője támogatja olyan program írását, amikor a program sorai strukturáltan látszanak. Használjuk ki e lehetőségeket a program gépelésekor.

2.2. Adattípusok a Turbo Pascalban

Az adat szó gyűjtőfogalom. Adatnak nevezünk minden objektumot, amivel a számítógép műveletet végez. A gépi kód szintjén az adatot bináris számsor jelenti. A magas szintű programozási nyelvek viszont lehetővé teszik, hogy az adatot logikai konstrukcióként kezeljük, és elvonatkoztassunk a tényleges ábrázolás részleteitől ([14]). A programnyelvek többségében fontos szerepet játszanak az adattípusok. Általánosan elfogadott szabály az, hogy a deklaráció fejezi ki explicit módon a konstans, változó vagy függvény típusát, és ez a deklaráció a programban megelőzi az illető konstans, változó vagy függvény alkalmazásait. Ez a szabály különösen akkor tűnik ésszerűnek, ha figyelembe vesszük, hogy egy fordítóprogramnak kell majd megválasztania az objektumok ábrázolását a számítógép tárában ([27]).

A Pascal programnyelv típuskoncepciója a következőket jelenti:

1. Egy adattípus meghatározza azt az értékhalmazt, amelyhez egy konstans tartozik, amelyből egy változó vagy egy kifejezés értéket vehet fel, vagy ahová egy művelet vagy egy függvény értéket generál.

2. Egy konstanssal, változóval vagy kifejezéssel megadott érték típusa mindig megállapítható formája vagy deklarációja alapján anélkül, hogy a számítást végre kellene hajtani.

3. Minden művelet vagy függvény meghatározott típusú argumentumokat vár, és meghatározott típusú eredményt ad vissza.

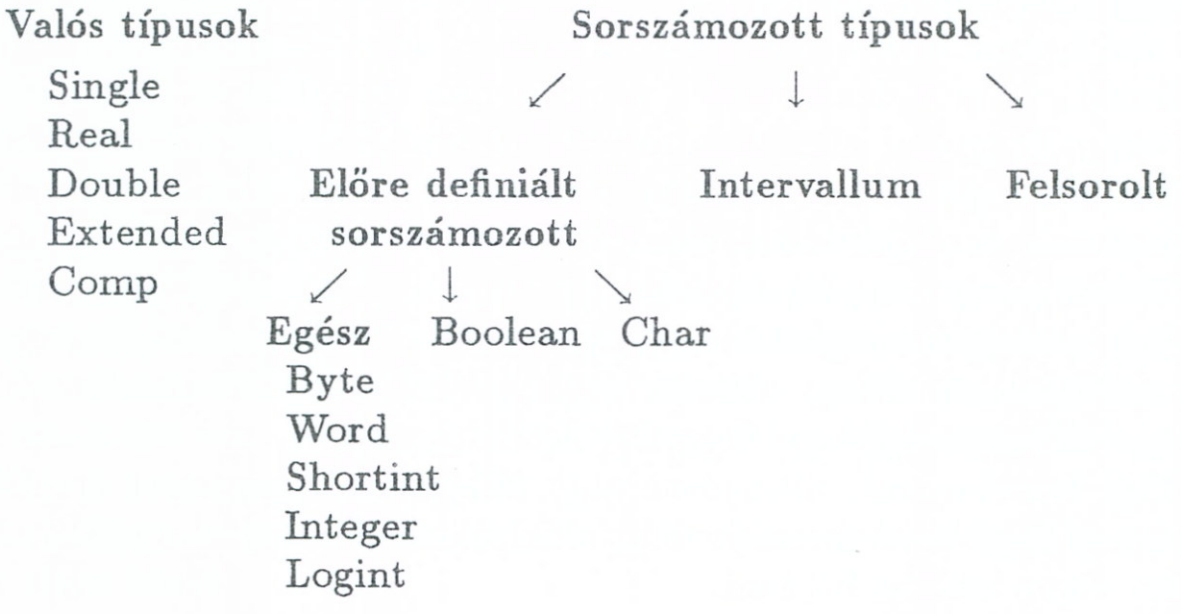
A fordítóprogram a típusokra vonatkozó információkat felhasználja az objektumok **ábrázolásához**, a műveletek elvégezhetőségének **ellenőrzéséhez** és olyan **típusok definíciójához**, amelyeket a fordító standard módon nem tartalmaz, vagy az összetett típust most deklaráljuk már definiált összetevőtípusokkal. Így lehetséges az, hogy a fordító már a forrásnyelvű program fordításkor — a típusok tekintetében — az adatok ábrázolásának optimalizálására törekszik, a műveletek elvégzése nélkül is kiszűri a típus összeférhetetlenségi hibáit, és a program kódja strukturáltsága és jó felépítettsége miatt jól definiált. Bevezetőnkben erre a tulajdonságra is gondoltunk akkor, amikor a Pascal nyelvet szigorú programozási nyelvnek neveztük. A Turbo Pascal típuskoncepciójával tananyagunkban még többször találkozunk. Az egyes adattípusok ismérveinek részletesebb tárgyalása végigvonul a tananyagon. Most először tekintsük át azokat az adattípusokat, amelyeket a Turbo Pascal fordító standard módon ismer!

Alapvetően a következő csoportokba sorolhatók a Turbo Pascalban megengedett adattípusok:

— egyszerű,
— karakterlánc,
— strukturált,
— mutató.

Az **egyszerű típusok** mindig egyetlen adat hovatartozását definiálják, és a programozás szempontjából tovább nem bonthatók. Az egyszerű adattípushoz minden esetben rendezett részhalmazok tartoznak. Az egyszerű adattípusok csoportjának két ága van.

Egyszerű adattípusok



Léteznek azonban olyan típusok is, amikor nem egy adat, hanem egy egész adatcsoport hovatartozását határozzuk meg. Az azonos vagy különböző típusú adatcsoportokat fogják össze egy szerkezetté az összetett vagy strukturált adattípusok. Az strukturált adattípusok a következő hierarchia alapján épülnek föl:

Tömb Rekord Fájl Halmaz

Ezek után vizsgáljuk meg néhány adattípus jellemzőit. Most csak azoknak az adattípusoknak az áttekintésével foglalkozunk, amelyeknek az elemzésére a továbbiak megértéséhez feltétlenül szükségünk van. A többi adattípus tárgyalására a későbbiekben térünk ki. A típusok tanulmányozása során mindig szem előtt kell tartanunk a típusok hierarchiáját mutató diagramokat.

2.2.1. Egyszerű adattípusok

2.2.1.1. Valós típusok

A Turbo Pascal nyelvben a valós típusú értékek halmaza nem azonos a matematikában tanult valós számok halmazával. Csak a véges tizedes törtek bizonyos részhalmazairól van szó az egyes valós típusoknál. A valós típusú számkonstansok programbeli használatával a Turbo Pascal szintaktikai elemeinél már foglalkoztunk. A tárban, lebegőpontos formában ábrázolja a fordító. Korábbi tanulmányainkból tudjuk:* ez azt jelenti, hogy minden számot bináris normálalakban $(m \cdot 2^k)$ képzelünk el, ahol m a mantisszát, k pedig a karakterisztikát jelenti az $\frac{1}{2} \leq |m| < 1$ feltételrendszer teljesülése mellett. Mivel az $|m|$ első, legmagasabb helyi értékű jegye mindig 1, ezért csak az ettől jobbra lévő számjegyeket ábrázolja, annyi pozíción, amennyi az adott típusnál rendelkezésre áll. Az így kapott értéket ábrázolt mantisszának (ÁM) nevezzük. A bitek számának növekedésével a szám ábrázolt számjegyeinek száma nő. Az előjeles karakterisztikát úgy ábrázolja binárisan, hogy az ábrázolandó számhoz hozzáad 2^{n-1}-et, és az így kapott számot ábrázolja (K). Neve: többletes ([21]) vagy feszített ábrázolás. Az n azt jelenti, hogy hány bit áll rendelkezésre, a karakterisztika ábrázolására. Ennek értéke is az ábrázolt típusoktól függ. Növelve a bitsorozat hosszát, nő az ábrázolható értéktartomány. A típus teljes méretéhez hozzátartozik 1 biten az előjel is. Ennek a bitnek az értéke pozitív mantissza esetén 0, negatívnál 1 (ME). Az ÁM és a K értékeire felhasználható bitek száma és sorrendje a Turbo Pascal nyelvben az ábrázolt valós típustól függ, de az ábrázolás elve minden típusban azonos.

---

* A számábrázolás részleteit korábbi tanulmányaiból elevenítse fel!

ME	ÁM	K

A táblázatban az egyes valós típusok esetén csak az ábrázolt szám decimálisan értendő pozitív intervallumát, a számjegyek számát és a tárolásnál felhasznált bitek számát tüntettük fel. Ezt az információt a Turbo Pascal Help menüpontjában is megnézhetjük ([25]).

Valós típusok

A Turbo Pascal-nak öt előre definiált valós típusa van. Minden egyes típus egy bizonyos tartományon és pontosságon belül van:

Típus	Tartomány	Szám-jegyek	Bájt méret
real	2.9e-39..1.7e38	11-12	6
single	1.5e-45..3.4e38	7-8	4
double	5.0e-324..1.7e308	15-16	8
extended	3.4e-4932..1.1e4932	19-20	10
comp	-9.2e18..9.2e18	19-20	8

A valós típusokat úgy használjuk a programokban, mint a matematikában a tizedes törteket. Néhány fontos megkülönböztetés az egész típusokkal szemben azonban van. Ezekre az eltérésekre, valamint az értelmezett műveletekre, eljárásokra és függvényekre az anyag tárgyalása során később kitérünk. Most csak néhány fontos esetet említünk:

— A Real típus minden Pascal verzióban használható megkötések nélkül. Azonban a többi típus csak numerikus társprocesszorral, vagy annak emulálásával léptethető működésbe {$N+, $E+}; ellenkező esetben hibaüzenetet kapunk. A valós konstans ezért általában Real, {N+} esetén Extended típusú lesz. A comp típus ábrázolási módja fixpontos ábrázolás 64 biten, előjelesen, de a programban valós típusként kell kezelni.

— A valós típusú számok nem lehetnek indexváltozók, és nem vezérelhetik a CASE szerkezetet sem!

— Egy osztás eredménye mindig valós típusú.

Műveletek

A következő sorokban — most csak — felsoroljuk azokat a műveleteket, amelyeket valós típusú operandusok esetén alkalmazhatunk. A *Kifejezések* című fejezetben az operandusok és az eredmény típusával ismét foglalkozunk.

Egy operandusú műveletek:

+, – A + művelet nem változtatja az operandust, míg a – művelet képezi az operandus ellentettjét.
Pl. +13.46E–3, –9.0, –43.54e–4.

Két operandusú, multiplikatív műveletek:

*, /	Szorzás, osztás. Osztásnál az eredmény mindig valós típusú. Pl. 43.23*12.4, -56.0*4.0E-2, 8.2/2, 10.4/2.0.

Két operandusú additív műveletek:

+, -:	Összeadás, kivonás, pl. 3.2E0+21-4.0, 78.4+2.5.

Összehasonlítások:

=, <>, <, <=, >, >=	Az eredmény logikai érték. Vigyázzunk ezeknek a relációjeleknek az írására! A két karakterből álló szimbólumok csak így írhatók a Turbo Pascalban. Pl. (-2.3<-2.2)=True, (2.34=0.234E+1)=True, (4.5<>45E-1)=False.

Eljárások, függvények

A valós típusú paramétereknél alkalmazható **standard eljárásokról** és függvényekről is részletesen szó lesz a későbbiekben. Most csak felsoroljuk nevüket.

Eljárások:	Randomize, Str, Val
Függvények:	Abs, ArcTan, Cos, Exp, Frac, Int, Ln, Random, Round, Sin, Sqr, Sqrt, Trunc

2.2.1.2. Sorszámozott (megszámlálható) típusok

Minden lehetséges értékükhöz egy-egy sorszám tartozik és értékei véges — a sorszámuk szerint rendezett — sorozatot alkotnak, melyből következik az is, hogy van legnagyobb és legkisebb elem is. A sorszámok: $0, 1, 2, \ldots$ értékek, kivéve az egész típusokat, ahol a sorszám maga a szám. Az értékek közötti reláció a sorszámok alapján értékelődik ki. Alkalmazhatók rájuk speciális függvények is. Ezekkel a függvényekkel egy későbbi fejezetben foglalkozunk.

1. Előre definiált sorszámozott típus

A Turbo Pascal több **előre definiált sorszámozott típust** is ismer. Ez azt jelenti, hogy a fordító alapértelmezés szerint definiáltnak tekint néhány típust.

(a) Egész típusú számok

Az egész típusú számok sorszáma maga a szám. Ezeket a számokat fixpontosan ábrázolja a fordító. Ez azt jelenti, hogy az egész számoknál a bináris pont fix helyen van. Előjel nélküli fixpontos számábrázolás esetén

a nem negatív egész számok pontos ábrázolása valósul meg a felhasználható bitek számától függően. Ezeknél a típusoknál negatív számokat nem lehet ábrázolni, de a kettes komplemens segítségével ebben a tartományban is elvégezhetők az alapműveletek. Az előjeles számok ábrázolására a program olyan fixpontos ábrázolást használ, amikor az előjelet a legnagyobb helyi értékű bit jelzi. A negatív számokat a kettes komplemens segítségével ábrázolja. Ekkor már a negatív számok körében is elvégezhetők az alapműveletek.

Tekintsük át az egész típusok fajtáit és értékhatárait:

TÍPUS	ÉRTÉKHATÁR	TÁROLÁS
Byte	0..255	előjel nélküli 8 bit
Shortint	−128..127	előjeles 8 bit
Word	0..65535	előjel nélküli 16 bit
Integer	−32768..32767	előjeles 16 bit
Longint	−2147483648..2147483647	előjeles 32 bit
Comp	$-2^{63}+1..2^{63}-1$	előjeles 64 bit (de valós típusnak számít)

A Comp típus ábrázolása ugyan fixpontos — mint már a valós típusoknál is említettük — de alkalmazásuk alapján a valós típusok közé soroljuk. Használatához aritmetikai segédprocesszor vagy annak emulálása szükséges. Az egész típusoknak — mint az ábrán is látható — több fajtája van. Azt, hogy melyiket választjuk ezek közül, az szabja meg, hogy milyen a feladat alaptermészete. Minden művelet, amelyik az egyik típuson elvégezhető, az elvégezhető a másikon is. Ha jól választottunk típust, akkor a típuson értelmezett műveletek pontosan hajtódnak végre. Az egész típusú számok estén mindig van legkisebb és legnagyobb szám.

Értéktartomány	Típus
−2147483648..−32769	Longint
−32768..−1	Integer
0..255	Byte
256..32767	Integer
32767..65535	Word
65536..2147483647	Longint

Az adatok típusa mindig egyértelmű a programban. A fordító már a fordítás előtt ellenőrzi, hogy az adatok típusai a deklarációkban, az adatokon végzett műveletekben, az eljárásokban és függvényekben megfelelnek-e a szabályoknak.

Az egész számkonstansnak is egyértelmű típusa van. Ezt a fordító határozza meg, törekedve az optimális megoldásra. Egy egész konstans konkrét típusát a látható táblázat mutatja.

Integer és Longint típusoknál előre definiált konstansok is vannak: Maxint=32767, MaxLongint=2147483647.

2147483647-nél nagyobb számok csak valós típusúként adhatók meg. Az előjel megadása a típust nem változtatja meg.

Műveletek

Egy operandusú aritmetikai műveletek:

Az eredmény is egész típusú lesz.

+	a változatlanul hagyás operátora, pl. +8, +$F00A, +V.
–	az operandus kettes komplementerét képezi, pl. –A, –5, –($2).

NOT (aritmetikai NOT): az egész típusú operandus egyes komplemensét képezi. A típust nem változtatja meg. Pl. NOT 3=–4.

Két operandusú, multiplikatív műveletek:

*	szorzás, pl. 4*B. Az eredmény csak akkor egész típusú, ha mindkét operandus egész típusú.
/	osztás, pl. A/B. Az eredmény mindig valós típusú.

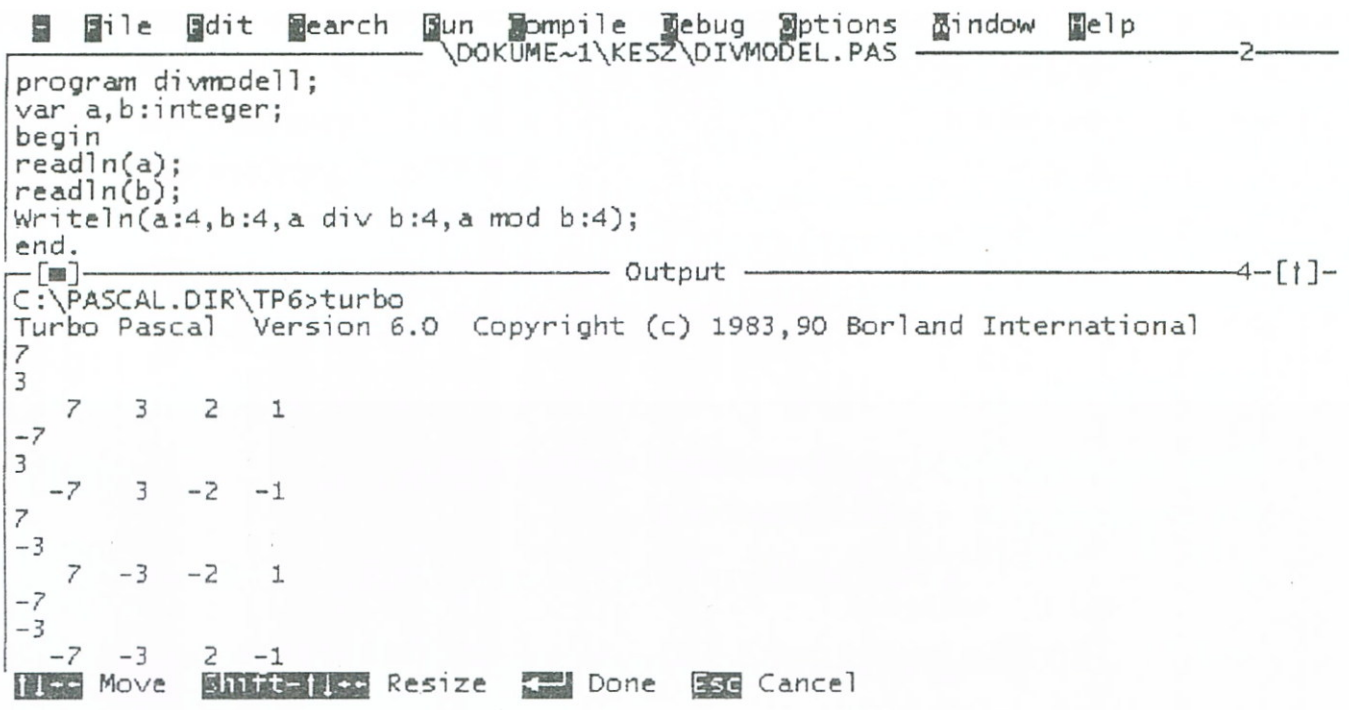
DIV: egész osztás. Az eredmény is egész típusú lesz. A matematikában szokásos maradékos osztástól eltérően, nem a hányados egész részét veszi. Az eredmény abszolút értékét az operandusok abszolút értékeinek maradékos osztása adja. Ha az operandusok azonos előjelűek, akkor az eredmény pozitív lesz, egyébként pedig negatív számot kapunk. Pl. 21 DIV 4 = 5, –9 div 4 = –2.

MOD: egész osztás maradéka (A MOD B = A–A DIV B *B.) Egész típusú eredményt kapunk. Pl. 21 MOD 4=+1, –9 mod 4 = –1. (Lásd a következő oldali ábrát.)

AND (bitenként AND): a következő értéktáblázat alapján:

AND	0	1
0	0	0
1	0	1

Pl. 5 AND 4 = 4.



SHL: balra tolás. A SHL n esetén A bitjei n értékével balra tolódnak ($n \geq 0$). Pl. 5 SHL 3 = 40, −5 SHL 1 = −10.

SHR: jobbra tolás. A SHR n esetén A bitjei n értékével jobbra tolódnak ($n \geq 0$). Pl. 5 SHR 2 = 1.

Két operandusú additív műveletek:

+: összeadás. Az eredmény csak akkor egész típusú, ha mindkét operandus egész típusú.

−: kivonás. Az eredmény csak akkor egész típusú, ha mindkét operandus egész típusú.

OR (bitenként OR): Értéktáblázata a következő:

OR	0	1
0	0	1
1	1	1

Pl. 5 or 4 = 5.

XOR (bitenként XOR): Értéktáblázata:

XOR	0	1
0	0	1
1	1	0

Pl. 5 XOR 4 = 1.

A NOT, DIV, MOD, AND, SHL, SHR, OR, és az XOR műveletek esetén az operandus(ok) és az eredmény is egész típusú.

Összehasonlító műveletek (relációk): <, >, <=, >=, <>.

Egész típusú operandosok esetén — mint minden más sorszámozott típusnál — az összehasonlító műveletek az operandusok sorszáma alapján értékelődnek ki. Például: 5 < 3 értéke FALSE.

A Pascal fordítónál nemcsak a konstansok típusa, hanem a műveletek eredményének típusa is egyértelműen meghatározott.

Ha a *, MOD, DIV, +, – vagy az AND, OR és XOR műveletek mindkét operandusa egész típusú, akkor a művelet eredményének típusát a következő táblázat ([3] adja:

	ShortInt	Integer	LongInt	Byte	Word
ShortInt	Integer	Integer	LongInt	Integer	LongInt
Integer	Integer	Integer	LongInt	Integer	LongInt
LongInt	LongInt	LongInt	LongInt	LongInt	LongInt
Byte	Integer	Integer	LongInt	Integer	Word
Word	LongInt	LongInt	LongInt	Word	Word

— Ha a *, + vagy – műveletek valamelyik operandusa valós, akkor az eredmény $N– esetén Real, $N+ mellett Extended típusú lesz.
— Az A SHL n és A SHR n operandusai csak egész típusúak lehetnek, és az eredmény típusa megegyezik az A típusával.
— A NOT B művelet operandusa csak egész típus lehet. A művelet a típust nem változtatja meg.

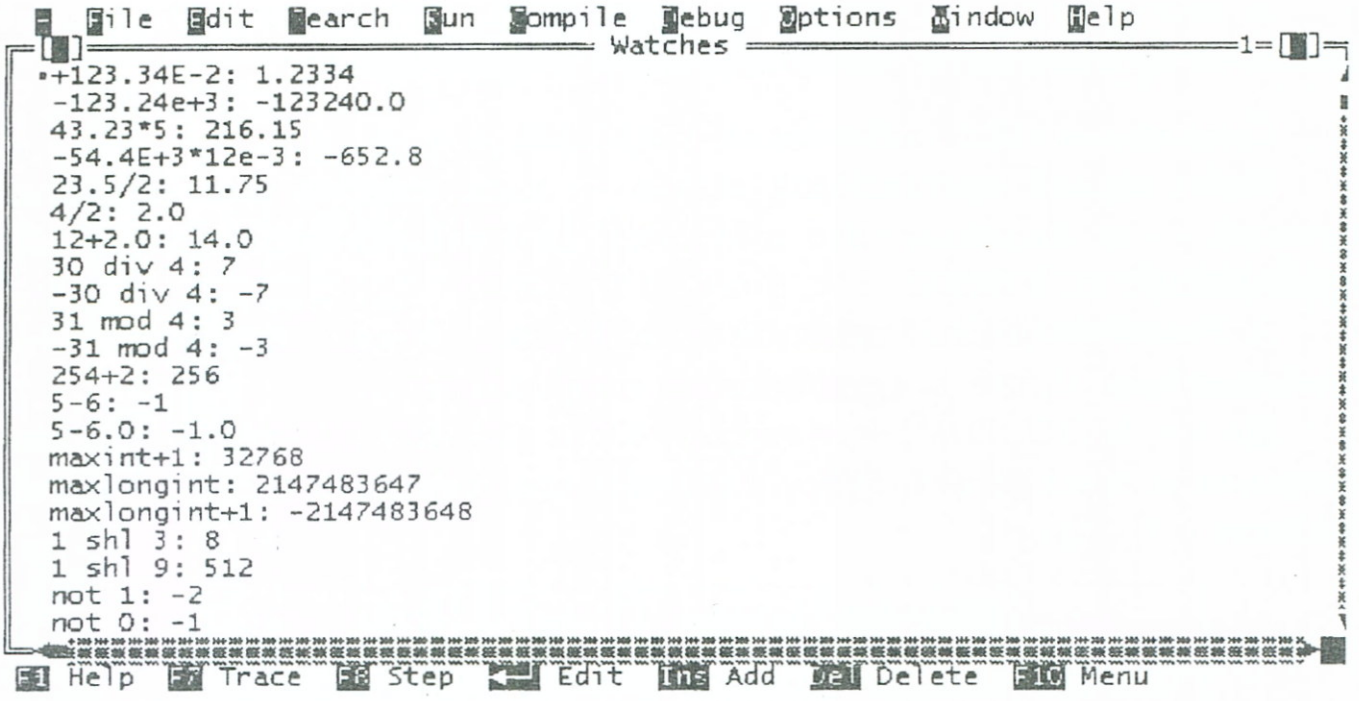
Feladat. A valós és az egész típusok elemzése után javasoljuk, hogy a fejlesztői környezet adta lehetőségeket segítségül híva, ellenőrizze az eddig tanult típusokról leírtakat! Használja a Turbo Pascalt egyszerűen számológépként! Nyissa ki a Window menüből a Watch ablakot maximális méretűre a Zoom (F5) segítségével, hogy minél több sort láthasson egyszerre az ablakban. Próbálja ki a bemutatott példához hasonlóan a tárgyalt típusokat és műveleteiket! (Az ábrát lásd a következő oldalon.)

(b) A logikai (Boolean) típus

A logikai típus (Boolean*) is sorszámozott típus. Tárolása 1 bájton történik.

---

* George Boole (1815—1864) a logika algebrájának úttörője. A matematika ezen ágát nevezzük Boole-algebrának.



Két logikai konstans van: a TRUE és a FALSE; a logikai igaz és a logikai hamis. A FALSE sorszáma 0, a TRUE sorszáma pedig 1. Köztük a FALSE<TRUE reláció áll fenn, sorszámuk alapján.

Logikai műveleteknél az operandusok és az eredmény is logikai típusú.

Igazságtáblázataik:

P	NOT P
FALSE	TRUE
TRUE	FALSE

P AND Q	FALSE	TRUE
FALSE	FALSE	FALSE
TRUE	FALSE	TRUE

P OR Q	FALSE	TRUE
FALSE	FALSE	TRUE
TRUE	TRUE	TRUE

P XOR Q	FALSE	TRUE
FALSE	FALSE	TRUE
TRUE	TRUE	FALSE

Összehasonlító műveletek (relációk): <, >, <=, >=, <>, IN

A halmazelem-vizsgálat (IN) esetén az egyik operandus halmaz típusú, de az eredmény mindig logikai típusú. Ezért szokták ezt az összehasonlító műveletet a logikai műveletek közé is sorolni. Pl. True IN [False,True] művelet eredménye True. A halmaz típusnál az IN műveletet pontosabban leírjuk.

A kifejezés címszó alatt a logikai kifejezésekről és a logikai műveletek prioritásáról is szólunk.

(c) A karaktertípus (Char)

```
[■]──────────────────────── KARAKTER.PAS ────────────────────────2-[↑]┐
program karakterek;{32..255}                                          ▲
uses crt;                                                             ■
var bkod:byte;                                                        ▒
begin clrscr;                                                         ▒
for bkod:=32 to 255 do write(bkod:4, char(bkod):4);                   ▒
readln                                                                ▒
end.                                                                  ▼
└─────1:1──────────────────────────────────────────────────────────────┘
```

32		33	!	34	"	35	#	36	$	37	%	38	&	39	'	40	(	41	)
42	*	43	+	44	,	45	-	46	.	47	/	48	0	49	1	50	2	51	3
52	4	53	5	54	6	55	7	56	8	57	9	58	:	59	;	60	<	61	=
62	>	63	?	64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O	80	P	81	Q
82	R	83	S	84	T	85	U	86	V	87	W	88	X	89	Y	90	Z	91	[
92	\	93	]	94	^	95	_	96	`	97	a	98	b	99	c	100	d	101	e
102	f	103	g	104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w	120	x	121	y
122	z	123	{	124	|	125	}	126	~	127	Ç	128	ü	129	é	130	â	131	ä
132	à	133	ů	134	ć	135	ç	136	ł	137	ë	138	Ő	139	ő	140	î	141	Ź
142	Ä	143	Ć	144	É	145	Ĺ	146	Ľ	147	ô	148	ö	149	Ľ	150	ľ	151	Ś
152	ś	153	Ö	154	Ü	155	Ť	156	ť	157	Ł	158	×	159	č	160	á	161	í
162	ó	163	ú	164	Ą	165	ą	166	Ž	167	ž	168	Ę	169	ę	170	¬	171	ź
172	Č	173	ş	174	«	175	»	176	▒	177	▓	178	▓	179	▒	180	┤	181	Á
182	Â	183	Ě	184	Ş	185	╣	186	═	187	╗	188	╝	189	Ż	190	ż	191	┐
192	└	193	┴	194	┬	195	├	196	─	197	┼	198	Ă	199	ă	200	╚	201	╔
202	╩	203	╦	204	╠	205	═	206	╬	207	¤	208	đ	209	Đ	210	Ď	211	Ë
212	ď	213	Ň	214	Í	215	Î	216	ě	217	┘	218	┌	219	▒	220	▄	221	Ţ
222	Ů	223	▀	224	Ó	225	ß	226	Ô	227	Ń	228	ń	229	ň	230	Š	231	š
232	Ŕ	233	Ú	234	ŕ	235	Ű	236	ý	237	Ý	238	ţ	239	´	240	-	241	˝
242		243	ˇ	244	˘	245	§	246	÷	247	¸	248	°	249	¨	250	˙	251	ű
252	Ř	253	ř	254	■	255													

A karaktertípus (Char) értékkészlete az ASCII karakterkészlet. Ebben a típusban a $[0, 255]$ intervallumban minden egyes egész számhoz egy karaktert rendelünk. Ezeket a sorszámokat nevezzük a karakterek ASCII kódjainak. A karakterek egy byte-ot foglalnak el a memóriában. Ezen byte értéke az ASCII kód, képernyőn való megjelenítési formája a karakter képe.

Műveletek: az összehasonlító műveletek: $<$, $>$, $<=$, $>=$, $<>$.
Az összehasonlítás a karakterek sorszáma alapján történik.

Függvények és eljárások: Chr, Dec, FillChar, Inc, Ord, Pred, Succ, UpCase.

A karakterek belső kódjaival kapcsolatosan nézze át a „Pascal program szintaktikai elemei" című fejezetben írottakat is. Ha a belső kódok és karaktereinek táblázatát kívánja tanulmányozni, akkor azt valamelyik szakirodalom alapján is megteheti ([3], [9]). A táblázat megtanulása nem szükséges, inkább kis programok gyors elkészítését és futtatását alkalmazza. (Lásd a fenti ábrát.)

2. Az intervallumtípus (résztartománytípus)

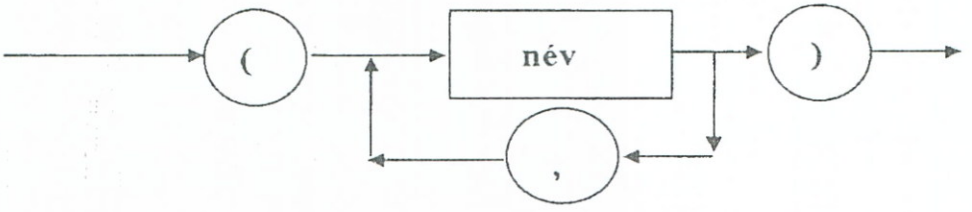
Az intervallumtípus (résztartománytípus) esetén egy már definiált, sorszámozott típus részsorozatát határozzuk meg. Ekkor meg kell adni az inter-

vallum alsó és felső végpontját meghatározó kifejezéseket. Ezek a kifejezések csak fordítási időben kiszámítható kifejezések* lehetnek. Fontos, hogy az alsó korlát (sorszáma) ne legyen nagyobb a felső korlátnál. Pl. 1..5, 0..127, 'a'..'z', 1..2*SOR, 1..OSZL, ahol SOR és OSZL azonosítók, deklarált konstansokat jelentenek.

Az intervallum elemeinek sorszáma az eredeti típusban lévő sorszám. A típus műveletei azok a műveletek, amelyeket az eredeti típuson végezhetünk. A sorszámozott típusok függvényei az intervallumtípus esetén is alkalmazhatók.

3. A felsorolt típus

A felsorolt típus azokat az adattípusokat tartalmazza, amelyeket a felhasználó hoz létre, azok neveinek puszta felsorolásával ([24]). Sorszáma a felsorolásbeli sorszám. A sorszám 0 értékkel kezdődik.



Pl. `(TOK,MAKK,ZOLD,PIROS)`
`(VASARNAP,HETFO,KEDD,SZERDA,CSUTORTOK,PENTEK,SZOMBAT)`
`(PIROS,FEHER,ZOLD)`
`(TAVASZ,NYAR,OSZ,TEL)`

— A felsorolt típusban lévő elemek nevek (azonosítók).
— A felsorolt nevek nem szerepeltethetők beolvasó vagy kiíró utasításokban.
— A típus értékeinek rendezettségét a felsorolás sorrendje adja.
— Az összehasonlító műveleteket (<, >, <=, >=, <>) használhatjuk. Így: TOK<PIROS, NYAR<OSZ
— Az ORD, PRED és SUCC függvények alkalmazhatók.

Körülbelül ennyi az, amit elengedhetetlenül fontos tudnunk a standard egyszerű típusok felépítéséről, illetve használatukról.

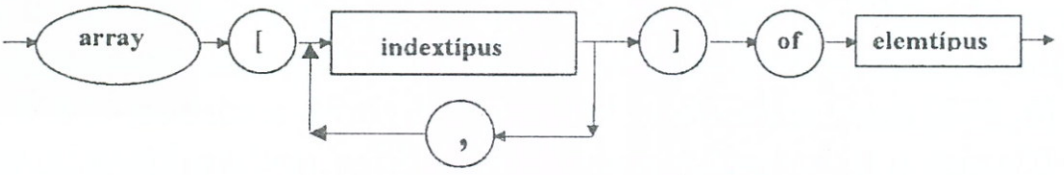
Mint már korábban is írtuk, az összetett (strukturált) adattípusok az azonos vagy különböző típusú adatcsoportokat fogják össze egy szerkezetté. Ebben a fejezetben csak a tömbtípussal — mint strukturált típussal — szeretnék bővebben foglalkozni. A karakterlánctípust (stringtípust) azért most elemezzük, mert a karakterlánc felfogható tömbként is. A rekord-, a fájl- és a halmaztípusokkal az azonos című fejezetekben foglalkozunk.

---

* Lásd a kifejezés témakörnél.

2.2.2. A tömbtípus

A tömbtípus olyan összetett adattípus, mely fix számú, azonos típusú komponensből áll. Az elemek számát és a komponensek típusát a deklarációban adjuk meg. A tömbszintaxis diagramja:



Az indextípus a logint kivételével minden sorszámozott típus lehet. Az indextípus gyakran intervallumtípus. A deklarációnál ekkor vigyázni kell arra, hogy az intervallum első eleme ne legyen nagyobb az utolsó elemnél.

A tömb deklarálásakor az indextípusú kifejezés csak a fordítási időben kiszámolható indextípusú konstans kifejezés lehet.

Az indextípust megadhatjuk típusnévként is, ha a tömb indexei befutják a típus teljes alaphalmazát.

Az indexek számát a tömb dimenziójának nevezzük.

A komponensek a tömb elemei. Az elemtípus — a fájltípus kivételével — tetszőleges típus lehet. Az elemekre indexeikkel hivatkozhatunk. A tömb elemszámát az egyes indextípusok számosságának szorzata adja meg. A tömb helyfoglalása az elemszám és az egyes elemek helyfoglalásának szorzatával egyenlő. A tömb elemei a memóriában sorfolytonosan foglalnak helyet. A tömb elemei maguk is lehetnek tömbök, mely elemei szintén lehetnek tömbök. Így deklarálhatunk két, három, illetve akárhány dimenziójú tömböt. A dimenzió számára a Turbo Pascalban nincs megkötés.

A tömb elemeit a programban úgy használjuk, hogy a tömbnév után szögletes zárójelben megadjuk az aktuális indexek listáját. Az indexek értékei sorszámozott kifejezések. A tömb elemeit típusaiknak megfelelően használhatjuk.

Pl.: A[1]:=0 B[I,J]:='g' c[2*k,FALSE]:='szöveg1' v:=d['z'].

Az értékadás végrehajtásához természetesen az értékadás kompatibilitásnak* teljesülni kell.

Ha A és B két azonos típusú* tömb, akkor minden, az indextípushoz tartozó indexre végrehajtható az elemenkénti értékadás helyett a következő, rövidebb alakot is használhatjuk: A:= B.

---

* A típusok azonosságát és az értékadás kompatibilitását később pontosan megfogalmazzuk.

Tömbtípusú konstans

A tömbtípusú konstanst deklarációja előtt nem használhatjuk. Csak a CONST deklarációs részben adhatjuk meg **típusos konstansként**. A fejezetben szó lesz a témáról.

Példák:

1. array[0..255] of real (Írható így is: array[byte] of real). Ez a tömb: egy 1 indexes, 256 elemű tömb, aminek elemei real típusúak. Indexének értékei a $[0, 255]$ intervallumban lehetnek. A tömb a tárban $256 \cdot 6$ byte helyet foglal el. Elemei a 0. indexű elemmel kezdődően sorban követik egymást. Az 1 indexes tömböt vektornak is nevezzük.

2. ARRAY[1..3,1..2] OF INTEGER Ez egy mátrix, aminek 3 sora és 2 oszlopa van. 2 indexének lehetséges értékeit a két sorszámozott típus mutatja. Minden eleme integer típusú. $3 \cdot 2 \cdot 2$ byte a helyigénye. Elemei a memóriában sorfolytonosan foglalnak helyet.

3. Array['a'..'z',–3..4] of Char Ez a tömb $26 \cdot 8 \cdot 1$ byte tárfoglalású. Elemei karakterek lehetnek.

4. A tömb indexhatárait konstans kifejezésekkel is megadhatjuk, hiszen a konstans kifejezést már fordítási időben kiértékeli a fordító. Ha a DB numerikus konstans, akkor a következő tömbről is beszélhetünk: ARRAY[1..2*DB] OF BOOLEAN. Ennek a tömbnek mind a 2*DB eleme logikai típusú.

2.2.3. A karakterlánctípus

A karakterlánc (string) ASCII karakterek egymáshoz fűzött láncolatát jelenti. A karakterlánc hossza a program futása közben dinamikusan változhat. Maximális hosszát azonban a deklarációban meg kell határozni, a következő feltételekkel: $1 \leq \text{maxhossz} \leq 255$. A maxhossz fordítási időben kiszámítható, előjel nélküli egész kifejezés. Ha nem adjuk meg ezt a kifejezést, akkor a srting maximális hossza 255 értékként tekinthető. Ha maxhossz=1, akkor a string[1] Char típusként is értelmezhető.

A **karakterlánctípusú konstanssal** szövegkonstans néven már foglalkoztunk.

A karakterlánc elemeire úgy hivatkozhatunk, mint a tömb elemeire. A string tehát kezelhető olyan tömbként is, melynek elemei Char típusúak. Pl. szoveg[1] a karakterlánc 1. karakterét, szoveg3] a 3. karakterét jelenti.

A string 0. karaktere tartalmazza a karakterlánc aktuális hosszát, pontosabban azt a karaktert, melynek kódja (sorszáma) a karakterlánc hossza. Ez az elem a string értékének változásával automatikusan változik. Ha például s='abc', akkor s[0] egyenlő a ♡ karakterrel, mivel ennek a karakternek

az ASCII kódja 3. Ha tehát valóban a string hosszát kívánjuk tudni, akkor ennek a karakternek venni kell a belső kódját. Ezt az Ord( ) függvénnyel tehetjük meg. A karakterlánc hosszát a Length( ) függvénnyel is meghatározhatjuk.

A string minimális aktuális hossza 0. Ekkor az üres stringről ('') beszélünk. Pl.: s:='' utasítás után az s[0] értéke 0 lesz.

Műveletek: A stringeknél is értelmezve vannak a különböző műveletek:

+: Az összefűzés az első operandus karaktersorozata után fűzi a második operandus karaktersorozatát. 'macska'+'nadrág'='macskanadrág', ''+'ALMA'='ALMA'. Ha V='A:\ALLOMANY\' és Fiznev='adatok.dat', akkor a V+Fiznev string tartalma: 'A:\ALLOMANY\adatok.dat'.

Összehasonlító műveletek: <, >, <=, >=, <>

Itt kell megjegyeznünk azt, hogy két string akkor és csak akkor egyenlő, ha karakterről karakterre megegyeznek. Tehát, ha két karaktersorozat nem egyenlő hosszúságú, akkor nem is lehetnek egyenlők. Az összehasonlítás karakterről karakterre történik, és addig folyik, amíg nem talál eltérést. Például: 'abcd'>'aacd', mert 'b'>'a'. Az összefűzés prioritása magasabb az összehasonlító műveletek prioritásánál.

A karakterlánc aktuális értékét megváltoztathatjuk értékadással és eljárásokkal, valamint függvényekkel is.

A karakterlánccal kapcsolatos eljárások: Delete, Insert, Str, Val; függvények: Concat, Copy, Length, Pos. A felsorolt eljárásokat később pontosan leírjuk.

A Pascal programnyelvben még további standard típusok is vannak. Sőt a programon belül mi magunk is építhetünk típusokat. Az egyes típusokhoz nevet is rendelhetünk. A kifejezések kiértékelésekor is nagyon fontos szerepet játszanak a típusok... Szóval nagyon sok mondanivalónk van még a típusokkal kapcsolatosan. Az egyes adattípusok elemzését most mégis felfüggesztjük. Ezt azért tesszük, hogy az eddig tanult anyagrészeket minél hamarabb a gyakorlatban is kipróbálhassuk. A típusok fontosságáról a későbbiekben sem feledkezünk meg.

Most pedig válaszoljon néhány kérdésre, oldjon meg néhány feladatot, és azután térjünk át a program felépítésével kapcsolatos anyagra!

2.2.4. Kérdések, feladatok

1. Fejtse ki részletesen a következő témaköröket:

(a) Milyen szintaktikai elemei vannak a Turbo Pascal programnak?

(b) Az egész számkonstansok írása

(c) A valós számkonstansok írása

(d) Szövegkonstansok

(e) Rajzolja le a Turbo Pascal adattípusainak összefoglaló grafikonját. Jellemezze az eddig tanult típusokat.

(f) Mit jelent a fixpontos és a lebegőpontos számábrázolás?

(g) Jellemezze azaritmetikai műveleteket, adja meg a művelet operandusainak és az eredménynek is a típusát.

(h) A logikai típus és műveletei

(i) Mit tud a karaktertípusról és műveleteiről?

(j) Az intervallumtípus jellemzése. Melyek a műveletei, eljárásai és függvényei?

(k) A felsorolt típus és alkalmazási lehetőségei

(l) A karakterlánctípus és művelete, eljárások, függvények

(m) A tömbtípus mint strukturált típus. A tömb indexe, eleme, dimenziója

2. Milyen típusúak a következő konstansok:

`-0, 255, 265, -1, -128, 32767, -32768, 65535`

`-$E2, $FF, -$FF, $ffff, $7FFF, $E1`

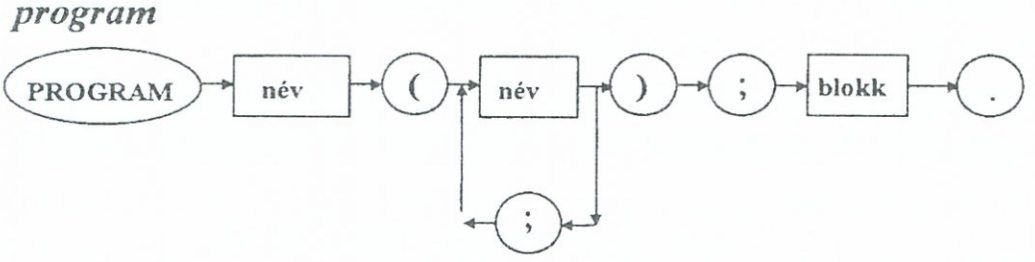
`-E1, 1.E1, 1.E, -1.E-1, -e2, 2.56E2`

`-'1', 'Alma', 'Jó tanuló, jó sportoló', '', '''HURRÁ'''`

3. Mennyi az eredménye a következő műveleteknek: $ABA+$EDE, $EDDA+$ABBA. Az eredményeket adja meg decimálisan és hexadecimálisan is.

2.3. A Pascal program szerkezete

Tekintsük meg először a Pascal program szintaxis diagramját ([12]). Ezen jól nyomon követhetjük egy Pascal program szerkezetét.



Ez a diagram a Turbo Pascal esetén kis mértékben kiegészül azokkal a sajátosságokkal, amelyek a Turbo Pascalra jellemzők.

PROGRAM név;	**Programfej**; a név tetszőleges név lehet, de ügyeljünk arra, hogy ezt a nevet csak a program azonosítására használhatjuk (a 4.0 verziótól). Ez a program logikai neve.
{Fordító direktívák...}	Olyan információk a Turbo Pascalban a fordítóprogram számára, melyek az egész programra érvényesek.
USES $név_1$,$név_2$,...$név_n$;	Egységek felsorolása.

Blokk

	Deklarációs rész
CONST ...;	{konstansok deklarálása}
TYPE ...;	{típusok deklarálása}
VAR ...;	{változók deklarálása}
PROCEDURE ...;	{eljárások deklarálása}
FUNCTION ...;	{függvények deklarálása}
LABEL ...;	{címkék deklarálása}
BEGIN	**Végrehajtandó rész, a program törzse**
utasítások...	
END.	**Pont, a program vége**

A programfej

Különösebb magyarázatot nem igényel. Feladata: a program azonosítása. Ez a program logikai neve. A név leírásával azonosítottuk is a programot, és más programelem azonosítására nem használhatjuk ezt a nevet. Javasoljuk — de nem kötelező —, hogy ez a név a háttértárolón tárolt. PAS kiterjesztésű fizikai névvel legyen megegyező!

A fordítódirektívák

A program fordítására vonatkozó információk. Speciális szintaxissal kell használni. Ott lehet alkalmazni, ahova megjegyzéseket is beszúrhatunk. Ha a program deklarációs része előtt helyezkednek el, akkor globálisak. Hatásuk az egész programra érvényes. Lehetnek kapcsolódirektívák vagy paraméterdirektívák. A direktíva nevét kapcsos zárójelek közé kell tenni. Több direktívát is felsorolhatunk, vesszővel elválasztva. A fordítódirektívákat az Options/Compiler... párbeszéddobozban is beállíthatjuk. A Help segítségével tanulmányozza feladatát. Néhány fordítódirektívával mi is foglalkozunk.

A USES

A USES alapszó után a programban használni kívánt egységeket sorolhatjuk fel. A Unit (egység), nem ebben a programban deklarált konstansok, típusok, változók, eljárások, függvények gyűjteménye a Turbo Pascalban. Külön fejezetben foglalkozunk velük.

2.3.1. A blokk

A Pascal programban minden nevet és címkét **deklarálni kell.** Automatikus deklaráció nincs! Ezeket a deklarációkat a blokk elején kell elhelyezni. A blokknak ezt a részét **deklarációs résznek** is nevezzük.

A deklarációs részben rögzített sorrend nincs, de bizonyos logikai sorrend van. A deklarációban is hivatkozhatunk bizonyos nevekre. Addig azonban nem hivatkozhatunk egy névre sem, amíg azt nem deklaráltuk.* Ez jelenti a logikai sorrendet.

Egy blokk deklarácós részében egy név, vagy címke csak egyszer szerepelhet. **Érvényességi köre** a deklarációtól a deklaráció blokkjának végéig tart. A blokkban deklarált elemek a blokk lokális nevei, illetve címkéi. Ugyanezek az elemek globálisak a program egy belső blokkjában.

Ha a Turbo Pascal program valamelyik eljárásának, függvényének vagy unitjának blokkjában ugyanaz a név ismét deklarált — ami megengedett —, akkor ez a név a belső blokkban deklarált elemet jelenti. A belső blokkban minősített névvel lehet hivatkozni a program deklarált neveire, ha a lokális névtől meg akarjuk különböztetni. **Minősítő névvel** a program és bármely egység (Unit) bármely nevére hivatkozhatunk.

2.3.1.1. Változódeklaráció

Tudjuk, hogy a programok adatokkal dolgoznak. A Turbo Pascal program esetén a programozónak egyértelműen közölni kell, hogy a program futásához mennyi memóriaterületre van szüksége. A **változó** a programban olyan memóriaterületet azonosít, aminek tartalma a változó értéke. Ennek a memóriaterületnek a tartalma a program futása során más és más értéket vehet fel. A változó neve azonban nemcsak a neki megfelelő memóriaterület kezdő címét azonosítja, hanem az ott található adat típusát is. Ezt csak úgy teheti, ha a változó deklarálásakor megadjuk a változó nevét és típusát is.

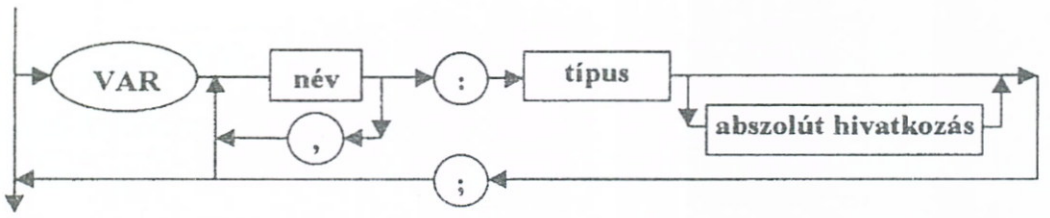
A program által használt összes változót deklarálni kell. A (fő) programban deklarált változók az alprogramok számára globális változók. Azokat az alprogramok is használhatják. A memóriában az adatszegmensben (Data

---

* Ez alól a szigorú szabály alól a mutató típusnál lesz egy kivétel.

Segment) foglalnak helyet. Ezért van az, hogy a program minden blokkjából láthatók, legfeljebb minősített változóként kell hivatkozni. A változó deklarálásának pontos szintaktikája a következő oldalon található diagramon látható.

A változók deklarálása tehát a VAR alapszóval kezdődik, a változók nevének felsorolásával folytatódik. A nevek felsorolását a kettőspont (:), majd a típus zárja.* Ha további változókat is kívánunk deklarálni, akkor nem szükséges — de lehet — a VAR szót megismételni. Az egymástól vesszővel elválasztott változók azonos típusúak.



VAR $név_{1,1}$,$név_{1,2}$,...$név_{1,n}$: $típus_1$;
$név_{2,1}$,$név_{2,2}$,...$név_{2,n}$: $típus_2$;
⋮

```
Pl.: Var      i,j,k:        integer;
              x,y:          real;
              vektor:       array[1..20] of real;
              vanhiba:      boolean;
```

A változók nevének választásakor törekedjünk arra, hogy ezek a nevek kellően beszédesek legyenek, és a matematikában megszokott jelölésektől csak a szükséges mértékben térjenek el. Ez azért jó szokás, mert ekkor a változó neve utal szokásos jelentésére.

2.3.1.2. Típusdeklaráció

A változó típusán azt a halmazt értjük, amely a változó által felvehető értékeket tartalmazza. A Pascal nyelvben minden változóhoz kötelező típust rendelni, és egy változóhoz csak egy típus tartozhat! Minden típushoz meghatározott értékhalmaz és műveletek tartoznak.* A változó deklarációja tehát azt jelenti, hogy egy változó névhez hozzárendeljük lehetséges értékeinek halmazát és a halmaz elemein értelmezett műveleteket.

---

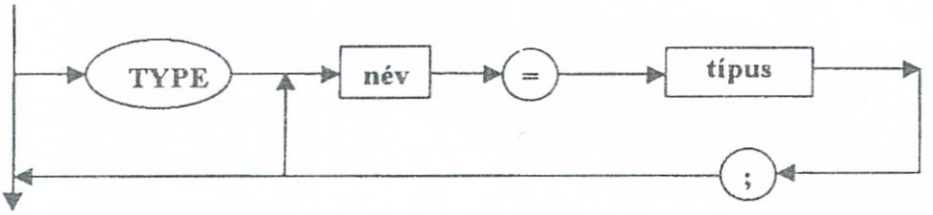
* Az abszolút hivatkozás egy későbbi fejezet anyaga.

* A Pascal típuskoncepciójánál pontosan megfogalmaztuk.

Mint tudjuk, a Turbo Pascal fordítónál különböző beépített adattípusok vannak. Ezekhez a típusokhoz típusneveket is rendeltek. Pl.: real, byte, string...

De a program során mi is deklarálhatunk típusokat. Ezt úgy tehetjük, hogy a típust névvel, azonosítóval látjuk el. A program további deklarációs részében — típusok vagy változók deklarációjakor — már csak erre a típusnévre hivatkozunk. A típus deklarációja nem jelenti a változók deklarációját. A változók deklarálásáról is gondoskodni kell.

A típus deklarálását a következő diagram alapján végezhetjük:



A deklarációban szereplő típus lehet egyszerű típus, stringtípus, strukturált típus, mutató típus, de típusnév is állhat a helyén.

Pl.: TYPE

```
Egesz       =integer;
Nevtip      =String[20];
Tomb1tip    =ARRAY[1..100] of Nevtip;
Jegytip     =1..5;
LOTTOtip    =ARRAY[1..5] of 1..90;
Tomb2tip    =ARRAY[1..100] of LOTTOtip;
```

A típusneveket vesszővel elválasztva felsorolni tilos.

A típusnév abban a blokkban ismert, amelyben deklaráltuk, a deklarációtól kezdődően. Mint látjuk, a típusdeklaráció a programozó kényelmét szolgálja, hiszen a típust csak egyszer kell leírni, később csak nevével kell rá hivatkozni. Ez bonyolultabb típusok esetén jól alkalmazható gyakorlati jelentőséggel bír. A típusdeklarációnak azonban nem csak praktikus jelentősége van. A rendszer több helyen megköveteli, hogy bizonyos változók vagy paraméterek azonos típusúak legyenek. Ekkor nagy segítség a típusdeklaráció.

Típusazonosság

Két típus akkor és csak akkor azonos ([3], [24]), ha típusnevük azonos vagy az egyik típust a másik típus nevével deklaráljuk.

```
Type
        tip1=byte;
        tip2=byte;
        tip3=tip2;
```

```
tip4=tip3;
tip5=array[byte] of char;
tip6=array[byte] of char;
tip7=tip6;
tip8=tip6;
```

Példánkban a tip1, tip2, tip3, tip4 típusok azonosak, de tip5 és tip6 típusok különbözők. A tip6, tip7, tip8 típusok ismét azonos típusok.

Tömbök, rekordok, halmazok értékadó utasításánál, eljárások változó szerint hívott aktuális paramétereinél a típusok azonosságáról soha ne feledkezzünk el.

2.3.1.3. Konstansok deklarációja

Bizonyára felmerült már az olvasóban is az a kérdés, hogy mi a megoldás akkor, ha egy elkészült programot át akarunk alakítani úgy, hogy a benne lévő azonos jellemzőkkel rendelkező ciklusokat többször akarjuk alkalmazni, de egy másik futtatáskor más értékekkel. Természetesen erre az lehet a válasz, hogy a ciklusparaméterek értékeit kijavítjuk. De mi a megoldás akkor, ha sok számot kell kijavítani egy 3000 soros programban? Meglehetősen aprólékos munka lenne ezt egyenként végigcsinálni. Hasonló a probléma, ha a tömb deklarációjában az indexhatárokat szeretnénk egy-egy futtatáshoz módosítani. Ezen problémák megoldását a konstansok deklarálásának bevezetésével segíthetjük. Előnyük pontosan az, hogy csak néhány helyen kell javítani a programban, ha változtatni akarunk az értékeken.

A konstansokról már több ízben is szóltunk, de arról még nem volt szó, hogy a deklarációs részben definiálhatunk is konstansokat. Minden deklarált konstansnak van egy neve és értéke, mely érték a program során nem változik.

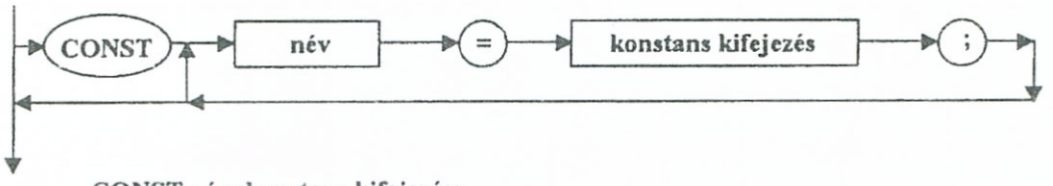
A konstans olyan objektum, mely értékével együtt fordításkor befordítódik a programba, mégpedig a kódszegmensbe, oda, ahol a program kódja is található.

Alapvetően két fajtáját különböztetjük meg a konstansoknak, így deklarációjuknak is. (*Nem tipizált*) *konstans*, amelynek értéke a programon belül nem változtatható meg, illetve az *igazi konstans* (*állandó*). (A konstans deklarációja a következő oldalon látható.)

A konstans neve után egyenlőségjel következik, majd konstans, amit a Turbo Pascal fordítónál konstans kifejezés is képviselhet ([24]). A konstans kifejezés* olyan kifejezés, amit a fordító már fordítási időben kiszámol, és ezután már csak a kiszámított érték képviseli a konstanst. A legegyszerűbb eset természetesen az, amikor egyszerű konstans, mint 255 vagy 'k' áll itt.

---

* A konstans kifejezés fogalmával a kifejezéseknél részletesen foglalkozunk.



CONST név=konstans kifejezés;

A nem tipizált konstans felhasználható további deklarációkban is. A deklarációk sorrendje — mint tudjuk — nem kötött, de ha más deklarációkban hivatkozunk a konstansra, akkor azt a konstans deklarációjának meg kell előznie!

Példánkban jól látható, hogy a sor típusát az 5 határozza meg, ami egy byte típusban is elfér. Az ertek konstans számértékét hexadecimális alakban adtuk meg.

```
CONST     sor       = 5;
          oszl      = 3;
          ertek     = $ABA;
```

Karakterlánctípusú konstans deklarálása a következő lehet:

```
CONST gyakori = 'Ügyes vagy!'
```

Korábban deklarált konstansokat is felhasználhatunk a deklarációban.

```
CONST matrix_max= sor*15;
```

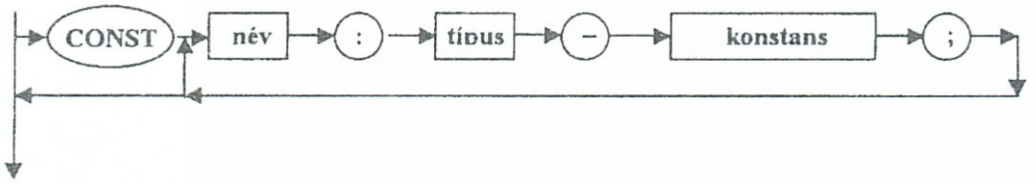
Ezek után már következhetnek azok a változódeklarálások, amelyek már tartalmaznak ilyen konstansokat. Például:

```
VAR matrix:ARRAY[1 .. sor, 1..oszl] OF BYTE;
```

Tipizált konstans

Tulajdonképpen nem is igazi konstans, hanem inicializált, azaz kezdőértékkel rendelkező változó. Úgy viselkedik a programban, mintha rendes változó lenne, de a program fordítása során kezdőértéket kap. Azt mondjuk, hogy inicializáltuk a változót. A tipizált konstansok értéke a program során megváltoztatható, s alapjában véve ez az, ami megkülönbözteti a nem tipizált konstanstól. Fordításkor csak a konstans neve íródik bele a kódszegmensbe. A konstans tartalma az adatszegmens elején helyezkedik el, ahol a változók értékei is vannak.

A tipizált konstans deklarálásában (lásd a következő oldali ábrát) nem a konstans határozza meg a típust. Ennek oka az, hogy a program futása során a deklarált tipizált konstans kezdőértéke megváltozhat. A megadott konstans (kifejezés) csak kezdőértéket ad (inicializál) a névnek, a típus lehetőségein belül. Természeténél fogva a változóhoz áll közelebb a tipizált konstans fogalma, ezért **későbbi deklarációkban nem alkalmazhatók** ezek a konstansok. Hasznosan alkalmazhatjuk viszont a tipizált konstansokat előkészületi munkálatoknál.



CONST név : típus = konstans;

A konstans lehet egyszerű típusú konstans (kifejezés), karakterlánc típusú konstans és strukturált típusú tömb, rekord és halmaz típusú konstans is.* Strukturált konstans csak tipizált konstans lehet!

Egyszerű konstans esetén olyan konstans értéket kell megadni, mely ábrázolható a jelzett típusként. Ez az érték lesz a tipizált konstans kezdő értéke. Egyszerű konstansok esetén különösen nagy a kísértés, hogy a deklarált konstanst felhasználjuk további deklarációkban. **Tipizált konstansok nevére más deklarációkban nem hivatkozhatunk!**

```
const     SZUM      : real      =0;
          FAKT      : integer   =1;
          MAX       : integer   =999;
          KAR       : char      =#13;
          VAN       : Boolean   =False;
```

Karakterlánc típusú konstans esetén a karakterlánc kezdő értékét kell megadni.

```
const     HIBA:     string[10] = 'nincs hiba';
          VANSZ:    string[12] = 'nincs benne';
```

Strukturált típus esetén a tömbtípusú konstansok nagyon praktikusan alkalmazhatók, mert nem szükséges a tömb elemeit külön feltölteni értékekkel a programban, mivel a kezdőértékek megadása igen egyszerűen megtehető már a deklarációban. Az értékek felsorolását zárójelben végezhetjük el.

```
const jelek : array[0..15] of char=
('0','1','2','3','4','5','6','7','8','9', 'A','B','C','D','E','F');
```

Karaktertömb esetén konstansként az elemek számával megegyező hosszúságú karakterlánc is megadható.

```
const jelek : array[0..15] of char=('0123456789ABCDEF');
```

Többdimenziós tömbök esetén a tömb elemeit sorfolytonosan kell megadni.

```
CONST matrix:ARRAY[1..2,1..3] OF CHAR=(('a','b','c'),('d','e','f'));
```

Feladat: Elemezze ki a következő példa deklarációját. Egy forintösszeg címletelésénél vagy szöveges leírásánál alkalmazhatjuk.

---

* Lásd a Kifejezések témakörben is!

```
const   F='forint';
        sor =2;
        oszl =13;
        cimletek:array[1..sor,1..oszl] of string[10]=(('1','2','5',
        '10','20','50','100','200','500','1000','2000','5000',
        '10000'),('Egy','Kettő','Öt','Tíz','Húsz','Ötven',
        'Egyszáz','Kettőszáz','Ötszáz','Egyezer','Kettőezer',
        'Ötezer','Tízezer'));
```

Hasonlóan egyszerű a rekord és a halmaz típusú konstans megadása is. A téma tárgyalásával a rekordoknál, illetve a halmazoknál találkozunk.

Strukturált típusú konstansok megadását csak tipizált konstansként tehetjük, és ezeket a konstansokat további deklarációkban nem használhatjuk!

2.3.1.4. Standard függvények, eljárások

Annak érdekében, hogy a sok megtanult anyagot a programozásban alkalmazhassuk, most egy kis kitérőt teszünk. Az eljárások és függvények alprogramokat jelentenek. Minden eljárást és függvényt deklarálni kell, kivéve a standard eljárásokat, és függvényeket. Az eljárások, függvények deklarálásával hamarosan részletesen foglalkozunk. Az eljárások, függvények meghívása a program végrehajtandó részében (a törzsben) történik. Meghívásuknál meg kell adni az aktuális paramétereket. Miután elvégzik feladataikat, visszakerül a vezérlés oda, ahonnan az alprogramokat meghívtuk. A függvények értékkel térnek vissza. Ez az érték az, amit a függvény kiszámolt. A formális és aktuális paraméterekről is részletesen írunk a következő részben. Most csak annyit, hogy a két paraméterlista számának, típusának, a felsorolás sorrendjének meg kell egyezni.

A Turbo Pascal rendszerben lévő standard függvényeket és eljárásokat a rendszer készítői deklarálták, ezért ez nem a mi feladatunk. Az viszont nagyon fontos, hogy az alprogram hívásakor pontosan tudjuk az aktuális paraméterek lehetséges típusát, és függvényeknél a visszatérési érték típusát. A standard függvények és eljárások ismertetésekor ezeket mi is feltüntetjük. Az alprogramok alkalmazásakor fontos azt is tudni, hogy mely paraméterek értékei változnak meg az alprogram végrehajtásakor. Ezeknek a paramétereknek a helyére **csak változókat írhatunk**, és nem írhatunk konstanst, függvényt vagy kifejezést. Nevezzük ezeket a paramétereket **változó paramétereknek**. A függvények és eljárások ismertetésénél a változó paraméterek elé a VAR alapszót írjuk, ami a változóra utaló szó. Ha nincs jelzés a paraméter előtt, akkor az érték szerint is hívható, vagyis az aktuális paraméter lehet konstans, változó, függvény, sőt kifejezés is.

Aritmetikai standard függvények és eljárások

ABS(x): x típusa: függvény. Bemenő értéke valamennyi aritmetikai típus lehet, az eredmény típusa a bemenő adat típusával megegyező típusú. Értéke a paraméter abszolút értéke.

ARCTAN(x):real: függvény. Bemenő értéke valamennyi aritmetikai típusú kifejezés lehet, az eredményt valós típusban fogjuk kapni, értéke x arcus tangense.

COS(x):real: függvény. A bemenő érték szintén valamennyi aritmetikai típus lehet, amit radiánban kell megadni, a visszaadott paraméter valós, értéke az x cosinusa. A függvény értékkészlete a $[-1, 1]$ intervallum.

EXP(x):real: függvény. A bemenő érték szintén valamennyi aritmetikai típus lehet, a visszaadott paraméter valós, értéke az e x-edik hatványa.

LN(x):real: függvény. Bemenő paraméter bármely aritmetikai típus pozitív értékű eleme lehet. Az eredmény típusa valós, értéke a paraméter természetes alapú logaritmusa.

PI:real: olyan függvény, amelynek nincs bemenő paramétere. Értéke $\approx$ 3,14159265358979323 85. Pontossága a hardver- és szoftverkörnyezettől függ. A 4.0-ás verzió előtt még beépített konstans volt, utána függvény lett.

RANDOMIZE: eljárás, a véletlenszám-generátort inicializálja.
A RANDOM függvény használata előtt érdemes használni.

RANDOM[(x)]: függvény. Paraméter nélküli formában egy véletlen valós számot generál a $[0, 1)$ intervallumban. Ha megadunk utána paramétert (x), amelynek a $[0, 65535)$ intervallumban kell lennie, akkor a $[0, x)$ intervallumban generál egy egész számot.

SIN(x):real: függvény. A bemenő érték tetszőleges számtípus lehet, az eredményt valós típusban kapjuk. Értéke a paraméter sinusa. A paramétert radiánban kell megadni.

SQR(x): x típusa: függvény. A bemenő érték bármilyen aritmetikai típus lehet, a visszaadott érték a bemenő paraméter típusával megegyező típusú. Értéke x négyzete.

SQRT(x):real: függvény. A bemenő érték bármilyen aritmetikai típus lehet, ami nem negatív, a visszaadott paraméter valós. Értéke a paraméter négyzetgyöke.

INT(x):real: függvény, az egészrész-függvény. A matematikában a $[x]$ jelölést használjuk. Azt a legnagyobb egész számot jelenti, ami még nem nagyobb x-től. A bemenő érték bármilyen aritmetikai típus lehet, a visszaadott érték típusa viszont valós lesz. A függvény típusára és az értékre vonatkozó megállapításokra figyeljünk a függvény használata során!

ROUND(x:real):longint: függvény, valós típusú adatot kerekít a kerekítési szabály szerint. A ROUND(x) értéke — a típustól eltekintve — egyenlő

INT(x+0.5) értékével. A bemenő érték bármilyen aritmetikai típus lehet, az eredmény a paraméter egészre kerekített értéke, mely longint típusú.

TRUNC(x:real):longint: valós típusú adat konvertálása egész típusúra, csonkítással. A függvény a tizedes jegyeket levágja, és így kapjuk az egész számot.

FRAC(x):real: függvény az x tört részét jelenti, azaz $x - [x]$-et. A matematikában néha használják az $\{x\}$ jelölést is. A bemenő érték bármilyen aritmetikai típus lehet, a visszaadott paraméter valós.

PRED(x):x típusa: Ennek a függvénynek a bemenete bármilyen sorszámozott típus lehet, és a kimenete is ugyanolyan típusú lesz. Ezért egész típusok esetén is alkalmazható. A függvény értéke a sorszámozott bemenet elődje a sorban. Pl. pred(12)=11; pred(−2)=−3; pred('B')='A'.

SUCC(x):x típusa: Ennek a függvénynek a bemenete bármilyen sorszámozott típus lehet, és a kimenete is ugyanolyan típusú lesz. A függvény értéke a sorszámozott bemenet következője a sorban.
Pl. succ(12)=13; succ(−3)=−2; succ('B')='C'.

DEC(var x [;y:longint]): eljárás. Az sorszámozott típusú x értékét csökkenti y értékével. Ezt az eljárást kétféle módon (szintaxisban) is szoktuk használni. Ha csak x paraméter van, akkor ez a paraméter változó paraméter, és egyaránt lehet kimenő és bemenő is, azaz x-ben fogjuk megkapni az „eredményt". Az eljárás x-et dekrementálja, tehát megadja x-ben a típus előző értékét. Természetesen ha x numerikus sorszámozott típusú, akkor az eljárás alkalmazása megfelel az x:=x−1 értékadásnak és a PRED(x) függvény értékének is. Ha két paramétert használunk, akkor hasonló lesz az eljárás eredménye, csak most a művelet az x:=x−y értékadással lesz ekvivalens. A DEC eljárással ellentétes funkciót lát el az INC eljárás.

INC(var x [;y:longint]): eljárás a DEC ellenkezőjét csinálja. Az ott elmondottak igazak az INC eljárásra is. Ha y nincs, akkor az eljárás hatására az x értéke a sorszámozott típusnak megfelelő következő érték lesz. Numerikus sorszámozott típus esetén x:=x+1 utasítást jelenti. Ugyanezt az értéket adja a SUCC(x) függvény is. Kétparaméteres formában az eljárás az x:=x+y értékadást jelenti.

Néhány függvény és eljárás sorszámozott típusra

CHR(x:byte):char: Ez a függvény egy byte típusú bemenő adatot char típusúvá konvertál. A függvény megadja a belső kódhoz tartozó karaktert.

ORD(x):longint: Ez a függvény a bemenő, sorszámozott kifejezésnek adja meg a sorszámát.

ODD(x:longint):boolean: A függvény értéke igaz lesz, ha a bemenet páratlan.

HI(x):byte: Ez a függvény megadja a bemenő, integer vagy word típusú

kifejezés felső bájtjának az értékét. A két bájton tárolt egészek ábrázolásánál a felső bájt a nagyobb helyi értékű biteket tartalmazza, de a memória nagyobb címén szerepel, mint az alsó bájt.

LO(x):byte: A függvény megadja a bemenő, integer vagy word típusú kifejezés alsó bájtjának az értékét: x=256*HI(x)+LO(x).

SWAP(x:):word: A bemenet bármilyen egész típusú lehet, de a függvény wordre fogja csonkítani. A végeredményül a bemenet felső és első byte-ját megcseréli.

A **DEC(var x[;n:longint])** és a **INC(var ;x[,n:longint])** eljárásokat, valamint a **PRED(x):x típusa**, **SUCC(x):x típusa**, **ROUND(x:real):longint**, **TRUNC(x:real):longint** függvényeket az aritmetikai eljárások és függvények között szerepeltettük, ezért leírásukat nem ismételjük meg.

2.3.1.5. Kifejezések kiértékelése

Kifejezést akkor kapunk, ha operandusokat operátorokkal (műveletekkel) kötünk össze. Operandus lehet konstans, változó, függvény vagy kifejezés. Pl.: 2*r*PI, (a+c)*m/2, NOT(A AND B).

Az első két kifejezés **aritmetikai kifejezés**, mert operandusaik számtípusúak. A harmadik kifejezés lehet aritmetikai is, de lehet logikai is. A kérdést az A és B változók típusa dönti el. A Pascal programban a kifejezésnek **nemcsak értéke, hanem típusa is van.** A kifejezés típusa az **operandusok típusától** és az operandusokkal végzett **operátoroktól** (műveletektől) függ.

Az operandusok típusával kapcsolatosan:

A **konstansról** a Turbo Pascal fordító el tudja dönteni, hogy milyen típusúak. Az egész és a valós konstansok típusaival korábban részletesen foglalkoztunk. Az egyszerű konstansok típusa — leírásukból — egyszerűen megállapítható. Strukturált típusú konstansokat pedig kezdőértékkel rendelkező típusként kell kezelni, és akkor ezeket a konstansokat deklarálni kell. (Lásd: tipizált konstans!)

A Pascal programban minden **változót** deklarálni kell. Ekkor adjuk meg a változó típusát. A **függvények** egyértelműen adott típusú értékkel térnek vissza. A standard függvényeknél a visszatérési érték kiszámítási módját és típusát is a Turbo Pascal tartalmazza. Ezekkel a függvényekkel már ebben a fejezetben foglalkoztunk. A saját készítésű függvények visszatérési értékének kiszámítási módját és típusát is a függvény deklarációjakor határozzuk meg. Ezzel a témával is nemsokára foglalkozunk.

A kifejezések típusának megállapítása érdekében az **operátorok** lehetséges operandusá(i)nak és a művelet eredményének **típusát** a műveletek ismertetésekor következetesen feltüntettük, és az egész típusú operandusoknál táblázatba is foglaltuk.

Annak érdekében, hogy a kifejezés értékét és típusát meghatározhassuk, ismerni kell a kiértékelés és az elvégzendő műveletek végrehajtási sorrendjét is.

(1) Először a *függvények* értékét számítja ki a fordító, és határozott típusú értékkel tér vissza. Ez az érték operandusnak számít.

(2) Ha *zárójelezést* is alkalmaztunk, akkor előbb a zárójelben lévő kifejezés értéke számítódik ki. Ez is operandusnak fogható fel. A fordító a redundáns zárójelezést is megengedi, de ennek alkalmazására ne szokjunk rá, mert lassítja a program futását. Tanuljuk meg inkább a műveletek prioritását!

(3) *A műveletek prioritása a következő:*

(a) Először az egy operandusú műveleteket hajtja végre a program,

(b) másodikként a multiplikatív operátorok,

(c) ezután az additív operátorok,

(d) és legvégül az összehasonlító műveletek (<, >, =, <=, >=, <>, IN) következnek.

(4) Ha egyenrangú operátorok is vannak a műveletek sorában, akkor a *balról jobbra szabály* érvényesül.

Gyakori hiba a feltételek írásakor, hogy ezeket a szabályokat nem gondoljuk át következetesen. Pl. az IF (I>2) AND (I<3) THEN A:=0 utasításhoz hasonlóaknál gyakran elfelejtjük a relációknál a zárójeleket kitenni, és akkor jogosan kapunk hibaüzenetet. Zárójelezés nélkül ugyanis a feltételnek szánt kifejezés kiértékelése az AND aritmetikai művelettel kezdődne. Ennek eredménye egy szám. Ezek után következne a reláció kiértékelése. Ekkor viszont olyan relációjelekkel felírt kifejezést kaptunk, ahol a relációkban nem csak két oldal szerepel. Ez pedig nem megengedett. *Type mismatch* (típusösszeférhetetlenség) hibaüzenetet kapunk.

Mint látjuk, a kifejezések kiértékelésében jelentős szerepet játszanak az operandusok típusai. Ezért fontos, hogy az operandusok típusát jól válasszuk meg. Ez rajtunk múlik. Átgondolatlan típusválasztások esetén kellemetlen meglepetésekben lehet részünk. Példaként elemezzük a következő kis programot. A programban az x változót integer típusúnak deklaráltuk. A read(x) utasítással beolvastuk az x értékét, majd a writeln('x*x=',x*x:10) utasítással kiírattuk az 'x*x=' karakterlánc konstanst és az x*x értékét 10 pozíción.

Láthatjuk, hogy több esetben a kiszámolt érték nem fért el az integer típusban, ezért csonkulás történt. Figyelje meg, hogy melyik adatoknál hibás az eredmény!

Milyen típusúnak deklarálná az x változót, hogy a hiba ne jelentkezzen?

A program tervezésekor gondoljon a változók értékhatáraira és a részeredmények típusaira is! Mindig a probléma jellegének megfelelően válassza

a típusokat. A feleslegesen „bő" típusok és a valós típusok lassítják a program futását. Ha viszont nem számíthatunk az adatok korlátjaira, akkor ne takarékoskodjunk az adatok típusával!

```
File  Edit  Search  Run  Compile  Debug
                                        VIGYAZAT.PA:
program vigyazat;
var x:integer;
begin
write('x=');read(x);
Writeln('x*x=',x*x:10);
end.
[■]                                          Output =
x=128
x*x=      16384
x=10
x*x=        100
x=256
x*x=          0
x=181
x*x=      32761
x=182
x*x=     -32412
x=-128
x*x=      16384
x=-256
x*x=          0
F1 Help  ↑↓→← Scroll  F10 Menu
```

```
File  Edit  Search  Run  Compile  Debug  Options  Window  Help
                    \DOKUME~1\KESZ\ERTKOMP.PAS                    5
program ertkomp;
var a:integer;
    b:shortint;
begin read(a);
      b:=a+1;
      writeln(a:4,b:5);
end.
[■]                          Output                          4-[↑]
126
 126  127
127
Runtime error 201 at 0828:0048.
2034
Runtime error 201 at 0828:0048.
\
126
 126  127
127
 127 -128
2034
2034  -13
                                  Runtime errors
                                  [X] Range checking
                                  Runtime errors
                                  [ ] Range checking
↑↓→← Move  Shift-↑↓→← Resize  ↵ Done  Esc Cancel
```

A kifejezés értékét gyakran egy változóba tesszük be az **értékadó utasítás** (A:=K) segítségével. Az utasítás szintaktikáját és a megfelelő végrehajtás pontos feltételeit a következő fejezetben részletesen elemezzük. Most csak — felszínesen — azt jegyezzük meg, hogy a bal oldalon álló változóba csak akkor tehetjük a jobb oldali kifejezés értékét, ha a kifejezés értéke elfér a változóban. Ha a változó és a kifejezés nem „értékadás-kompatibilis", akkor szintaktikai vagy futási hiba keletkezhet. Előadódhat az is, hogy nincs hibajelzés, „csak" éppen a kapott eredmény hibás. Ez a legkellemetlenebb hiba, ami létezik.

Konstans kifejezés

Konstans kifejezésnek nevezzük azt a kifejezést, amit már a fordító ki tud értékelni a fordítás során. Fordítási időben kiszámítható kifejezésnek is szoktuk nevezni ezeket a kifejezéseket. Ebből a meghatározásból következik, hogy a konstans kifejezésben az operandus változó nem lehet. Néhány függvény, konstans kifejezésben való meghívása viszont megengedett ([24]) Abs, Chr, Hi, Length, Lo, Odd, Ord, Pred, Ptr, Round, SizeOf, Succ, Swap, Trunc.

Ha a programban valahol konstans szerepel, akkor ott konstans kifejezés is állhat. Gyakran alkalmazzuk a konstans kifejezéseket a kifejezésekben és a deklarációkban is. Az intervallumtípusnál és a tömb deklarálásánál mi is hivatkoztunk a konstans kifejezésekre. Alkalmazásuk célszerű, mert a kiszámított konstans kifejezéseket a fordító csak a fordítás idején, egyszer értékeli ki, és a kódszegmensben már a kiszámított értéket tárolja. Ezzel csökkenhet a program futási ideje.

2.3.1.6. Adatok beolvasása és kiírása

Mint megígértük, jöjjön akkor a dolgok kissé talán gyakorlatiasabb része. Mondjuk ezt mindazért így, mert a soron következő fejezet is jócskán fog elméletet tartalmazni, de az első harmadában valóban megpróbáljuk összeépíteni eddigi tudásunk alapján az első programunkat. Ehhez azonban még néhány egyszerű fogalomra is szükségünk van.

Nevezetesen arról van szó, hogy meg kellene vizsgálnunk: hogyan is lehet beolvasni adatokat a billentyűzetről, illetve hogyan lehet a képernyőre kiíratni. Mindkét tevékenységet **eljáráshívó utasításokkal** tehetjük meg programjainkban. Ez azt jelenti, hogy mindkét eljárást megírta a Borland cég, és most nekünk csak a kész eljárásokat kell meghívni az eljárások nevével és aktuális paramétereinek megadásával. Az **eljáráshívó utasítások** (rövidebben eljárásutasítások) utasítások közé való besorolásával a következő fejezetben részletesen foglalkozunk. Most csak az adatok bevitelére és kiírására szorítkozunk.

Adatbevitel

Kezdjük mindjárt a beolvasással, hiszen egy programban is ez a szokott sorrend. A beolvasó eljárások nevei a READ és a READLN. Mivel ezek ún. eljárások (ezekkel később részletesen foglalkozunk), ezért a név után meg kell adni az eljárás aktuális paramétereit. A paraméterek tulajdonképpen a már deklarált változók lehetnek. Azaz a két szintaktika a következő lesz:

```
READ(V1[,V2, ... Vn]);
READLN(V1[,V2, ... Vn]);
```

A V_1 V_2, ... V_n változónevek sorozata, de egyetlen változó is állhat itt paraméterként. A Read és a ReadLn eljárásokkal adatokat (karaktersorozatokat) olvashatunk be TEXT típusú fájlból.* Az adatbevitel a billentyűzetről történik a felsorolt változók memóriaterületére. Amikor a program vezérlése ezekhez az utasításokhoz ér, akkor a billentyűzeten leütött karakterek megjelennek a képernyőn. A beütött karaktersorozat átmenetileg a billentyűzet pufferében tárolódik, és az Enter billentyű leütéséig szerkeszthető a BackSpace és az Esc billentyűkkel. Az Enter leütése után, a szabályok szerint, a változók típusainak megfelelően egymás után betöltődnek a pufferből az adatok. Ha numerikus változókba olvasunk be, akkor a betöltéskor konverzió is történik, vagyis a karaktersorozatból a változó típusának megfelelő szám képződik. Ekkor fontos, hogy csak olyan karakterek kerüljenek sorra, amelyek a számnak is megfelelő karakterei lehetnek, mert ellenkező esetben hibajelzésre leáll a program. A szóköz (Space), valamint a Tab és az Enter karakterek számelválasztó karaktereknek minősülnek.

A két eljárás között különbség, hogy a ReadLn eljárás a feladat végrehajtása után a pufferben maradt szemetet kitakarítja, és így a következő beolvasó eljárások tiszta lappal indulnak. A paraméter nélküli ReadLn eljárás vár egy bevitelt, majd üríti a puffert ([1]). A Read eljárás működése különösen több változó megadása esetén nehezen követhető, ezért alkalmazását nem javasoljuk. **Javasoljuk, hogy csak a ReadLn eljárást használja, és azt is csak egy paraméterrel! Sok bonyodalomtól és fejtöréstől kímélheti meg magát.**

Adatkiírás

Ha már tudunk beolvasni, akkor tanuljunk meg kiíratni is a képernyőre. A kiíratásnak is — akárcsak a beolvasásnak — két eljárása van. Ezek az eljárások is TEXT állományokra — jelen esetben képernyőre — vonatkoznak.

```
WRITE(K1 [,K2, ... KN]);
WRITELN(K1 [,K2, ... KN]);
```

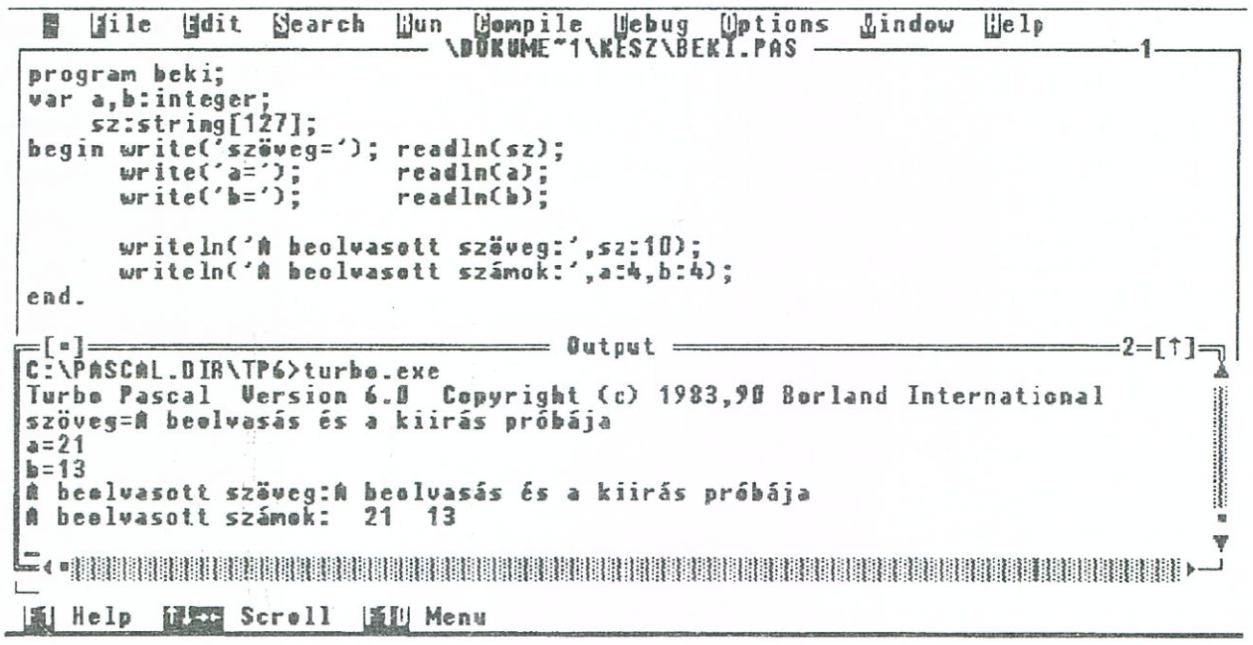
A K_1, K_2, ... K_N sorozat kifejezések sorozata, így ezeken a helyeken konstansok, változók, függvények és kifejezések is állhatnak. Az érdekesség az, hogy minden kifejezés után megadhatunk mezőszélességet is a következő formában: :M1[:M2]. A mezőszélességet a listaelem típusától függően kell megadni. Ez egy kicsit bonyolultnak látszik, de mindjárt megértjük. M1 jelöli a kiírandó érték teljes hosszát, M2 pedig azt, hogy a tizedespont után hány karaktert írjon ki a program. Értelemszerűen: ha a típus nem valós, akkor M2 értékét nem adható. Az is érthető, hogy a mezőszélességet nem

---

* Mindkét eljárásról a TEXT fájloknál tanulunk részletesebben.

kell megadnunk, ez csak a szépérzéket szolgálja. Karakterláncra és mutató típusra is alkalmazható a mezőszélesség megadása. Ha az érték „hossza" nagyobb, mint a mezőszélesség, akkor sincs baj: a fordító felrúgja az utasításunkat, és hagyományosan írja ki azt. A két eljárás között az a különbség, hogy a WRITE nem emel sort az értékek kiírása után, a WRITELN viszont igen. A paraméter nélkül írt WRITELN utasítás egy soremelést jelent.

Írjon az ábrán látható egyszerű programokat, és elemezze az eljárások működését.



2.3.1.7. Kidolgozott feladat

Nézzünk akkor egy példát, amivel kipróbálhatjuk eddigi tudásunkat. A feladat a következő: a program kérjen be 4 számot (először egészet, majd valósat), és adja össze, a matematikában szokásosan tanított módon, azaz a képernyőn egymás alatt, szépen, rendben jelenjen meg a négy bekért szám, legyen aláhúzás, és az alatt jelenjen meg az összeg úgy (ott), ahogy kell.

```
PROGRAM OSSZEAD_EGESZET;
USES CRT;
VAR elso, masodik, harmadik, negyedik, osszeg: LONGINT;
BEGIN
    CLRSCR;
    WRITE(' Kérem az első számot: '); READLN(elso);
    WRITE(' Kérem a második számot: '); READLN(masodik);
    WRITE(' Kérem a harmadik számot: '); READLN(harmadik);
    WRITE(' Kérem a negyedik számot: '); READLN(negyedik);
    CLRSCR;
```

```
    WRITELN(elso:14);
    WRITELN(masodik:14);
    WRITELN(harmadik:14);
    WRITELN(negyedik:14);
    WRITELN('+___________');
    osszeg:=elso+masodik+harmadik+negyedik;
    WRITELN(osszeg:14);
    READLN;
END.
```

Eddigi ismereteink alapján így néz ki a programunk. Aki már programozott Pascal nyelven, az tudja, hogy sokkal rövidebben is meg lehet ezt a programot írni a ciklusok segítségével, de ezt majd csak a fejezet végén tehetjük meg. Most elemezzük programunkat sorról sorra, hogy mit miért csináltunk?

A programfejjel kezdtünk, a program azonosítója: OSSZEAD_EGESZET. Az ez után következő sorról még nem beszéltünk. Mint azt az előző fejezetben is mondtuk, lehetőségünk van ún. unitok felhasználására a programban. A CRT unit a karakteres képernyőkezelést teszi lehetővé, illetve könnyebbé. Vele a későbbiekben még részletesen fogunk foglalkozni. Használatát a CLRSCR képernyőtörlő eljárás kiadása tette szükségessé. Most elégedjünk meg ennyivel. Ezután a változódeklaráció következett. Négy számot kellett összeadnunk, ezért négy különböző azonosítójú változót vettünk fel. (A későbbiekben is használjunk „beszédes" változóneveket.) A típusukat a lehető legnagyobbra vettük, hogy minél nagyobb számokat tudjunk összeadni. Ezen kívül deklaráltunk még egy osszeg nevű változót is, aminek rendeltetése nyilvánvaló.

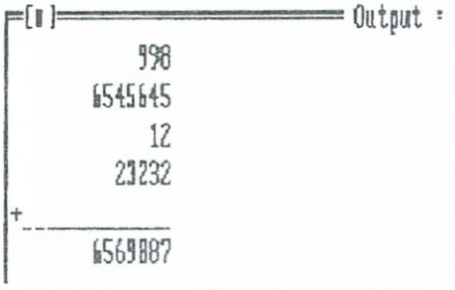
Eddig tartott a deklarációs rész, innentől következik az ún. programblokk, vagy végrehajtási rész. Először letöröltük a képernyőt (CLRSCR), majd kiírattuk a következő sort a képernyőre: Kérem az első számot:, és nem tettünk soremelést, hogy a számot a felhasználó a logikailag hozzá tartozó sorba írhassa. Ezután bekértük az első számot. A változókat nem nulláztuk ki a felhasználás előtt, de erre a beolvasás miatt nincs szükség feltétlenül. Hasonló módon jártunk el a többi érték bekérésekor is.

Újbóli képernyőtörlés után kiírattuk „szépen" a számokat egymás alá: a képernyő 14. oszlopához mérten jobbra igazítva rendeztük a kiírást. Mivel a longint típusú változó legfeljebb 12 karakter hosszú lehet, 14-re állítva az igazítást biztosan elkerültük a gondokat. Ezután aláhúztuk az összeadandókat, majd az osszeg nevű változónak értéket adtunk:

```
osszeg:=elso+masodik+harmadik+negyedik;
```

Ezt úgy olvassuk, hogy: osszeg legyen egyenlő első, ... összegével. A

legyen egyenlő (:=) az értékadó utasítást jelenti. Nemsokára pontosan megfogalmazzuk. Végül kiírattuk a számok összegét „oda, ahova kell", s hogy ne szaladjon el a képernyő, várunk egy ENTER-re (READLN;).



Ilyen az Output képernyő egy része futtatáskor.

2.3.1.8. Feladatok

1. Írja meg a programot úgy, hogy valós számokat is tudjon összeadni, s két tizedes jegyre írja ki őket.

2. Írjunk a háromszög kerületét számoló programot, feltételezve, hogy a felhasználó „jó" adatokat ad meg.

3. Kérjünk be 5 valós számot, írassuk ki a képernyőre úgy, hogy 2 tizedes jegy jelenjen meg a számokból, és a tizedespont a sor közepére kerüljön.

4. Számítsa ki és írja ki a program a háromszög területét, ha adott a háromszög három oldala.

5. Számítsa ki a háromszög területét, ha adott a háromszög két oldala és a közbezárt szög.

6. Számítsa ki a program egy téglalap kerületét és területét.

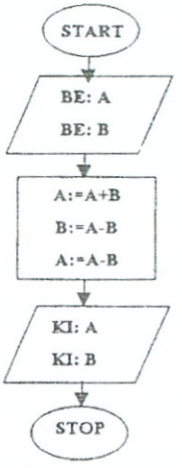
7. Számítsa ki egy rombusz területét, ha adott két párhuzamos oldala és magassága.

8. Írja ki a program a kör kerületét és területét, bemenő sugár esetén.

9. Olvasson be két számot az A és B változókba. A következő értékadó utasításokkal számítsa ki és írja ki a NAGYOBB és a KISEBB változók értékét.

```
NAGYOBB:=(A+B)/2+ABS(A-B)/2
KISEBB:=(A+B)/2-ABS(A-B)/2
```

Mit csinál ez a program? Érdemes megjegyezni a két kifejezést. Tegye programja futásának képernyőjét áttekinthetővé és beszédessé.



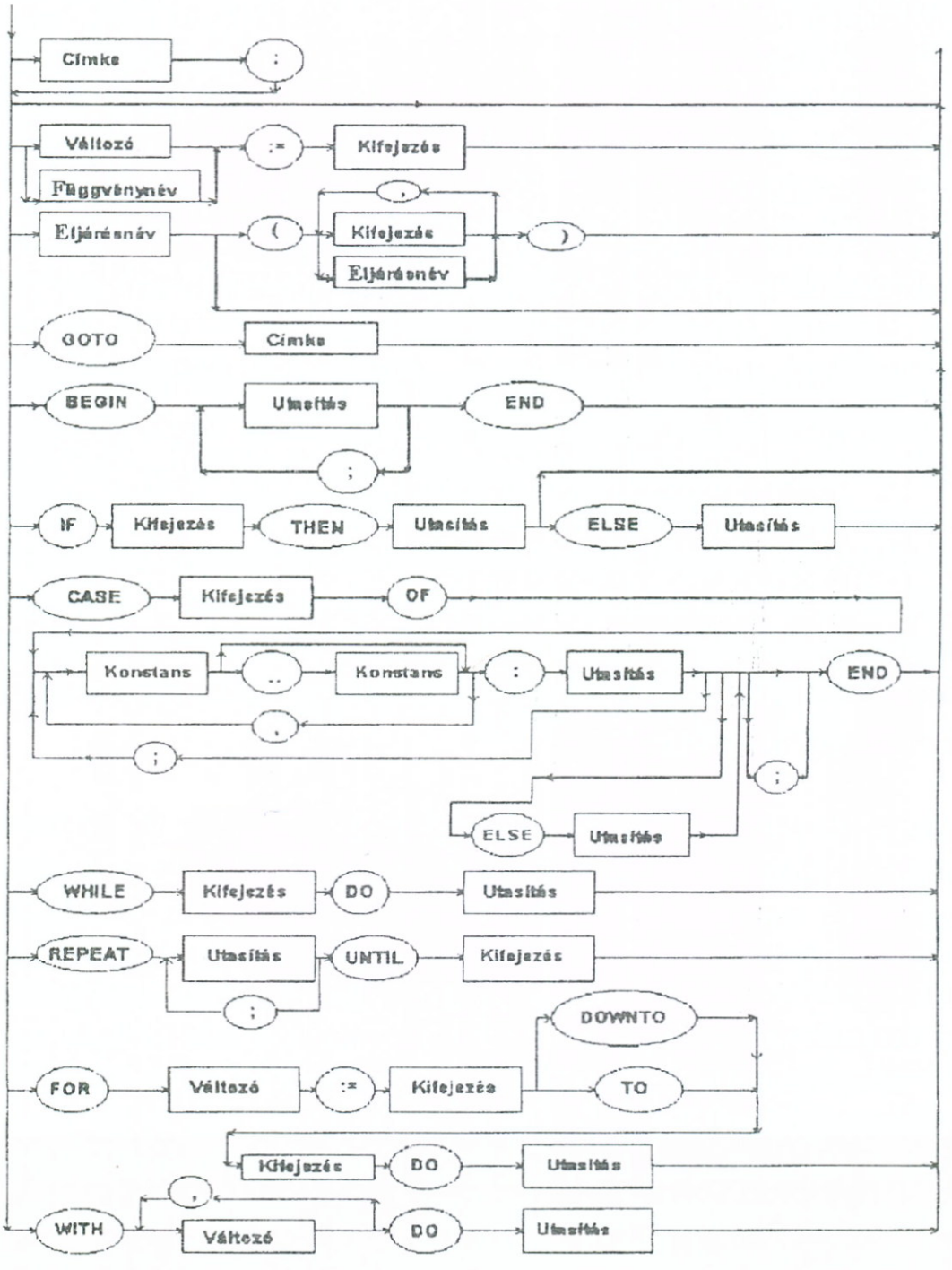
10. Mit csinál a lap szélén látható folyamatábra szerinti algoritmus? Írja meg a programot.

2.3.1.9. A program törzse

A programnak ez a része tartalmazza mindazokat az utasításokat, amelyeket a programnak végre kell hajtania. Ezért szokták végrehajtandó résznek is nevezni.

A következőkben térjünk át az utasítások elemzésére. Az utasítás: a program, az eljárás, a függvény végrehajtási elemét jelenti. Minden utasítás egy tevékenységet végez.

Az Utasítások szintaxisdiagramja:



Az utasításokat helyjelölő, azaz címke előzheti meg. A Turbo Pascal programban címke név vagy az [1, 9999] intervallumba eső előjel nélküli egész szám lehet. A címkét kettőspont (:) követi. Az utasítások között elválasztójelként a ; karaktert használjuk. A ; tehát nem tartozik az utasításhoz, hanem csak az őt követő utasítástól való elválasztásra szolgál.

2.3.1.10. Utasítások

Alapvetően két csoportra oszthatjuk az utasításokat a Pascal nyelvben.

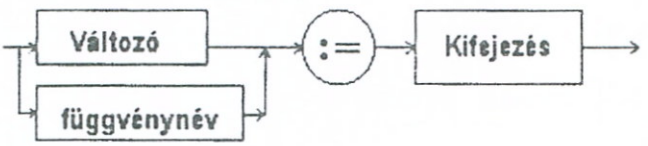
Utasítások

I. Egyszerű utasítások	II. Strukturált utasítások
1. Üres utasítás	1.Összetett utasítás
2. Értékadó utasítás	2.Feltételes utasítások
3. Eljárásutasítás	3.Ciklusutasítások
4. Ugró utasítás	4.With utasítás

I. Egyszerű utasítások: Ezek tulajdonképpen olyan utasításokat jelentenek, melyeknek egyetlen valódi része sincs, amely utasítást jelentene. Nem lehet még elemibb utasításokra bontani őket.

1. Üres utasítás: Mondhatni, hogy ez a legfontosabb utasítás. Az üres utasítás nem tartalmaz semmilyen szimbólumot sem. Funkciója az, hogy nem csinál semmit, hanem jelez valamit. Jelzi azt, hogy ott egy utasítás áll. Hibásan mondják azt, hogy az üres utasítást a ; jelzi. A ; csak elválasztójelet jelent az utasítások között. Így az üres utasítást is ; választja el egy őt követő utasítástól. Sok helyen alkalmazzuk az üres utasítást. Leggyakrabban feltételes utasítás valamelyik ágában és a Case utasításnál használjuk. Ezekre az esetekre a megfelelő utasítások elemzésekor kellő mélységig kitérünk.

2. Értékadó utasítás:



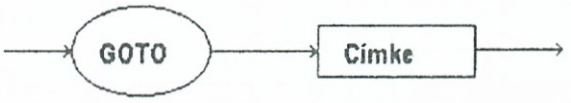
Szintaktikája: V:=K, ahol V változót, a K pedig kifejezést jelent. Nagyon vigyázzunk arra, hogy az értékadó utasítás két oldala nem cserélhető fel! Talán ez az utasításfajta az, amit legtöbbet használunk A kifejezés jobb oldalon álló kifejezés értéke számítódik ki előbb, majd ez után ez az érték töltődik be a változóba. A változó helyén állhat — a fájltípus kivételével — tetszőleges típusú egyszerű, strukturált vagy mutató típusú teljes változónév. Strukturált típusú változók esetében lehet a változó helyén a strukturált változó egy alkotórésze is. Ez tömböknél a tömb elemét, rekordoknál a rekord mezőjét jelentheti.

Az értékadás — a fájlokat kivéve — tetszőleges típusú változókra vonatkozhat, de a változó és a kifejezés értékadás szempontjából kompatibilis kell, hogy legyen ([14]). Az értékadás-kompatibilitás teljesülésének fontosabb esetei a következők:

— A változó és a kifejezés azonos típusú, de egyik sem állománytípus.
— A változó valós, a kifejezés egész típusú.
— A változó és a kifejezés is ugyanahhoz a sorszámozott típushoz, illetve annak egy-egy résztartományához tartozik, és a kifejezés értéke a változó típusa által meghatározott intervallumban van.
— A változó karakterlánc, a kifejezés karakter vagy karakterlánctípus.
— A változó karakterlánc, a kifejezés egy dimenziós karaktertömb.
— A változó és a kifejezés ugyanahhoz a halmaztípushoz vagy két olyan halmaztípushoz tartozik, amelynek alaptípusa vagy ugyanaz a megszámlálható típus, vagy annak egy-egy résztartománya. A kifejezés értékének benne kell lennie a változó típusa értékkészletében.

3. Eljárásutasítás: Olyan egyszerű utasítást jelent, amikor egy korábban deklarált eljárást aktuális paramétereinek megadásával leírunk, azaz meghívjuk az eljárást annak érdekében, hogy az elvégezze tevékenységét. Az előző részben említett adatbeolvasó és kiíró eljárások alkalmazása is az eljárásutasítás esete. Pl.: `ReadLn(A)` vagy `Write('A=',A)`. Az eljáráshívásokkal az eljárások témakörnél még részletesen foglalkozunk.

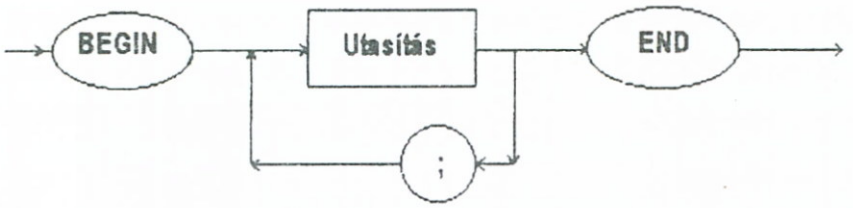
4. Ugró (GoTo) utasítás: Szintaktaxisának diagramját a a következő ábrán láthatjuk:



Ez az utasítás azt csinálja, hogy egy — már előre deklarált és elhelyezett — címkére adja a vezérlést, vagyis a végrehajtás a megadott címkére ugrik, és ott folytatódik a program. Itt említjük meg, hogy az ugró utasítások felesleges alkalmazásával rontjuk a program strukturális felépítését. Csak kivételes esetben alkalmazzuk ezt az utasítást!

II. Strukturált utasítások: Olyan utasításokról van szó, amelyek más utasításokból épülnek fel. Ezen részutasítások végrehajtási módja szerint osztjuk csoportokba az összetett utasításokat.

1. Összetett utasítás: Ilyen utasítást akkor kapunk, amikor két vagy több utasítást együtt kívánunk kezelni. Ekkor a részutasítások ; jellel elválasztott sorozatát a Begin és az End foglalt szavak közé zárjuk. Az utasítászárójelek közé zárt utasítások a leírt sorrendben hajtódnak végre.

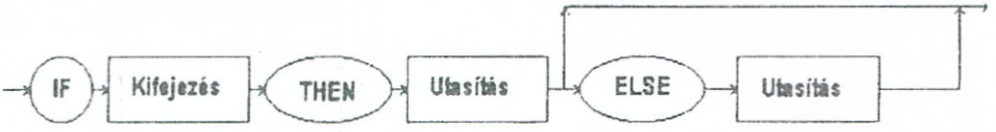


BEGIN $u_1;u_2;\ldots;u_n$ END

A program törzse is egy összetett utasítás.

2. Feltételes utasítások: Ezeknél az utasításoknál a végrehajtandó utasítás valamilyen feltételnek a teljesítésétől, illetve nem teljesülésétől függ.

Az IF utasítás diagramja pontosan leírja az utasítás szintaktikáját.

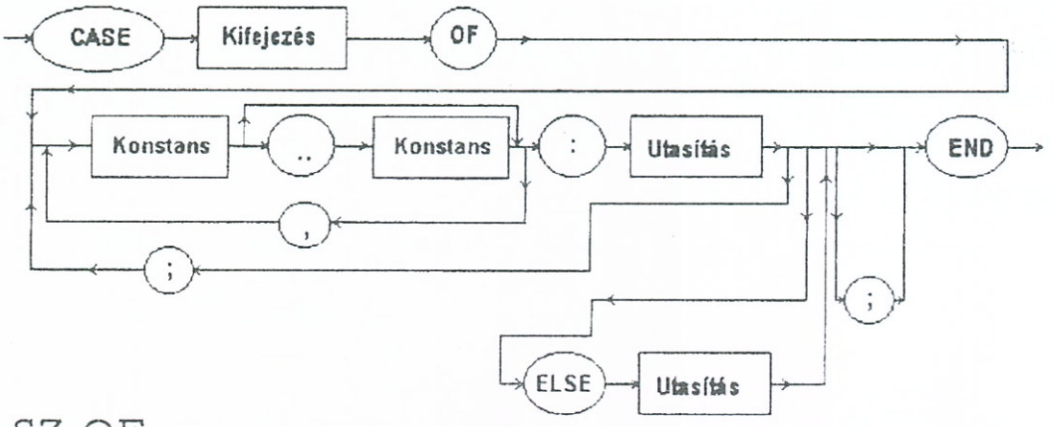


A Kifejezés logikai kifejezést jelent, és az utasítás további elemzésénél L betűvel jelöljük. Az Utasítás tetszőleges utasítást jelenthet, és a továbbiakban U betűt írunk helyette. A feltételes utasítás szintaktikája: IF L THEN U_1 [ELSE U_2] azt jelenti, hogy alapvetően kétféle módon fogalmazhatjuk meg a feltételes utasításokat. Ezt az opcionális jelölés is jelzi.

Az IF L THEN U megfogalmazás esetén: az U utasítás hajtódik végre, ha az L logikai kifejezés értéke TRUE. Nem hajtja végre az U utasítást a fordító, ha a logikai kifejezés értéke FALSE. Ekkor a feltételes utasítás nem hajt végre semmit. Az U utasítás tetszőleges egyszerű vagy strukturált utasítás lehet.

Az IF L THEN U_1 ELSE U_2 alakú feltételes utasítások esetén is az L logikai kifejezés értékétől függ az, hogy melyik utasítás végrehajtására kerül sor. Ha az L logikai kifejezés értéke igaz, akkor az U_1 utasítást hajtja végre, különben nem az U_1 utasítást hanem az U_2 utasítást. Jegyezzük meg, hogy az U_1 utasítás az üres utasítás kivételével tetszőleges utasítás lehet. Ezért van az, hogy az ELSE alapszó előtt soha nem állhat pontosvessző. Az U_2 utasításra nézve semmilyen megszorítás nincs. Így igen bonyolult utasításszerkezeteket hozhatunk létre.

A CASE utasítás valójában többirányú elágazást tesz lehetővé azáltal, hogy kiválasztott utasítása egy sorszámozott típusú kifejezés felvett értékétől függ. Szintaktikája a következő:



CASE SZ OF

c_1 : u_1;

c_2 : u_2;

⋮

c_n : u_n;

[ELSE u_{n+1}]

END;

Az SZ kifejezést szelektornak nevezzük. Ez a szelektor a Word határon belüli tetszőleges sorszámozott típusú lehet. A Case utasítás azt az utasítást választja ki, amelyik az SZ kifejezéssel egyenlő c_i érték után található. A c_i érték megadásakor több konstanst is felsorolhatunk — egyszerre vesszővel elválasztva őket —, és akár intervallumot is megadhatunk. Tulajdonképpen SZ értékeitől függően szelektál, mégpedig úgy, hogy ha az SZ értéke c_i értékével egyenlő, akkor az u_i utasítás hajtódik végre.

Érdekes eset az, amikor az SZ aktuális értéke egyik c_i értékkel sem egyenlő. Ekkor a Case utasítás nem választ egy utasítást sem végrehajtásra, hacsak egy kiegészítő utasításról nem gondoskodunk ELSE u_{n+1} alakban. Ez utóbbi esetben ez a rendkívüli utasítás hajtódik végre.

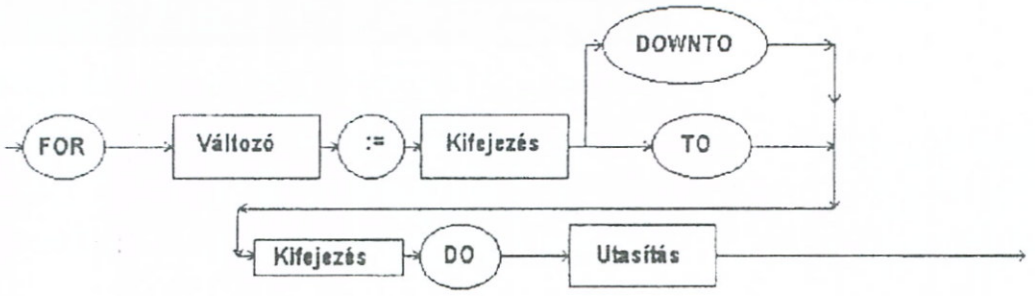
A Case utasítással kapcsolatban meg kell jegyezni, hogy hatékonyan dolgozik, mert ha a szelektor értéke valamelyik konstanssal egyenlő, akkor a neki megfelelő utasítást végrehajtva a teljes Case utasítás végrehajtása befejeződik, és további vizsgálatokra nem kerül sor.

Ha jól megfigyeljük a lerajzolt és leírt szintaktikát, akkor további érdekes megállapításokat tehetünk. A leírt ELSE elé ; karaktert tettünk. Erre azért van szükség, hogy a Case szerkezet eldönthesse, hogy az utolsó c_i értékhez tartozó utasítást — ami az egész választható szerkezet vége is — elválasszuk a rendkívüli esettől.

Az is érdekes szintaktikai megjelenés, hogy úgy szerepel az END alapszó, hogy nem tartozik hozzá BEGIN. Ez azért van, mert most az END szó nem az összetett utasítás záró zárójelét jelenti, hanem a Case strukturált utasítás végét jelzi.

3. Ciklusutasítások: Bizonyos utasítások ismételt végrehajtásáról gondoskodnak. A ciklus azon utasítását vagy utasítássorozatát, amelyet ismételten végrehajt, a ciklus magjának nevezzük. A Pascal rendszerben háromféle ciklusutasítást ismerünk.

A FOR ciklus vagy számlálós ciklus: elöltesztelő ciklus. Szintaktikájának diagramja a következő:



FOR V:=K_1 TO K_2 DO U

A V változó a ciklus paramétere. A K_1 kifejezés a V változó kezdőértéke, K_2 pedig a végértéke, U pedig a ciklus magja, ami tetszőleges utasítás lehet. V, K_1, K_2 csak sorszámozott típusú lehet. Azt mondhatjuk, hogy ez a ciklusutasítás K_1 értékkel kezdve $+1$ lépésenként K_2 értékig hajtja végre a ciklus magját.

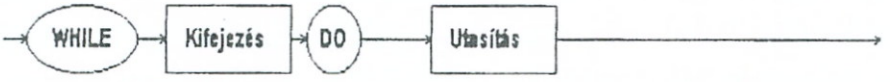
Ha csökkenteni akarjuk a ciklusparaméter értékét, akkor a FOR V:=K_1 DOWNTO K_2 DO U utasítást írhatjuk. Ekkor a lépésköz -1 lesz. A ciklusutasítás K_1 értékkel kezdve -1 lépésenként K_2 értékig hajtja végre a ciklus magját.

Fontos pozitív tulajdonsága ennek a ciklusutasításnak, hogy elöltesztelő. Ha A_1 és K_2 értékeit alkalmas módon adtuk meg, akkor mindkét esetben elérhető az, hogy az utasítás egyszer se hajtsa végre a ciklus magját.

A Turbo Pascalban a számláló ciklusok lépésközének értéke csak $+1$ vagy -1 lehet. Fontos azt is megjegyezni, hogy a ciklusutasításban szereplő változó és a kifejezések sorszámozott típusúak lehetnek.

Jó tudni, hogy V értékét a ciklusmagban megváltoztathatjuk, de az A_1 és A_2 értékeit nem.

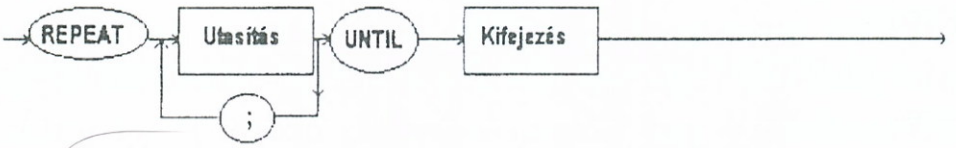
Az elöltesztelő ciklusok másik fajtája a **WHILE ciklus.** Ekkor a ciklus magjának végrehajtása egy logikai kifejezés értékétől függ.



Szintaktikája a következő: WHILE L DO U, ahol L logikai kifejezés, U pedig utasítás. Mivel U rendszerint befolyásolja L értékét, nem lesz végtelen a ciklus, azaz ki tudunk belőle lépni. A ciklusmag mindaddig hajtódik végre,

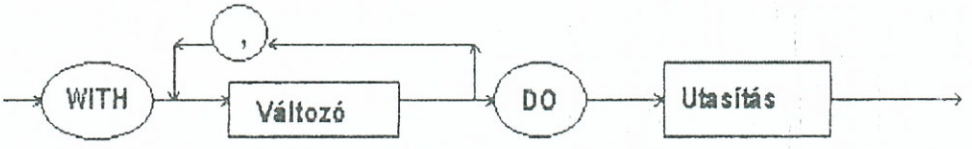
amíg L értéke igaz. Fontos hangsúlyozni, hogy a ciklus elöltesztelő, így ha az első vizsgálatkor L értéke FALSE, akkor az utasítás egyszer sem hajtja végre a ciklus magját. Sokszor éppen ezt akarjuk az algoritmusban biztosítani.

A **REPEAT ciklusutasítás** hátultesztelő ciklusutasítás:



Szintaktikája: REPEAT U UNTIL L, ahol U a ciklus magja, L pedig logikai kifejezés. Egyszer mindenképpen végrehajtja a ciklus magját az utasítás. Az L logikai kifejezés értékétől függ, hogy ezután mi történik. Ha az L értéke TRUE, akkor kilép a program a ciklusból. Ha L értéke FALSE, akkor ismét a ciklusmag utasításainak végrehajtását végzi. A két kulcsszó, a REPEAT és az UNTIL szavak ellátják az utasítás-zárójelek feladatát is, így az összetett utasítást BEGIN END utasítás-zárójelek nélkül is írhatjuk.

4. A **WITH utasítás**: Minősítő utasítás. Segítségével a rekordok mezőit egyszerűbben tudjuk írni. Most csak az utasítás szintaxisdiagramját mutatjuk meg, hogy az összefoglaló utasításdiagramban látható minden utasítást megemlítsünk.



Részletes elemzésével a rekordok tárgyalásakor foglalkozunk.

2.3.1.11. Kidolgozott feladat

Körülbelül ennyi lenne ennek a résznek az elmélete. Most pedig, mint ígértük, megírjuk előző programunkat rövidebben. Felhasználjuk a tömbökről és a ciklusokról tanultakat.

```
PROGRAM OSSZEAD_EGESZET;
USES CRT;
VAR osszeg: LONGINT;
    szamok: ARRAY[1..4] OF LONGINT;
    i: BYTE;      {Ez a ciklusváltozó}
BEGIN
    CLRSCR;
    FOR i:=1 TO 4 DO
```

```
    BEGIN
        WRITE('Kérem a(z) ',i,'. számot:'); READLN(szamok[i]);
    END;
    CLRSCR; osszeg:=0;
    FOR i:=1 TO 4 DO
    BEGIN
        WRITELN(szamok[i]:14); osszeg:=osszeg+szamok[i];
    END;
        WRITELN('+___________'); WRITELN(osszeg:14);
    READLN;
END.
```

Jól látszik, hogy ez a verzió csak strukturáltságában különbözik az előbbitől. Lehet, hogy nem olyan áttekinthető, mint előző társa, de gondoljunk bele: mi lett volna, ha például 100 számot kellett volna összeadnunk. A deklarációs rész csak abban más, hogy a számoknak most nem külön-külön foglaltunk helyet, hanem egy tömbben, illetve föl kellett vennünk egy indexváltozót a ciklus miatt. Így foglalkozzunk a programblokkal részletesebben. A képernyőtörlés után indítottunk egy számlálós ciklust, ami négyszer fog lefutni. Ennek a ciklusmagja beolvassa az aktuális tömbelemhez tartozó számot, csak ezt nem nyolc sorban teszi, hanem négyben, tehát a program helyfoglalása csökkent. Az újabb képernyőtörlést követő hat sor — mint látható — az összegzés tétele kiíratással összevonva. A program befejezése pedig láthatóan ugyanaz, mint az előzőben.

Nézzünk egy másik feladatot: olvassunk be számokat végjelig (0) úgy, hogy közben állapítsuk meg, melyik a legnagyobb közülük? Mint látható, két változóra lesz szükségünk: az egyik az aktuális beolvasott szám, a másik a maximum. Mivel végjelig kell beolvasni, ezért elöltesztelő ciklust kell használnunk.

A program:

```
PROGRAM MAX_KIV;
USES CRT;
VAR max, szam:LONGINT;
BEGIN
  CLRSCR;
  WRITE('A szám: ');READLN(szam);
  max:=szam;
  WHILE szam<>0 DO
    BEGIN
      IF szam>max THEN max:=szam;
      WRITE('A következő szám: ');READLN(szam);
```

```
    END;
  WRITELN('Az adott számok közül a legnagyobb a(z) ',max);
  READLN;
END.
```

Mivel a deklarációs rész megint egyértelmű, nem foglalkozunk vele. A képernyőtörlést követően bekértük az első számot, ez most az előkészítő rész, hiszen ettől függ a ciklusba való belépés lehetősége. Ha megfelel a szám a kritériumnak, akkor belépünk a ciklusba, amely egy összehasonlítást és egy újbóli beolvasást tartalmaz. Ha a beadott szám nagyobb, mint az összes előző, akkor azt megjegyzi a max-ban. A program végén ezt kell kiíratni.

A könyvben a későbbiekben is gyakran találkozunk kidolgozott feladatokkal. Most viszont az a fontos, hogy mindenki kipróbálja saját tudását. Ezért javasoljuk, hogy a következő feladatok közül oldjon meg annyit, amennyi ahhoz szükséges, hogy egyszerű feladatok deklarációit és a gyakran használatos utasításokat megbízhatóan tudja kódolni!

2.3.1.12. Feladatok

1. Olvassunk be végjelig számokat, közben válasszuk ki a legnagyobb és a legkisebb elemet.

2. Számoljuk ki végjelig beolvasott pozitív számok átlagát és szórását.

3. Határozzuk meg a leütött betűről, hogy a betű magánhangzó vagy mássalhangzó volt-e. Fejlesszük programunkat úgy, hogy csak a betűkaraktereket vegye figyelembe.

4. Kérjük be a háromszög három oldalát, és ha „jók a megadott adatok", mondja meg a program, hogy a háromszög derékszögű-e vagy sem.

5. Kérjünk be egy n pozitív egész számot. A program adja meg 2^n értékét.

6. Számoljuk ki az első n pozitív egész szám összegét.

7. Számoljuk ki az első n pozitív egész szám négyzetének az összegét.

8. Kérjünk be egy vektorba (egydimenziós tömb) számokat, rendezzük nem növekvő sorrendbe a számokat.

9. Töltsünk fel pozitív egészekkel egy $n \times m$-es mátrixot, majd adjuk meg a legnagyobb, illetve legkisebb elemét.

10. Számolja ki a program $n!$ értékét. Törekedjünk arra, hogy a program nagy n értékekre is tudjon helyesen dolgozni.

11. Számítsa ki a program bemenő n szám átlagát.

12. Számítsa ki bemenő n szám átlagát és szórását. A szórás képletét úgy alakítsa át, hogy csak egy ciklust használjon a programban.

13. Írja ki a program az angol ábécé nagybetűit először betűrendben, majd visszafelé. A kódoláskor gondoljon arra, hogy a számláló ciklusnál a

ciklusparaméter karaktertípusú is lehet.

14. Olvasson be a program * végjelig neveket egy elegendően nagy tömbbe, majd rendezzük a neveket ábécérendbe. A rendezés után írjuk ki a névsort.

15. Olvassunk be számokat végjelig, és közben határozzuk meg a legkisebb és legnagyobb számokat. Írjuk ki ezek átlagát is.

16. Beolvasott nevek között keressünk inputként megadottakat. Az újabb név keresését végjelig ismételjük.

17. Beolvasott számok közül írjuk ki az adott számnál nagyobbakat. Írjuk ki ezek számát és átlagát, legkisebb és legnagyobb elemének eredeti sorszámát.

18. Egy beolvasott számról állapítsuk meg, hogy hány jegyű.

2.3.1.13. Kidolgozott feladatok

Akkor most ismét következzen néhány példaprogram.

Elsőként nézzünk meg a chr és ord függvények leggyakoribb használatát, vagyis a konvertálást *char* és *byte* típusú adatok között.

```
program ord_es_chr;
begin
  writeln(chr(64)); {Az eredmény a kukac lesz (@).}
  writeln(ord('A')); {Az eredmény 65 lesz}
end.
```

A blokk első sora így is írható: writeln(#64);

Nézzük meg, hogy milyen értékkel tér vissza az ord függvény, ha különböző sorszámozott típusokra alkalmazzuk.

```
program ord;
type tevszak=(tavasz,nyar,osz,tel);   {Deklarálunk egy felsorolt
                                       típust}
var evszak:tevszak;
  egesz:integer;
  logikai:boolean;
begin
  evszak:=osz;
  egesz:=-12;
  logikai:=true;
  writeln(ord(evszak));
  writeln(ord(egesz));
  writeln(ord(logikai));
end.
```

Próbálja ki a programot! Ha a felsorolt kiíratások végeredményeként eggyel többet várt, akkor annak a hibának az a magyarázata, hogy a Pascal program a nullától kezdi a sorszámozást a nem numerikus sorszámozott típusok esetén. Ezek szerint a tavasz sorszáma 0, a nyár sorszáma 1 és így tovább. Reméljük, hogy a 12 sorszámának kiíratásán már senki sem lepődött meg, hiszen az egész számoknak is van sorszáma, csak ez éppen megegyezik magával a számmal. És végül azt is megtudtuk, hogy a true sorszáma 1.

A pred és a succ parancsok is kapcsolatban vannak a sorszámmal, de visszatérési értékük nem egy szám, hanem a bemenő adat típusával egyezik meg. Ha az előző típus definícióját használjuk, akkor:

```
pred(nyar)=tavasz
succ(nyar)=osz
pred(-12)=-13
```

Még egy érdekes megállapítást tehetünk:

pred(true)=false és pred(false)=true

Tehát a pred és succ túlcsordulásánál (vagy éppen alulcsordulásánál) nem történik hibajelzés, azonban pred(tavasz)<>tel! Miért? Erre azonnal választ kapunk, mihelyt kipróbáljuk a writeln(ord(pred(tavasz))) eljárást.

Az eredmény sajnos −1. Valószínű, hogy a felsorolt típust hasonlóan tárolja a gép, mint az egész típusokat, ezért a túlcsordulás az egész típusok határainál történik meg.

A round és a trunc függvények valós típust alakítanak egész típusúvá. Sajnos könnyű őket összekeverni, ezért érdemes megtanulni a magyar jelentésüket. A *round* kerekítést, a *truncation* levágást, csonkítást jelent. Kipróbálásukhoz az alábbi programot ajánljuk:

```
program round_es_trunc;
begin
  Writeln(1.4:4:1, 'round(1.4)=', Round(1.4));
  Writeln(1.5:4:1, 'round(1.5)=', Round(1.5));
  Writeln(-1.4:4:1, 'round(-1.4)=', Round(-1.4));
  Writeln(-1.5:4:1, 'round(-1.5)=', Round(-1.5));
  Writeln(1.4:4:1, 'trunc(1.4)=', Trunc(1.4));
  Writeln(1.5:4:1, 'trunc(1.5)=', Trunc(1.5));
  Writeln(-1.4:4:1, 'trunc(-1.4)=', Trunc(-1.4));
  Writeln(-1.5:4:1, 'trunc(-1.5)=', Trunc(-1.5));
end.
```

A dec és inc két eljárás, a gyors növelést és csökkentést segíti elő. Ezeket érdemes ciklusban használni, például kereséskor:

```
program keresgel;
var tomb:array[1..100] of integer;    {Ebbe kerülnek az adatok.}
```

```
  i:integer;          {Kereséshez}
  max:integer;          {Elemek száma}
  mit:integer;          {Mit kell keresni?}
begin
  max:=1; writeln('Tömb feltöltése (kilépés 0-val)');
  readln(tomb[max]);
  while tomb[max]<>0 do
  begin          inc(max);
    readln(tomb(max));
  end;
  dec(max);          write('Mit keressek?'); readln(mit);
  i:=1;
  while (i<max)and(tomb[i]<>keres) do inc(i);
  if tomb[i]=keres then writeln('Megvan');
end.
```

Természetesen az inc(i) helyett használhattuk volna az i:=i+1-et is, de ez lassúbb lenne.

Mint már említettük, a Pascal nyelv eléggé szigorú a változók típusára nézve. Ha programunkban egy karaktertípusú K változó sorszámát szeretnénk használni, akkor azt az ord(K) konvertálással tehetjük meg. Ez a függvény jól alkalmazható stringek aktuális értékének meghatározásához is a következő módon: Hossz:=Ord(Szoveg[0]); A string felfogható egy tömbnek is, melynek 1. eleme adja a string első karakterét, 2. eleme a másodikat stb. A string hosszát a 0. elem tartalmazza, de úgy, hogy a szám helyett azt a karaktert tartalmazza, melynek sorszáma a hossz. A string hosszát ugyan meghatározhatjuk a Length(Szoveg) standard függvénnyel is.

Matematikai ismereteinket felhasználva a függvények sorát bővíthetjük. Az aritmetikai függvények felsorolásakor nem találkoztunk az e-től különböző alapú exponenciális és a logaritmusfüggvényekkel sem. Az a^x ($a > 0$, $x \in \mathbf{R}$) és $\log_a(x)$ ($x > 0$, $a > 0$, $a \neq 1$) értékét kiszámolhatjuk az

$$a^x = \texttt{exp(x*ln(a))}, \quad \text{illetve} \quad \log_a x = \texttt{ln(x)/ln(a)}$$

formulák segítségével. Az x^n ($n \in \mathbf{N}$) hatvány kiszámítására is használhatjuk az $x^n = \texttt{exp(a*ln(x))}$ képletet, de ciklusutasítással is dolgozhatunk.

Mielőtt folytatnánk további feladatok kidolgozását, egy nagyon fontos dolgot még el kell mondanunk: az előbb felsorolt függvények a SYSTEM unitban vannak elhelyezve. Ez a unit a program fordításakor magától beépül programunkba (természetesen csak akkor, ha szükség van rá), ami azt jelenti: anélkül, hogy a unitok deklarációs részében hivatkoznánk rá, azaz

beírnánk azt, hogy USES SYSTEM, a programrendszer fordítója automatikusan beépíti programunkba.

A következőkben nézzünk meg három egyszerű példát az imént tanultakra. Írassuk ki az első 20 pozitív egész szám négyzetgyökét 2 tizedes pontossággal. A feladatot próbáljuk meg strukturáltan és minél rövidebben megoldani. A megoldáshoz az SQRT(x) függvényre lesz szükségünk. Lássuk a megoldás forráslistáját!

```
PROGRAM negyzetgyok;
USES CRT;
VAR i : INTEGER;
BEGIN    CLRSCR;
FOR i:=1 TO 20 DO
WRITELN('A szám: ',i, ', négyzetgyöke: ', SQRT(i):2:2);
READLN;
END.
```

Mint láthatjuk, magát a feladatot a deklarációs rész nélkül 2 sorban meg lehet oldani, a többi szinte csak a szépérzetünket szolgálja. Természetesen ennek az az oka elsődlegesen, hogy a feladat meglehetősen egyszerű. Mivel az első 20 pozitív egész négyzetgyökét kellett kiíratni, egy számlálós ciklust indítottunk, s maga a változónk fogja adni a soron következő egészet. A WRITELN eljárásban négy paramétert adtunk meg:

'A szám:': egy string,
i: maga a változó,
', négyzetgyöke: ': még egy string,
SQRT(i):5:2: a változó négyzetgyöke a mezőszélességgel együtt.

Ezt megoldhattuk volna úgy is, hogy egymás után használtuk volna a WRITE, illetve WRITELN utasításokat, de pontosan ez adja meg a struktúrának a lényegét: minél összetettebb, de még jól áttekinthető programsorokat adjunk meg. Miután a WRITE és a WRITELN eljárásoknál a paraméterek „hosszú sorát" megadhatjuk, semmi értelme a kód hosszúságát növelni, ami az átláthatóságot és a követhetőséget is ronthatja.

A következő feladat az legyen, hogy írassuk ki a képernyőre a hárommal osztható pozitív egész számokat az $[1, 100]$ intervallumban visszafelé. Próbáljuk meg ismét az előbbi technikát!

```
PROGRAM harmasok;
USES CRT;
VAR i:INTEGER;
BEGIN
  CLRSCR;
  WRITELN('99-től csökkenő sorrendben a pozitív számok a
```

```
következők:');
  WRITELN;
  FOR i:=99 DOWNTO 1 DO
  IF i/3=INT(i/3) THEN WRITE(i:3);
  READLN;
END.
```

Jól láthatjuk, hogy megint nagyon egyszerűen megoldható a probléma. Miután a számlálós ciklus nemcsak előre, de hátrafelé is indítható, a visszafelé való kiíratás nem lehet gond. A FOR-ciklus magját tulajdonképpen egy elágazás alkotja. Ez azt csinálja, hogy megvizsgálja a következő értéket, hogy osztható-e hárommal. Ha osztható, akkor a hányados egész része megegyezik a hányadossal. S ha ez így van az adott értéknél, akkor kiírathatjuk. Jól megfigyelhető esetünkben a típus-kompatibilitás, jobban mondva a típusazonosság: az egyenlőség mind a két oldala valós típusú, mivel az osztás mindig valós, az INT függvény is valós típusú értéket ad vissza.

Ezután feladatunk a következő: egy dobókockával dobjunk úgy egymás után valahányszor (értsd: bekérhető legyen), hogy a páros dobások valószínűsége háromszorosa legyen a páratlanokénak. A különböző dobások eredményét a futás végén írassuk is ki.

```
PROGRAM kocka;
VAR x,n,i,par,prt : INTEGER;
BEGIN
  par:=0;
  prt:=0;
  WRITE('Kérem a dobások számát:');
  READLN(n);
  FOR i:=1 TO n DO
  BEGIN
    x:=RANDOM(3);
    IF x>0 THEN par:=par+1
    ELSE prt:=prt+1;
  END;
  WRITELN('Páros= ',par);
  WRITE('Páratlan= ',prt);
  READLN;
END.
```

Érthető, hogy a deklarációs részben sokkal több helyet kellett lefoglalnunk, mint az eddigiekben. A szükséges előkészületek után (nullázás stb.) elindíthatjuk a ciklust, ami pontosan annyiszor fog lefutni, ahányszor a felhasználó akarja. A ciklusmag egy összetett utasítás. Egy értékadással kez-

dődik, amelyben x értéke $[0, 2]$ intervallumbeli érték lesz. Tulajdonképpen ez és az elágazás teszi lehetővé a „furcsa dobókocka" működését. 3-nál nagyobb érték pontosan 3-szor annyi van, mint előtte vele bezárólag: így fogja adni a páros-páratlan arányt.

Mint látható, nem is a kódolás itt az igazán nagy művészet, hanem a probléma helyes megfogása, matematikai ismerete. Hasonló példákat magunk is készíthetünk.

2.3.1.14. Feladatok

1. Alkossunk programot, amely bemenő összeget címletel, és szépen elrendezve kiírja.

2. Készítsük el polinomok helyettesítési értékének kiszámítását a Horner-féle séma alapján. A polinom $F(x) := a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$, a bemenő paraméterek: $n, a_n, \ldots, a_0, x$.

3. A program döntse el, hogy bemenő 2 oldalból és közbezárt szögből milyen háromszög szerkeszthető, majd adja meg a területét is.

4. Készítsen programot, mely kiszámítja a háromszög területét a három oldal ismeretében. A háromszög létezését is vizsgálja.

5. A szinusztétel egy háromszögben két oldal hossza és a velük szemközti szögek szinusza közötti összefüggést adja meg. Készítsen olyan programot, mely három adat esetén kiszámítja a hiányzó negyediket.

6. Készítsük el a tg függvény táblázatát a $[0°, 90°)$ intervallumban $1°$ lépésenként.

7. Írassuk ki egy bemenő szám tetszőleges alapú logaritmusának értékét.

8. Határozzuk meg egy bemenő pozitív egész szám minden osztóját. A program hibás adatokra is reagáljon.

9. Írja ki a program egy egész szám osztóinak számát.

10. Döntse el a program egy egész számról, hogy prímszám-e.

11. Határozzuk meg két egész szám legnagyobb közös osztóját és legkisebb közös többszörösét.

12. Bemenő számot bontson a program törzstényezőire.

13. Szimulálja a színes korong feldobását. A program írja ki az események gyakoriságait és relatív gyakoriságait is.

14. Írjunk totószelvényt kitöltő programot.

15. Szimuláljuk a lottószámok húzását.

16. Készítsünk olyan konvertáló programot, amely egész számoknál áttér 10-es számrendszerből a alapú számrendszerbe.

17. Készítsünk olyan programot, mely a alapú számrendszerben adott egész számot tízes alapúra ír át. A kényelmes adatmegadás érdekében jól fogalmazza meg a feladatot.

18. Készítsünk programot, ami bemenő értéket kerekít. (Ne a Round(x) függvény használatával.)

19. Írja ki a program bemenő számokról, hogy egész vagy törtszámokat adtunk meg. A program végjelig kérjen adatokat.

2.4. String függvények és eljárások

Most egy újabb típusra nézzük át a használható operációkat. Röviden foglaljuk össze azokat az ismereteket, amelyeket korábban a string (karakterlánc-)típussal kapcsolatban tettünk. Mint már korábban is láttuk, a karakterláncokban szövegeket tárolhatunk. Minden string típusnak van deklarált hossza. Amikor pedig megadjuk a string értékét, akkor a szövegnek aktuális hossza is van. Ez az információ a string 0. elemében tárolódik. Az aktuális hossz nem lehet nagyobb a deklarált (maximális) hossznál. Ha valamelyik operációnál a string aktuális hossza nagyobb lenne, mint a maximális hossz, akkor a szöveg csonkul. Az előző meggondolásból is láthatjuk, hogy a szöveg betűi a tömb elemeiként is kezelhetők. A stringet nemcsak stringként, hanem egy char típusú tömb elemeiként is elérhetjük. Például: ha s egy string, akkor s[1] az első karaktere, s[2] a második és így tovább. Ennek segítségével karakterenként is vizsgálhatjuk a megadott stringet. Figyelem: bár az s[0] a string hosszát adja meg, ezt nem szám formájában teszi, hanem a hossznak az ASCII karakterét adja vissza (hiszen ez char típus). És most lássuk a stringekkel kapcsolatos operációkat.

+: Ez az operáció egy művelet. Segítségével két sztringet összeadhatunk. Ez a művelet az operandusok karaktersorozatait egymás után fűzi leírásuk sorrendjében. Használata két stringre: s1+s2, de használhatjuk többre is: s1+s2+s3. Pl.: 'vas'+'út' eredménye 'vasút'. A kapott eredményt — értékadó utasítással — string típusú változóba tehetjük: s:= s1+s2+s3.

CONCAT(s1,s2,[s3,...,sn]:string):string E függvény visszatérési értéke megegyezik az előző művelet értékével. s:=CONCAT(s1,s2,s3). Gyakran alkalmazzuk helyette az előző műveletet.

COPY(s:string;i:integer;db:integer):string Ez a függvény az s string i-edik karakterétől db darab karaktert ad vissza. Ha az s stringben már nincs annyi karakter, amennyi kellene, akkor kevesebbet fog visszaadni.

DELETE(Var s:string;i:integer;db:integer) Ez az eljárás az s stringből az i-edik karaktertől kezdve db darab karaktert kitöröl.

INSERT(s1:string;Var s2:string;i:integer) Ez az eljárás a s2 stringbe az i-edik karaktertől beszúrja az s2 stringet. Ha i nagyobb, mint s2 aktuális hossza, akkor s2 végéhez fűzi az s1 karaktereit. Ha a végeredmény hosszabb lenne, mint az s2 deklarált hossza, akkor a felesleg levágódik.

LENGTH(s:string):integer A függvény az s string hosszát adja vissza.

UPCASE(c:char):char Ez a függvény csak egy karakterre érvényes. A megadott karaktert nagybetűssé konvertálja. Csak a 97, . . . , 122 ASCII kódú karaktereket konvertálja, az ékezetes betűket nem.

POS(substr:string;s:string):byte A függvény megadja, hogy az s stringben, hol van a substr. Ha nincs benne, akkor a visszatérési érték 0.

STR(x[:m1[:m2]]; Var s:string) Ez az eljárás az x számot konvertálja karakterlánctípusúvá. Az m1 és m2 feladatát már láttuk korábban a kiíró eljárásoknál. Az m1 jelöli a változó teljes hosszát, m2 pedig azt, hogy a tizedespont után hány karaktert írjon ki.

VAL(s:string; Var v; kod:integer) Ez az eljárás pedig az s stringet konvertálja — a v változóba — számmá. A kod változóban 0-át kapunk, ha az átalakítás sikeres volt. Ha valamelyik karakternél „elakad" a konvertálás, akkorannak sorszáma kerül a kod változóba.

A következő két függvény nem teljesen a stringekhez kapcsolódik, de máshová nem akartuk helyezni. Amikor az EXE-re lefordított kész programunkat indítjuk, akkor a neve után adhatunk át paramétereket. Ha a fejlesztőből indítjuk, akkor a RUN/PARAMETERS dialógusdobozban adhatjuk meg. A következő két függvény ezeket a paramétereket kezeli.

PARAMCOUNT:word A függvény megadja, hogy hány paramétert gépeltünk be. Ha értéke 0, akkor csak az indítási név lett begépelve.

PARAMSTR(Index):string A függvény visszaadja az index-edik paramétert.

2.4.1. Kidolgozott feladatok

És most lássunk néhány példát ezekre a függvényekre és eljárásokra.

1. Programunk egy bemenő karakterláncot nagybetűs karakterlánccá konvertál. Sajnos az upcase csak az angol ábécé betűit konvertálja.

```
program nagybetu;
var s:string;
  i:integer;
begin
  write('Szöveg: '); readln(s);
  for i:=1 to length(s) do s[i]:=upcase(s[i]);
                  {nagybetűkké konvertál}
  writeln(s);
  readln
end.
```

2. Most számoljuk meg, hogy milyen betűből mennyi van a bevitt szövegben?

```
program betuszamtan;
var tomb:array[0..255] of byte;
  i:byte;
  s:string;
begin
  write('Szöveg:');readln(s);
  for i:=0 to 255 do tomb[i]:=0;
  for i:=1 to length(s) do inc(tomb[ord(i)]);
  for i:=1 to 255 do if tomb[i]>0 then
  writeln(i,' ',chr(i),' ',tomb[i]);
end.
```

3. Gyakori feladat az is, hogy egy mondatból az első szót kell leválasztani. Feltételezzük, hogy a beolvasott string elején nincsenek szóközök és tabulátorok, de két szó között lehet több is.

```
program darabol;
var s,s1:string;
  i:integer;
begin
  i:=1;
  while (i<=length(s)) and not(s[i] in [#32,#9]) do inc(i);
        {megkerestük az első szó végét}
  s1:=copy(s,1,i-1);
  while (i<=length(s))and (s[i] in [#32,#9]) do inc(i)
        {kihagyjuk a felesleget}
  delete(s,1,i-1); writeln('Első szó:',s1); writeln('Maradék:',s);
end.
```

Ezt a módszert később jól használhatjuk majd szövegelemzésre.

4. Következő feladatunk, hogy egy adott szövegben cseréljünk ki bizonyos részeket.

```
program strcsere;
var s,mit,mire:string;
  i:byte;
begin
  write('Szöveg:');readln(s);
  write('Mit cseréljek:');readln(mit);
  write('Mire:');readln(mire);
  i:=pos(mit,s);
  while i>0 do
  begin
    delete(s,i,length(mit));
```

```
      insert(s,mire,i);
      i:=pos(mit,s);
    end;
    writeln('Csere után: ',s);
end.
```

5. Kérjük be a mai dátumot (külön az évet, hónapot és a napot) egész változókba, majd írjuk ki ÉÉÉÉ-HH-NN formában.

```
program datum1;
var ev:word;
ho,nap:byte;
s,s1:string;
begin
  write('Év:'); readln(ev);
  write('Hó:'); readln(ho);
  write('Nap:'); readln(nap);
  str(ev,s1);          s:=copy('000'+s1,length(s1),4)+'-';
  str(ho,s1);          s:=copy('0'+s1,length(s1),2)+'-';
  str(nap,s1);          s:=copy('0'+s1,length(s1),2);
  writeln('Dátum:',s);
end.
```

6. Most pedig kérjünk be egy dátumot ÉÉÉÉ-HH-NN formában, és szedjük szét év, hónap és napra, majd tároljuk egészként. Ha valami hiba van a bevitelben, akkor szóljunk.

```
program datum2;
var s:string;
  ev:word;
  ho,nap:byte;
  i:integer;
begin
  write('Kérem a dátumot ÉÉÉÉ-HH-NN formában:');readln(s);
  val(copy(s,1,4),ev,i);
  if i>0 then writeln('Hibás az évszám!')
  else
    if s[5]<>'-' then writeln('Hiányzik az első kötőjel!') else
    begin
    val(copy(s,6,2),ho,i);
    if (ho>12) or (i>0) then writeln('Hibás a hónap!') else
    if s[8]<>'-' then writeln('Hiányzik az második kötőjel!')
    else
    begin
```

```
        val(copy(s,9,2),nap,i);
        if (i>0) or (nap>31) then writeln('Hibás a nap!') else
          begin
              writeln('Év:',ev);
              writeln('Hó:',ho);
              writeln('Nap:',nap);
          end;
    end;
    end;
end.
```

2.4.2. Feladatok

1. Írja ki a program a $[0, 255]$ intervallumban a karaktereket.

2. Leütött betűknek, számoknak írjuk ki a belső kódját (a betűt vagy számot Enter zárja).

3. Egy bemenő szöveget írjon ki a program visszafelé.

4. A bemenő szöveget függőlegesen írja ki a program.

5. Egy beolvasott szöveget írjon ki a program úgy, hogy minden karakter után legyen szóköz.

6. Bármely beírt szót írjon ki a program felülről lefelé úgy, hogy a betűk mellett kódjaikat is lássuk.

7. Határozza meg a program, hogy egy beolvasott szövegben az előforduló karakterek milyen gyakorisággal fordulnak elő.

8. Számolja meg a program, hogy egy szövegben hány magánhangzó, mássalhangzó és egyéb jel van. A magyar ábécé betűivel dolgozzunk.

9. A beolvasott szöveget írja ki a program madárnyelven (évérteved uvugyeve?).

10. A program írjon ki a képernyőre bemenő szót a következő alakban:

A
AL
ALM
ALMA

11. Helyezzen el a program egy bemenő szót N sorban a következő módon:

MATEMA
ATEMATI
TEMATIK
EMATIKA

A szöveg most a MATEMATIKA szó, a bemenő N pedig 4.

12. Egy adott szöveg nagybetűit változtassuk kisbetűkre, a kicsiket nagyra, és mindkét szöveget írjuk ki. Csak betűket fogadjon el a program.

13. Készítsen programot, mely az [1, 100] intervallumban adott egész számot szövegesen is leír.

14. Készítse el a számkiíró programot más nyelven is (angol, német, eszperantó stb.).

15. Határozza meg két bemenő szó közös betűit. Mutassa ki azokat a betűket külön, amelyek ugyanazon a helyen állnak mindkét szóban.

2.5. Konstansok alkalmazása

Két fajtáját különböztetjük meg a konstansoknak.

A nem tipizált konstans igazi konstans. Olyan objektum, melynek értéke már fordításkor ismert, és ez az érték is befordítódik a kódszegmensbe, oda, ahol a program kódja is megtalálható. Értéke a programon belül nem változtatható meg: igazi állandó. Természetesen deklarálni kell. A nem tipizált konstans neve felhasználható további deklarációkban is.

A tipizált konstans tulajdonképpen nem is igazi konstans, hanem inicializált, azaz **kezdőértékkel rendelkező változó.** Úgy viselkednek, mintha rendes változók lennének, de a program fordításakor már értéket kapnak. A program futása során más értéket is felvehetnek. Későbbi deklarációkban nem alkalmazhatók. Hasznosan alkalmazhatjuk viszont a tipizált konstansokat előkészületi munkálatoknál.

Például:
```
CONST    gyujto:INTEGER=0;
  fakt:LONGINT=1;
  matrix: ARRAY[1..sor,1..oszl] OF CHAR=(('a','b'),('a','b'));
```

2.5.1. Kidolgozott feladatok

A feladat az hogy egy véletlenszerűen generált mátrix sorminimumainak maximumát, illetve oszlopmaximumainak minimumát kell megállapítani. Most csak a sorminimumok maximumát kiszámító program eredményét, majd a programot mutatjuk meg.

Az eredmény a következő oldalon látható.

```
File  Edit  Search  Run  Compile  Debug  Options  Window  Help
[■]═══════ Output ═══════[↕]
 69  35   2  85  31  99  52  99  65  75
 29  52  10 100  88  27  72  83  61  87
 56  96  18  30  60   6  61  53  68  40
 84   5  42   9   5  76  48  85  10  87
 32  37  44  87  10  39  14  95 100   8
  8  84   3  79  51  75  28  31  27  98
 30  98  36  56  70  23  38  43  79  94
 60  94  16  26  75  77  39  68  42  55
  7  74  76  70  33  20  80   2  50   2
 45   6  40   7  55  20   3  10  36  89
Sorminimumok:
2 10 6 5 8 3 23 16 2 3          Ezek maximuma: 23
```

A program:

```
PROGRAM soroszl;
USES CRT;
CONST n=10; m=10;
VAR smm,i,j:INTEGER;
  t:ARRAY[1..n,1..m] of INTEGER;
  smin:ARRAY[1..n] of INTEGER;
  smax:ARRAY[1..m] of INTEGER;
BEGIN
  CLRSCR;
  FOR i:=1 TO n DO
  BEGIN
    FOR j:=1 TO m DO
    BEGIN
      t[i,j]:=RANDOM(100)+1;
      WRITE(t[i,j]:4);
      IF j=m THEN WRITELN;
    END;
  END;
  WRITE;
  WRITELN('Sorminimumok: ');
  FOR i:=1 TO n DO
  BEGIN
    smin[i]:=t[i,1];
    FOR j:=2 TO m DO IF t[i,j]<smin[i] THEN smin[i]:=t[i,j];
    WRITE(smin[i],' ');
  END;
  smm:=smin[1];
  FOR i:=2 TO n DO IF smm<smin[i] THEN smm:=smin[i];
  WRITE(' Ezek maximuma: ',smm);
  READLN;
END.
```

Jól látható, hogy egy $n \times m$-es mátrixszal kívántunk dolgozni. Ehhez segítségül hívtuk a nem tipizált konstansokat. Programunkban egy 10×10-es tömbbel dolgoztunk. Most gondoljuk meg, ha egy 20×20-asra szeretnénk áttérni, akkor mennyi javítást kellene eszközölnünk. Tíz helyen kellene javítani a nem is hosszú programban. Talán nem is kell tovább ecsetelni alkalmazásuk jelentőségét.

Nézzünk másik példát is. Ez pedig a címletezés legyen.

```
PROGRAM cimlet;
USES CRT;
CONST penz:ARRAY[1..10] OF STRING[8]= ('ötezres','ezres','ötszá-
zas','százas','ötvenes','húszas','tízes','ötös','kettes','egyes');
ft:ARRAY[1..10] OF INTEGER=(5000,1000,500,100,50,20,10,5,2,1);
tovabb : BOOLEAN=FALSE;
VAR i,db : INTEGER;
  f : LONGINT;
  c :CHAR;
BEGIN
  REPEAT
    CLRSCR;
    WRITE(' Kérem az összeget: '); READLN(f); WRITELN;
    WRITELN(' ',f,' Ft címletelése:');
    i:=1;
    WHILE (f>0) AND (i<11) DO
    BEGIN
      db:=0;
      WHILE f>=ft[i] DO
      BEGIN
        db:=db+1;
        f:=f-ft[i];
      END;
      IF db>0 THEN WRITELN('      ',penz[i],' ',db,' db');
        i:=i+1;
    END;
    WRITE(' Akar tovább számolni? (I/N)'); c:=READKEY;
    tovabb:= Upcase(c) ='N';
  UNTIL tovabb;
END.
```

Mivel a logikai változót inicializáltuk, időt nyertünk a kilépés feltételének vizsgálatakor. A két tömböt a következő okok miatt deklaráltuk ezzel a módszerrel:

— így eléggé jól érthetők a funkciók, jól láthatók a szerkezetek,

— ily módon könnyen megváltoztathatjuk a tömbelemek számát és a címleteket.

Aktualizálja a programot a ma használt címletekre.

2.5.2. Feladatok

1. Az előbbi feladatban adjunk lehetőséget a felhasználónak arra, hogy megváltoztathassa a címletelés sorrendjét, azaz ha valamely címletet előnyben kívánja részesíteni, akkor azt megtehesse.

2. Dobókockával dobjunk n-szer, kérjünk be egy számot, s írassuk ki, hogy az adott számot hányszor dobtuk ki, és hányadikként dobtuk először.

3. Legyen adott a NEVEK[1..db] string típusú adatokkal. Töltsük fel a tömböt úgy, hogy legyenek konstansként megadott VEZETEK (ebben vannak a vezetéknevek) és KERESZT (ebben pedig a keresztnevek) tömbök, melyek tipizált konstansok. Az összes lehetséges módon állítsunk elő neveket. Rendezzük névsorba a tömb elemeit, keressünk benne lineárisan, binárisan.

2.6. A CRT Unit

A megírt példaprogramok többségében már találkozhattunk a következő eljárással: ClrScr. Ez a karakteres képernyőtörlés eljárása, és csak akkor működik, ha a deklarációban szerepeltettük a USES CRT sort. Azt is mondottuk, hogy a későbbiekben még kitérünk ezekre a deklarációkra. Most szeretnénk ezzel az egységgel foglalkozni, ami a karakteres képernyőkezelést teszi lehetővé. Mielőtt azonban rátérnénk a konkrét elemzésre, néhány szót mindenképpen ejtenünk kell a unitokról általában.

A unit olyan speciális programegység, amiben konstansok, típusok, változók, eljárások és függvények vannak deklarálva. Ezen alprogramokat beépíthetjük saját programjainkba, mindössze csak annyit kell tenni, hogy programunkban a unitot deklarálni kell. Ez a következő szintaktika alapján történik: USES unitnév;. Azzal, hogy hogyan épül fel egy Unit, majd később foglalkozunk.

Biztosan felmerült már az a kérdés a kedves olvasóban, hogy minek ez a sok hivatkozás ide-oda, miért ne lehetne a unitot is szervesen, mintegy fizikailag beleépíteni a programba. Pontosan ezért hozták létre: végül is fizikailag csak az a része épül be az aktuális programba, amelyik ténylegesen szükséges, és anélkül, hogy a forráslista hosszát akár egy sorral is megnövelné. Erre pedig azért van szükség, mert egyrészt szeretünk takarékoskodni időnkkel, és kétszer nem megírni azt a programrészt, amit már egyszer megírtunk, másrészt a fordító „mindössze" 64 K hosszú forráslistát tud kezelni, ami bizonyos esetekben kevés is lehet. Alapvetően két fajta unitot különböztetünk meg: standard és saját készítésű unitot. Standard unitnak nevezzük azokat az egységeket, amelyeket a Turbo Pascal fejlesztői a rendszerrel együtt adtak a számunkra. Ilyen például a SYSTEM unit is, amiről már a korábbiakban is esett szó, és később is foglalkozunk vele.

A CRT unit is standard unit. A fejlesztőrendszer alkotói ebben az egységben foglalták össze a karakteres képernyő kezelésére vonatkozó kellékeket, illetve a billentyűzetkezelést s a hanggenerálást. Tartalmaz beépített konstansokat is. Ezeket a főprogramban nem kell deklarálni, mert a unit által ezek a konstansok is globálisak lesznek. Ezek a megállapítások igazak más egységeken belüli deklarációkra is.

2.6.1. Konstansok

1. A TEXTMODE konstansai: Ezek a képernyő üzemmódjának a konstansai. A TEXTMODE(c:BYTE); eljárás a képernyő üzemmódját állítja be a c konstansnak megfelelően, melynek értékei a következők lehetnek:

BW40=0	40 × 25 fekete-fehér CGA
CO40=1	40 × 25 színes CGA
BW80=2	80 × 25 fekete-fehér CGA
CO80=3	80 × 25 színes CGA
MONO=7	80 × 25 fekete-fehér HGC
FONT 8 × 8=256	80 × 43 színes EGA
	80 × 50 színes VGA

2. Az írás- és háttérszínek konstansai:.

BLACK=0	fekete
BLUE=1	kék
GREEN=2	zöld
CYAN=3	türkiz
RED=4	piros
MAGENTA=5	lila
BROWN=6	barna
LIGHTGRAY=8	sötétszürke

3. Írásszínek vagy tintaszínek:

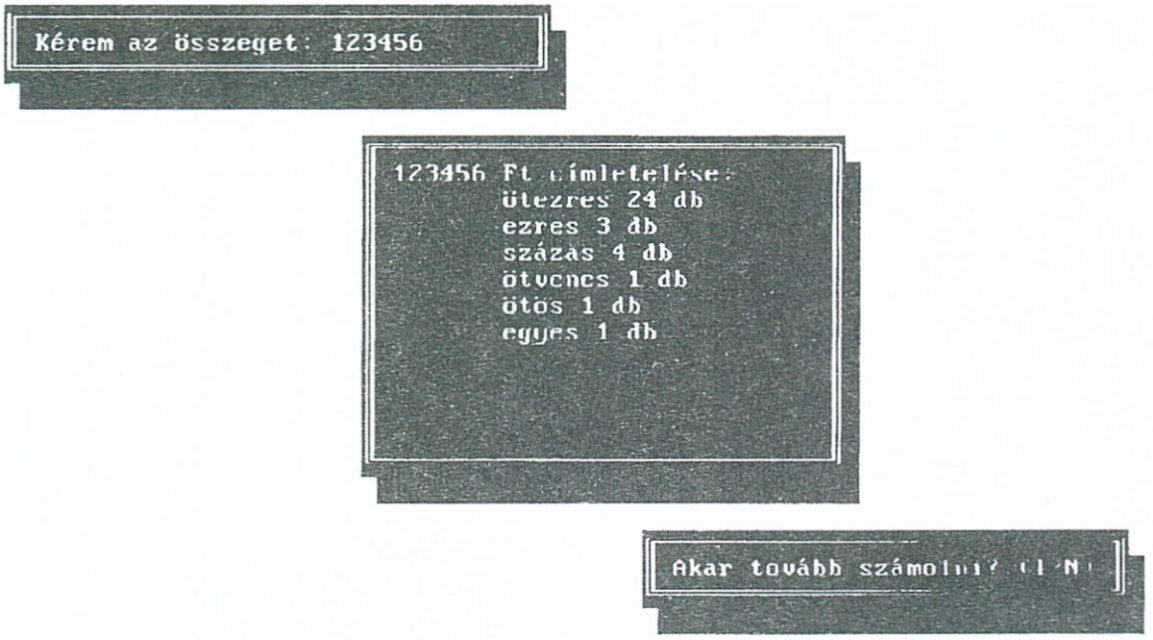
DARKGRAY=8	világosszürke
LIGHTBLUE=9	világoskék
LIGHTGREEN=10	világoszöld
LIGHTCYAN=11	világostürkiz
LIGHTRED=12	világospiros
LIGHTMAGENTA=13	világoslila
YELLOW=14	sárga
WHITE=15	fehér

Ha az írásszínnel (kivétel a fekete) villogtatni akarjuk, akkor adjuk hozzá a következő konstanst:

BLINK=128	villogás

Egy fontos változójáról is ejtsünk szót. Ez a TEXTATTR nevű változó, ami byte típusú. Mindig az aktuális szövegkiíró színt tartalmazza. Akit részletesebben érdekel a változó „szerkezete", tájékozódjon még szakirodalmakban.

A következőkben térjünk rá a különböző eljárások és függvények ismertetésére. Illusztrálásként mutatjuk a képet. A következő anyag ismerete után már ilyen ablakokat is tudunk készíteni. Ilyen eszközökkel programjaink barátságosabbak lesznek.



2.6.2. Eljárások

A **TEXTBACKGROUND** eljárás a képernyő háttérszínének a beállítását végzi el. Szintaktikája: TEXTBACKGROUND(szin:BYTE), ahol a szin értéke 0 és 7 közötti egész (BLACK,...,LIGHTGRAY) lehet, ami színkódot jelent. Hasonló hatású a **TEXTCOLOR** eljárás is, ami a szövegszínt állítja. Szintaktikája: TEXTCOLOR(szin:BYTE), ahol a szin 0 és 15 közötti egész lehet (BLACK,...,WHITE) lehet.

A **CLRSCR** eljárás az aktuális ablakot törli. Ez alapértelmezésben a teljes képernyő. CO80 üzemmód esetén alapértelmezésben a teljes képernyőnek 25 sora és 80 oszlopa van. A törlés valójában az aktuális háttérszínre való festést jelenti. Az alapértelmezéstől eltérő ablakokat is lehet készíteni.

Ezt a **WINDOW** eljárás végzi, amelynek szintaktikája a következő: WINDOW(X_1,Y_1,X_2,Y_2:BYTE), ahol X_1,Y_1 a megnyitandó ablak bal felső, X_2,Y_2 pedig a jobb alsó csúcsának koordinátái. A koordináták abszolútak, tehát mindig az alapértelmezésbeli ablakra vonatkoznak. CO80 üzemmódban az alapértelmezésű ablak koordinátái $(1,1)$ és $(80,25)$. A Window-utasítás hatása még az is, hogy az ablak definiálása után a kurzort az ablak bal felső sarkába pozícionálja.

A **GOTOXY** eljárás a kurzort pozícionálja az aktuális ablakban. A GOTOXY(X,Y:BYTE) a kurzort a képernyő Y. sorának X. oszlopába viszi. Két függvény, a **WHEREX**:BYTE; és a **WHEREY**:BYTE; a kurzor aktuális pozícióját adják vissza Byte típusban.

Van két olyan eljárás, amelyet inkább csak érdekesség miatt említünk meg. A **DELLINE** paraméter nélküli eljárás törli az aktuális kurzorpozíció

sorát, azaz a képernyőn minden e kurzorpozícióhoz tartozó sor alatti sor eggyel feljebb kerül. Ellentéte az INSLINE szintén paraméter nélküli eljárás, amely pedig az aktuális kurzorpozíció sorába egy üres sort szúr be. E két eljárással egész gyors képernyőkezelést lehet elérni.

A billentyűzetkezeléssel kapcsolatban ismerkedjünk meg két eljárással. A billentyűzetről bemenő adattól függ a függvény értéke.

A KEYPRESSED:BOOLEAN; függvény értéke akkor TRUE, ha az adott pillanatban volt lenyomva billentyű. A függvény nem veszi észre a CTRL, ALT, SHIFT, NUMLOCK, SCROLL LOCK leütését.

A READKEY:CHAR; függvény vár egy billentyű leütésére. Addig nem folytatja tovább a futást, amíg a KEYPRESSED értéke FALSE. Értéke a leütött karakter lesz. Tehát egy karakter beolvasására használjuk. Fontos megjegyezni, hogy Enter nem kell a bemenő adat lezárására, és hogy a bemenő karakter nem jelenik meg a képernyőn. Ha ALT+billentyű kombinációt ütünk be, akkor a READKEY értéke 0 lesz, azaz mintha két billentyűt nyomtunk volna le: a következő READKEY érték valamilyen szám lesz. Ez egyébként más funkcióbillentyűkre is igaz.

A DELAY(MS:WORD) eljárás a programunk futását függeszti fel MS millisecundum ideig, Ezt az eljárást dallamok előállításakor is jól tudjuk alkalmazni.

A hanggeneráláshoz két eljárás tartozik.

A SOUND(Hz:WORD) bekapcsolja a beépített hangszórót, és Hz frekvenciájú hangot generál. Ez addig szól, amíg a NOSOUND eljárással ki nem kapcsoljuk a generálást.

2.6.3. Kidolgozott feladatok

Nézzünk néhány érdekes példát.

Generáljunk véletlenszerűen számokat az ablak pozíciójához és színéhez is. Rajzoljuk ki az ablakot, majd tartsunk rövid szünetet. A program addig dolgozzon, amíg egy billentyűt le nem ütünk.

```
PROGRAM ablakok;
USES CRT;
VAR s,x1,x2,y1,y2:INTEGER;
BEGIN
  RANDOMIZE;
    REPEAT
      x1:=RANDOM(80)+1;
      y1:=RANDOM(25)+1;
      REPEAT
          x2:=RANDOM(80)+1;
```

```
            y2:=RANDOM(25)+1;
        UNTIL (x2>x1) AND (y2>y1);
        s:=RANDOM(7);
        WINDOW(x1,y1,x2,y2);
        TEXTBACKGROUND(s);
        CLRSCR;
        DELAY(50);
      UNTIL KEYPRESSED;
END.
```

Az ablak nyitáskor a belső REPEAT-UNTIL ciklus tulajdonképpen csak a program biztonságosságát szolgálja.

A következőkben írjuk meg ezt a programot úgy, hogy az ablakokat keretekkel, illetve árnyékkal is ellátjuk. Ez nemcsak az időtöltést szolgálja, hanem a karakteres képernyő igényes megjelenéséhez is hozzátartozik.

```
PROGRAM keret;
USES CRT;
VAR k,j:INTEGER;
x1,y1,x2,y2,szin:BYTE;
BEGIN
  RANDOMIZE;
  TEXTBACKGROUND(black); CLRSCR;
  REPEAT
  x1:=RANDOM(77)+1; y1:=RANDOM(22)+1;
  REPEAT
    x2:=RANDOM(80)+1; y2:=RANDOM(25)+1;
  UNTIL (x2-1>x1) AND (y2-1>y1);
  WINDOW(x1+1,y1+1,x2+1,y2+1);
  TEXTBACKGROUND(black); CLRSCR;
  WINDOW(x1,y1,x2,y2);
  TEXTBACKGROUND(red);
  CLRSCR;
  WRITE(chr(201));
  FOR k:=x1+1 TO x2-2 DO WRITE(chr(205));
  FOR j:=2 TO y2-y1 DO
  BEGIN
  WRITE(CHR(186)); GOTOXY(x2-x1,j);WRITE(CHR(186),' ');
  END;
  WRITE(CHR(200));
  FOR k:=x1+1 TO x2-2 DO WRITE(CHR(205));
  WRITE(CHR(188));
```

```
    WINDOW(x1+1,y1+1,x2-1,y2-1);
    UNTIL KEYPRESSED;
END.
```

Jól látható, hogy az árnyék tulajdonképpen egy jól elhelyezett ablak, a keret pedig valójában az ablak szélére rajzolt karakterek sora. De a hatás mindenképpen nagy lehet, ha ezt a módszert ügyesen használjuk.

Következő feladatunk legyen az, hogy írjunk olyan programot, amely adott billentyűről megmondja annak belső kódját. A programot készítsük fel arra is, hogy bizonyos billentyűknek kettős, kétbájtos kódja van. Ekkor az első kódja 0.

```
PROGRAM kód;
USES CRT;
VAR k:CHAR;
BEGIN
  CLRSCR;
  REPEAT
    k:=READKEY;
    WRITE(ORD(k));
    IF ORD(k)=0 THEN
    BEGIN
      k:=READKEY;
      WRITE(',',ORD(k));
    END;
    WRITELN;
  UNTIL ORD(k)=27;
END.
```

Nagyon alaposan elemezzük ki a programot, mert a Repeat Until ciklus programbeli alkalmazása nagyon indokolt. Megfigyelhetjük, hogy gond akkor adódik, ha a beolvasott karakter kódja nullával kezdődik. Ekkor tudjuk azt, hogy a következő beolvasás után már a karakter mindkét kódja rendelkezésünkre áll. A kilépést az Esc billentyű lenyomásával érhetjük el.

2.6.4. Feladatok

1. Írjunk olyan programot, amely bemenő összeget címletel. Tervezzünk igényes képernyőképet.

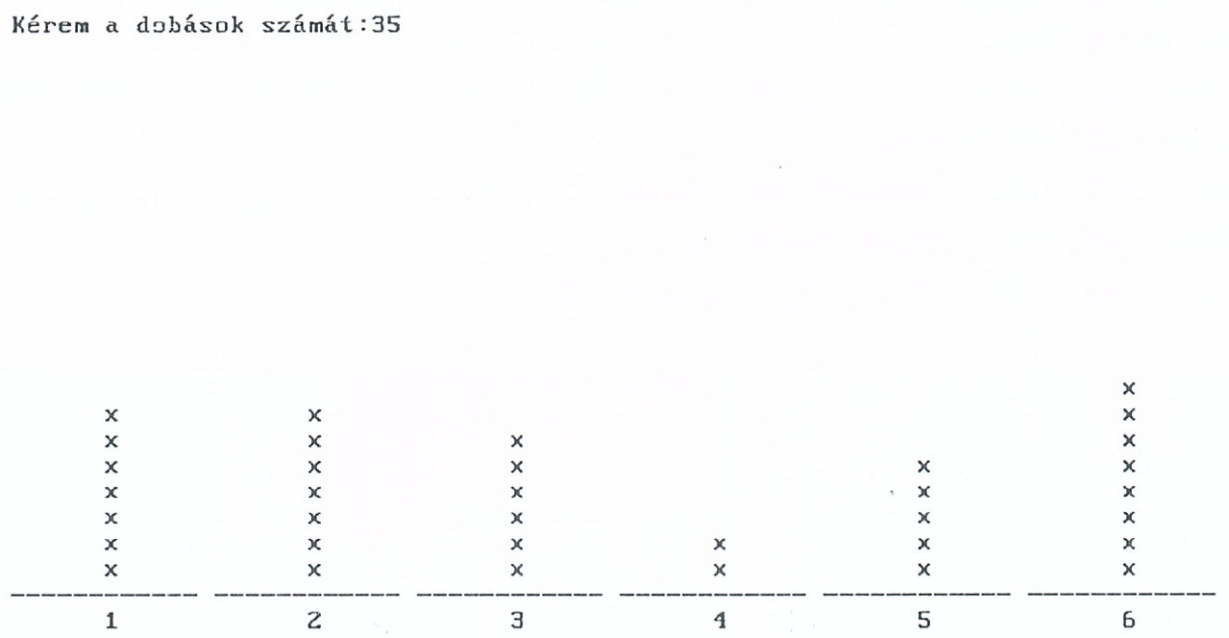
2. Készítsünk zongoraszimulátort.

3. Készítsünk fényújságot, mely a 80 × 25-ös CGA képernyőn a 12 sorban jobbról balra egy bekért szöveget mozgat.

4. Készítsünk menüt. A címkék és a hely legyenek állandóak, az itemek között a kurzormozgató billentyűkkel lehessen mozogni. (A kész programot

tegyük el, mert amikor majd megtanulunk unitokat írni, saját unitot fogunk belőle készíteni. A Crt Unit eszközeit a jövőben alkalmazzuk programjainknál.)

5. Dobjunk dobókockával. Minden dobáshoz más hangmagasság társuljon. A program mutassa meg a dobásokat és azok gyakoriságát.



6. Készítsen programot, mely ablakban lévő menüpontokat kezel úgy, hogy a kívánt, menüpont kiválasztása után a teljes képernyőre tervezett üzenőablakba írja ki az aktuális választás szövegét. A választható menüpontok között legyen VÉGE is.

7. Készítsen olyan menürendszert, amelyben az első ablak menüpontjaiból ismét ablak nyílik.

8. Egy állandó ablakba írja ki a program a leütött billentyű kódját, illetve kódjait.

9. Készítsen digitális órát, amely egy ablakban mutatja az időt. A GetTime(var ora,perc,mp,mp100: word) eljárás, amely leolvassa az idő értékeit a Dos Unitban van.

10. Készítsen a képernyőre nagyobb méretű digitális órát. Vigyázzon arra, hogy a nagy rajz a program futását kedvezőtlenül befolyásolhatja.

11. Egy ablakban mutassa végig a program a karaktereket az $[1, 255]$ intervallumban. A kiírást lehessen lassítani és gyorsítani.

12. Oldja meg az előző feladatot úgy, hogy a sorszámok és karakterek kiírását a kurzor fel/le billentyűkkel lehessen lapozni. A rajzolásokhoz szóba jöhető karaktereket és a kódokat jegyezzük fel.

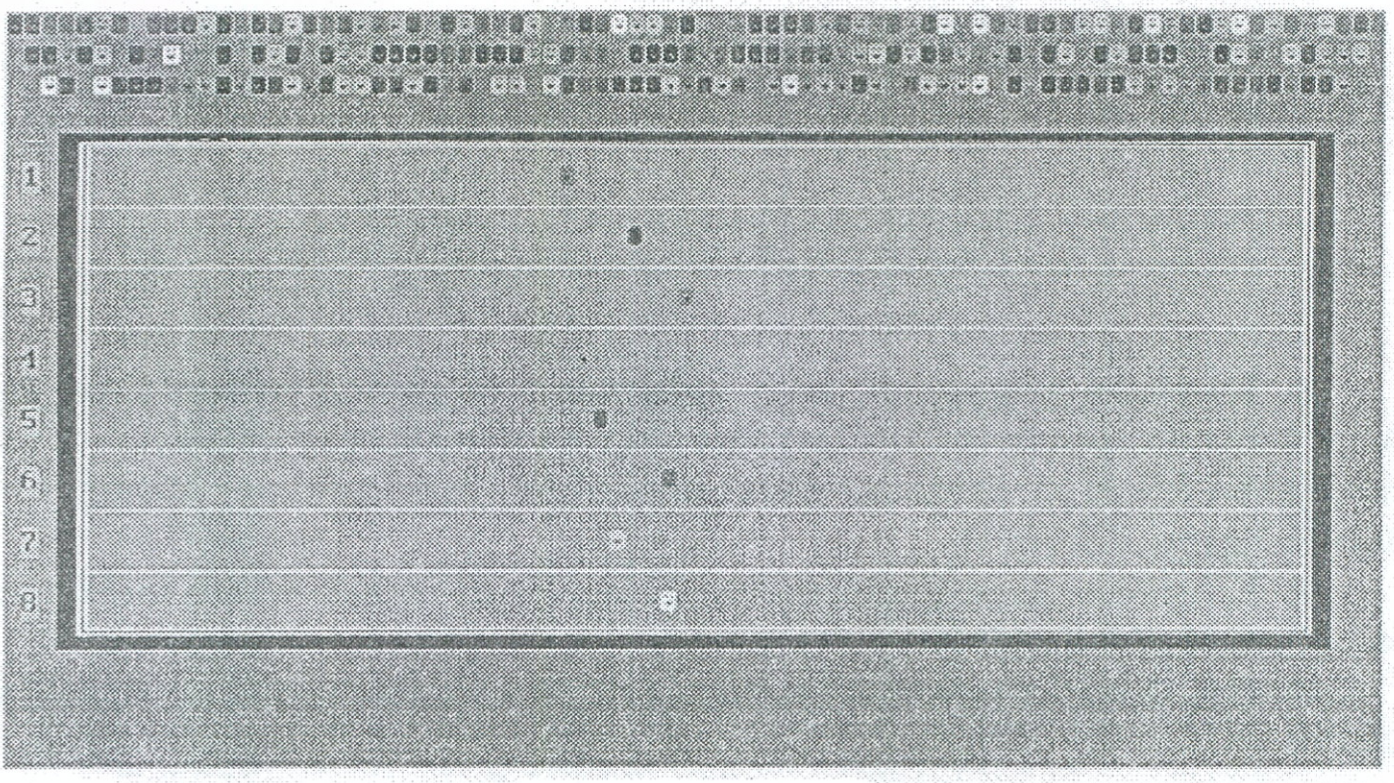
13. Rajzolja ki a program annak a dobókockának egy oldalát, amelyen nem számok, hanem színek jelzik a pontokat. A biztonság kedvéért egy

ablakban a számokat is látni szeretnénk, miközben a gyakorisági grafikon alakulását is látjuk.

14. Egy kis érdekesség vagy inkább játék. Készítsünk olyan programot, amely egy hátúszóversenyt szimulál. Ügyeljünk a következőkre:

— legyen szurkoló közönség is,
— a medencéből ne jöjjön ki a víz és az úszó sem,
— az úszósávok legyenek elválasztva egymástól,
— minden úszót meg lehessen különböztetni egymástól, azaz sapkájuk színe eltérő legyen,
— az úszók véletlenszerű sebességgel ússzanak,
— lehessen eredményt hirdetni.

A képernyőkép valahogy így nézzen ki:



A feladat nem bonyolult, csak kissé időigényes.

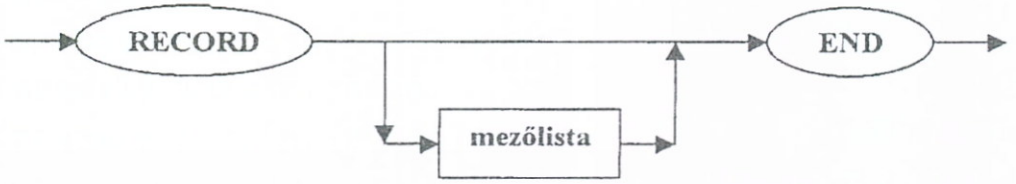
2.7. A rekordtípus és a With utasítás

2.7.1. A rekord

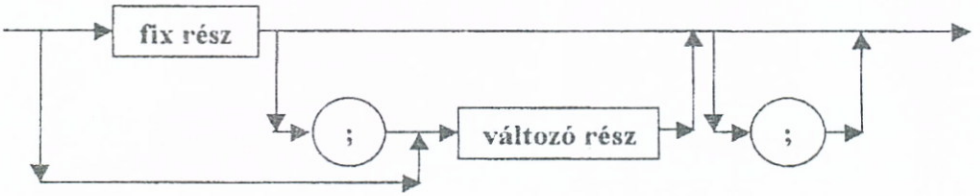
Ebben a részben egy olyan problémával ismertetjük meg a kedves olvasót, amelynek használata azon feladatok megoldását egyszerűsítheti, amelyeket másképpen csak meglehetősen strukturálatlanul lehetne megoldani. Gondoljunk csak az adatállományok kezelésére. Az adatfeldolgozásra a Pascal programnyelvtől alkalmasabb szoftvereket is ismerünk, mégis érdemes a

nyelv ilyen irányú törekvéseit is megismerni.

A **rekord** olyan strukturált adattípus, mely komponensekből áll. A komponenseket mezőknek nevezzük, és ezek **különböző típusúak** (is) lehetnek. Tulajdonképpen ezek a mezők alkotják és definiálják a rekordot.

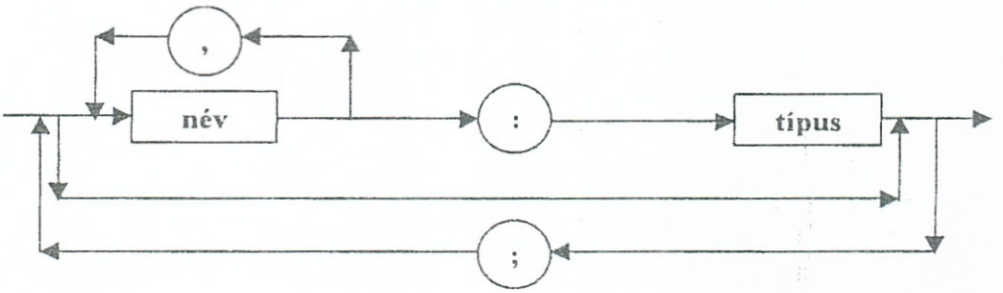


A mezőlista két részből állhat.



Mint láthatjuk, a mezőlistát a fix résszel kell kezdeni, és nem kezdhetjük a változó résszel.

Az állandó (fix) részben fel kell sorolni a mezőket típusaikkal együtt.



Pl.:

```
Record
  ev: integer;
  ho: 1..12;
  nap: 1..31;
end
```

A rekordtípust elhelyezhetjük típus- és változódeklarációban is. Mint típusos konstansnak kezdőérték is adható.

```
TYPE ktip= RECORD
    cim: STRING[5];
    szerzo: STRING[20];
    ar: REAL;
    isbn_szam: INTEGER;
```

```
  END;
VAR konyv1,konyv2: ktip;
  maskonyv:        RECORD
         c:STRING[5];
         kiev:integer;
             end;
CONST
  ezaz:RECORD
       cim:STRING[5];
       kiad:integer;
    END=
  (cim:'A Pascal programozási nyelv elmélete és gyakorlata';
  kiad:1998);
```

Ha típusos konstansként adjuk meg a rekordot, akkor a rekord minden mezőjét és azok értékét meg kell adni. Az elemeket zárójelben, pontosvesszőkkel elválasztva kell leírni. Minden érték elé ki kell írni annak nevét is!

A deklarációk alapján sejthető, hogy a rekordra és mezőire való hivatkozás más, mint a szokásos változóknál, illetve a tömböknél.

Ha a teljes rekordra kívánunk hivatkozni, akkor rekordnevet (azonosítót) kell írni. Pl.: konyv1, konyv2, maskonyv.

A rekord mezőire úgy hivatkozunk, hogy a rekord neve után pont (.) és a mező neve következik (konyv1.szerzo, maskonyv.kiev,...). A mezőket típusaiknak megfelelően használhatjuk. Ha egy programban azt szeretnénk, hogy a felhasználó adja meg a könyv címét, akkor azt ily módon tehetjük:

```
WRITE('A könyv címe:');    READLN(konyv1.cim)
```

A rekord mezői lehetnek egyszerű és strukturált típusok is. Deklarálhatunk mezőnek tömböt, sőt rekordot is.

```
VAR nev:STRING[15];
    sz,szemely:RECORD
      nev : STRING[20];
      szemszam : STRING[11];
      lakcim: RECORD
          varos : string[15];
          utca : string[30];
          END;
      END;
```

A rekordon belüli mezőazonosítóknak egyedieknek kell lenniük. Ha azonos nevű változónk is van a programban és mint mező a rekordban, akkor

azokat meg tudja különböztetni egymástól, hiszen másként kell rájuk hivatkozni.

Az előző deklarációk alapján nézzük át a következő utasításokat:

```
nev:=sz.nev;
nev:=sz.lakcim.varos;
Writeln(szemely.nev:15, ' Lakcíme:');
Writeln(szemely.lakcim.varos);
Writeln(szemely.lakcim.utca);
```

A rekord neve értékadó utasításban is szerepelhet, ha teljesül az értékadás-kompatibilitás. Előző deklarációnk után az sz:=szemely utasítás is adható, és nem szükséges az egyes mezők értékeit egyenként áttölteni. Használhatjuk a rekordnevet fájlból való olvasáskor és fájlba íráskor is. Ezekről az eljárásokról a fájloknál tanulunk.

A rekordokkal a programozás során nagyon gyakran találkozunk. Biztos alkalmazásukat gyakorlással sajátíthatjuk el.

2.7.2. Kidolgozott feladat

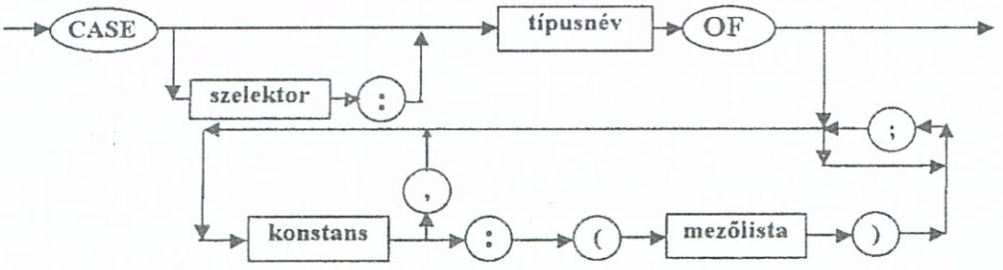
Nézzünk most egy példát. Kissé talán erőltetett a probléma, de bemutatja mindazt a lehetőséget, amit a rekordról eddig tanultunk.

```
PROGRAM rekord;
USES CRT;
VAR a,b:  RECORD
          nev:STRING[20];
          varos:STRING[15];
          irsz:STRING[4];
              END;
      c:char;
BEGIN
CLRSCR;
REPEAT
  WRITE('Neve: ');READLN(a.nev);
  WRITE('Varos: ');READLN(a.varos);
  WRITE('Ir. szám: ');READLN(a.irsz);
    b:=a;
  WRITELN;WRITELN;WRITELN;
  WRITELN('Az adott adatok',b.nev,' nevü emberről szólnak, aki');
  WRITELN(b.varos,' -on/en/ön/án/ban... lakik,');
  WRITELN('melynek irányítószáma: ',b.irsz,'.');
  WRITELN;
  WRITELN('Még egyet akar kiíratni? (I/N)');
```

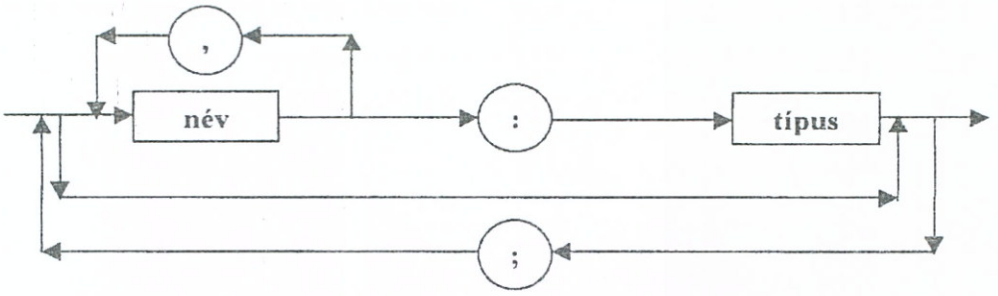
```
UNTIL 'N'=UPCASE(READKEY);
END.
```

2.7.3. Változó rekord

Már a rekordról szóló rész elején szó volt arról, hogy a rekordok mezőinek felsorolása két részből állhat. A szintaxisdiagramból is látható, hogy a mezők felsorolását az állandó résszel kell kezdeni. De az állandó rész után lehet egy változó mezője is a rekordnak. Ennek szintaktikája a következő:



A mezőlistában most is fel kell sorolni a mezőneveket típusaikkal együtt. Szintaktikája:



Nézzünk egy konkrét példát! A szemelyek tömb elemei rekordok. A rekord utolsó mezőjének mezőlistája a szelektormező (no) értékétől függ. A szelektormező is egy mező, amely csak sorszámozott típusú lehet. Példánkban logikai típusú. A deklarált adattípusnál, ha no=TRUE, akkor van egy leanykori nevű mező, amely string[20] típusú karakterlánc. Ha no=FALSE, akkor nincs mező. Ezt ; jelzi a mezőlista helyén.

```
CONST max=40;
VAR szemelyek : ARRAY[1..max] OF
      RECORD
        nev : STRING[20];
        szsz : STRING[11];
        CASE no:BOOLEAN OF
          TRUE:(leanykori:STRING[20]);
          FALSE:;
      END; {RECORD}
```

A CASE szerkezet adja a változó rekord deklarációjának lényegét. A CASE szerkezet végére itt azért nem kell az END, mert a változó rekord esetén a szelektormező (no) minden lehetséges sorszámozott típusú értékét fel kell sorolni. A látható END a RECORD END-je. A változó rekord szelektormezője és a változó mezője is az állandó mezőkhöz hasonlóan használható.

Pl.: `Szemelyek[i]. no`
`Szemelyek[i].leanykori`

Fontos azonban megjegyeznünk azt, hogy:

1. Fizikailag ezek a rekordok is annyi helyet foglalnak le, amennyit Case nélkül lefoglalnának.

2. A deklarációban a mezőfelsorolásokat mindig az állandó mezőkkel kell kezdeni.

3. Mindig csak egy változó mezője lehet a rekordnak, és az csak az utolsó mező lehet.

2.7.4. Kidolgozott feladat

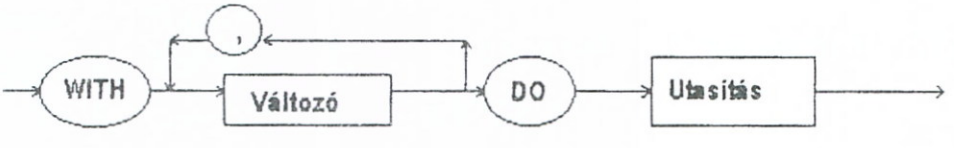
Kérjünk be két dátumot, és számoljuk ki a köztük lévő napok számát, feltételezve azt, hogy az első dátum a későbbi.

```
PROGRAM evek;
USES CRT;
CONST honap:ARRAY[1..12] OF BYTE=(31,28,31,30,31,30,31,31,
        30,31,30,31);
VAR a,b: RECORD
  ev:WORD;
  ho:BYTE;
  nap:BYTE;
    END;
  sub:LONGINT;
  jo:BOOLEAN;
  i,i1,hossz,hossz1:INTEGER;
  h:LONGINT;
  s:STRING;
  c:CHAR;
BEGIN
CLRSCR; WRITE('Kérem az 1. dátumot: '); jo:=FALSE;
REPEAT
  READLN(s);
  IF LENGTH(s)=10 THEN
  BEGIN
```

```
    VAL(COPY(s,1,4),a.ev,i); VAL(COPY(s,6,2),a.ho,i1); i:=i OR i1;
    VAL(COPY(s,9,2),a.nap,i1);
    IF (i OR i1)=0 THEN
    IF (a.ho>0) AND (a.ho<13) AND (a.nap>0) AND
        (a.nap<=honap[a.ho])
      THEN jo:=TRUE;
  END;
    IF NOT jo THEN WRITELN('Hibás a dátum vagy a formátum!');
UNTIL jo;
WRITE('Kérem az 2. dátumot: '); jo:=FALSE;
REPEAT
  READLN(s);
  IF LENGTH(s)=10 THEN
  BEGIN
    VAL(COPY(s,1,4),b.ev,i); VAL(COPY(s,6,2),b.ho,i1);i:=i OR i1;
    VAL(COPY(s,9,2),b.nap,i1);
    IF (i OR i1)=0 THEN
    IF (b.ho>0) AND (b.ho<13) AND (b.nap>0) AND
        (b.nap<=honap[b.ho])
      THEN jo:=TRUE;
    END;
      IF NOT jo THEN WRITELN('Hibás a dátum vagy a formátum!');
  UNTIL jo;
  h:=a.nap; FOR i:=1 TO a.ho-1 DO h:=h+honap[i]; hossz1:=h; h:=b.nap;
  FOR i:=1 TO b.ho-1 DO h:=h+honap[i];
  hossz:=h; sub:=hossz1-hossz+(a.ev-b.ev)*365;
  WRITELN(sub); c:=READKEY;
END.
```

2.7.5. A With utasítás

Talán sikerült is bemutatnunk azt a problémát, amit a következőkben vázolni szeretnénk. Jól látható az, hogy a rekordazonosítót akkor is fel kell tüntetnünk, ha már vagy huszadszor ugyanazzal a rekorddal dolgozunk. Ezen segít a WITH utasítás, melynek szintaktikája a következő:



WITH rekordnév1, ... DO utasítás

A DO utáni utasítás tetszőleges utasítás lehet. A DO utáni utasításon belül a program a WITH alapszó után írt rekorddal dolgozik. Ez azt jelenti, hogy az utasításban nem kell a rekordnevet „ismételgetni" a mezőnevek előtt. Elegendő csak a mezőnevekkel hivatkozni.

A WITH utasítás érvényessége a WITH után felsorolt rekordok aktuális értékeire és csak a DO utáni utasításra vonatkozik.

A rekordnév tömbelem is lehet. Ekkor gyakori hiba, hogy a DO után írt ciklusutasítás magjában megváltozik a tömb indexe, és ezzel a rekord aktuális értéke. A WITH utasításban ez után nem ugyanarra a rekordra hivatkoznánk, mint korábban.

2.7.6. Kidolgozott feladat

Egy tömb elemei rekordok. Készítsünk programot, amely beolvassa a tömb elemeinek rekordmezőit.

```
PROGRAM rekord;
USES CRT;
VAR a,b:  RECORD
          nev:STRING[20];
          varos:STRING[15];
          irsz:STRING[4];
              END;
    c:CHAR;
BEGIN CLRSCR;
REPEAT
  WITH a DO
    BEGIN
      WRITE('Neve: ');READLN(nev);WRITE('Varos: ');READLN(varos);
      WRITE('Ir. szám: ');READLN(irsz);
    END;
  b:=a; WRITELN;WRITELN;WRITELN;
WITH B DO
    BEGIN
      WRITELN('A ',nev,' nevű személy');
    WRITELN(varos,' -on/en/ön/án lakik, ');
    WRITELN('melynek irányítószáma ',irsz,'.');
    END;
  WRITELN; WRITELN('Még egyet akar kiíratni? (I/N)');
UNTIL 'N'=UPCASE(READKEY);;
END.
```

2.7.7. Feladatok

1. Készítsünk olyan programot, amely egy iskolai osztály tanulóinak nevét, magatartás- és szorgalomjegyeit veszi fel. Az adatok beolvasását a névnek adott csillag végjelig folytassuk.

2. Az előző programot egészítsük ki azzal, hogy a program az adatok beolvasása közben számolja a tanulókat. Írja ki az osztálynévsort az adott két jeggyel, majd adja meg a két tantárgy átlagát is.

3. A program dicsérje meg mindazokat a gyermekeket, akiknek mindkét jegyük jeles. Érdemes lenne a programot olyan vizsgálatokkal kiegészíteni, hogy azokat a gyerekeket is felvehessük, akiknek még nincs jegyük valamelyik tárgyból.

4. Állítsunk elő véletlenszerűen egy olyan tömböt, melynek 100 eleme van, és elemei rekordok.

A rekord mezői:

```
SORSZAM:BYTE;
MAGAS:REAL;
SULY:REAL;
```

A testmagasságot a következőképpen generáljuk:

A magasságból következik a súly is:

Írjuk ki a tömb elemeit a sorszám alapján úgy, hogy minden fejléccel ellátott oldalon 20 személy adatai legyenek, és az oldalak kiírását csak billentyű leütésére folytassa.

5. Rendezze úgy a tömb elemeit, hogy a legkisebbek legyenek előbb (növekvő sorrendben). A rendezés után ismét írjuk ki a tömb elemeit az előzőhöz hasonlóan.

6. A program számítsa ki a magasságok és a testsúlyok átlagát és szórását. Állapítsa meg a program, hogy hány egyén van mindkét tulajdonság esetén az átlag ugyanazon oldalán.

7. Határozza meg a program az $Ax^2 + Bx + C = 0$ egyenlet gyökeit a valós számok halmazán. A program minden bemenő adat esetén adjon megfelelő választ. Programjában használja a változó rekordot.

A feladatok megoldásakor elég sokat kell írni. Bizonyos programrészleteket többször is leírunk. Ezen a problémán némileg segítenek az eljárások. A fáradsággal begépelt adatok megőrzése szintén fontos feladat. Ezt a kérdést a fájlok kezelésekor fogjuk megoldani. Az anyag végén visszatérünk e kérdésekre.

2.8. Alprogramok

A legbonyolultabb programokat is felépíthetjük elemi tevékenységekből, sokszorosan összetett szerkezetekkel. Az ilyen programok azonban nem áttekinthetők, nehezen érthető a működésük, és nehézkesen módosíthatók. Világosabbá és rövidebbé tehetjük programunkat, ha bizonyos résztevékenységeket elemi tevékenységként kezelünk ([5]). Ezt a moduláris programozási módszert támogatják a Pascal nyelvben az alprogramok.

Alprogramokat akkor célszerű írni, ha:

— bizonyos tevékenység többször előfordul a programban ugyanúgy vagy más adatokkal,

— a program tevékenységét és kódolását felosztottuk önálló részfeladatokra,

— a túl nagy programot tagolni, olvashatóságát növelni szeretnénk. Túl nagy programrész nehezen áttekinthető, és a srukturált programkód előállítása is nehezebbé válik.

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 tevékenységnek tűnnek.

Az alprogram egy jól meghatározott tevékenység önállóan megfogalmazott 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, amely aktivizálja őket. Egy alprogramot aktivizáló programot gyakran nevezünk meghajtóprogramnak, keretprogramnak vagy hívóprogramnak. Az alprogram tevékenységének befejezése után a hívóprogram mindig az aktuális hívóutasítást közvetlenül követő utasítás végrehajtásával folytatódik.

Azok a programok, amelyek másik programon belül nem használhatók, csak önállóan, főprogramok. Eddig megírt programjaink ilyen főprogramok voltak. Nemcsak főprogram hívhat alprogramokat, hanem alprogram is, sőt alprogram is hívhatja saját magát.

A meghajtóprogram a hívással egy időben információt is átadhat az alprogramnak, amit az alprogram feldolgoz, majd visszatéréskor visszaadhatja a hívóprogramnak. Az információátadást paraméterekkel valósítjuk meg.

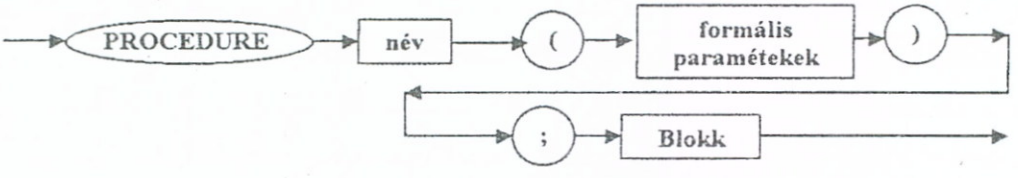
Kétféle alprogramot használhatunk a Pascal programnyelvben: eljárásokat és függvényeket.

2.8.1. Eljárások

Az általánosabb értelemben vett alprogramok az eljárások, feladatuk egy meghatározott — és önmagában is értelmes — tevékenység elvégzése. Két alapvető dolgot kell megvalósítanunk: *hogyan definiáljuk* és *hogyan használjuk* az eljárásokat.

Az alprogramokat a hívóprogram blokkjában, a deklarációs részben definiáljuk. Lapozza fel könyvünk „A blokk" című részét, ahol mi is megmutattuk az eljárások deklarációját. Az eljárások deklarációja hasonló a program leírásához.

A definíció az eljárásfejjel kezdődik. Az eljárásfejet a **procedure** alapszó vezeti be, majd az eljárás neve, utána — ha vannak — zárójelben a paraméterek leírása következik, amelyek az eljárás és a hívóprogram 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 hívóprogramtól a híváskor, és milyen értékeket juttat vissza a főprogramnak, ha elvégezte feladatát.

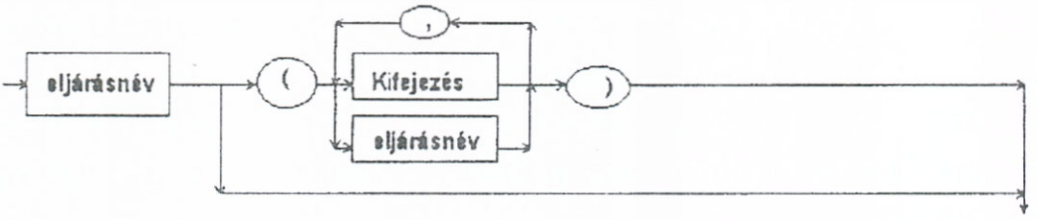


Az eljárásfej után az eljárásblokk következik, amely tartalmazhat deklarációkat, akár alprogram-definíciókat is. A formális paramétereket deklarálni tilos, hiszen azokat specifikáltuk. Begin és end között a tevékenység leírása, az eljárástörzs következik. Az eljárásblokk és a programblokk szintaktikusan teljesen azonos nyelvi szerkezetek, ezért a továbbiakban egyszerűen blokkról beszélünk.

Az eljárásfej ismerete elegendő, hogy az eljárást használni, hívni tudjuk. Az eljárás hívásakor ugyanis az eljárás nevét adjuk meg, és a formális paraméterek helyébe a hívóprogramegység objektumait, az aktuális paramétereket helyettesítjük.

Az eljárások hívása tehát a következő: eljárásnév(aktuális_paraméterek)

Vigyázni kell arra, hogy az aktuális paraméterek számának, sorrendjének és típusának is meg kell egyeznie a formális paraméterekével.



Most lássuk gyakorlatban is az eljárások alkalmazását!

2.8.2. Kidolgozott feladatok

1. Első eljárással megírt programunk neve: proba. A programban deklaráltunk egy eljárást, aminek neve: elso_eljarasunk. Ez az eljárás kiírja a képernyőre, hogy: Hahó. A hívóprogram a szöveget vonalakkal aláhúzza.

```
PROGRAM proba;
PROCEDURE elso_eljarasunk;
BEGIN
  WRITELN('Hahó');
END;
BEGIN
  elso_eljarasunk;
  WRITELN('----------------');
  elso_eljarasunk;
END.
```

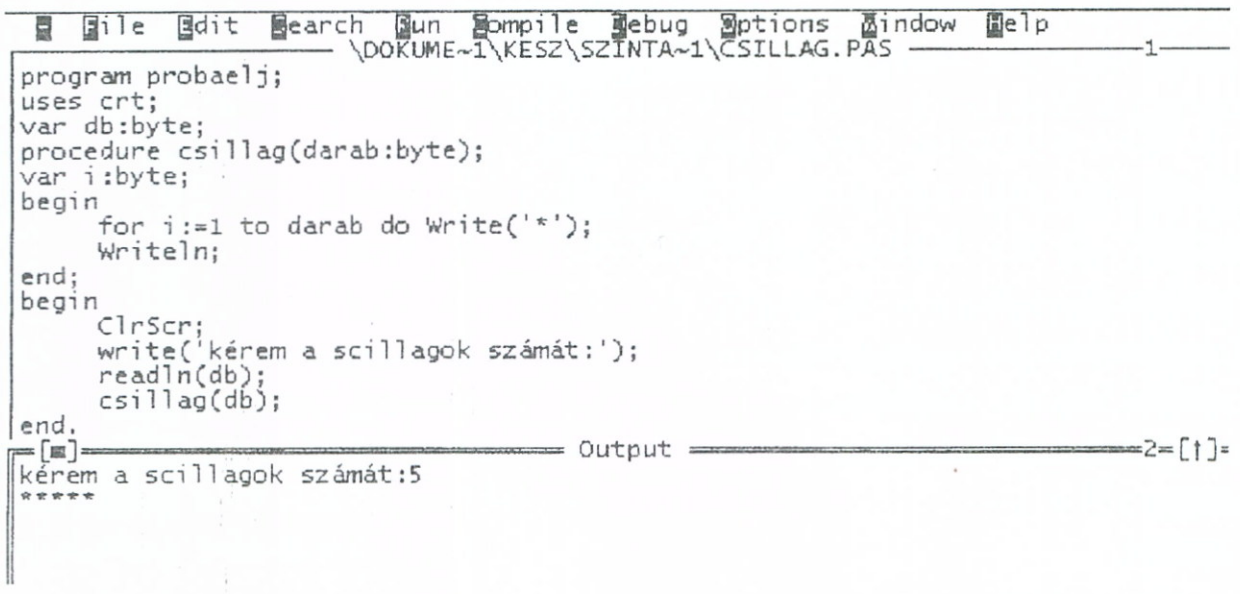
Elemezzük a programot! Nincs szükségképpen minden eljárásnak paramétere. A paraméter nélküli eljárások hívásakor zárójelet is elhagyjuk. Láthatjuk ezt pogamunknál.

Az eljárások nyomon követésére is kiválóan alkalmas a Turbo Pascal nyomkövetője, amit az F7 funkciógombbal használhatunk. Érdemes megnézni egy-egy programot, hogy az utasítások milyen sorrendben hajtódnak végre.

2. Készítsünk programot, amely egész számokat kér, majd — képernyőtörlés után — az adott számnak megfelelő darab csillagot ír ki egy sorba.

```
program probaelj;
uses crt;
var db:byte;
procedure csillag(darab:byte);
  var i:byte;
    begin
      for i:=1 to darab do Write('*');
```

```
        Writeln;
    end;
begin
    ClrScr;
    write('kérem a csillagok számát:');
    readln(db);
    csillag(db);
end.
```



Az eljárásfejet nagyon gondosan kell kialakítani. A paraméterezés határozza meg az eljárás más programokhoz való hozzákapcsolásának lehetőségét.

Az eljárás hívása utasítást jelent. Az utasításoknál már beszéltünk az eljárásutasításokról. Az eljárás hívásakor az aktuális paraméterek számának, sorrendjének és típusának is meg kell egyezni a formális paraméterekével.

3. Készítsünk olyan programot, mely egy dobókocka dobását rajzolja ki a képernyőre úgy, ahogy a dobókocka egy lapjának pettyeit látjuk. A dobókocka lapjának színe azzal a színkóddal legyen kifestve, amennyit dobtunk. Minden dobás után kérjen a program megerősítést a további dobáshoz.

```
program Kockadobas;
uses crt;
type
dbtip=1..6;
var
d:dbtip;
procedure dobas(var db:dbtip);
begin
randomize; db:=random(6)+1;
5:begin
gotoxy(5,4);
write('o');
gotoxy(5,8);
write('o');
gotoxy(20,4);
write('o');
gotoxy(20,8);
write('o');
```

```
end;
procedure rajz(dob:dbtip);
begin
window(25,8,50,18);
textbackground(dob); clrscr;
textcolor(7);
case dob of
1:begin
gotoxy(12,6);
write('o');
end;
2:begin
gotoxy(5,4);
write('o');
gotoxy(20,8);
write('o');
end;
3:begin
gotoxy(5,4);
write('o');
gotoxy(12,6);
write('o');
gotoxy(20,8);
write('o');
end;
4:begin
gotoxy(5,4);
write('o');
gotoxy(5,8);
write('o');
gotoxy(20,4);
write('o');
gotoxy(20,8);
write('o');
end;
gotoxy(12,6);
write('o');
end;
6:begin
gotoxy(5,4);
write('o');
gotoxy(5,8);
write('o');
gotoxy(12,4);
write('o');
gotoxy(12,8);
write('o');
gotoxy(20,4);
write('o');
gotoxy(20,8);
write('o');
end;
end;
end;

begin
clrscr;
repeat
dobas(d);
rajz(d);
until upcase(readkey)='V';
textbackground(0);
textcolor(7);
clrscr;
end.
```

2.8.3. Függvények

A függvények készítése nagyon hasonló az eljárásokéhoz, csak itt mindig van egy visszatérési érték is. Ennek az értéknek a típusát is meg kell adni a függvény deklarálásakor. A függvény értéke egy, a függvény nevével azonos változóban tárolódik. Ezért a függvény deklarálásakor a nevének — zárójel

és paraméterek nélkül — értéket kell kapni, vagyis egy értékadás bal oldalán kell szerepelnie! Pl.: Fuggveny:=x*x;

```
FUNCTION nev(formális_paraméterek) :típus;
        {deklarációs rész}
BEGIN
    :                {a függvény törzse}
nev:=érték;
    :
END
```

A függvény hívása viszont már az aktuális paraméterekkel történik.

```
BEGIN
    :
Y:=függvénynév(aktuális paraméterek):
    :
END.
```

Pl.: Y:=Fuggveny(X). Ügyeljünk arra, hogy a függvény mindig egy értéket képvisel. Ezért mindig olyan környezetben használjuk, amilyen környezetben az értékek lehetnek. Pl.: WRITELN('Terület=',Ter(a)); vagy T:=2*Fv(r)+.5; Nem állhat értékadó utasítás bal oldalán.

2.8.4. Kidolgozott feladat

Ezek után nézzünk egy feladatot. Készítsünk egy olyan függvényt, amelyik kiszámolja A–B értékét. Kérjük be először az adatokat, majd a kiszámított függvényértéket írjuk ki.

```
PROGRAM KIVON;
VAR A,B:REAL;
FUNCTION kivonas(a,b:real):REAL;
BEGIN
  kivonas:=a-b;  {A kivonas név képviseli a visszatérő értéket}
END;
BEGIN
  WRITE('Az egyik szám:');READLN(A);WRITE('A másik:');READLN(B);
  WRITELN('Eredmény=',kivonas(A,B))
END.
```

Programunkban A és B globális változók, a függvényben a és b lokális változók. Mivel a Pascal programnyelv nem tesz különbséget kis- és nagybetűk között, ezért a megkülönböztetés csak nekünk szól. Gondoljuk

át programunk ténykedését a változók láthatósága szempontjából is. Ekkor a függvényünkben (és az eljárásunkban is) lesz egy A és B nevű változó, amiben a két adatot megkapjuk függetlenül attól, hogy híváskor milyen változót adtunk meg. Az elsőként adott szám kerül az A-ba, amit másodiknak adtunk, az a B-be. Ezek szerint, ha nem az a-ból akarjuk kivonni a b-t, hanem fordítva, akkor csak meg kell cserélni a két változót a híváskor: c:=vonas(b,a);

2.8.5. Paraméterátadás

A Pascalban kétféle paraméterátadás létezik az eljárásoknál és függvényeknél. Érték szerinti és cím szerinti. A fenti példánkban **érték szerinti paraméterátadást** láthattunk. Ilyenkor az alprogram csak az **értékparaméter** értékét kapja meg a külvilágtól, és a címét nem. Az alprogram hívásakor az aktuális paraméterek lehetnek konstansok (értékek) és kifejezések is, hiszen ők is képviselnek értéket. Ekkor, ha az alprogramban megváltoztatjuk az aktuális változó értékét, akkor ez a főprogram megfelelő változójára hatástalan lesz. Annak címét az alprogram nem ismeri, hiszen csak az értéken keresztül tartja a kapcsolatot az aktuális paraméter az alprogrammal.

A **cím szerinti paraméterátadáskor** a paraméterként átadott adat memóriacíme adódik át. Ilyenkor ha megváltoztatjuk az alprogramban a változó értékét, akkor az a főprogramban is megváltozik. Cím szerinti átadáskor a paraméter csak változó lehet. Ha cím szerinti átadást akarunk a paraméternél, akkor az alprogram deklarációjánál a paraméterlistában a VAR kulcsszóval jelezzük. A paramétert **változó paraméternek** is mondjuk. Például ha az előző kivonást nem függvénnyel, hanem eljárással akarjuk megvalósítani, akkor a következőt kell tennünk:

```
PROCEDURE vonas(a,b:REAL; VAR c:REAL);
BEGIN
  c:=a-b;
END;
```

És ezek után az eljárás hívása már ismerős: vonas(a,b,c);

Készítsük el a következő programot. Elemezzük ki részletesen a program eredményét:

```
PROGRAM MINTA;
VAR A,B: INTEGER;
PROCEDURE VALTOZIK(X:INTEGER;VAR Y:INTEGER);
BEGIN X:=X+2; Y:=Y+2;
    WRITELN(X:4,Y:4);
END; {procedure}
```

```
BEGIN A:=1; B:=1;
  WRITELN(A:4,B:4);
  NAGYOBB(A,B);
  WRITELN(A:4,B:4);
END.
```

A futtatás eredménye:

```
   1   1
   3   3
   1   3
```

Ha egy paraméter érték szerint hívott, akkor az eljárás vagy függvény hívásakor a paraméter helyén olyan kifejezés állhat, mely értékkel rendelkezik. Ha a paraméter cím szerint hívott, akkor csak változó állhat aktuális paraméterként ott, hiszen csak ennek van címe. Ha egy eljárás paramétere csak befogadja a külső értékeket, akkor azt érték szerint hívottnak kell tekinteni. De amikor az eljárásban megváltoztatott változó értékeit a külvilágnak ki is akarjuk adni, akkor ezeket a paramétereket cím szerint hívottnak kell specifikálni. A cím szerinti paraméter hívásánál külső érték befogadása is történhet!

2.8.6. Globális és lokális változók

Mint látjuk, az alprogramokban újabb változókat is deklarálhatunk. Ilyenkor felülbíráljuk a főprogramban deklaráltakat, de amint elhagyjuk az alprogramunkat, ezek a változók tovább nem láthatók. A főprogrambeli változókat globális változóknak nevezzük, az alprogramokban deklaráltakat pedig lokálisaknak. A változók láthatóságának az a magyarázata, hogy a főprogram változói a memória Dataszegmensében vannak, és az egész program futása alatt élnek. Az alprogramokban deklarált változók a Stackszegmensben keletkeznek, akkor, amikor az eljárást meghívjuk. Életük során az ugyanolyan nevű globális változót eltakarják, az eljárás elhagyásakor viszont megszűnnek. Ezért fontos tudnunk a változók hatáskörét. A Pascal nyelvre érvényes, hogy azonos szinten nem, de más szinteken lehetőség van a változók felüldefiniálásra. A következő programvázlaton összefoglaljuk, hogy egy programban és alprogramjaiban mely változókat látjuk, azaz hol, melyik változóra hivatkozhatunk.

A P főprogramban deklarált változókat mindenhol láthatjuk, ha megjelöljük, hogy a főprogram globális változójáról van szó. A jelölés formája: P.I vagy P.J. A változók hatáskörének kérdésével összefügg az eljárások és függvények hatáskörének kérdése is. Mivel az eljárások és függvények változói a Stackszegmensben vannak, ezért az egyes eljárások is onnan és addig alkalmazhatók, amíg láthatók annak kellékei.

Ezután tételesen összefoglaljuk azokat az alapeseteket, amikor az eljárásokat/függvényeket hívhatjuk. Jelölésünkben a következő vázlatra hivatkozunk.

```
Program P;
  Var I,J:Integer;

  Procedure A;
    Var I,K:Integer;
    Procedure B;
      Var I:Integer;
        Begin
        A B blokkja. Ismert a B-ben deklarált I, az
        A-ban deklarált K és a P-ben deklarált J, valamint P.I
        End;

      Begin
      Az A eljárás blokkja. Ismert az A-ban deklarált I és K,
      valamint a P-ben deklarált J és a P.I
      End;

  Procedure C:
    Var I:Integer;
    Begin
    A C blokkja. Ismert a C-ben deklarált I,
    valamint a P-ben deklarált J és P.I
    End;

Begin
Ez a P blokkja. Használhatók a Blokkban deklarált változók. Ezek az egész
programra nézve globálisak. A többi lokális változó már nem él.
End.
```

Egy eljárás blokkjából hívható saját maga. Ezt nevezzük rekurzív hívásnak.

A B eljárás blokkjában hívhatja a A eljárást, mert B deklarálása is az A deklarációjában van.

Egy eljárás ismer — a deklarálás helyétől lefelé — minden olyan azonosítót, így eljárást is, amellyel azonos deklarációs szinten van, azaz deklaráció szempontjából mellérendeltek.

Példánkban C ismeri az A eljárást.

Az A eljárás blokkjában ismeri mindazokat az eljárásokat, amelyeket az A eljárásban deklaráltunk. Így A-ból hívható B, és természetesen P-ből hívható A és C is.

2.8.7. Forward opció

Már láttuk, hogy egy eljárásból (vagy függvényből) csak akkor hívhatunk meg egy másik azonos szinten deklarált eljárást, ha azt már előtte

deklaráltuk. Előfordulhat azonban, hogy két eljárás egymást szeretné meghívni. Ilyenkor segít a FORWARD kulcsszó. Az első eljárásnak — ezt hívja a második — csak a fejét adjuk meg, kifejteni majd csak a másik kifejtése után fogjuk. Így szervezve a deklarációt már nem lesz hibajelzés, ha a második eljárás hívja az elsőt.

```
PROCEDURE egyp;FORWARD;
```

```
PROCEDURE masp;
BEGIN
...
  egyp;   {meghívjuk az egyp-t}
  ...
END;
```

```
PROCEDURE egyp;
BEGIN
  ...
  masp;
END;
```

2.8.8. Rekurzió

Az alprogramok hívásának különös esete az, amikor az eljárások vagy függvények önmagukat hívják — közvetlenül vagy közvetve. Ezt nevezzük rekurziónak. A rekurzív eljárás/függvény leírásakor két dologról kell gondoskodni.

Gondoskodni kell arról, hogy hogyan kezdődik az önmagát hívó eljárás.

Meg kell adni azt az algoritmust, hogy az előző esetből hogyan számítható a mostani eset. Az öröklődés szabályáról van itt szó. Elve a teljes indukciós bizonyításra hasonlít. Ne felejtse, hogy minden eljárás/függvény hívásakor lokális változók keletkeznek a veremben. Ez így van a rekurziónál is. Visszaszedésük a kezdőlépés elérésekor kezdődik. Túl hosszú rekurziónál a stack el is fogyhat. Méretét az Options/Memory size Stack size paraméterében is növelhetjük.

2.8.9. Kidolgozott feladatok

1. A program kérjen egy egész számot, majd törölje le a képernyőt, és tegyen annyi $\star$ jelet, amennyit az imént mondtunk. Most rekurzív eljárással fogalmazzuk meg az algoritmust.

```
Program Rekcsillag;
Uses Crt;
```

```
Var Db:Byte;
Procedure Csillagok(N:Byte);
Begin
  if N=1 then Begin ClrScr; gotoxy(1,10); Write('*'); End
    else
      Begin Csillagok(N-1);
          Write('*');
      End;
End; Procedure
Begin
  ClrScr;
  Write('Hány csillagot rajzoljak?');Readln(Db);
  Csillagok(Db);
  Readln
End.
```

Azt javasoljuk az olvasónak, hogy kövesse végig fejben a program futását Db=3 esetében.

2. A rekurzió alkalmazása függvényeknél is gyakori. Mint tudjuk, minden függvény rendelkezik visszatérési értékkel. Ha a függvényt mi készítjük, akkor erről nekünk kell gondoskodnunk. Mivel a függvény azonosítója az értékadó utasítás bal oldalán kell, hogy szerepeljen, ezért különösen érdekes lesz ez a sor a rekurzív függvényeknél. A feladat most az, hogy számítsa ki a program N! értékét. Most rekurzív függvényt készítsünk!

```
Program Rekfakt;
Uses Crt;
Var N:Longint;
Function Fakt(M:Longint): Longint;
Begin
  if M=0 then Fakt:=1 else Fakt:=Fakt(M-1)*M;
End;
Begin
  ClrScr;
  ReadLn(N);
  Write(N,'!=',Fakt(N); Readln;
End.
```

3. A rekurzív sorozatok klasszikus példája a matematikában is a Fibonacci-számsorozat tagjainak kiszámítása. Oldjuk meg mi is a feladatot. Olvassunk be egy egész számot, majd írja ki a program a Fibonacci-sorozat kért tagját.

```
PROGRAM fibi;
VAR sorszam:INTEGER;
FUNCTION fibker(melyik:INTEGER):LONGINT;
BEGIN
  IF melyik<2 THEN fibker:=1
  ELSE fibker:=fibker(melyik-1)+fibker(melyik-2);
END;
BEGIN
  WRITELN('Hányadik számot keresi:');READLN(sorszam);
  WRITELN('A sorozat ',sorszam,'. eleme: ',fibi(sorszam));
END.
```

Kérjük, hogy elemezze a programot, majd próbálja ki!

2.8.10. Feladatok

Ezek után azt javasoljuk, hogy a gyorsabb és hatékonyabb gyakorlás érdekében lapozza fel az eddig javasolt és megoldott feladatokat, és alakítsa át azokat úgy, hogy most eljárásokkal és függvényekkel dolgozzon. A korábban javasolt feladatokat most nem ismételjük meg.

1. Készítsen olyan programot, mely az $Ax^2 + Bx + C = 0$ egyenletet oldja meg a valós számok halmazán, tetszőleges bemenő együtthatók esetén. Minden adatsor esetén helyes reagálás legyen.

2. Olvasson be — * végjelig — számokat, majd határozza meg azok átlagát. Az összegzést készítse el eljárással, függvénnyel rekurzió nélkül és rekurzióval is.

3. Számítsa ki $\sqrt{x\sqrt{x\sqrt{x\ldots}}}$ értékét, ha adott az x, valamint az x-ek száma.

4. Számítsa ki $\sqrt{x+\sqrt{x+\sqrt{x+\cdots}}}$ értékét, ha adott az x értéke és az x-ek száma.

5. Adott x érték és az x-ek számának ismeretében számolja ki az

$$\cfrac{1}{x+\cfrac{1}{x+\cfrac{1}{x+\cdots}}}$$

kifejezés értékét.

6. Határozza meg $\sqrt{A}$ értékét adott A esetén az

$$x_{i+1} = \frac{x_i + \frac{A}{x_i}}{2}$$

rekurzív képlet alapján, megadott pontossággal. Kezdje az iterációt $x_1 = 1$ értékkel.

7. Az előző feladathoz hasonlóan számítsa ki $\sqrt[k]{A}$ értékét bemenő pontossággal. Az iterációs képlet:

$$x_{i+1} = \frac{(k-1)x_i + \frac{A}{x_i^{k-1}}}{k}.$$

8. A program generáljon véletlenszerűen 25 egyjegyű egész számot. Írja ki a sorozatot egy sorba. Rendezze a számokat csökkenő sorrendbe, és írja ki azokat, majd növekvő sorrendben is írja ki a számokat. Dolgozzon eljárásokkal.

9. Készítsen programot, mely bemenő két számról eldönti, hogy melyik a nagyobb. Ne feltételes utasítással, hanem a következő függvényekkel dolgozzon: NAGYOBB(A,B)=(A+B)/2+ABS(A−B)/2, KISEBB(A,B)= (A+B)/2−ABS(A−B)/2.

10. A program generáljon legalább 100 véletlen számot $[0, 99]$ intervallumban. Keresse meg a legkisebb és a legnagyobb számot közöttük, majd írja ki az indexét és magát a számot. Csinálja meg a programot feltételes utasítással és az előző feladatban közölt függvénnyel is.

11. Készítsen olyan programot, mely tiszta képernyőre két véletlen számot kiír. Az egyik az $1, 2, \ldots, 30$ számok közül, a másik az $1, 2, \ldots, 20$ közül kerüljön ki. A program addig szimulálja a tételhúzást, amíg Esc billentyűt nem ütünk. Könnyen lehessen a határokat változtatni.

12. Segítsen a program tippelni 1, 2, X jelekkel a totózónak, de úgy, hogy minden tipp előtt mondjuk meg, hogy a játékos hány százalék esélyt ad az 1 és 2 tippeknek.

13. Készítsen programot, mely egész számot konvertál a alapú számrendszerből b alapú számrendszerbe.

14. Készítsen olyan programot, amivel egy beolvasott szöveg fényújságszerűen jelenik meg.

15. Olvasson be a program egy tetszőleges hosszú szöveget, és közben készítsen betűstatisztikát. A szöveg végét az F6 billentyű jelenti. A beolvasás után a statisztika is jelenjen meg.

16. Olvasson be egy stringet. Egy menüből válasszuk ki a szöveg újbóli kiírásának módját. A kiírás módja lehet a betűk közötti 0, 1, 2, 3 szóköz (space).

17. Programja törölje le a képernyőt, majd kérjen be egy jelszót, végén Enter billentyűvel. A jelszó karakterei helyett a képernyőn csillagok jelenjenek meg. Hibás jelszó esetén a jelzés után kérjen újat. A helyes jelszó esetén ér véget a program.

18. Kérjen a program a billentyűzetről egy természetes számot, de csak akkor jelenítse meg a leütött karaktereket, ha az Enterrel lezárt karaktersorozat valóban természetes szám. Ha nem ilyen az adat, akkor az ne is jelenjen meg a képernyőn, és változatlan helyen kérjen újabb számot.

19. Készítsen A_ad_N(A:real; N:byte):real függvényt, mely kiszámolja A^N értékét minden egész kitevő esetén. Készítsen a függvényt működtető keretprogramot is. (Minden adat esetén helyesen reagáljon a program.)

20. Határozza meg a program egy mértani sorozat első n tagjának összegét. Beolvasott adatok: a_1, q, n. A függvényt rekurzióval készítse:

$$S_1 := a_1, \qquad S_i := S_{i-1} + a_1 q^{i-1}.$$

21. Egy 100 elemű vektorba generáljon véletlen számokat, melyek a $[0, 9]$ intervallumban vannak. Számítsa ki az elemek átlagát ($\overline{x}$) és szórását (D).

$$\overline{x} = \sum_{i=1}^{100} x_i \qquad D^2 = \sum_{i=1}^{100} x_i^2 - n\overline{x}^2.$$

22. Generáljon egy legalább 10000 elemű tömböt, melynek elemei real típusú számok. A tömb elemeit rendezze növekvő sorrendbe különböző rendezési algoritmusokkal. Minden rendezéskor mérje az időt. Az egyes módszerekre kapott időket foglalja táblázatba. Az idők mérésére használja a Dos Unit **GetTime(var ora,perc,mp,szmp: word)** eljárást.

23. Készítsen most olyan programot, amelyben a rendezetlen tömbben való keresés, a rendezett tömbben való lineáris keresés és a logaritmikus keresés felhasznált időit foglalja táblázatba. Az időmérést különböző keresendő adatokra is lehessen alkalmazni. Használja fel az előző program rendezetlen és rendezett tömbjeit.

24. Határozza meg n elem k-ad osztályú ismétlés nélküli kombinációinak számát:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Alkalmazza az $n!$ kiszámításánál tanult függvényt.

26. Számítsa ki $\binom{n}{k}$ értékét úgy, hogy a tört számlálójának és nevezőjének — egyszerűsítés utáni — kiszámítására készítsen eljárásokat:

$$\binom{n}{k} = \frac{(n-k+1)(n-k+2)\cdots n}{1 \cdot 2 \cdots k}.$$

27. Számítsa ki $\binom{n}{k}$ értékét rekurzióval a következő képlet alapján:

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

28. Készítsen olyan programot, amivel kiírja a Pascal-háromszög — képernyőn elférő — részét.

29. Készítsen olyan programot, amivel egy adott személyi szám helyességét ellenőrizheti. Ha True a logikai függvény értéke, akkor írja ki a személy nemét és születési dátumát is. A személyi számot számként kezelje.

30. Oldja meg az előző problémát úgy, hogy most a személyi számot stringként kezeli. A program minden bemenő adatra helyesen válaszoljon, és ne akadjon ki.

31. Készítsen egy menürendszert. A megjelenő menüben a kurzormozgató billentyűkkel lehessen mozogni. A menüpontok számokat tartalmazzanak szövegesen. Kiválasztva, majd Entert ütve jelenjen meg a választott szám számjegy alakjában a képernyő valamely pontján. A vége menüpontot választva van vége a programnak.

32. Egy 20 sorból és 80 oszlopból álló mátrix elemeit töltse fel 0 vagy 1 számokkal. Írja ki a mátrixot a képernyőre, és a program várjon egy billentyű leütésére. Ahol 0 számjegy áll, ott egy sejt található, a másik esetben nincs sejt. Elpusztul az a sejt, amelynek nincs egy szomszédja sem, vagy háromnál több szomszédja van. (Egy sejtnek legfeljebb 8 szomszédja lehet.) Rajzolja ki ismét a mátrixot. Azon a helyen, ahol nem volt sejt, de a helynek 1, 2 vagy 3 szomszédja van, egy sejt keletkezik. Megint rajzolja ki a mátrixot. A fejlődési folyamat kirajzolását a szóköz billentyű leütéséig végezze a program.

33. Egy 20 sorból és 80 oszlopból álló mátrix elemeit töltse fel véletlenszerűen a $[0, 9]$ intervallumba eső számokkal. Írja ki a mátrixot a képernyőre. Jelölje zöld színű számmal azokat a pozíciókat, amelyek gödröt jelentenek. Egy helyen akkor van gödör, ha az ott lévő szám nem nagyobb egyetlen szomszéd számnál sem.

34. Valósítsa meg a torpedójátékot úgy, hogy a számítógép készíti a hajókat és adminisztrálja a játékot. A játékos a lövést koordinátáival adja meg.

2.9. Halmaztípus

A halmaz azonos tulajdonságú elemek összessége. A matematikai halmaz mintájára a Pascal nyelvben is létrehoztak halmaztípust. Sajnos csak kis tartományokban számolhatunk vele, és a műveletek száma is kevés. A számítógép tulajdonságaiból következik, hogy csak véges halmazokat használhatunk. A Turbo Pascalban a halmaz elemeinek száma legfeljebb 256 lehet, és elemei csak sorszámozott típusúak lehetnek. A halmaz fogalmából következik, hogy egy elemről csak azt tudjuk, hogy benne van-e a halmazban vagy nincs. Arról, hogy a halmaz hányadik eleme, vagy hogy egyszer vagy többször tettük bele a halmazba, nem kaphatunk információt.

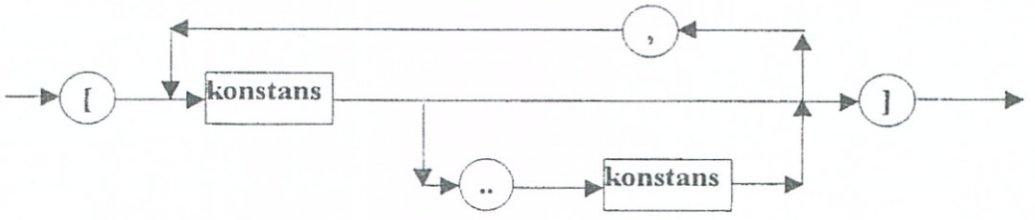
A Pascal nyelvben a halmazok definiálásához a set alapszó szükséges, és meg kell adnunk a halmaz elemeinek típusát:

Az alaptípus csak olyan sorszámozott típus lehet, amelyben az elemek sorszáma a $[0, 255]$ intervallumban van. Különben hibaüzenetet kapunk. Pl.:

```
TYPE Htip = SET OF 1..6;
Betutip = SET OF 'a'..'z'
VAR kocka, dobas :Htip;
kisbetu :Betutip;
halmaz :SET OF BYTE;
```

Megjegyezzük, hogy egy halmaz jóval kevesebb helyet foglal a memóriában, mint a tömb.

A halmazkonstanst úgy adhatjuk meg, hogy szögletes zárójelben felsoroljuk az alaptípusú konstansokat (konstans kifejezéseket).

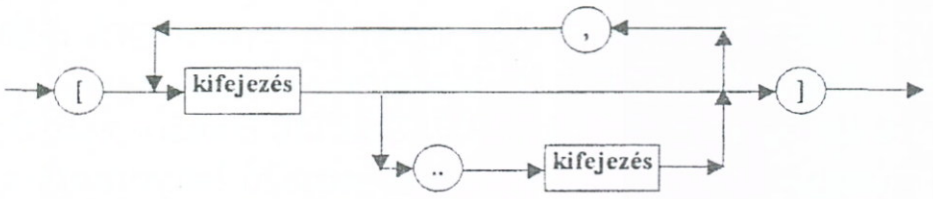


Pl.: [0,2,4,6,8], [0..9], ['0'..'9','A'..'Z','a'..'z'], [], [2*K,3*K], ahol K konstanst jelent. CONST Ures=[]; Paros = [0,2,4,6,8,]; LOTTO = 1..90; TOTO = ['1','2','x'];

Hasonlóan adhatunk meg tipizált konstansoknak kezdőértéket is.

CONST Magyar : SET OF Char = ['A'..'Z','a'..'z','Á','á','É','é','Í','í', 'Ó','ó','Ö','ö', Ő,'ő','Ú','ú','Ü','ü','Ű,'ű'];

A halmazkonstruktornál a felsorolt értékeket a halmaz típusának megfelelő kifejezések is adhatják ([1]):



[2*i], [1..N], [Kar]

A halmaztípusok körében is értelmezve van az értékadó utasítás, amennyiben teljesül az értékadás-kompatibilitás. Pl. egeszh:=[4*12]; egy értékadás. Ez azt jelenti, hogy felvettük a halmazba a 48 értéket. Egyszerre több elemet is betehetünk a halmazba: egeszh:=[12,23,4];.

2.9.1. Műveletek

A halmazok körében műveleteket is végezhetünk, ha a halmazok azonos típusúak. Az alábbi táblázatban összefoglaljuk a matematikai és a Pascal nyelvbeli jelöléseket.

Matematikai jelölés	Turbo Pascal jelölés
$A \cap B$	A*B
$A \cup B$	A+B
$A \setminus B$	A-B
$A = B$	A=B
$A \neq B$	A<>B
$A \subseteq B$	A<=B
$A \supseteq B$	A>=B
$x \in A$	x IN A

Metszet: A * B olyan halmaz, amelynek elemei mindkét halmaznak elemei.

Unió: A + B olyan halmaz, amelynek elemei A-nak vagy a B-nek elemei.

Különbség: A-B olyan halmaz, amelynek elemei elemei A-nak, de nem elemei B-nek.

A matematikában megszokott relációs műveleteket is végezhetünk.

Egyenlő: Az A = B reláció értéke akkor és csak akkor igaz, ha a két halmaz elemei ugyanazok.

Nem egyenlő: Az A <> B reláció értéke akkor és csak akkor igaz, ha a két halmaz elemei nem egyeznek meg.

Részhalmaz: A<=B csak akkor igaz, ha A minden eleme a B halmaznak is eleme.

Részhalmaz: A>=B csak akkor igaz, ha B minden eleme az A halmaznak is eleme.

A matematikában szokásos eleme vizsgálatra az IN operátor szolgál.

x IN A: x eleme A-nak értéke igaz, ha x szerepel A halmaz aktuális értékei között. Használa: Elem IN Halmaz.

Pl.: IF 12 IN egeszh THEN WRITELN('Benne van!');

Ha azt szeretnénk tudni, hogy a felhasználó lenyomott-e bizonyos gombokat, de az nem szükséges, hogy közülük melyiket, akkor a c változóba beolvassuk a lenyomott billentyű kódját és IF c IN ['1','2','3',#27,#13] THEN... feltétellel tesztelhetjük.
(Egyszerűbben: IF Readkey IN ['1','2','3',#27,#13] THEN...)

A nem eleme kérdést is eldönthetjük az IN operátor segítségével. NOT(x IN A) értéke ugyanis akkor igaz, ha x nem eleme A-nak.

Egy halmaztípusú kifejezés kiértékelésekor a műveleti prioritás hasonló az aritmetikai kifejezéseknél tanultakhoz.

1. Multiplikatív művelet: *
2. Additív műveletek: +, -
3. Relációs műveletek: =, <>, <=, >=, IN

Azonos műveleti prioritás esetén most is a balról jobbra szabály érvényes. Zárójelezéssel a műveletek elvégzésének sorrendjét itt is módosíthatjuk.

2.9.2. Kidolgozott feladat

A következő feladat prímszámokat keres Eratoszthenész módszerével 2-től 255-ig. Ezt úgy végezzük, hogy egy halmazba növekvő sorrendben beletesszük 2-től 255-ig az egész számokat. Kiválasztjuk a halmaz első elemét, és kiírjuk. Ez biztosan prím. Ezután a halmazból kivesszük ennek a számnak a többszöröseit, majd megkeressük a következő halmazbeli számot, és kiírjuk. Többszöröseit kivesszük... Ezt addig csináljuk, amíg el nem érünk a halmaz végére.

```
program szita;
uses crt;
type szamtip =set of 2..255
const maxertek =255;
    szita:szamtip =[2..255];
var szam,i:byte;
begin ClrScr
  writeln('MEDDIG?[2..255]=',maxertek);
  readln(maxertek);
  writeln('PRÍMSZÁMOK ',maxertek,'-IG');
```

```
    For szam:=2 To maxertek Do
    If szam IN szita Then
      Begin
        write(szam:10);
        For i:=2 To maxertek div szam Do szita:=szita-[i*szam];
      End;
  readln;
end.
```

2.9.3. Feladatok

1. Készítsen programot, mely megszámolja egy bemenő szóban lévő magánhangzókat.

2. Döntse el a program, hogy számot vagy betűt ütöttünk le.

3. Egy halmazban tároljuk a 3-mal osztható számokat a lehetséges intervallumon belül. Egy beolvasott számról döntse el a program, hogy osztható-e hárommal.

4. Az előző feladathoz hasonlóan állapítsa meg egy bemenő pozitív egész számról, hogy osztható-e héttel.

5. A halmazok műveleteinek felhasználásával készítsen olyan programot, mely eldönti egy számról, hogy igaz-e a következő állítás: osztható 3-mal és páros, de nem osztható 7-tel.

6. A billentyűzetről töltsön fel két halmazt karakterekkel, majd menüből választhatóan adja meg a két halmaz metszetét vagy unióját.

7. Olvassunk be két számhalmazt billentyűzetről. Menüből választhasson a használó, hogy a két halmaz metszetét, unióját vagy különbségét kívánja látni.

8. Határozza meg két számhalmaz Descartes-szorzatát.

9. Írja ki a program a lehetséges határon belül a hattal osztható számokat.

10. Egy bemenő stringben hány olyan karakter van, amelyik kisbetű, nagybetű, nem betű?

11. Egy vizsgán három kérdésre kell válaszolni. A válaszokat $1, 2, \ldots, 5$ jegyekkel minősítik. Az egész vizsga eredménye: „kitűnő", ha minden részjegy 5; „jól megfelelt", ha legalább egy 5 van és nincs 1 és 2; „nem felelt meg", ha nincs 5 és van 1; egyébként „megfelelt".

2.10. Unitok

Az egység (UNIT) készítése és alkalmazása a Turbo Pascalban a moduláris programozás része. Arról van szó, hogy a lefordított kód mérete maximum egy szegmensnyi, azaz 64 Kbyte lehet. Ez a korlát problémák megoldásakor igen gyakran gondként jelentkezett, ugyanis 64 Kbyte-nál nagyobb programot csak külső programhívással vagy overlay technikával futtathattunk. A Turbo Pascal 4.0-ás verziójától lehetőségünk van arra, hogy forrásnyelvű programjainkban olyan egységeket is meghívjunk forrásnyelvű programunkban, amelyek már lefordított állapotban vannak a Unit könyvtárban ([16]). A használatuk szempontjából nézve a Unit nem más, mint egy olyan gyűjtemény, amely konstansokat, típusokat, változókat, eljárásokat, függvényeket tartalmaz úgy, hogy mindezek az objektumok abban a programban globálisak, amelyből a Unitot meghívjuk. A Unitok bevezetése miatt bővült a Turbo Pascal deklarációja egy új típusú deklarációval. Közvetlenül a programfej után, a Uses szó után deklaráljuk a használni kívánt Unitokat, nevük felsorolásával. Igen fontos tulajdonsága a Unitoknak, hogy alkalmazásukkal a lefordított program mérete nagyobb lehet 64 Kbyte-nál. Ennek felső határát az operációs rendszer, illetve a gép memóriájának nagysága szabja meg. Így a lefordított program méretének felső határa általában 640 Kbyte.

2.10.1. Unitkönyvtár

A Turbo Pascal 4.0-ba egy óriási eszközt építettek be, ami jelentősen megkönnyítette a programozói munkát. Ez a lehetőség a programkönyvtárak létrehozását jelentette (továbbiakban csak Unitok). A Pascal nyelvben viszonylag kevés a beépített utasítás, eljárás és a függvény. Ezek számát azonban a Unitok segítségével növelték, illetve mi magunk is növelhetjük. A UNIT (egység) nem más, mint Konstansok, Változók, Típusok, Eljárások, Függvények és iniciáló programrész gyűjteménye, amelyeket programjainkban használhatunk. A Turbo Pascal nyelvhez több hasznos Unitot szállítanak. Ezek a standard Unitok. Ezek közül a legfontosabbak:

SYSTEM: A standard eljárásokat, függvényeket és sok hasznos könyvtárkezelő eljárást tartalmaz. Minden programhoz automatikusan hozzászerkesztődik, ezért külön nem kell meghívni.

DOS: A DOS alkalmazások változóit és sok DOS parancsot megvalósító eljárást tartalmaz. Rendszerközeli programozást tesz lehetővé.

CRT: A karakteres képernyő, a billentyűzet és a hangadás kezelését segíti.

PRINTER: A nyomtatást támogatja.

OVERLAY: Az átlapolási technikát támogatja.

GRAPH: Grafikus modulokat tartalmaz.

GRAPH3: Teknőcgrafika. Kompatibilitás miatt maradt meg e régi grafikai rész.

TURBO3: Néhány megszűnt parancsot tartalmaz.

Ezek közül az első 5 Unit a **TURBO.TPL** fájlban található. Ez a könyvtárfájl a Turbo Pascal indításakor betöltődik a memóriába, és így gyors elérést tesz lehetővé. A többi standard Unit a UNIT könyvtárban található.

Unit	Code	Data	Uses
System	22 757	702	
Overlay	1 859	26	System
Crt	1 567	20	System
Dos	1 591	6	System
Printer	54	256	System

A TURBO.TPL-hez kapcsolódó külső program a **TPUMOVER.EXE** segítségével tudunk Unitot bevinni, törölni és kivenni a TURBO.TPL-ből.

Használata: TPUMOVER [[TURBO.TPL[Parancs]]]

TPUMOVER	Tájékoztatást ad a program használatáról.
TPUMOVER TURBO.TPL	Listát ad a programkönyvtárról.
TPUMOVER TURBO.TPL Parancs	**Parancs.**

Parancs lehet:

+unitnév	Hozzáadás a TPL-hez.
−unitnév	Törlés a TPL-ből.
*unitnév	Kimásolás a TPL-ből.

Pl.: TPUMOVER TURBO.TPL+UJ.TPU.

Készített Unitjaink használatára a saját készítésű Unitok után visszatérünk.

Ha szeretnénk használni egy Unitban lévő információkat, akkor a Unitot meg kell hívni programunk deklarációs része előtt a következő módon: Uses **unitnév1,unitnév2...**; Pl.: Uses Crt, Graph;

Ezután a Unitban lévő összes változót, konstanst, típust, eljárást és függvényt használhatjuk ugyanúgy, mintha a saját programunkban deklaráltuk volna őket.

2.10.2. Saját készítésű Unit

A standard Unitok mintájára mi magunk is készíthetünk Unitokat, amikben a leggyakrabban használtakat gyűjthetjük össze. Érdemes minden nagyobb eljárást vagy függvényt eltenni, mert még később is szükség lehet rá. Ha úgy döntünk, hogy készítünk saját gyártmányú Unitokat, akkor egy Unitba az egymáshoz kapcsolódó deklarációkat és alprogramokat tegyük. Végül ne feledkezzünk meg a Unit leírásáról. A dokumentáció segít később is felidézni a Unit tartalmát.

Akkor lássuk, hogy készíthetünk Unitot! Először készítsük el a UNIT forrásnyelvű fájlját. A Unitnak van egy forrás fájlja — ami ugyanúgy, mint Pascal programjaink, egy szöveg — amely .PAS kiterjesztésű fájl. A fájl felépítése:

UNITnév; Itt adunk egy nevet a Unitunknak. Ezzel tudunk rá hivatkozni a USES alapszónál. Fontos, hogy ilyen néven is mentsük majd el.

INTERFACE Globális deklarációk. Azokat a neveket deklarálhatjuk itt, amelyek a Unitból, és az őt használó programból, illetve más egységből is elérhetők. Kezdődhet Unitok hívásával, majd konstansok, típusok, változók, alprogramok deklarálása következhet. Az eljárásoknak és a függvényeknek csak a feje kerül ide, a többi a implementációs részbe kell.

IMPLEMENTATION Ide kerülnek azok az elemek, amelyeket csak a Unitban szeretnénk látni. Itt fejtjük ki az eljárásokat és a függvényeket. Most is használható a USES alapszó, de ha itt hivatkozunk egy Unitra, akkor annak deklarációit nem használhatjuk az INTERFACE-ben. Ide csak akkor érdemes tenni, ha két Unit egymást hívja.

BEGIN Ez a Unit inicializáló része. Itt beállíthatjuk a kezdőértékeket. Ez a rész a főprogram előtt fog lefutni. Ha nincs semmi, amit ide írnánk, akkor teljesen elmaradhat, még a BEGIN-t sem kell kiírni, de a záró END.-et igen.

END.

Ez a saját készítésű Unitoknak a forrása. Az INTERFACE/IMPLEMENTATION közötti különbség jobban szemléltethető egy példával. Ha szeretnénk létrehozni egy típust, és azt szeretnénk használni majd a főprogramban is, akkor azt az INTERFACE után kell használni. Így a főprogramban lévő alprogramok is és a Unitunkban lévő alprogramok is használhatják. Ha a típust az IMPLEMENTATION után írjuk, akkor csak a Unitban használhatjuk. Így elkerülhetjük az esetleges átdefiniálást. Megengedett egy ugyanolyan típus egy másik Unitban is. A fordító nem jelez hibát. Ilyen esetben a főprogramunk mindig a később hívott Unitban lévőt veszi figyelembe.

Ha készen vagyunk a forrásnyelvű kóddal, akkor le kell fordítanunk lemezre a Compile menüponttal C/D opcióval, és így egy TPU kiterjesztésű fájlt kapunk. A lefordítás el is hagyható, mert a Pascal egy új program fordításakor megnézi a hozzá kapcsolódó Unitokat, és azokból — amelyek változtak az utolsó fordítás után, vagy még nem voltak lefordítva — elkészíti a lefordított UNIT-ot, az új .TPU-t. A forrásnyelvű Unitot is őrizzük meg.

Az új, lefordított Unitot a főprogram számára elérhető helyen kell elhelyezni. Elérhető a Unit, ha:

— ugyanott van, ahol a főprogram,
— a TURBO.TPL egységkönyvtárban van, vagy
— csinálunk egy könyvtárat, amiben a Unitokat összegyűjtjük, és az Option/Directories/Unit Directories mezőjében ezt a könyvtárat jelöljük meg.

Ezután próbáljuk is ki a Unitot egy program segítségével.

2.10.3. Kidolgozott feladatok

Most már minden szabályt ismerünk a Unitok készítéséhez. Nézzünk néhány példát.

1. Első rövidke Unitunk összevonja a GOTOXY és a WRITELN eljárások használatát. Így egy stringet a megadott pozícióba egyetlen eljárással írhatunk:

```
UNIT elso;
INTERFACE
USES CRT;          {Innen vesszük majd a GOTOXY-t}
PROCEDURE writexy(x,y:BYTE;s:STRING);
                   {A szabályoknak megfelelően deklaráljuk
                   az eljárás fejét}
IMPLEMENTATION
PROCEDURE writexy(x,y:BYTE;s:STRING);
                   {Megismételtük a fejet. A paraméterlistát
                   elhagyhatjuk, mert azt már megadtuk az
                   INTERFACE-ben. Ha nem írjuk ki az nem
                   baj, de ha mást írunk, mint felül, akkor
                   hibaüzenetet kapunk}
BEGIN
    GOTOXY(x,y);
    WRITELN(s);
END;
                   {Most nincs inicializáló rész}
END.
```

Ezzel készen is vagyunk a forrásnyelvű Unittal. Mentsük el ELSO.PAS néven, fordítsuk le, helyezzük el elérhető helyen, és próbáljuk ki egy programmal.

```
Program Proba;
USES elsc;
BEGIN
    WRITEXY(10,10,'Hahó');
    READLN;
END.
```

2. Most következzen egy nehezebb példa. Ebben számos string és számkezelő függvény található, amelyek szerintünk hasznosak. A következő függvényeket készítjük el:

function upstr(s:string):string;

A megadott stringet nagybetűssé konvertálja az upcase segítségével. Csak az angol ábécé betűit módosítja.

function potnul(s:string; l:byte):string;

A stringként megadott számot l hosszúságúra igazítja. Ha a karakterlánc rövidebb, mint l, akkor nullákkal pótolja. Ez a függvény első látásra nem tűnik fontosnak, de gyakran kell négyjegyű hexadecimális számokat írni, így a később lévő tento eljárással kombinálva ezt egyszerűen tehetjük meg. Pl.: WRITELN(potnul(tento(312,16),4));

function str(w:word):string;

A w egész számot stringgé konvertálja. Ez a hagyományos str eljárásnak a függvény változata. így például paraméterként is használhatjuk. A hagyományosnál két lépésben kellett ezt megtenni. Egyébként majd minden nyelvben az str mint függvény szerepel. Vajon miért kellett a Turbo Pascalban eljárást csinálni belőle? Aki a hagyományos str-t akarja használni, az nevezze el a függvényt másnak, vagy ahogy a függvényben is meghívjuk, használja a hagyományost. Így: SYSTEM.STR(x,s) Ebből is látja az olvasó, hogy a névadással nagyon kell vigyázni, mert érhetnek bennünket meglepetések.

function val(s:string;base:byte):longint;

A val eljárás javított változata. Itt nemcsak az a változás, hogy függvény lett, hanem hogy megadhatunk egy számrendszeralapot is. Ha például hexadecimális számokat szeretnénk beolvasni, akkor azt mindenképpen stringbe kell tenni. Majd konvertálás, külön változófeltartás. Fontos, hogy a tízes alapú számrendszernél nagyobbakban a betűket nagybetűkkel kell írni. Ajánlott eljárás az upstr.

function tento(l:longint;base:byte):string;

A val függvény ellentéte tízesből base alapút készít (ez mindenképpen string lesz).

function minimstr(s:string):string;

Levágja a string elején és végén lévő felesleges space-eket. Könnyíti a szövegfeldolgozást.

function killdupspace(s:string):string;

Az egymás mellett lévő két vagy több space-ből csak egyet csinál. Könnyíti a szövegfeldolgozást.

```
unit strnum;
interface
  function upstr(s:string):string;
  function potnul(s:string;l:byte):string;
  function str(w:word):string;
  function val(s:string;base:byte):longint;
  function tento(l:longint;base:byte):string;
  function minimstr(s:string):string;
  function killdupspace(s:string):string;
implementation
  function upstr(s:string):string;
    var i:byte;
      begin
        for i:=1 to length(s) do s[i]:=upcase(s[i]); upstr:=s;
      end;
  function potnul(s:string;l:byte):string;
    var i:byte;
      begin for i:=length(s)+1 to l do s:='0'+s; potnul:=s;
      end;
  function str(w:word):string;
    var s:string;
      begin system.str(w,s); str:=s;
      end;
  function minimstr(s:string):string;
    var i:byte;
      begin i:=1;
        while (i<length(s))and(s[i]='')do inc(i);
        delete(s,1,i-1); i:=length(s);
        while (i>0)and(s[i]='') do dec(i);
        delete(s,i+1,length(s)); minimstr:=s;
      end;
  function val(s:string;base:byte):longint;
    var szam:longint;
```

```
        i:byte;
        nega:shortint;
      begin if s[1]='-' then
        begin delete(s,1,1); nega:=-1;
        End else nega:=1;
        szam:=0;
        for i:=1 to length(s) do
    szam:=szam*base+ord(upcase(s[i]))-48-7*byte
        (upcase(s[i])>='A');
        val:=nega*szam;
      end;
  function tento(l:longint;base:byte):string;
    var s:string;
        b:byte;
        nega:boolean;
      begin if l<0 then begin
                nega:=true; l:=-l;
            end else nega:=false;
        s:='';
        while l>0 do
        begin b:=l mod base;
              s:=chr(b+7*byte(b>9)+48)+s; l:=l div base;
        end;
        if nega then s:='-'+s;
        tento:=s;
      end;
  function killdupspace(s:string):string;
    var i:integer;
      begin
        i:=2;
        while i<length(s) do
        if (s[i]='') and (s[i-1]='') then delete(s,1,1)
        else inc(i);
        killdupspace:=s;
      end;
end.
```

2.10.4. Feladatok

1. Készítsen eljárásgyűjteményt, melynek segítségével minden hatványozással kapcsolatos függvény ismert. Az eljárások matematikailag helyesen működjenek.

2. Gyűjtse össze egy saját Unitba az eddig készített jelentősebb alprogramjait.

2.11. Abszolút változók, rádefiniálás

Talán az egyik legnehezebb részhez érkeztünk el. Kérjük az olvasót, hogy elsősorban ne megtanulni akarja az itt leírtakat, hanem megérteni, mert ezen áll az amúgy igen jól, hatásosan használható elsajátítanivaló alkalmazása. De kezdjük a legelején: tekintsük át nagy vonalakban a változókról tanultakat.

A deklarálásról nem is tudnánk sok újat mondani az olvasónak. Ha mégis felelevenítésre szorulna az ide tartozó anyag, úgy lapozza fel az ide vonatkozó fejezetet. A mi problémánk most a változó elhelyezkedése körül forog. Az „eddig megismert deklarációk" a helyfoglalásról maguk gondoskodtak. A főprogram változói az adatszegmensben helyezkednek el. Ezek a változók statikusak, a program futása alatt végig a memória ugyanazon területén foglalnak helyet, a teljes programra nézve globálisak. A program bármely pontjáról hivatkozhatunk rájuk. A **lokális változók**, vagyis az eljárásokban, függvényekben deklarált változók csak az alprogram működése során léteznek, és a stackszegmens elején helyezkednek el a Stack nevű verem típusú adatszerkezetben. A verem foglalása a stack végén indul, és visszafelé növekszik a lokális változók deklarálásakor. Ez nagyon kényelmes megoldás, mert a fordító minden definiálást úgy végez el, hogy abból baj nem származhat, azaz még véletlenül sem kerülhet változó például az operációs rendszer által elfoglalt memóriaszeletbe. A deklarált változók tárbeli helyfoglalását, memóriabeli kezdőcímét a fordítóprogram automatikusan végzi.

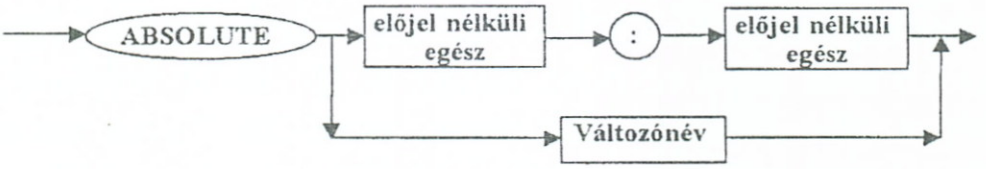
Változók memóriabeli kezdőcímét azonban mi is megadhatjuk.

A kezdőcím megadása történhet úgy, hogy a változót rádefiniáljuk

— egy már deklarált változó címére,

— vagy pedig egy abszolút címre.

A most deklarált változónak a kezdőcíme a már ismert változó kezdőcíme, illetve a megadott cím lesz. Ennek a résznek a szintaxisa:



Az ABSOLUTE alapszó segítségével címre definiált változót **abszolút változónak** nevezzük. Az abszolút változó esetében — közvetlenül vagy köz-

vetve — egy memóriacímet is megadunk, így egy előre megadott memóriacímre rádefiniáljuk a változót. A változó deklarálása tehát:

VAR név : típus ABSOLUTE cím; vagy

VAR név : típus ABSOLUTE már_deklarált_változó; alakú lehet.

A memóriacím közvetlen megadása a Turbo Pascalban két előjel nélküli egész konstans segítségével történik. Az első: a memóriának egy abszolút címét jelenti, egy alapcímet, más néven szegmenscímet. A második pedig egy ehhez viszonyított relatív cím. Ezt nevezzük offsetnek. A konstansok értéke a 0..65535 intervallumban lehet. Hexadecimálisan megadva az intervallum: $0000..$FFFF . A két konstans közé kettőspontot kell tenni. Pl.:

```
Var KarMod : Byte Absolute $0040:$0049;
  KarErnyo : ARRAY[1..25,1..80,1..2] OF BYTE ABSOLUTE $B800:$0;
  KarLanc : string[50];
  LancHossz: byte Absolute KarLanc;
```

A felsorolt példákból is látható az abszolút változók bevezetésének előnye.

Segítségükkel olyan hardverelemek közvetlen elérését és vezérlését biztosíthatjuk, amelyeknek ismerjük a memóriabeli kezdőcímét és szerkezeti felépítését. Jó példa erre a karakteres képernyő abszolút változóval való kezelése. Kidolgozott feladatunkban részletesen taglaljuk a kérdést.

Az abszolút változók használatával mód nyílik arra is, hogy változók tartalmát többféle módon is feldolgozhassuk. A LancHossz byte típusú változót rádefiniáltuk a KarLanc string[50] típusú változóra. Ezt azt jelenti, hogy memória-kezdőcímük azonos. A karlanc változó értékének változásával közvetlenül kapjuk a LancHossz változóban a string aktuális hosszát. A string aktuális hosszának meghatározására eddig már két megoldást is tanultunk, de ez a módszer mindkettőnél gyorsabb. Érdemes alkalmazni.

2.11.1. Kidolgozott feladatok

1. Írassunk ki a képernyőre véletlenszerűen generált karaktereket különböző színekkel a CRT-Unit eljárásai és függvényei nélkül. (Csak a képernyőtörlés parancsot használjuk.) Ehhez viszont szükségünk van bizonyos előismeretekre. Mint tudjuk a karakteres képernyő vezérlését a Crt Unit tartalmazza. A karakteres üzemmód beállításait a $0040:$0049 címen lévő byte tartalmazza. Állítsuk ezt 3 értékre. Ez színes $80 \cdot 25$-ös üzemmódot jelent. A karakteres képernyő a $B800:$0000 címen kezdődik. Talán azt senkinek sem kell elmondani, hogy egy tömböt kell deklarálnunk. A probléma a tömb méretével van. Egy karakteres képernyőn 25 sor és 80 oszlop van. De a tömbbe még kell két adat: a karakter ASCII kódja (BYTE típusú adat) és a színe (ez is elfér bőven a BYTE típusban). Innen már rögtön adódik a méret:

egy $25 \cdot 80 \cdot 2$-es tömböt kell lefoglalni a $B800:0 pozíciótól kezdve. Talán a programot már mindenki meg tudná írni.

```
PROGRAM kep;
USES CRT;
VAR KarMod : Byte Absolute $0040:$0049;
    k:ARRAY[1..25,1..80,1..2] OF BYTE ABSOLUTE $B800:$0;
    i,j:INTEGER;
BEGIN
  KarMod:=3;
  CLRSCR;
    REPEAT
      i:=RANDOM(25)+1;
      j:=RANDOM(80)+1;
      k[i,j,2]:=RANDOM(16);
      k[i,j,1]:=RANDOM(255)+1;
  UNTIL KEYPRESSED;
END.
```

2. Aki már lapozta az Angster Erzsébet—Kertész László szerzőpár feladatgyűjteményét ([4]), annak bizonyára ismerős lesz a következő feladat. Lássuk a forráslistát, majd az elemzést.

```
PROGRAM Mem1 ;
VAR Rek: RECORD
       b : BYTE ;
       w : WORD ;
       s : STRING[3] ;
       END;
    RekKep : ARRAY [1..100] OF BYTE ABSOLUTE Rek ;
    i : BYTE ;
BEGIN
  WRITE('B= ') ; READLN(Rek.b) ;
  WRITE('W= ') ; READLN(Rek.w) ;
  WRITE('S= ') ; READLN(Rek.s) ;
  FOR i := 1 TO SIZEOF(Rek) DO WRITE(RekKep[i]:5) ;
  WRITELN; READLN;
END.
```

Mit is csinál ez a program? Jól láthatóan, a második fajta rádefiniálásra példa ez. Deklaráltunk egy rekordot, majd erre a rekordra „tettük rá" a byte-okból álló tömböt. Ez a tömb — mint a szerzőpáros találó elnevezése is mutatja — a rekordunk képét fogja szolgáltatni. A rekord elemei egymás

után, sorban helyezkednek el a memóriában: előbb a b nevű változó 1 BYTE-ja, aztán a w 2 BYTE-ja, s legvégül a string 4 BYTE-ja. A tömb első 7 eleme is így helyezkedik el: az első BYTE-jában (elemében) a b van stb. Pontosan ez a lényege a rádefiniálásnak. Ettől kezdve már csak ki kell íratni a rekord méretének megfelelően a tömb elemeit.

3. A már előbb említett könyvben találhatunk egy érdekes példát. Nézzük csak meg alaposan.

```
PROGRAM Mem3 ;
CONST NumLock = $20 ;
      CapsLock = $40 ;
VAR ValtoBill : BYTE ABSOLUTE $0:$417 ;
PROCEDURE BeKapcsol(Bill: BYTE) ;
BEGIN ValtoBill := ValtoBill OR Bill AND $E0 ;
END;
PROCEDURE KiKapcsol(Bill: BYTE) ;
BEGIN ValtoBill := ValtoBill AND ($FF XOR Bill AND $E0) ;
END ;
BEGIN BeKapcsol(CapsLock) ;
      WRITELN('Most a Caps Lock aktív -<ENTER>'); READLN;
      BeKapcsol(NumLock) ;
      WRITELN('Most a Caps Lock és a Num Lock aktív -<ENTER>');
      READLN;
      KiKapcsol(CapsLock+NumLock) ;
      WRITELN('Most a Caps Lock és a Num Lock inaktív -<ENTER>');
      READLN;
END.
```

Ha jól áttekintjük a programot, könnyedén rájöhetünk arra, hogy mit csinál. A Num Lock és Caps Lock gombok működését „teszi lehetővé" a gombok lenyomása nélkül. Talán nem kell tovább mondani: valahol itt van értelme az ilyen típusú változóknak. Természetesen a hasonló programok megírásához ismerni kell a különböző memóriacímeket. Ezekről az ismeretekről is többet olvashatunk a szerzőpáros könyveiben. Talán most már megértjük a könyv eszmefuttatásait. De a feladatok kipróbálása nélkül nem elég egyetlen könyvet sem olvasgatni.

2.11.2. Feladatok

1. Olvassunk be egy karakterláncot, mondjuk meg a hosszát. Természetesen a rádefiniálás segítségével dolgozzon.

2. Írjunk ki a képernyő közepére egy bekért szöveget, s véletlenszerűen változtatgassuk karaktereinek színét.

3. Olvassuk le a képernyő karakteres videomódját, változtassuk meg és közben mindig írassuk ki, hogy hány sor és oszlop áll a rendelkezésünkre. A szükséges címek:

$0040:$49 1 BYTE-on van a videomód

$0040:$4A 1 BYTE-on van az oszlopok száma

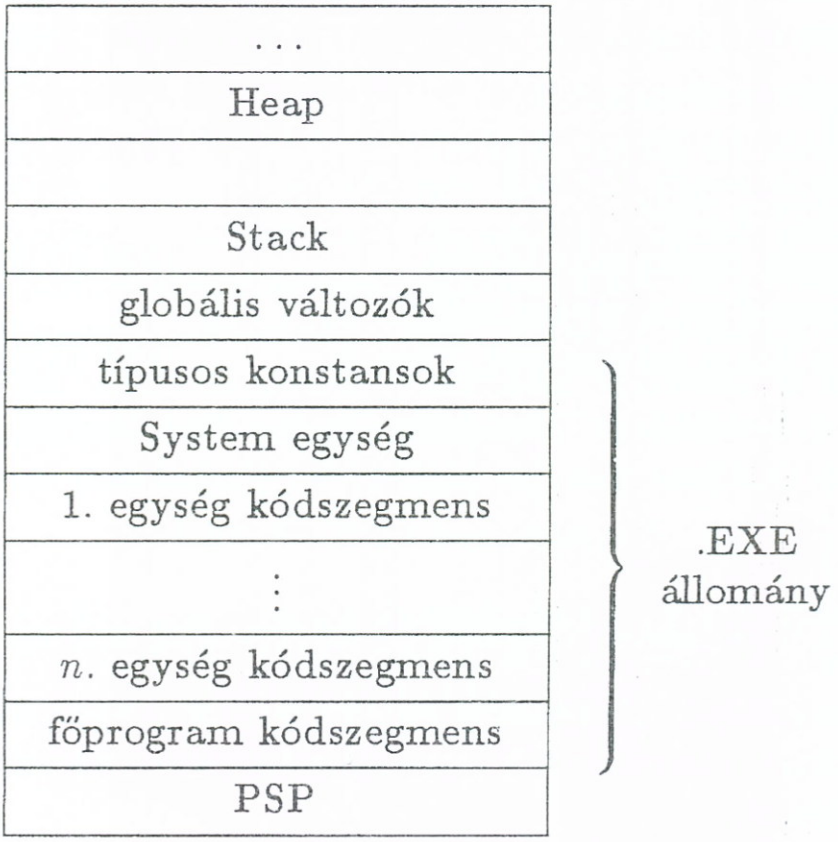
$0040:$84 1 BYTE-on van a sorok száma

4. Használja a képernyőt különböző videomódokban.

5. Készítse el a példában megoldott — képernyőre kiíró — programot különböző videomódokban.

2.12. Dinamikus változók

2.12.1. A Heap szerkezete



A Heapnek a pointeres változók, illetve a dinamikus változók használatában van nagy jelentősége. A dinamikus változók fő területe a stackszegmens végén a Heapben van. A Heap, azaz halom egy **verem** típusú adatszerkezet ([17]). Az utoljára betett adatot vehetjük ki először. Erre utal ennek az adatszerkezetnek az angol neve, a LIFO (last in first out), azaz: az utolsó először távozik. A Heapet a rendszer minden program futásának kezdetén a szabad memóriaterület kezdetére helyezi, és hozzárendel egy belső mutatót, amelyet a futás kezdetekor a Heap elejére állít (HeapPtr). A New eljárás ezt a mutatót állítja a változó méretének megfelelően feljebb, oly

módon, hogy ez a mutató mindig a szabad memóriaterület elejére állítódik, így a Heap következő szabad helyére mutat. Mindez azt jelenti, hogy a Heap és a Stack — két veremtípusú struktúra — egymással szemben, ellentétes irányban híznak. A végüket két mutató, a HeapPointer (HeapPtr) és a StackPointer mutatja. A memóriatúlcsordulást a New eljárás ellenőrzi. A túlcsordulás figyelését az S direktívával szabályozhatjuk. Alapértelmezésben a direktíva aktív: {$S+}. A figyelés kikapcsolását nem javasoljuk, mert annak beláthatatlan következményei lehetnek. Az M direktíva a program memóriafoglalását határozza meg: {$M veremméret, heapmin, heapmax}. A verem méretét (1024 és 65520 között) a veremméret, a heapméret alsó határának minimumát (0 és 655360 között) a heapmin, felső határának maximumát (a heapmin és 655360 között) a heapmax állítja be.

Pl.:

A verem mérete 20 000, a Heap mérete 0 legyen:	{$M 20000,0,0}
A verem mérete maximális, a Heap mérete 64 Kbyte legyen:	{$M 65520,0,$10000}
A verem mérete 1024, a Heap mérete maximális legyen:	{$M 1024,1024,655360}
Alapértelmezésben a verem mérete 16 Kbyte, a Heap mérete 640 Kbyte:	{$M 16384,0,655360}

Ugyanezek a beállítások elvégezhetők az Options|Memory sizes paranccsal a fordító Options menüpontjából is. Unitokban ezek a beállítások hatástalanok.

2.12.2. Dinamikus változó a Turbo Pascal nyelvben

A változókhoz alapértelmezésben a memóriakiosztás automatikus. Tárbeli helyfoglalásukat, memóriabeli kezdőcímüket a fordítóprogram automatikusan rendeli hozzá.

A főprogram és az egységek globális változóinak memóriabeli helyfoglalása már a fordítási időben megtörténik. Ez a terület az adatszegmensben (Data Segment) van.

Az alprogramokban deklarált változóknak a futtató rendszer futás közben — az alprogramba való belépéskor — foglal helyet. Helyfoglalásuk a veremszegmens (stackszegmens) elején a stackben van. A veremben futás közben foglalódik, illetve felszabadul a hely, a szubrutin meghívásakor, illetve elhagyásakor. Minderről a rendszer automatikusan gondoskodik. Az alprogram rekurzív hívása esetén újra belépünk ugyanabba az eljárásba anélkül, hogy befejezve elhagytuk volna azt. Így újabb belépéskor az alprogram úgy

kezelhető, mintha ez egy másik szubrutin volna, és ezért változói számára az újabb belépés esetén újabb helyet jelöl ki a rendszer a veremben.

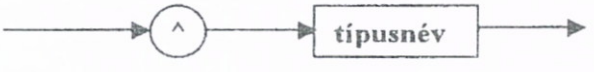
Az abszolút változók címét maga a programozó jelöli ki. Ezek címének foglalása az abszolút változó jellegétől függ.

A szakirodalmak ([26], [5], [3]) **statikus változóknak** nevezik azokat a változókat, amelyek címkijelölése már a fordítási időben megtörténik, míg dinamikusnak azokat, amelyeknek a futtató rendszer futás közben — az igények függvényében — foglal helyet.

A Turbo Pascal által dinamikusnak nevezett változó a dinamikus változók egy fajtája, a programozó által vezérelt helyfoglalású változó.* Egy dinamikus változót a program futása közben hozunk létre, illetve szüntetünk meg. A létrehozott változóra mutatóval hivatkozunk. A deklaráció — a típus-, méret- és kezdőcím-attribútumok — megadása a futás közben történik és ezek az attribútumok futás közben is változhatnak. Ezt jelenti a dinamikus elnevezés. Mivel közvetlen deklarációjuk nincs (VAR szakaszban nem fordulnak elő), ezért közvetlen módon, azaz azonosítóval nem hivatkozhatunk rájuk ([12]). A dinamikus változóra azonosító (név) helyett közvetve ún. mutatóval hivatkozunk. A mutató egy speciális típussal rendelkező változó, amely egy tárcímet, a létrehozott dinamikus változó címét tartalmazza. A mutatóváltozó típusmeghatározása annyit jelent, hogy meg kell mondani a hozzá tartozó dinamikus változó típusát. Mivel a dinamikus változó típusa akár rekord is lehet, ezért dinamikus változók láncolata is létrehozható, ha a rekord mezői között ismét szerepel mutató.

2.12.3. Típusos mutató

A dinamikus változók létrehozásának alapvető eszközei a típusos mutatók. A mutatók a nyelv pointer típusú objektumai. A mutatótípust a ^ karakter jelzi. Ezt követi annak a típusnak a neve, amelyre a most definiáló pointer mutatni fog: **^típusnév**.



Ez a szerkezet egyaránt szerepelhet a típusdefiníciós részben és változódefiníciós részben is. Minden mutató típusú változó felvehető értékei között ott szerepel a Nil mutatókonstans. A Nil konstans olyan mutató, mely nem mutat egyetlen változóra sem. A mutatott változóra **változónév^** formában hivatkozhatunk.

A mutató típusú változókkal a következő műveleteket végezhetjük:

---

* A dinamikus változó megnevezést a továbbiakban ebben az értelemben használjuk.

— értékadás (csak azonos típusú mutatóváltozók között),

— szerepelhetnek alprogramok paramétereiként,

— az = és a <> hasonlítások elvégezhetők az azonos típusú mutatók között,

— segítségükkel **dinamikus változókra hivatkozhatunk.**

2.12.4. Típusnélküli mutató

A Turbo Pascal nyelvben típusnélküli mutató is van. Ekkor nincs meghatározva, hogy milyen típusra mutat a mutató. A rendszerben a típusnélküli mutató előre definiált típus, ezért a típus- és a változódeklarációs részben is a Pointer szót kell használni.

```
Type Mut=Pointer;
Var Masmut:Pointer;
```

A mutatott változóra ugyanúgy hivatkozhatunk mint a típusos mutatóknál: **változónév^**. Pl.: Writeln(Masmut^);

Mivel a mutatott változónak nincs típusa, ezért csak a típust nem igénylő operációknál alkalmazhatjuk ezt a típust. Természetesen a New( ) eljárásnak sincs értelme a típusnélküli mutatóknál.

A továbbiakban a típusos mutatók alkalmazási területét elemezzük kissé részletesebben. Amikor mutatóról (Pointer) vagy mutatótípusról lesz szó, nem a Turbo Pascal előre definiált Pointer típusáról, hanem általában a mutatóról, illetve a típusos mutatóról lesz szó. A Pointer szót az irodalmakban is így használják.

2.12.5. Dinamikus változók létrehozása

Mint korábban is mondtuk; a dinamikus változókat a program futása közben hozzuk létre. Ennek programbeli ajánlott lépései:

A dinamikus változó létrehozásához célszerű **először** a mutatóváltozó típusát definiálni. Az így definiált, névvel azonosított típust **mutatótípusnak** (pointertípusnak) nevezzük. A dinamikus változó címét egy ilyen módon definiált típussal rendelkező változó tartalmazhatja. A típusdeklarációban a típusnevet egy másik típushoz, az alaptípushoz rendeltük. Az alaptípus a mutatótípushoz tartozó dinamikus változó típusa.

```
Type Tomb =Array[1..100] Of Word;
  Mut =^Tomb;
  Mut1 =^Tomb1;
  Tomb1 =Array[-10..10] Of Byte
```

Mint azt a példából is látjuk, a típusdeklarációs részben a Mut mutató típust a Tomb típus deklarációja után (amelyre mutat) definiáltuk. A Mut1

mutatótípust viszont azon Tomb1 típus előtt deklaráltuk, amelyre mutat. Az általános szabály, hogy a típusdefiníciós részben egy típus és a hozzátartozó mutatótípus deklarálásának sorrendje tetszőleges. Csak az a kikötés, hogy ha a mutatót deklaráljuk, akkor valahol definiálnunk kell azt a típust is, amelyre mutat.

Ezután a változódefiníciós részben deklarálhatjuk a mutatótípusú statikus változokát.

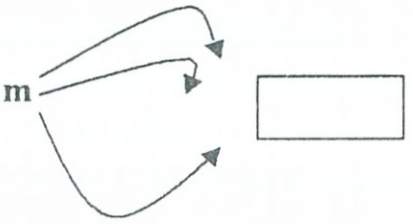
```
Var p,q : Mut;
      u : ^Tomb;
      r : Mut1;
      s,t : ^Word;
      i,j : Word;
```

Dinamikus változónk eddig még nincs. Eddig csak statikus változókat hoztunk létre. A dinamikus változó **még nem foglalja a helyet a tárban. Tárbeli helyfoglalása csak a program végrehajtási részében történik meg.** A példában p egy olyan tömbre mutat, mely a deklarálás pillanatában még nem létezik. Ahhoz, hogy a tárban ténylegesen lefoglalásra kerüljön a hely, a végrehajtási részben meg kell hívni a New( ) eljárást.

A New(Var P:Pointer) eljárás **létrehoz** a Heapben egy dinamikus változót — mely típusát a mutatótípusú P határozza meg —, és címét elhelyezi P-ben ([3]). Ekkor létrejön a Heapben a deklarációnak megfelelő memóriafoglalás. Összekapcsolódik a pointer a memóriabeli objektummal. Az aktuális paraméter — a deklarációs részben már definiált — típusos mutató lehet. A New eljárás oly módon látja el funkcióját, hogy mindössze két mutatót állít. A paramétereként szereplő mutató felveszi HeapPtr értékét, a HeapPtr értékét pedig a lefoglalt területnek megfelelően feljebb állítja, mely az eljárás után is a szabad terület kezdetét jelzi. A lefoglalt területet a mutatóhoz tartozó blokknak nevezzük. Példánkban: New (P) eljárás meghívásakor létrejön a Heapben egy, a TOMB típusnak megfelelő hosszúságú memóriaterület, amelyre a P mutat. De hogy pontosan megértsük a folyamatot, tekintsük a következő egyszerű deklarációt.

```
type intpoint = ^integer;
var m : intpoint;
  p : ^real;
```

A New eljárás előtt a dinamikus változó még nem foglal helyet a memóriában, azaz nincs kapcsolat az m pointer és az egész típusú változó tárolására alkalmas tárterület között ([5]).



A New(m) eljárás végrehajtása után viszont az m pointer egy egész típus tárolására alkalmas tárterületre mutat.

Mivel a mutatott változónak saját neve nincs, ezért a programban csak a mutatóval hivatkozhatunk rá. Az m-hez kapcsolt objektumot m^ jelöli, azaz így hivatkozhatunk rá. Ezt a hivatkozást ugyanúgy használhatjuk bármely nyelvi konstrukcióban, mint a változóneveket. Pl.: m^ :=4 utasítással értéket adtunk neki. Ez azt jelenti, hogy az a memóriaterület, amelyre az m mutató mutat, 4 értéket kap.

Az azonos típusú mutatók között megengedett az értékadó utasítás is. Ha P és Q két azonos típusú mutatóváltozó, akkor P:=Q utasítás azt jelenti, hogy P mutató is ugyanoda mutasson, mint a Q mutató.

A dinamikus változók bármely tárkijelölését feloldhatjuk a **Nil konstans** alkalmazásával ([12]). Pl.: P:= Nil; Ekkor a P mutató nem mutat sehová, vagyis feloldja a kapcsolatot a mutató és a tárterület között. A Nil konstans **kompatibilis minden mutatótípussal.**

Ha P és Q azonos típusú mutatók, akkor P:= Q értékadás azt jelenti, hogy a P-hez rendelt cím legyen egyenlő a Q-hoz rendelttel. Tehát a P ugyanoda mutasson, ahova a Q mutat.

A P^ := Q^; értékadás viszont azt jelenti, hogy a P által mutatott tárbeli érték vegye fel a Q által mutatott tárbeli értéket.

A pointerhez kapcsolt objektumot le is „akaszthatjuk" a mutatóról. Ezt a Dispose eljárással tehetjük meg. Ez azt jelenti, hogy az eljárás meghívása után nem lesz meg a kapcsolat a mutató és a tárterület között.

A Dispose(Var P: Pointer) eljárás felszabadítja a P típusos mutatóval mutatott Heapbeli területet. A felszabadított terület nagyságát az aktuális paraméter típusa határozza meg. A felszabadítás után a P^-ra hivatkozás hibát okozhat. Futási hiba lesz, ha a P nem a Heapbe mutat. Ha a felszabadítás a Heap tetejéről történik, akkor a HeapPtr lejjebb mozog, egyébként a Heapben egy „lyuk" keletkezik, amelyet a későbbi New felhasználhat.

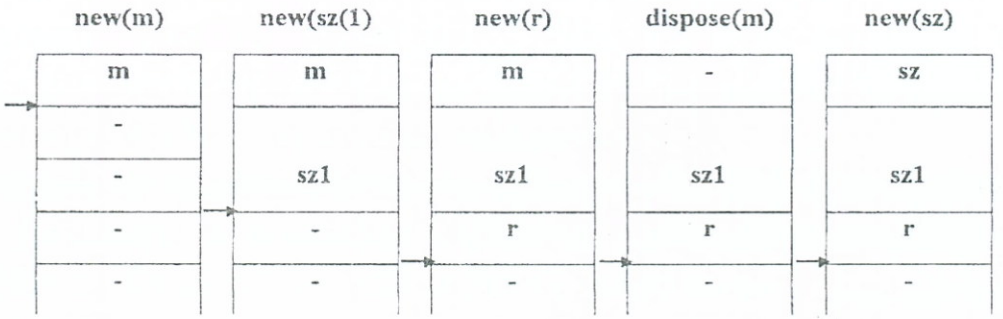
Egy példán keresztül tekintsük át, hogy mi történik a halomban a New és a dispose utasítások hatására. Legyen a deklaráció a következő:

```
Type klanct = string[15];
     klanct1 = string[20];
Var m : ^integer;
  r : ^real;
  sz : ^klanct;
  sz1 :^klanct1;
```

Tekintsük meg a halom állapotait a következő utasítások végrehajtásakor:

```
new(m);
new(sz1);
new(r);
dispose(m);
new(sz);
```

A $\longrightarrow$ a HeapPtr pillananyi értékét mutatja.



A New és a Dispose eljárások nem bájtonként, hanem 8 bájtos darabokban kezelik a halmot. Ez általában nem okoz nagy tárpazarlást, mert nem integer értékeket, hanem többnyire hosszabb rekordokat szoktunk a mutatókkal elérni.

A dinamikus változóknak különösen nagy jelentőségük a különböző adatszerkezetek létrehozásakor van. Létrehozhatunk listákat, fastrukturákat, gráfokat. Láncolt adatszerkezetekről akkor beszélünk, ha mutatókkal kapcsolunk össze objektumokat. Az összekapcsolt objektumok rekordok lehetnek, hiszen az adatmezők mellett mutatókat — azaz pointer típusú mezőket — is kell tartalmazniuk. Egy iskolai tanulmányi pályázaton a tanulók adatait tartalmazó rekordokat helyezési sorrendjük alapján fűzhetjük össze. Az ilyen, lineárisan láncolt szerkezeteket listaszerkezeteknek nevezzük ([5]).

2.12.6. Kidolgozott feladatok

1. Az értékadó utasítások megértése érdekében gépelje be a programot, és lépésenként kövesse végig a rendszer nyomkövetésének segítségével. (A Watch ablakban p, q, p^, q^, majd F7.)

```
PROGRAM proba;
TYPE ipoitip:^INTEGER;
VAR p,q: ipoitip;
BEGIN
   NEW(p); p^:=5;
   NEW(q);q^:=3;      {Eddig ott tartunk, hogy lefoglaltuk a két
                       változónak a helyet. A p mutatott értéke 5,
                       q mutatott értéke 3 lett,
                       de a címük különböző.}
     p^:=q^;          {Most mindkét változó értéke 3 lett,
                       de a címük különböző.}
     p^:=5;           {Vissza az eredetihez.}
     p:=q;            {Ez a címek közti értékadás: mindkettő
                       mutatott értéke 3, de most
                       a címük is megegyezik.}
     DISPOSE(p);      {Szétbontottuk a kapcsolatokat a p és q
                       mutatók és a halom között, vagyis
     DISPOSE(q);       megszűntek a dinamikus változók.}
END.
```

2. A következő példában bemenő decimális számot átalakítunk 10-esnél kisebb alapú számrendszerbe.

```
PROGRAM dinam;
USES CRT;
  TYPE mutato=^elem;
       elem= RECORD
sz:BYTE;
elozo:mutato;
END;
  VAR a,szam: INTEGER;
    verem,h:mutato;
BEGIN
  ClRSCR;
  WRITE('Kérem a számot decimálisan: '); READLN(szam);
  WRITE('Kérem a számredszer alapját: '); READLN(a);
  h:=NIL;
  WHILE szam<>0 DO
    BEGIN
      NEW(verem);
      verem^.sz:=szam mod a;
      verem^.elozo:=h;   {Az osztás maradékainak elraktározása a verembe.}
      szam:=szam DIV a;
```

```
      h:=verem;
    END;
  WHILE verem<>NIL DO
    BEGIN
      WRITE(verem^.sz);    {Az elraktározott számjegyek kiolvasása
      verem:=verem^.elozo;   és kiírása a veremből.}
    END;
  READKEY;
END.
```

Elemezze részletesen a program működését.

2.12.7. Feladatok

1. Olvassunk be számokat végjelig, átlagoljunk, számítsunk szórást.

2. Olvassa be egy osztály tanulóinak adatait (név, tan. átlag), majd számítsa ki az osztály átlagát. Dolgozzon dinamikus változóval.

3. Olvasson be számokat végjelig. Tároljuk le az adatokat egy dinamikus változóban. Ezután tárolja le az adatokat az előbbi tárolásnak fordított sorrendjében egy újabb dinamikus változóban.

4. Olvasson be pozitív számokat 0 végjelig. Tárolja azokat dinamikus változóban. Rendezze a számokat csökkenő sorrendbe, majd írja ki a rendezés sorrendjében. Dinamikus változókkal dolgozzon.

5. Egy kérdőív kitöltésekor nem biztos, hogy minden kérdésre válaszoltak. A kérdések nagyság szerint következnek, azaz sorszámozottak. Írassuk ki növekvő sorrendben azt, hogy mely kérdésekre adott választ az illető.

2.13. Grafika

Talán elérkeztünk a könyv legszebb, de minden bizonnyal leglátványosabb részéhez. A képek, az ábrák, a grafikus felületek mindig is a legérdekesebb témák voltak. E témákkal könnyű lekötni a figyelmet. De be kell mutatni azokat a nehézségeket és problémákat is, amelyek a különböző hardverkonfigurációkból adódnak.

A fejezet logikailag két részre tagozódik. Az első részben a grafika Pascal programnyelvbeli megoldásáról lesz szó: a grafikus képernyő létrehozásáról, kezeléséről, magáról a rajzoló eljárásokról, függvényekről. A második rész talán mindenki számára érdekes lesz. A számítógépes grafika egyik alkalmazási területéről ejtünk szót: a függvényábrázolásokról.

2.13.1. A BGI fájl

Mielőtt teljesen belemerülnénk a témába, boncolgassunk egy közhelyet: rengeteg monitortípus van. Adott géphez, adott monitorvezérlőhöz egészen másképp kell hozzányúlni, mert egészen mások azok a bizonyos vezérlőjelek, amelyeknek ugyanazt a funkciót kell megvalósítaniuk a képernyőn. Sajnos — vagy nem? — ez már régóta így van, s tudták ezt a keretrendszer írói is. A különböző grafikus kártyákhoz — CGA, MCGA, EGA, VGA, Hercules, AT&T400 soros, 3270 PC, IBM 8514 — a Borland cég grafikus meghajtókat szállít. A kódot és az adatot tartalmazó meghajtót a .BGI (Borland Graphics Interface) állomány tartalmazza, melynek a grafika használatakor a tárban kell lennie. Többek között ez az állomány biztosítja számunkra azt, hogy egyrészt kapcsolatot tudjunk tartani az egyes monitorfajtákkal, másrészt minden monitoron ugyanúgy (hasonlóan) nézzen ki a programunk által készített kép.

2.13.2. Graph Unit

Ha már adott a grafikus képernyő kezelése, használjuk is ki az adott lehetőségeket. A Graph Unitban vannak azok az eljárások és függvények, amelyekre szükségünk lesz. Ne feledjük tehát minden grafikus jellegű programunkba beilleszteni a fent említett unit meghívását: **USES GRAPH;**.

Első lépésként le kell olvasni a grafikus meghajtót és annak üzemmódját. A **DETECTGRAPH (VAR i, j : INTEGER);** eljárás teszi ezt meg, ahol i a meghajtóra, j pedig az ezen belüli üzemmódra jellemző értéket tartalmazza.

Miután ismerjük ezeket a jellemzőket, az

INITGRAPH (i,j : INTEGER; bgi_ut : string);

eljárással tudjuk inicializálni a képernyőnket. Az i,j a DETECTGRAPH által felismert értékek, az őket követő string pedig a már előbb említett BGI kitejesztésű fájl elérési útja. Például ha a BGI fájl a C:\PASCAL\BGI-ben található, akkor az előbbi sor így fog kinézni:

INITGRAPH (i, j, 'C:\PASCAL\BGI');.

Ha az elérési útnál a string üres (''), akkor az elérési út a beállított elérési út lesz.

Ha már nem használjuk a grafikus képernyőt, akkor azt a **CLOSEGRAPH;** eljárással tudjuk bezárni.

A továbbiakban beszéljünk egy kicsit a színekről. Mint ahogyan a festők keverőpalettájukról kiválasztják a nekik megfelelő színeket, nekünk is hasonlóan kell eljárnunk. A Unitban deklarálták az ún. palettatípust, melynek felépítése a következő:

```
TYPE    Palettetype  =RECORD
        Size         : BYTE;
        Colors       : ARRAY[0..MAXCOLOR] OF SHORTINT;
                     END;
```

ahol az első mező a színek számát, a második pedig a „leírását" tartalmazza. A MAXCOLOR konstans, értéke a színek maximális számát jelzi. Nézzük meg tehát, hogy mit lehet kezdeni a színekkel, a palettával.

A **SETBKCOLOR**(szin : WORD); eljárás a paletta szin-edik színére állítja a háttérszínt. Természetesen ettől még nem lesz ilyen színű a háttér, de a következő képernyőtörléskor ilyenre fogja festeni.

A **SETCOLOR**(szin : WORD); magát a tintaszínt állítja.

Térjünk át a palettára. Mit is lehet vele kezdeni? Például lekérdezhetjük a Size értékét a

GETMAXCOLOR : WORD; függvény segítségével direkt módon is.

A **GETPALETTESIZE** :WORD; függvény az aktuális grafikus meghajtóhoz és üzemmódhoz tartozó színek számát adja vissza.

Deklaráljunk egy paletta nevű Palettetype típusú változót.

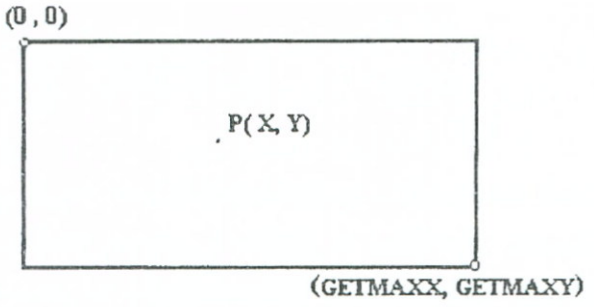
A **GETPALETTE**(VAR paletta : Palettetype); kiolvassa a Unitból a palettára vonatkozó információkat, s ezután mint rekordot lehet kezelni, azaz minden további nélkül átdefiniálhatjuk a palettát.

A **SETPALETTE**(VAR paletta : Palettetype); eljárás pedig visszaírja az általunk módosított palettát.

Ha mégis az eredeti palettát szeretnénk visszaállítani, akkor hívjuk meg a **GETDEFAULTPALETTE**(VAR paletta : Palettetype); eljárást! Ezt egyébként alapértelmezésben az INITGRAPH elvégzi, tehát nem kell rögtön ezzel kezdenünk. Nézzük meg a továbbiakban: milyen is valójában a grafikus képernyő, hogyan méretezhető, törölhető stb.?

A CRT Unitnál a karakteres képernyő nem volt olyan változatos, mint a grafikus képernyő lesz. A TextMode eljárás paraméterei néhány üzemmódot engedtek meg. A képernyő bal felső sarkának koordinátája $(1, 1)$ volt. A 25 sor és 80 oszlop alkalmazása általánosnak tekinthető, így a jobb alsó sarok koordinátái: $(80, 25)$.

A grafikus képernyők koordinátarendszerében a bal felső sarok koordinátáit $(0, 0)$-nak választották. A grafikus képernyő felbontásának ismeretétől függetleníthetjük magunkat, ha használjuk a GetMaxX és a GetMaxY függvényeket. A GETMAXX, illetve a GETMAXY függvény a $(0, 0)$ ponttól való maximális értékeket adja meg az x, illetve az y tengelyen.



Javasoljuk, hogy lehetőleg ne használjon a képernyő felbontására jellemző konkrét számokat, mert akkor a program merev lesz, és csak ugyanolyan felbontás esetén ad elfogadható képet. Fontos, hogy az ominózus sarkot ne (639, 479)-nek vegyük, mert, ha egy ábrát megterveztünk 640 × 480-as felbontásra, de a monitorunk 1024 × 768-assal működik, akkor az ábra nem lesz középen. Torz és nem jól illesztett vonalakat láthatunk grafikánk helyett.

A rendszer ablakot is tud kijelölni a

SETVIEWPORT(x_1,y_1,x_1,y_2 : WORD; vagas : BOOLEAN);

eljárás segítségével, ahol x_1, y_1, x_2, y_2 az ablak bal felső és jobb alsó csúcsainak koordinátái. Ha vagas=TRUE, akkor az aktív ablakba rajzolt objektumok csak akkor látszanak, ha azok nem lógnának ki belőle, ellenkező esetben (FALSE) a kinyúló részek is láthatók. Az eljárás kiadásától a minden — a rajzoláshoz megadott paraméter — erre az ablakra vonatkozik mindaddig, amíg nem alkalmazzuk más paraméterekkel az eljárást. A képernyőtartalom, a korábbi ablakok csak akkor törlődnek ha meghívjuk a grafikus képernyőt törlő eljárást. Ezt a CLEARVIEWPORT; eljárás hívásával tehetjük meg, ami tehát az aktív ablak háttérszínre festését végzi. Alapértelmezésben az aktív ablak a teljes grafikus képernyő.

Most már térjünk rá a tényleges rajzoló eljárásokra. Természetesen a karakteres képernyőhöz hasonlóan itt is találhatunk kurzort — jelöljük CP(CP_y,CP_y)-nal —, melyet lehet mozgatni.

A MOVEREL(x, y : INTEGER); eljárás hatására a kurzor az eredeti pozíciójához képest (x, y) vektorral mozdul el, azaz a kurzor koordinátái

$$CP_x := CP_x + x, \quad CP_y := CP_y + y$$

alapján változnak. Ez az eljárás relatív változást hoz létre a kurzor koordinátáinál.

A másik kurzorváltoztató eljárás a MOVETO(x, y : INTEGER);, amely abszolút módon változtatja a kurzort, azaz $CP_x := x$ és $CP_y := y$.

2.13.3. Szöveg a grafikus képernyőn

Elsőként nézzük meg a szövegek kiíratását a képernyőre, hiszen ez is grafikusan történik. Az OUTTEXT(x, y : INTEGER; s : STRING); eljárás

(x, y) pozíciótól kezdve írja ki az s szöveget az aktuális ablakra vonatkoztatva. Megváltoztathatjuk a kiírandó szöveg stílusát is, melyet a

SETTEXTSTYLE(font, irany : WORD; meret: CHARSIZETYPE);

eljárás tesz lehetővé. Ennek paraméterei:

— font: számokat jelent, a betűtípust szabja meg.

0: alapértelmezésbeli
1: triplex font
2: kicsi betűk
3: egyszerű betűk
4: gót betűk

— irany: ezzel lehet állítani az írás irányát.

0: balról jobbra ír
1: lentről fölfele ír

— meret: és 10 közötti egész szám, ennyiszeres nagyítást jelent.

A CHARSIZETYPE beépített típus a Unitban.

Az alapértelmezésbeli karaktereket csak teljes karakterként láthatjuk (ha elférnek). A magyar ékezetes karakterek közül az Ő és az Ű betűk kalaposan jelennek meg, ezért olyan programokban, amelyekben fontos a helyes magyar írás, nekünk kell „repülő ékezetekkel" megoldani a nevezett betűk feletti ékezeteket. Már létezik olyan karakterkészlet, amely a teljes magyar betűkészletet tartalmazza.

A SETTEXTJUSTIFY(v, f : WORD); eljárás a kiírandó szöveget igazítja az aktuális kurzorhoz képest. Ha

v=0, akkor balra igazít,
v=1, akkor középre igazít,
v=2, akkor jobbra igazít,
f=0, akkor az aljára igazít,
f=1, akkor közepére igazít,
f=2, akkor tetejére igazít.

A szöveg magasságát a TEXTHIGH(s : STRING)WORD;, szélességét a TEXTWIDTH(s : STRING)WORD; függvény adja meg. Természetesen minden kiírás az aktuális színnel és háttérszínnel történik.

2.13.4. Pontok, egyenesek, görbék rajzolása

Térjünk át a pontok rajzolására. Egy pont kirajzolását vagy inkább kifestését a PUTPIXEL(x, y : INTEGER; szin: WORD); eljárás teszi lehetővé, ahol (x, y) a koordináták, a szin pedig a kifestőszínt jelenti. Természetesen a MOVETO és a MOVEREL eljárások érvényesek most is.

A **GETPIXEL(x,y)**: WORD; függvény lekérdezi az adott koordinátájú pont színkódját a palettáról.

A **GETX**: INTEGER; és **GETY**: INTEGER; függvények az aktuális képpont x, illetve y koordinátáit adják meg.

Vonalat húz a **LINE(x1,y1,x2,y2** :INTEGER); eljárás az éppen érvényes vastagságban és stílusban.

Ezt a **SETLINESTYLE(stilus, minta, vastagsag** : WORD); állítja be. Paraméterei:

— stilus: stílust határoz meg.

0: normál,
1: pontozott,
2: szaggatott 1,
3: szaggatott 2,
4: a felhasználó által definiált.

— minta: akkor van értelme, ha a stílus 4 volt. Word típusú hexadecimális érték adja meg a mintát. Ha a szó byte-ja 1, akkor ott pont van, azaz a $FFFF a tömör, a $0001 alig látszik.

— vastagsag: 1 a normál, 3 a vastag vonal.

Vonalat húzhatunk úgy is, hogy az aktuális kurzorpozíciótól megadott koordinátáig megyünk. Ezt a **LINEREL(x, y** : INTEGER); teszi lehetővé, s ekkor a kurzor pozíciója nem változik.

A **LINETO(x, y** : INTEGER); eljárás is hasonlót eredményez, annyi különbséggel, hogy az aktuális kurzorpozíció (x, y) lesz.

Ejtsünk néhány szót a poligonokról vagy más néven sokszögekről.

A **DRAWPOLY(pontszam** : WORD; VAR **pontok)**; eljárás összeköti a pontokat az aktuális stílussal és színnel. A pontszam adja meg a poligon csúcsainak a számát, a pontok egy deklarált tömb a következő beépített típussal:

```
Ponttype = RECORD
x, y : INTEGER;
END;
```

A poligon sokszög lesz, ha az első és utolsó pontja megegyezik.

A **RECTANGLE** (x_1,y_1,x_2,y_2 : INTEGER); eljárás egy négyszöget rajzol a megadott koordinátákkal az aktuális színben, stílussal, alakban.

Az egyenesek rajzolási módját a **SETWRITEMODE**(v : INTEGER); szabályozza. Ha v=0, akkor a most kitett pontok lefedik az előzőt. Ha v=1, akkor a már meglévő pontokra és a most kitettekre XOR művelet hajtódik végre.

Keretvonal nélküli kitöltött négyszöget a BAR(x_1,y_1,x_2,y_2: INTEGER); eljárás rajzol.

A kitöltés mintáját és színét a SETFILLSTYLE(stilus, szin: WORD); állítja be, ahol a stilus 0 és 12 közötti egész szám (a 12-es a felhasználó által definiált stílus), a szin pedig palettaszín. Ha a stilus=12, akkor alkalmazhatjuk a SETFILLPATTERN(minta: Fillpatterntype; szin:WORD); eljárást. A minta egy 8 BYTE-on tárolt hexadecimális kód, a szin pedig a palettaszínt jelenti.

A BAR3D(x_1,y_1,x_2,y_2:integer; mely : WORD; t: BOOLEAN); eljárás egy perspektivikus téglatestet rajzol, ahol a mely a téglatest mélységét jelzi (ha 0, akkor téglalap), t pedig azt, hogy legyen-e teteje a téglatestnek vagy ne.

Körök, körívek rajzolásakor a rendszer a grafikus kártyára jellemző arányossági tényezőt használja fel, így korrigálja a torzítási problémákat. A torzítási tényezőt a GETASPECTRATIO(VAR x, y :WORD); adja meg. A CIRCLE(x, y : INTEGER; r : WORD); a SETCOLOR által meghatározott színnel kört rajzol. Az ARC(x, y : INTEGER; a, b, r : WORD); eljárás (x, y) középpontú, a foktól b fokig terjedő r sugarú körívet jelenít meg, ahol a szöget fokokban kell megadni. Ez sem torzít.

Ellipszisvonalat az ELLIPSE(x, y :INTEGER; a, b, r_x, r_y : WORD); segítségével rajzolhatunk, ahol x, y, a, b ugyanaz, mint az ARC esetében volt, r_x, r_y pedig a megfelelő sugarak.

A FILLELLIPSE(x, y :INTEGER; r_x, r_y : WORD); pedig kitöltött ellipszist rajzol az aktuális értékekkel.

2.13.4.1. Kidolgozott feladatok

A következőkben jöjjenek a példák, hiszen már eléggé sok mindent megtanultunk. Jó lenne alkalmazni is a tanult eljárásokat. Természetesen még sok eljárást tud a Graph Unit, de ezek voltak azok, amelyeket leggyakrabban használni fogunk.

1. Először rajzoljunk meg egy biliárdasztalt felülről.

```
PROGRAM golyok;
USES CRT,GRAPH;
VAR dm,dg,i,j:INTEGER;
    x:ARRAY[1..10] OF INTEGER;
    y:ARRAY[1..10] OF INTEGER;
BEGIN
  DETECTGRAPH(dm,dg); INITGRAPH(dm,dg,'c:\tPascal6.0\bgi');
  SETCOLOR(3); MOVETO(80,80); LINETO(GETMAXX-80,80);
  LINETO(GETMAXX-80,GETMAXY-80); LINETO(80,GETMAXY-80);
  LINETO(80,80);
```

```
  FOR i:=1 TO 10 DO
    BEGIN
      x[i]:=ROUND(RANDOM*(GETMAXX-200)+100);
      y[i]:=ROUND(RANDOM*(GETMAXY-200)+100);
      SETCOLOR(ROUND(RANDOM*16);
      SETFILLSTILE(1,ROUND(RANDOM*15));
      FILLELLIPSE(x[i],y[i],10,10);
    END;
  READKEY; CLOSEGRAPH;
END.
```

2. Most írjunk meg egy képernyővédő szimulátort.

```
PROGRAM csillagok;
USES CRT,GRAPH;
CONST db=5000;
TYPE ttip=ARRAY[1..db,1..3] OF INTEGER;
VAR t:ttip;
  dm,dg,c:INTEGER;
PROCEDURE cs;
  const m=6;
  n=4;
  BEGIN
  SETBKCOLOR(1);
  CLEARVIEWPORT;
  MOVETO(55+(6-m)*n*10,0);
  SETTEXTSTYLE(n,0,m);
  OUTTEXT('CSILLAGOK © 1998');
  SETVIEWPORT(0,90,GETMAXX,GETMAXY,true);
END;
PROCEDURE inic(VAR t:ttip);
VAR i:INTEGER;
BEGIN
  FOR i:=1 TO db DO
  BEGIN
    t[i,1]:=ROUND(RANDOM*GETMAXX);
    t[i,2]:=ROUND(RANDOM*GETMAXY);
    t[i,3]:=ROUND(RANDOM*GETMAXCOLOR);
  END;
END;
PROCEDURE kirak(VAR t:ttip);
VAR i:INTEGER;
```

```
BEGIN
  FOR i:=1 TO db DO PUTPIXEL(t[i,1],t[i,2],t[i,3]);
END;
PROCEDURE csillag(a,b,c:INTEGER);
CONST r=3;
VAR i,j:INTEGER;
BEGIN
  SETFILLSTYLE(1,GETMAXCOLOR);
  FOR i:=1 TO r DO
  BEGIN
    IF i<r-(r/2) THEN j:=0 ELSE j:=i-ROUND(r-(r/3));
    FILLELLIPSE(a,b,i,j);
    IF i<r-(r/2) THEN j:=0 ELSE j:=i-ROUND(r-(r/3));
    FILLELLIPSE(a,b,j,i);
    DELAY(30);
  END;
  DELAY(50);
  SETCOLOR(0);
  SETFILLSTYLE(1,0);
  FOR i:=1 TO r DO
  BEGIN
    IF i<r-(r/2) THEN j:=0 ELSE j:=i-ROUND(r-(r/3));
    FILLELLIPSE(a,b,i,j);
    IF i<r-(r/2) THEN j:=0 ELSE j:=i-ROUND(r-(r/3));
    FILLELLIPSE(a,b,j,i);
  END;
  FILLELLIPSE(a,b,r,ROUND(r/2));
  FILLELLIPSE(a,b,ROUND(r/2),i);
  SETCOLOR(c);
END;
PROCEDURE valt(VAR t:ttip);
VAR g:INTEGER;
BEGIN
  g:=ROUND(RANDOM*db)+1;
  c:=GETPIXEL(t[g,1],t[g,2]);
  PUTPIXEL(t[g,1],t[g,2],0);
  csillag(t[g,1],t[g,2],c);
  t[g,1]:=RANDOM(GETMAXX);
  t[g,2]:=RANDOM(GETMAXY);
  t[g,3]:=RANDOM(GETMAXCOLOR);
  DELAY(100);
```

```
END;
BEGIN
  DETECTGRAPH(dm,dg);
  INITGRAPH(dm,dg,'');
  cs;
  inic(t);
  kirak(t);
  REPEAT
    valt(t);
    kirak(t);
  UNTIL KEYPRESSED;
  CLOSEGRAPH;
END.
```

2.13.4.2. Feladatok

1. Írjunk demoprogramot, mely megmutatja, mit tudunk. Varázsoljunk „csillagos égboltot" képernyőnkre.

2. Írjunk járó órát megvalósító programot. Az idő kijelzését megoldhatjuk analóg módon vagy digitálisan is.

3. Rajzolja meg az ötkarikás olimpiai jelvényt. A színezésre is gondoljon.

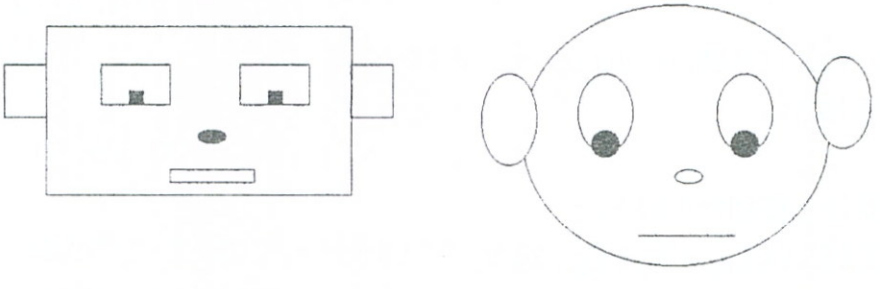
4. Rajzoljon robotembert.

5. Grafikus képernyőn kérje be a kör sugarát, majd írja ki kerületét, területét. Rajzzal szemléltesse a feladatot.

6. Kérje be grafikus képernyőn egy téglalap oldalainak hosszát, majd rajzolja ki úgy a téglalapot, hogy középpontja a képernyő közepe legyen.

7. Készítsen képernyő-pihentető programot. A program adjon hangot is futás közben.

8. Rajzoljon grafikus képernyőre a látható ábrákhoz hasonló fejeket.



9. Rajzoljon a grafikus képernyőre pirossal lehető legnagyobb keretet, majd a belsejét fesse zöldre. Húzza meg a téglalap mindkét átlóját is pirossal.

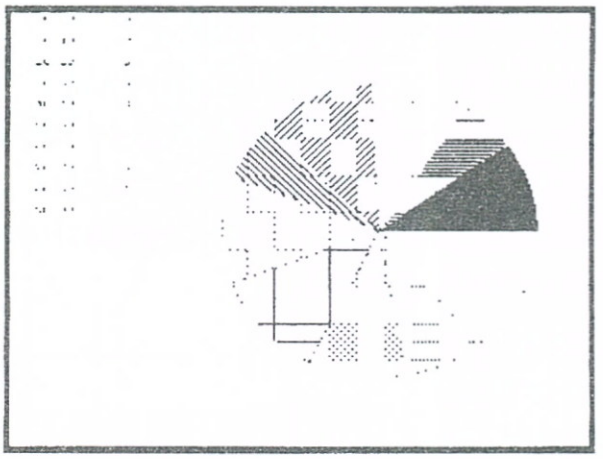
10. Rajzolja ki a grafikus képernyőre a színes dobókocka egy lapját.

11. Rajzoljon Wenn-diagramot, amivel a valós számok halmazát mutatja be.

12. Mutassa be diagramon a Pascal nyelv adattípusait.

13. Készítsen olyan programot, amivel a billentyűzettel rajzolni lehet. Törekedjen arra, hogy a program jól használható legyen.

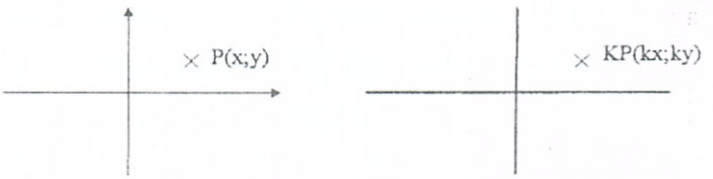
14. Bemenő adatokból (lehet generált is) készítsen jól áttekinthető diagramokat (kör-, oszlop-, vonaldiagram stb.).



2.13.5. Függvényábrázolás

2.13.5.1. Torzítás nélkül

Ez esetben a Descartes-féle koordinátarendszer két szomszédos rácsponti pontja a képernyő két szomszédos pontjának felel meg. Egy kölcsönösen egyértelmű hozzárendelést kell megvalósítanunk. Az origó a képernyő közepére kerüljön.



A P(0, 0)-hoz a KP(GETMAXX DIV 2, GETMAXY DIV 2) tartozik a feltételeknek megfelelően. Ezek alapján a levezetési szabály a következő:

P(x, y) ⇒ KP(GETMAXX DIV 2+x, GETMAXY DIV 2+y),

azaz

kx= GETMAXX DIV 2+x és ky= GETMAXY DIV 2+y.

Az egyszerűség kedvéért jelölje a képernyőre rajzolást a PLOT(kx,ky) eljárás, ami a KP(kx, ky) képernyőbeli pont kigyújtását végzi. Lássuk az algoritmust!

```
CIKLUS kx:=0 TO GETMAXX
  kx ⇒ x
  x ⇒ F(x)
  F(x) ⇒ ky
  PLOT(kx,ky)
CIKLUS VÉGE.
```

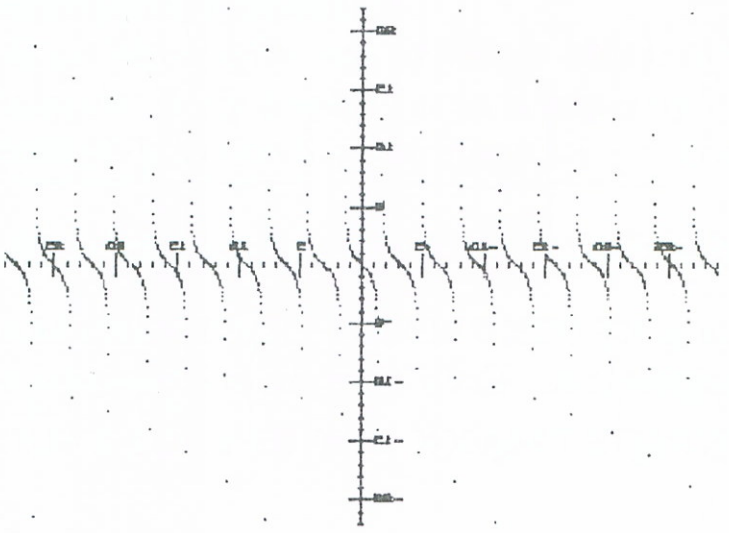
Először kiszámoljuk kx-hez a Descartes-féle koordinátarendszerbeli x-et, ezután az F függvény értékét az x helyen, majd F(x)-hez kiszámítjuk a képernyőbeli ky-t. Végül kirajzoltatjuk a pontot.

Innentől kezdve csak a kódolás van hátra.

Próbálja ki a programot a SIN(x) függvényre is. Ne ijedjen meg. A program jól működik, de a függvény értékkészlete olyan „kicsi", hogy a monitoron egy egységet szinte észre sem veszünk. Ezért érdemes rajzunkat felnagyítani. Ehhez az előbbi algoritmust kell kiegészíteni a következő módon, ahol T az arányossági tényezőt jelenti:

```
CIKLUS kx:= 0 TO GETMAXX
  kx ⇒ x   x:=x*T  x ⇒ F(x)
  F(x):=F(x)/T   F(x) ⇒ ky
PLOT(kx,ky)
CIKLUS VÉGE.
```

Most már megírhatjuk a programot. Ügyeljünk a változók kompatibilitására! Rajzoljuk meg a tengelyeken az egységeket is. Próbáljunk minél érdekesebb függvényt megadni, és megkeresni hozzá a legjobb arányossági tényezőt.



2.13.5.2. Függvényrajzolás az $[a, b]$ intervallumon

Az alábbiakban más oldalról közelítjük meg a függvényábrázolás kérdését. Egy $[a, b]$ intervallumon folytonos függvényt szeretnénk ábrázolni.

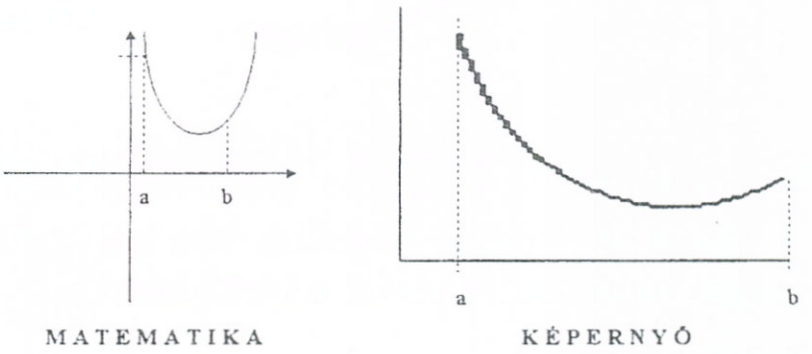
Elvárások:

— a függvény teljes $[a, b]$-beli gráfja látható legyen,

— mindig legyenek láthatók a koordinátatengelyek,

— a gráf maximálisan töltse ki a képernyőt. A kép tetején legyen a legnagyobb függvényérték, a $\max f([a, b])$, alul a legkisebb, a $\min f([a, b])$. Ha $\min f([a, b]) > 0$, akkor alul legyen az x tengely, ha pedig $\max f([a, b]) < 0$, akkor felül. Ha $a > 0$ és $b > 0$, akkor a képernyő bal oldalán legyen az y tengely, ha pedig mindkettő negatív, akkor a jobb oldalán.

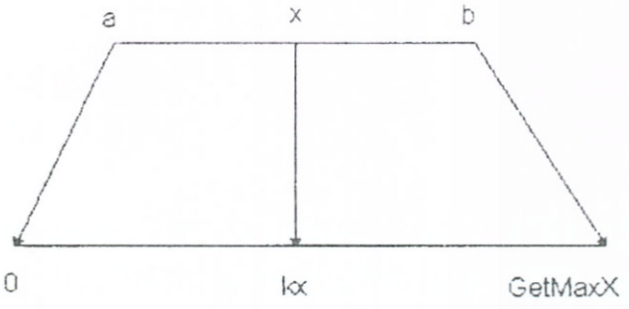
Talán ábra segítségével könnyebb lesz.



A megoldás menete a következő:

1. Számoljuk ki $\max f([a, b])$ és $\min f([a, b])$ értékét. Ez egy egyszerű keresés, egy speciális értékadással az elején. Ha MIN jelöli a $\min f([a, b])$-t, MAX a $\max f([a, b])$-t, akkor MIN:=0, MAX:=0 kezdőértékekkel biztosítjuk azt, hogy az x tengely mindig benne lesz a képben.

2. Ezután a transzformációs képlet meghatározása következik. Ez azt jelenti, hogy a kx $\Rightarrow$ x hozzárendelést kell megvalósítani. Az a-hoz a képernyőn a 0 tartozik, b-hez a GETMAXX.



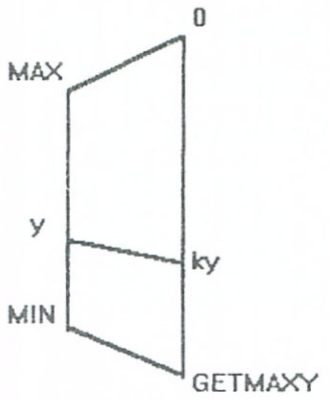
A párhuzamos szelők tétele alapján könnyen meghatározható a transzformációs képlet:

$$(x - a) : (kx - 0) = (b - x) : (GETMAXX - kx)$$

Miután ebben a képletben csak x az ismeretlen, könnyedén kifejezhető,

és a kx ⇒ x transzformáció készen is van.

Hasonló ábrával az y ⇒ ky is megoldható.



$$(y - MAX) : (MIN - MAX) = (ky - 0) : (GETMAXY - 0)$$

3. Már csak a vázolt algoritmus kódolása van hátra. Ha az előző függvényes feladatot megoldotta, akkor ezt is sikerül kódolni.

```
PROGRAM fuggveny;
USES CRT,GRAPH;
VAR dg,dm:INTEGER;
  min,max:REAL;
  a1,a,b:LONGINT;
FUNCTION fx(x:REAL):REAL;
BEGIN
  fx:=x*sin(x);
  END;
PROCEDURE kiv(VAR min,max:REAL;a,b:LONGINT);
VAR i,d:REAL;
  BEGIN
    i:=a;
    d:=(b-a)/GETMAXX;
    WHILE i<=b DO
    BEGIN
    IF fx(i)<min THEN min:=fx(i);
    IF fx(i)>max THEN max:=fx(i);
    i:=i+d;
    END;
  END;
FUNCTION kxx(kx:LONGINT):REAL;
  BEGIN
    kxx:=(b-a)*kx/GETMAXX+a;
```

```
  END;
FUNCTION yky(y:REAL):LONGINT;
  BEGIN
    yky:=GETMAXY-ROUND((y-min)*GETMAXY/(max-min));
  END;
FUNCTION xkx(x:REAL):LONGINT;
  BEGIN
    xkx:=ROUND((x-a)*GETMAXX/(b-a));
  END;
PROCEDURE yteng;
VAR y,i:LONGINT;
  BEGIN
    y:=xkx(0);
    FOR i:=0 TO GETMAXY DO PUTPIXEL(y,i,3);
  END;
PROCEDURE xteng;
VAR y,i:LONGINT;
  BEGIN
    y:=yky(0);
    FOR i:=0 TO GETMAXX DO PUTPIXEL(i,y,3);
    LINE(xkx(a1),yky(0)-10,xkx(a1),yky(0)+10);
    LINE(xkx(b),yky(0)-10,xkx(b),yky(0)+10);
  END;
PROCEDURE rajzol;
VAR i:LONGINT;
  BEGIN
    FOR i:=0 TO GETMAXX DO PUTPIXEL(i,yky(fx(kxx(i))),15);
  END;
  BEGIN
  CLRSCR;
  WRITE('a: ');READLN(a);
  WRITE('b: ');READLN(b);
  a1:=a;
  DETECTGRAPH(dg,dm); INITGRAPH(dg,dm,'c:\TP\bgi');
  IF (a>0)AND(b>0) THEN a:=0;
  IF (a<0)AND(b<0) THEN b:=0;
  MIN:=0; MAX:=0;
  KIV(MIN,MAX,A,B); XTENG; YTENG; RAJZOL;
  READLN;
  CLOSEGRAPH;
END.
```

2.13.5.3. Feladatok

1. Készítsen programot, amely meghatározza az $[a, b]$ intervallumon folytonos, nem negatív függvény Riemann-integrálját (görbe alatti területét) a Monte-Carlo-módszerrel. Programja szemléltesse a módszert ([11]).

2. Programunk vizsgálja meg, hogy milyen kapcsolat van egy iskolai osztályban a tanulók hiányzási óráinak száma és a tanulók tanulmányi átlaga között. Rajzolja fel a program a ponthalmazt és a regressziós egyenest is ([11]).

3. A grafikáknál fontos tényező a nyomtatás kérdése. A Turbo Pascal nem tud grafikát nyomtatni magától. A WINDOWS tud segíteni ezen a problémánkon. Ha WINDOWS alól indítjuk el a Pascal programot, akkor megoldhatjuk ezen problémát is. Amikor a grafika megjelent a képernyőn, a Print Screen billentyű lenyomásával a vágólapra kerül a kép, amit már például a PAINTBRUSH programmal nagyszerűen alakítgathatunk. Szöveges állományba is elhelyezhetjük a grafikont, és ki is nyomtathatjuk.

4. Készítsen olyan programot, amely segítségével az első- és másodfokú polinomok transzformációit szemlélteti.

5. Készítsen az $[a, b]$ intervallumon folytonos f függvényhez az $f(x) = 0$ egyenletet megoldó programot az intervallumfelezés módszerével. Szemléltesse a megoldást grafikonnal is.

6. Programja szemléltesse a határozott integrál közelítését a téglalap-módszerrel.

7. A program szemléltesse a határozott integrál közelítését a trapézmódszerrel.

2.14. Adatállományok

2.14.1. A fájl fogalma, felépítése

A világ leggyorsabb számítógépe sem érne sokat, ha nem tudná az adatokat valamilyen külső adathordozón tárolni. A könyvünkben lévő feladatoknál is kényelmetlen volt néha, hogy az egyszer beolvasott adathalmazt a továbbiakban programunk vagy programjaink nem használhatták. Programjaink segítségével a tárban elhelyezett adatok — a következő program indításával vagy a számítógép kikapcsolásával — elvesztek. Ezen a problémakörön segítenek a számítógép háttértárolóin tárolt adatállományok.

Az adatállomány helyett a szakmai terminológia igen gyakran „file"-t mond ([22]). Az angol szót magyarul fájlnak mondjuk. A magyar helyesírási szabályok szerint írni is így kell. A fájl valamilyen szempontból összetartozó egyedek összessége. Ilyen halmaz lehet például egy város telefonnal ellátott lakóinak összessége, vagy egy iskola, egy osztály tanulóinak halmaza.

A telefonkönyvben egy előfizetőről nyilvántartott adatok rekordot alkotnak. Ezek az adatok például a következők: az előfizető neve, címe foglalkozása, telefonszáma. A rekord mezői tehát az előfizetők halmazának egy elemét határozzák meg.

A fájl a rekordoknak a nyilvántartás, a hozzáférés és a megőrzés szempontjából kialakított összessége ([26]). A fájl általában nagy méretű, és ezért rendszerint háttértárban tároljuk. Az adatállománynak a külső táron szerkezete van. Ez azt jelenti, hogy az állományba tartozó mondatok (rekordok) valamilyen meghatározott rendben találhatók az adattárban ([22]). Az elhelyezés figyelembe veheti a mondatazonosítót (pl.: Személy, Tanuló, Rajzszám, Árukód) is. Az adatállományok kezelése a mondatazonosítónak és a mondatnak az adattárban való elhelyezése közötti összefüggést jelenti.

Az adatszerkezet a mondatok és adatainak logikai összefüggését jelenti, a **tárolási szerkezet** pedig a mondatok fizikai elhelyezését az adattárban.

Az adatszerkezet szempontjából beszélhetünk **egyszerű** és **összetett állományszerkezetről.**

Az egyszerű állományokban a keresés csak a rekordok mezőinek a segítségével is megoldható. Az állomány szerkezetére vonatkozó egyéb tárolt információkra nincs szükség. Az egyszerű állományok lehetnek: szeriális (soros), szekvenciális (sorrendi), direkt, random állományok.

Az összetett állományszerkezetek rekordjaiban a tényleges adatokon túl más segédinformációk is vannak, az úgynevezett szerkezethordozó adatok. Ezek a fizikai állománynak a háttértáron való elhelyezésére vonatkozó adatokat tartalmazzák. Az összetett állomány is lehet: szeriális, szekvenciális, direkt vagy random. A Pascal nyelv összetett állományszerkezeteket nem ismer. Ha ilyen állományokat kívánunk létrehozni vagy kezelni, akkor azt a programban nekünk kell megvalósítani. Összetett állományszerkezet megvalósítására alapvetően kétféle technika létezik: a láncolás és az indexelés.

Szalagok (lyukszalag, mágnesszalag) esetén a tárolási szerkezet és a visszakeresés csak **soros** (szeriális) lehet, ugyanis a szalagon az adatok csak egymás után (sorosan) helyezhetők el, és ugyanígy kereshetők vissza.

A soros elhelyezés lehet rendezetlen és rendezett (sorrendi, szekvenciális). Mágneslemezen a tárolási szerkezet és a visszakeresés szempontjából a következő adatállomány-szervezési módszereket ismerjük ([22]):

— soros,
— indexszekvenciális,
— random.

Soros szervezési módszernél a mondatazonosító (rekordnév) és a mondatnak az adattárbeli elhelyezkedése között nincs összefüggés. Ez azt jelenti, hogy az adatállomány elejétől kezdődően minden rekordot be kell vinni a

központi egységbe, amíg a keresett rekordot megtaláljuk.

A rekordok egymás után következő sorrendje lehet rendezetlen vagy rendezett.

A rendezett szeriális állományoknál a rekordok az azonosító sorrendjében (növekvő vagy csökkenő) következnek egymás után. Az ilyen szervezésű állományokat szoktuk **sorrendi** vagy **szekvenciális állományoknak** nevezni. A rekordhoz való hozzáférés csak a rendezettségből adódó sorrendben lehetséges.

Soros állományokra jellemző, hogy új rekordok belépése vagy törlése esetén az egész állományt új adattárba kell átírni.

Soros állomány lehet lyukszalagon, lyukkártyán, mágneslemezen és mágnesszalagon is. Ez a tárolási módszer a helykihasználás szempontjából gazdaságos, mert a rekordok hézagmentesen követhetik egymást egymás után.

Indexszekvenciális módszer esetén a rekordok az azonosító szempontjából rendezetten helyezkednek el. Ehhez a rendezett állományhoz azonban egy indextábla (keresési táblázat) tartozik, amelyik megmutatja, hogy a mondatazonosító szerint a mondat a mágneslemez melyik részén található ([22]). Ebben az esetben **közvetlen hozzáférésről** beszélünk.

A **random adatállomány-kezelési módszer** legfőbb jellemzője, hogy a mondat azonosítója és a mágneslemezen található helyének címe között nincs kölcsönösen egyértelmű hozzárendelés. Nagy méretű állományokban gyakran előfordul, hogy több rekord is ugyanarra a helyre kerülne. Ezért bonyolult matematikai összefüggések és eljárások szükségesek a cím kiszámításához ([21]).

2.14.2. Fájlok a Turbo Pascal nyelvben

A Turbo Pascal fájljai (a dBase állományai is) **alapvetően szeriális fájlok**, de a rendszer minden rekordhoz egy sorszámot, ún. **rekordmutatót** is rendel. Ez a sorszám nem a rekord azonosítójából keletkezik, hanem abból a sorrendből, ahogyan felvittük őket egymás után. Ezért úgy kezelhetjük az állományt, mintha direkt lenne, hiszen a rekordmutató alapján tetszőleges rekordra állhatunk rá. Ezeket az állományokat **kvázidirekt állományoknak** is szokták nevezni ([21]).

Fizikai fájlnak nevezzük a fizikai névvel ellátott és háttértárolón elhelyezett adatok összességét. Az állomány fizikai neve az a karakterlánc, amely karakterlánccal megnevezve a háttértárolón az állományt tároltuk. A fizikai állomány alapegysége a fizikai rekord. A fizikai rekord elhelyezkedésének az alapegysége a blokk. A fizikai rekord nem feltétlenül fedi le a logikai rekordot. Előfordulhat, hogy egy blokkban több logikai rekord kerül tárolásra, de az is lehet, hogy egy rekord több blokkban helyezkedik el.

A logikai fájl az a név, amely a fájlt képviseli a programban, tehát programjainkban az állomány logikai nevével dolgozunk. A fájl fizikai neve és logika neve egymással összerendelt fogalmak. A logikai fájl tartalmazza azokat az információkat is, amikkel elérhetők az adatállomány alkotóelemei, komponensei. Ezek a komponensek gyakran rekordok, de lehetnek elemi adatok is. A rekord mezői és az elemi adatok is önálló névvel rendelkeznek. Azt a mezőt, amelynek értéke minden rekordban más és más, szokták elsődleges kulcsnak is nevezni. Ilyen elsődleges kulcs lehet például a személyi szám vagy az árukód. A többi mezőt másodlagos kulcsnak nevezzük.

A programban a fizikai fájl adataival dolgozni szeretnénk. Ezért először a fizikai fájlt hozzárendeljük a logikai fájlhoz. Ezt a műveletet a fájl létrehozásának is szokták nevezni ([9]).

A puffer speciális tárolóterület, ahol az információ ideiglenesen tárolódik. Feladata az, hogy kiegyenlítse a memória és a periféria közötti sebességek különbségét. Amikor feldolgoztunk egy rekordot, nem a háttértárra, hanem a pufferbe kerül. Ha megtelik a puffer, azaz összegyűlik annyi logikai rekord, mint amilyen a puffer mérete, akkor kerül a háttértárolóra. Előfordulhat azonban az is, hogy nem telik meg a puffer a rekordokkal, és most egy műszaki hiba miatt megszakad a program futása. Ekkor az aktuális rekord elveszett.

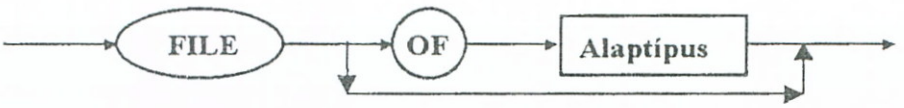
Az állománykezelés klasszikus műveletei:

— létrehozás,
— megnyitás,
— bővítés,
— keresés,
— törlés,
— módosítás,
— lezárás.

Az adatállományokkal kapcsolatosan a következő alapvető műveletekről szoktunk beszélni:

Az állományműveletek konkrét megvalósítása függ a programnyelvtől és az állománytípustól is ([21]). A Turbo Pascal nyelvben a klasszikus állománykezelésnek csak a csíráit találjuk. Minden műveletet a programozónak kell megvalósítania. Csak hozzáférések adottak: a közel fizikai szintű, soros és kvázi direkt. A következő részben a Turbo Pascal nyelv fájljait és az azokhoz kapcsolódó eljárásokat, függvényeket fogjuk áttekinteni. A TP nyelv tanulása során erre fektesse a fő hangsúlyt. A típusos fájloknál részletesebben foglalkozunk a klasszikus fájlműveletek megvalósításával is annak érdekében, hogy egyszerűbb adatfeldolgozással kapcsolatos feladatokat is meg tudjunk valósítani.

Az adatállományok megvalósítására a Turbo Pascal nyelv a FILE adattípust használja. A **fájltípus** olyan adattípus, amely meghatározatlan számú azonos típusú és/vagy méretű komponensek sora ([24]). A korábbi „rekord" szóhasználatról azért tértünk át a komponens szóra, mert a Turbo Pascal nyelvben a rekord megnevezéssel inkább az adattípusra utalnánk, a komponens azonban nem feltétlenül rekord típusú.



A diagram is mutatja, hogy a komponensek típusát nem mindig jelöljük. Ekkor nem tipizált fájlokról beszélünk.

A Turbo Pascal ismer egy standard fájltípust is, aminek a típusneve TEXT.

A program mindig logikai fájlt kezel, annak nevével dolgozik. Ezt a nevet deklarálni szükséges a program **deklarációs** részében.

A logikai állományt hozzá kell rendelni a fizikai állományhoz. Ezt az összerendelést az ASSIGN eljárás végzi.

A logikai fájllal még ekkor sem dolgozhatunk, mert előbb meg kell nyitni a fájlt. A **megnyitás** történhet: írásra, olvasásra. Az olvasás, illetve írás egysége a komponens. A komponenseknek egyértelmű sorszámuk (pozíciójuk, mutatójuk) van. Az elérés általában soros. Ez azt jelenti, hogy az I/O művelet az aktuális mutatóra vonatkozik, és a művelet végrehajtása után +1 értékkel növekszik. Erről maga a rendszer gondoskodik. Az elérés a komponensek sorszáma szerint direkt módon is történhet. Ekkor a komponens sorszáma alapján állítódik be az aktuális mutató. Írás a fájlba azt jelenti, hogy egy komponest a memóriából a fájlba helyezünk a mutató által mutatott helyre. Olvasás esetén a mutató által mutatott helyről egy komponens a memóriába kerül. Az I/O eljárások mindig egy komponensre vonatkoznak.

Most már **dolgozhatunk** a fájllal. Alkalmazhatjuk azokat az eljárásokat és függvényeket, amelyeket az aktuális fájlban a nyelv megenged a feldolgozás érdekében.

A feldolgozó lépések befejeztével a fájlt **le kell zárni**. Ez az eljárás nagyon fontos, mert ennek hatására ürül a puffer is, és így adataink nem vesznek el.

A Turbo Pascalban a fájlokat három csoportba osztjuk:
— tipizált fájlok,
— szöveges fájlok,
— nem tipizált fájlok.

2.14.2.1. Tipizált fájlok

A tipizált fájloknál jelöljük az állomány komponenseinek a típusát. Minden komponensnek azonos a típusa. Ez az alaptípus a jellemzője az állománynak is. A tipizált fájltípus szintaxisdiagramja a következő:

Az alaptípus lehet bármely egyszerű és strukturált típus, csak fájltípus nem. A komponensek száma elvileg korlátlan, annak csak a lemez kapacitása szab határt. Hogy megértsük egy fájl felépítését, vegyünk egy példát. Szeretnénk ismerőseinknek a címét nyilvántartani. Ennek érdekében alakítsunk ki egy fájlszerkezetet. A fájl komponensei most rekordtípusúak lesznek. A rekord neve legyen adatrec. A rekord mezőkből áll. Példánkban szükségünk lesz egy névmezőre, amibe az ismerős nevét írjuk. Továbbá kell város, utca irányítószám, telefonszám, egyéb mező.

Munka tipizált fájlokkal

Ebben a fejezetben azokat a követendő lépéseket szedjük pontokba a konkrét példa és a Turbo Pascal alapján, amelyeket a fájlokkal kapcsolatos munkák során követnünk kell. A felsorolt pontok lényegében mindenféle fájl kezelésénél követhetők a megfelelő sajátosságok figyelembevételével.

1. Deklaráljuk a rekord típusát. Arról a rekordról van szó, amely a fájl alkotóeleme. Vázolt példánk szerint ez így nézhet ki:

```
TYPE tadatrec=RECORD
     nev:STRING[30];
     varos:STRING[20];
     utca:STRING[30];
     irszam:word;
     tel:STRING[13];
     megj:STRING[30];
   END;
```

2. Szükséges a rekord változójának és magának a logikai fájlnévnek a deklarálása. A fájl logikai nevét úgy deklaráljuk, mint a változókat (a VAR kulcsszóval), típusát komponenseinek típusa adja: VAR NEV: FILE OF elemtípus.

A programban ezzel a névvel hivatkozunk a fájlra. Példánkban:

```
VAR adatrec:tadatrec;
adatfile:FILE OF tadatrec;
```

3. Mivel a lemezes, fizikai fájllal kívánunk dolgozni, ezért a fájl logikai nevét és a fizikai fájlt összerendeljük. Ezt az ASSIGN(VAR logikainev;fizikainev:string) eljárással tehetjük meg. A fizikai név egy string. A karakterláncban használhatunk meghajtónevet, elérési utat is, ugyanúgy, ahogyan azt a DOS-ban megszoktuk. Az eljárás redundáns módon is meghívható.

4. Ha megtörtént az összerendelés, akkor megnyitjuk a fájlt. Ezt a RESET(VAR logikainev) vagy a REWRITE(VAR logikainev) eljárással tehetjük.

A REWRITE(logikainev) egy üres fájlt hoz létre írásra, és a fájlmutatót 0-ra állítja. Az üres fájl mutatója 0. Ha már létezett a fájl, akkor azt törli, és újat hoz létre.

A RESET(logikainev) már meglévő fájlt nyit meg írásra és olvasásra is, és a mutatót 0-ra állítja. Ha még nem létezik a fájl vagy nem érhető el, akkor hibaüzenetet kapunk.

5. Ha a fájlt megnyitottuk, akkor következhetnek a feldolgozó lépések. Hogyan írhatunk típusos fájlba, illetve hogyan olvashatunk be típusos fájlból?

Az írás a fájlba a WRITE eljárással történik, a következő módon: WRITE(logikainev,V1[,V2,...]).

Az fájlból történő olvasás is hasonlóan történik a READ eljárással: READ(logikainev,V1[,V2,...]).

Mindkét eljárásban a változó(k) típusának és a fájl alaptípusának egyeznie kell. Az eljárások az aktuális mutatóra vonatkoznak, és minden változó kiírása, illetve beolvasása esetén a mutató eddigi értékéhez +1 adódik anélkül, hogy azon mi változtatnánk.

Az EOF(VAR logikainev):BOOLEAN függvénnyel megtudhatjuk, hogy a fájl végén vagyunk-e már. E függvény értéke TRUE, ha már a fájl végére értünk, és FALSE, ha nem. A FILEPOS(VAR logikainev):LONGINT függvénnyel a mutató aktuális értékét kaphatjuk meg. A SEEK(VAR logikainev;pozicio:LONGINT) eljárással mi is beállíthatjuk a mutatót a kívánt értékre.

A FILESIZE(VAR logikainev): LONGINT függvénnyel a fájl méretét, azaz a rekordok számát kaphatjuk meg.

A TRUNCATE(VAR logikainev) eljárás a nyitott logikainev nevű fájlt csonkítja, vagyis a fájl rekordjait törli a fájlmutató aktuális értékétől kezdődően.

6. A feldolgozás végén a korábban megnyitott állományokat le kell zárni. Ezt a CLOSE(VAR logikainev) eljárással tehetjük meg. Megjegyezzük, hogy a rendszer — programunk futásának befejeztével — az összes nyitva felejtett fájlt lezárja, de ajánlatos nekünk gondoskodni a lezárásról a

CLOSE eljárással. Fontos dolog, hogy a lemezre írás egy pufferen keresztül történik. Ha írunk egy fájlba, vagy olvasunk egy fájlból, akkor nem biztos, hogy normálisan befejeződött már a transzfer. Ez gond lehet egy esetleges áramszünetnél is. Ezért ha huzamosabb ideig nem használjuk a fájlt, akkor zárjuk le! A lezárás üríti a puffert is.

7. Lezárt fájlokra alkalmazhatjuk a következő eljárásokat:

Az ERASE(VAR logikainev) eljárás törli a lezárt fájlt a lemez FAT táblájából, ezért ez a fájl a továbbiakban nem érhető el.

A RENAME(VAR logikainev; 'újfiznév') eljárás átnevezi a lezárt fájlt új fizikai névre. Külön figyelmet érdemel, hogy első paraméterként az átnevezendő fájlnak a programban használt logikai nevét kell megadni, de a második paraméter egy string érték legyen. Ez a string lesz a fájl fizikai neve a lemezen a továbbiakban.

Ezzel felsoroltuk azokat a lépéseket, eljárásokat és függvényeket, amelyeket a fájlokkal kapcsolatos munka során alkalmazhatunk. Úgy gondoljuk, hogy a fájlkezelés nem tartozik a programozás legegyszerűbb témái közé, viszont segítségével valóban életszerű feladatokat oldhatunk meg valóban életszerű módon. Megoldott vagy vázolt példáink jobbára csak modellként szerepelnek, ezért kérjük, hogy a modelleket életszerű problémák megoldásával helyettesítse.

A következő részben az algoritmusok gyakorlása és a felsorolt eljárások, függvények lényegének jobb megértése érdekében elemezzünk egy egyszerű feladatot.

A feladatot két módon is elemezzük. Szalagos háttértárolók esetén a fájl egyes rekordjainak közvetlen elérésére nem volt módunk, hiszen ott csak azt tudtuk a rekordokról, hogy a szalagon fizikailag követik egymást. A fájlokat ekkor csak sorosan kezelhettük. Lemezes háttértárolók esetén a rekordokat a rendszer már jobban tudja adminisztrálni. A rekord deklarációjából tudja annak lemezen lefoglalt hosszát, így a rekord sorszámának ismeretében közvetlenül (direkt) is eléri a lemezen a fájl kívánt rekordját. Feladatunkban kövessük végig mindkét hozzáférési módszerrel a fájl kezelését.

2.14.2.1.1. Soros fájlkezelés

Soros fájlkezelés esetén a fájlkezelés lépései annyiban térnek el a direkt elérés lépéseitől, hogy feldolgozó lépésként a mutatóra vonatkozó információk nem állnak rendelkezésünkre, illetve ezeket az eljárásokat és függvényeket nem szabad használni, mert ezek az eszközök a soros kezelésnél még nem léteztek. Az I/O műveleteket természetesen a READ és a WRITE eljárások valósítják meg. A rendszer maga kezeli a fájl mutatóját úgy, hogy minden I/O művelet esetén 1 adódik hozzá. Tehát a soros elérésnél mindig csak előre haladhatunk. Egy rekordot csak úgy tudunk keresni, hogy a fájlt

megnyitjuk, és elemeit újból végignézzük addig, amíg a keresett rekordot megtaláljuk, vagy vége van a fájlnak. Egy újabb rekordot csak úgy tudunk felkeresni, hogy a fájlt lezárjuk, majd újból megnyitjuk, és a keresést újból kezdjük.

Soros kezelésnél a fájlmutatóval és a fájl rekordjainak számával nem operálhatunk. Csak az EOF(Fajl) logikai függvényt kérdezhetjük le, vagyis csak azt tudjuk eldönteni, hogy vége van-e már a fájlnak. Soros állománykezeléskor mindig arra gondoljunk, hogy szalagnál hogyan oldanánk meg a problémát.

Viszont **alkalmazhatjuk fájlok törlését és az átnevezést.** Törölni és átnevezni csak lezárt fájlokat lehet.

Törlés: ERASE(Fajlnev)

Átnevezés: RENAME(Fajlnev,'újfiznév').

Kidolgozott feladat szeriális állományra

A szeriális (soros) állomány a legegyszerűbb állomány. Fájlszerkezete azt jelenti, hogy a komponensek minden összefüggés nélkül követik egymást. A komponensnév és a komponens fizikai helye között nincs összefüggés. Szeriális állományokat a fájl soros elérésével kezelhetünk. A feladat az, hogy készítsük el a korábban megadott rekordszerkezettel nyilvántartó programunkat.

A program menüjéből a következő lehetőségeket választhassuk:

1. Adatfájl létrehozása
2. Fájl bővítése
3. Személy keresése
4. Törlés
5. Módosítás
6. Összefűzés
7. Gyors lista
8. Kilépés

A program:

```
Program szerialis_soros_kezeles;
```

Programunk típusainak és globális változóinak deklarációja a következő:

```
TYPE tadatrec=RECORD
        nev:STRING[30];
        varos:STRING[20];
        utca:STRING[30];
        irszam:word;
        tel:STRING[13];
```

```
        megj:STRING[30];
      END;
  fajlnevtip:string;
VAR adatfile:file of tadatrec;
  valaszt:byte;
  fajlnev:fajlnevtip; adatrec:tadatrec;
```

Ezek után nézzük azokat az algoritmusokat, amelyeket a program menüjéhez megvalósítunk.

A **fájl létrehozása** csak azt jelenti, hogy létrehozunk egy üres fájlt. A fájlt megnyitása után azonnal le is zárjuk.

```
Procedure letrehoz(fajlnev:fajlnevtip);
begin
  assign(adatfile,fajlnev);
  rewrite(adatfile);
  close(adatfile);
end;
```

Célszerű megoldani azt a praktikus ellenőrzést is, hogy programunk megnézze: van-e már ilyen adatállomány a lemezen, és ha nincs, akkor létrehozza azt. Ilyenkor ki kell kapcsolni a fájl hibakezelést. Ezt a {$I} lokális direktíva kikapcsolásával tehetjük meg. A direktíva kikapcsolása azt jelenti, hogy I/O műveleteknél bekövetkezett hiba esetén a program futása nem áll le, csupán a hiba bekövetkezésének regisztrálása történik meg az IOResult függvényben. Az **IORESULT** függvény adja a hiba kódját. Ha értéke nulla, akkor nem történt hiba. Más értékek esetén hiba történt a megnyitásnál, és természetesen nincs a fájl megnyitva. Az **IORESULT** lekérdezése után értéke automatikusan törlődik, vagyis ismét 0 lesz.

```
{$I-}
RESET(adatfile)
{$I+}
IF IORESULT<>0 THEN REWRITE(adatfile);
...
```

A fájl bővítése azt jelenti, hogy most olvassuk be a billentyűzetről a rekordokat, és helyezzük el azokat a fájl végén. A fájl végére úgy jutunk, hogy a fájlból a rekordokat beolvassuk, majd eldobjuk azokat. A rekordok beolvasása addig tart, amíg a NEV mezőnek végjelet nem adunk. Programunkban a végjel az üres név, vagyis az Enter leütése. Minden rekord beolvasása után kiírjuk a rekordot a fájlba, és beolvassuk a billentyűzetről a következő rekord NEV mezőjének értékét.

```
procedure bovit(fajlnev:fajlnevtip);
```

```
var adat:tadatrec;
begin
  assign(adatfile,fajlnev); reset(adatfile);
  while not EOF(adatfile) do read(adatfajl,adat);
  with adat do
  begin
    write('Név:');readln(nev);
    while nev<>'' do
    begin
    write('Város:'); readln(varos);
      write('Utca, házszám:');readln(utca);
      write('Irányítószám:'); readln(irszam);
      write('Telefon:'); readln(tel);
      write('Egyéb megjegyzés:'); readln(megj);
      write(adatfile,adat);
      write('Név:');readln(nev);
    end;
  end; {with} close(adatfile);
end; {bovit}
```

A keresés egyszerű lineáris keresés lesz. A fájl rekordjait addig kell beolvasni, amíg nem találtuk meg a keresettet, és nincs vége a fájlnak. Más keresési algoritmust most nem is alkalmazhatunk, hiszen a fájl rekordjai nem rendezettek. Név szerint keressük a személyt. A keresett egyének nevének megadását most is végjelig folytatjuk. A végjel most is az Enter. Az egyes nevek keresését addig végezzük, amíg a fájl végére nem értünk, vagy meg nem találtuk az egyént. Ha megtaláltuk a keresett egyént, akkor kiírjuk teljes rekordját. Ha nem található az egyén a fájlban, akkor azt is jelezzük.

```
procedure keres(fajlnev:fajlnevtip);
var adat:tadatrec;
  kit:string;
begin
  assign(adatfile,fajlnev); write('Kit keressek?:'); readln(kit);
  while kit<>'' do
  begin
  reset(adatfile); adat.nev:='';
  while not eof(adatfile) and (adat.nev<>kit) do
            read(adatfile,adat);
  if adat.nev=kit then
    begin
      writeln('Név:',adat.nev); writeln('Város:',adat.varos);
```

```
        writeln('Utca és házszám:',adat.utca);
        writeln('Irányítószám:',adat.irszam);
        writeln('Telefon:',adat.tel);
        writeln('Egyéb megjegyzés:',adat.megj);
      end
      else writeln('Nem találtam ilyen nevet');
    close(adatfile); write('Kit keressek:');readln(kit);
    end;
end;{keres}
```

Rekord törlése a fájlból soros kezelésnél úgy történik, hogy minden rekordot átmásolunk egy új fájlba, csak a törlendő rekordot nem.

```
procedure torol(fajlnev:fajlnevtip;kit:tadatrec);
var adat:tadatrec; ujfajl:file of tadatrec;
begin
  assign(adatfile,fajlnev); reset(adatfile);
  assign(ujfajl,'uj'); rewrite(ujfajl);
  while not eof(adatfile) do
  begin read(adatfile,adat);
    if adat.nev<>kit.nev then Write(ujfajl,adat);
  end;
  Close(adatfile); Close(ujfajl);
  erase(adatfajl); rename(ujfajl, fajlnev);
end; {kihagy};
```

A fájl valamelyik rekordjának módosítása már megoldható a TOROL és a BOVIT eljárások segítségével.

Két fájl összefűzése (összeadása) soros kezelés esetén úgy oldható meg, hogy

— nyitunk egy harmadik fájlt (F3),
— az egyik fájlt (F1) átmásoljuk F3-ba,
— majd a másik fájlt (F2) is átmásoljuk F3-ba.

```
Procedure Hozzaad;
var rek: tadatrec;
    F1, F2, F3 : file of tadatrec;
    Elso, Masodik, Harmadik: String;
BEGIN WRITE('ELSŐ FÁJLNÉV='); READLN(ELSO); ASSIGN(F1,ELSO);
  WRITE('MÁSODIK FÁJLNÉV='); READLN(MASODIK); ASSIGN(F2,MASODIK);
  WRITE('HARMADIK FÁJLNÉV='); READLN(HARMADIK);
                ASSIGN(F3,HARMADIK);
    REWRITE(F3); RESET(F1);
    While Not EOF(F1) do
```

```
      Begin
        Read(F1, rek); Write(F3, rek);
      End;
    Close(F1);
    Reset(F2);
    While Not EOF(F2) do
      Begin
        Read(F2, rek); Write(F3, rek);
      End;
    Close(F2);    Close(F3);
End; {Hozzaad}
```

A gyors lista azt jelenti, hogy az összes személynek csak a nevét és a városát írjuk ki, és nem írjuk ki a teljes rekordot. Képernyőre való kiíráskor ez egyébként is elég áttekinthetetlen.

```
procedure lista(fajlnev:fajlnevtip);
var adat:tadatrec;
  begin
  assign(adatfile,fajlnev); reset(adatfile);
  while not eof(adatfile) do
    begin
      read(adatfile,adat); writeln(adat.nev,' ',adat.varos);
    end;
  close(adatfile); readln;
end;
```

A főprogram blokkjának megírása már nem jelenthet gondot. A program induláskor megkérdezi a fájl fizikai nevét. Ezután kiírja a program a menüt, amiből a felhasználó választhat. A választástól függően működteti a főprogram az eljárásokat.

```
begin write('Kérem a fájl nevét:'); readln(fajlnev);
  repeat
    writeln('Menü');
    writeln('1 Adatfájl létrehozása');
    writeln('2 Fájl bővítése');
    writeln('3 Személy keresése');
    writeln('4 Törlés');
    writeln('5 Módosítás');
    writeln('6 Összefűzés');
    writeln('7 Gyors lista');
    writeln('8 Kilépés');
    write('Mit választ:'); readln(valaszt);
```

```
      case valaszt of
      1: LETREHOZ(FAJLNEV);
      2: BOVIT(FAJLNEV);
      3: KERES(FAJLNEV);
      4: TOROL(FAJLNEV);
      5: BEGIN TOROL(FAJLNEV);BOVIT(FAJLNEV) END;
      6: HOZZAAD;
      7: LISTA(FAJLNEV);
      end;
    until valaszt=8;
end.
```

Kidolgozott feladat szekvenciális állományra

Ezek után fussuk át a karbantartási algoritmusokat, eljárásokat szekvenciális állományoknál soros elérés esetén. A szekvenciális jelző azt jelenti, hogy a fájlok rekordjai rendezettek. Az elsődleges kulcs a rekord NEV mezője legyen. A fájl kezelése továbbra is soros kezelés. Ezek az eljárások a következők:

Előállítás, bővítés
Keresés
Törlés
Módosítás
Összefésülés
Kiírás

Új fájl létrehozása mindössze azt jelenti, hogy egy fájlnév bekérése után megnyitunk egy üres fájlt a Rewrite(F) eljárás segítségével, majd azt le is zárjuk. Vigyázzunk arra, hogy már létező fájlt ne szüntethessünk meg meggondolatlanul. Ezért praktikusabb az a módszer, amikor új fájl létrehozását külön nem is írjuk a menüpontok közé, hanem a bővítéskor akkor valósítjuk meg, amikor még nincs bővítendő fájl. Ezt a módszert a Pascal eljárásai is támogatják. Soros állománykezeléskor a fájlok rekordjainak rendezése csak új fájl készítésével oldható meg. Rendezett fájlok előállítása érdekében válasszuk azt a módszert, amikor a rendezett fájlt mindig úgy bővítjük, hogy a most hozzáadott rekord is úgy kerüljön a fájlba, hogy a rendezettség vele is fennálljon. Legyen a rendezett fájl logikai neve TORZS és fizikai neve is 'TORZS'. Az elnevezéssel a törzsállomány megnevezésre szeretnénk utalni. Ezt a fájlt fogjuk — továbbra is rendezetten — bővíteni.

Rendezett fájl előállítása (bővítése)

Olyan algoritmusról van szó, amikor úgy hozzuk létre a fájlt, hogy az minden egyes bővítés esetén rendezett lesz. Így természetesen az eljá-

rás végén is rendezett fájlt kapunk. Az algoritmus a következő lépésekkel végezhető:

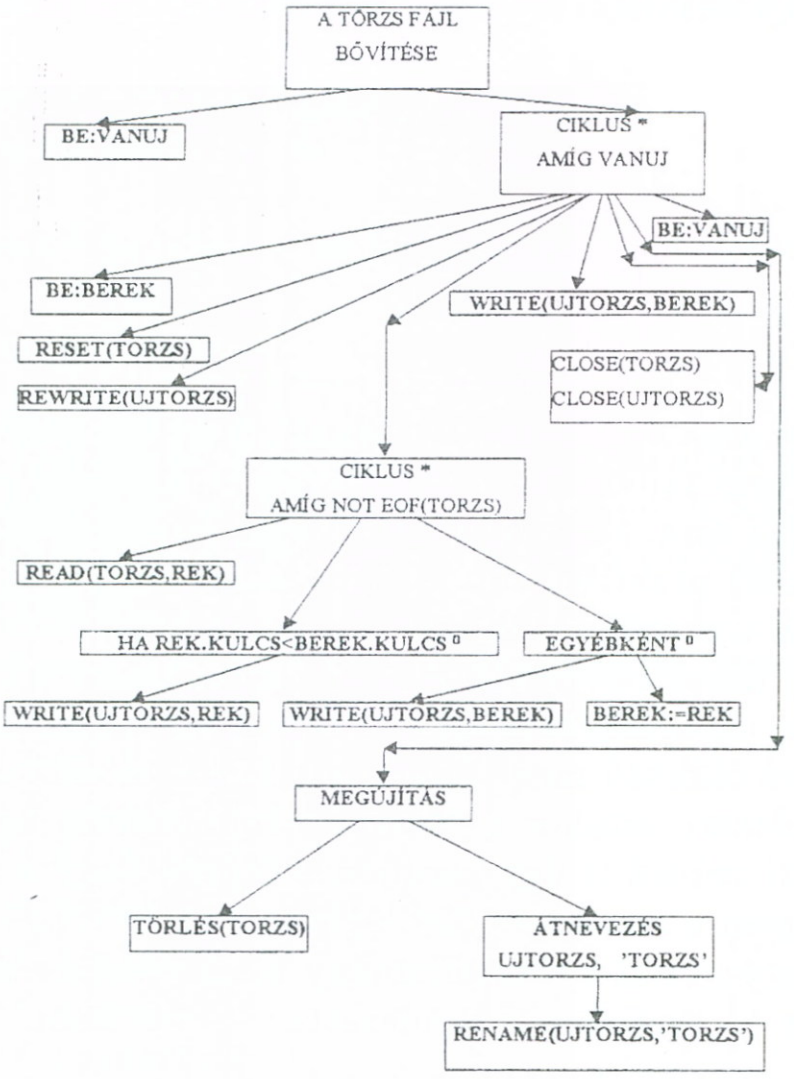
Olvassunk be a billentyűzetről egy rekordot (BEREK). A beolvasást természetesen mezőnként végezzük, de algoritmusunkban ezt most nem részletezzük. Ezzel a BEREK rekorddal szeretnénk a fájlt rendezetten bővíteni.

Egy UJTORZS('UJTORZS') nevű fájlba másoljuk be a TORZS fájl REK rekordjait és a billentyűzetről bejövő BEREK rekordot úgy, hogy az UJTORZS fájlba mindig a kisebb kulcsmezőjű rekord kerüljön. Az így keletkezett fájl ismét rendezett lesz.

A TORZS fájlt töröljük.

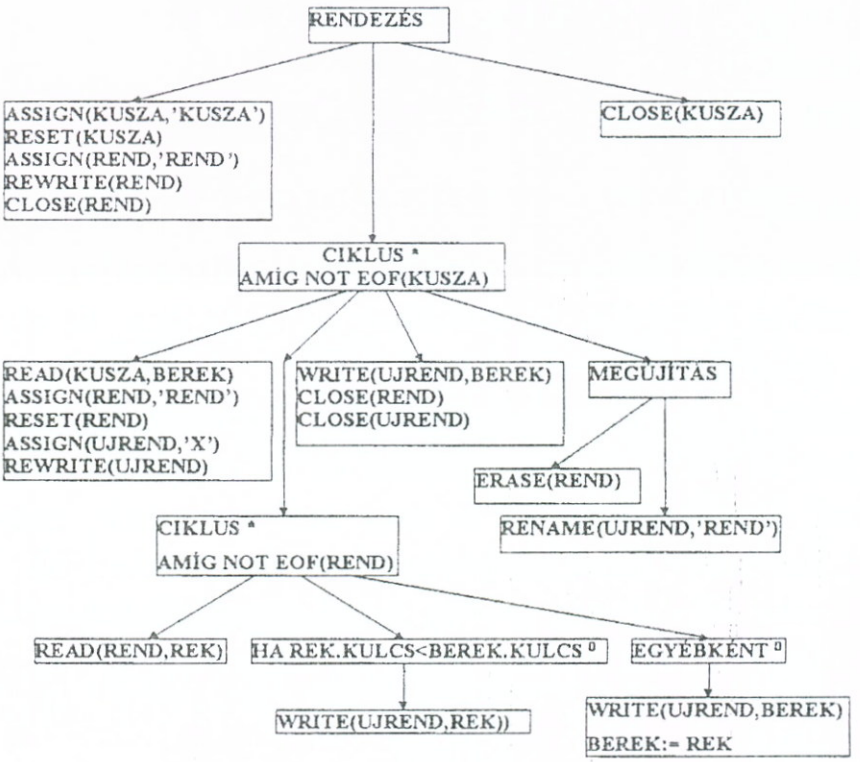
Nevezzük át az UJTORZS fájlt 'TORZS' fizikai névre.

Szekvenciális fájl létrehozásának, illetve bővítésének részletesebb algoritmusát írtuk le **Jackson-féle diagrammal** a következő oldalon. Az algoritmust megvalósítását az olvasóra bízzuk.



Előző algoritmusunkkal olyan fájlt állítottunk elő minden egyes billentyűzetről való bővítés esetén, amely fájl mindig rendezett. Ez az eljárás

nagyon elegáns, ha minden bővítés esetén van elég időnk az új fájl előállítására. Ha azonban szeretnénk elkerülni minden bővítés után a fájl állandó megújítását, akkor — szükség esetén — külön kell gondoskodni a rendezetlen fájl kulcs szerinti rendezéséről. Ezt az algoritmust is el kellene készíteni. Előző algoritmusunk azonban tartalmazza rendezetlen **fájl rekordjainak rendezését** is. Ha ugyanis előző algoritmusunknál a rekordok billentyűzetről való beolvasása helyett input adatnak a rendezendő fájl rekordjait tekintjük, akkor a keletkezett új fájl nem más, mint a rendezett fájl. Legyen a rendezendő fájl logikai neve KUSZA, a rendezetté REND. A rendezetlen fájl is maradjon meg változatlanul a további felhasználásokhoz. És most nézzük az algoritmust.



A keresés rendezett fájlban már hatékonyabb, mert csak addig kell a rekordokat beolvasni, amíg a keresendő rekord előtt van a fájl mutatója, és nincs vége a fájlnak. A logaritmikus keresést most sem alkalmazhatjuk, mert soros elérésnél a fájl mutatóját és a méretére vonatkozó függvényt nem használhatjuk.

```
procedure rendkeres(fajlnev:fajlnevtip);
var adat:tadatrec;
kit:string;
begin
assign(adatfile,fajlnev); write('Kit keressek?:'); readln(kit);
while kit<>'' do
begin
```

```
reset(adatfile); adat.nev:='';
while not eof(adatfile) and (adat.nev<kit) do
read(adatfile,adat);
if adat.nev=kit then
begin
writeln('Név:',adat.nev);
writeln('Város:',adat.varos);
writeln('Utca és házszám:',adat.utca);
writeln('Irányítószám:',adat.irszam);
writeln('Telefon:',adat.tel);
writeln('Egyéb megjegyzés:',adat.megj);
end
else writeln('Nem találtam ilyen nevet');
close(adatfile);
write('Kit keressek:');readln(kit);
end;
end; {keres}
```

A keresés után a kihagyás algoritmusa változatlan. A megtalált rekord kivételével a többi rekordot átmásoljuk az új fájlba, és elvégezzük a szükséges törlést és átnevezést.

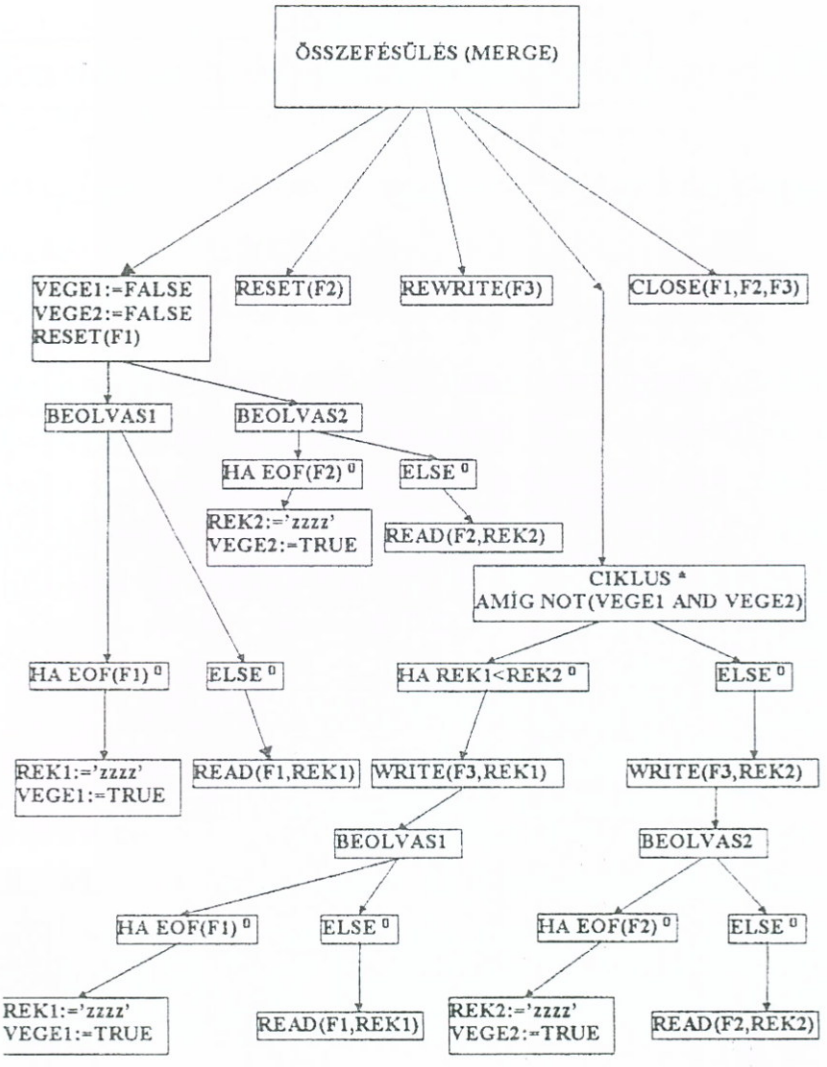
A **módosítás** algoritmusa is hasonló. A megtalált rekord után nem ezt a rekordot írjuk ki az új fájlba, hanem az új mezőkkel rendelkező rekordot. A többi rekord átmásolását itt is meg kell tennünk, mert különben a fájl csonka lesz. A módszer kissé kényelmetlen, de ne feledjük, hogy a fájlmutató állítása a szekvenciális elérés esetén nem engedhető meg.

A **kiírás** algoritmusa esetén minden rekordot ki kell írni a képernyőre, ezért amíg nem a fájl végén vagyunk, addig be kell olvasnunk a következő rekordot, és annak mezőit ki kell írni.

Két — azonos kulcs szerint rendezett — fájl összefésülése szekvenciális kezelés esetén a következő oldalon lévő diagram alapján készíthető el. Ez a Jackson-féle struktúradiagram a részleteket nem tartalmazza, hanem a szerkezeti felépítésre koncentrál. Ennek ellenére reméljük, hogy az eljárás megírása ez után nem okoz nehézséget (az ábrát lásd a következő oldalon).

Feladat

Az előzőekben vázolt problémát kódolja teljes részletességgel. Törekedjen a program biztos működésére és a felhasználó kellő tájékoztatására. Ne feledje, hogy kidolgozott feladatunk az életszerű feladatok megoldásához modellként szolgálhat. Az éles feladatokban Önnek kell eldöntenie az alkalmazható lépéseket, de azután az algoritmusoknak és a kódolási fogásoknak gyorsan, automatikusan kell menniük.



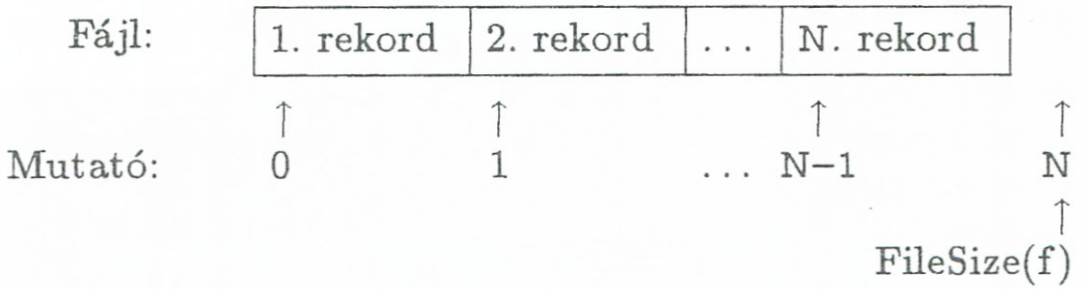
2.14.2.1.2. Direkt fájlkezelés

A fájlokról szóló eddig leírt algoritmusainkban és programjainkban a fájlok rekordjait csak szekvenciálisan kezeltük. A Pascal programozási nyelv azonban a rekord deklarációjából tudja annak a lemezen lefoglalt hosszát, így a rekord sorszámának ismeretében közvetlenül (direkt) is eléri a lemezen a fájl kívánt rekordját.

Direkt kezelés esetén a fájlban minden rekordhoz egy fájlmutató tartozik, amivel a programban mi is operálhatunk. A fájl rekordjainak I/O eljárásai mindig a mutató által „mutatott" pozícióra vonatkoznak. Ha egy fájlnak N rekordja van, akkor azt mondjuk, hogy a fájl mérete N. Ezt a méretet a **FileSize(f)**:Longint függvény is megadja. A fájl mutatójának aktuális értékét a **FilePos(f)**:Longint függvény adja meg, amelyre a következő reláció igaz: $0 \leq m \leq \mathbf{FileSize(f)}$. Az üres fájlnak csak 0 mutatója van, hiszen a fájl mérete 0.

A fájl rekordjainak fizikai sorszáma is van. A sorszám azt jelenti, hogy hányadik rekordja az aktuális rekord a fájlnak. Az s sorszám és az m mutató

között a következő szoros összefüggés van: m = s−1.*



Az előzőekben megfogalmaztunk egy modellfeladatot, és soros fájlkezeléssel meg is oldottuk. A fájlmutatóval való munkálkodást akkor nem engedtük meg. Csak azt kérdezhettük meg, hogy a fájl végén vagyunk-e már. Ebben a részben direkt (közvetlen) kezeléssel vázoljuk egy példa megoldását. Ekkor használható minden olyan függvény és eljárás is, amelyek a fájl mutatójára vonatkoznak.

```
FILEPOS(VAR logikainev):LOGINT
SEEK(VAR logikainev;pozicio:LOGINT)
FILESIZE(VAR logikainev):LOGINT
```

Kidolgozott feladat

Modellfeladatunk egy NOTESZ nevű program megvalósítása lesz. A programban lévő fájl rekordszerkezetét és a deklarációkat a következő módon adjuk meg:

```
PROGRAM NOTESZ;
USES CRT;
TYPE NEVT=STRING[30];
  SZEMELYT=RECORD
        NEV:NEVT;
        CIM:STRING;
    END;
  FIZNEVT=STRING[20];
  FAJLT=FILE OF SZEMELYT;
VAR SZEMELY:SZEMELYT;
  NEVSOR:FAJLT;
  FIZNEV:FIZNEVT;
  SZAM:LONGINT;
  KNEV:NEVT;
```

A programban a következő feladatokat valósítsuk meg:

---

* Az m, s, N betűk nem Pascal változók. Jelölésükkel csak a magyarázatot segítettük.

1. Hozzunk létre egy üres állományt.
2. Bővítsük az állományt.
3. Keressünk az állományban név szerint.
4. Töröljünk az állományból.

Az adatfájl létrehozása teljesen megegyezik a soros kezelésnél leírtakkal.

```
PROCEDURE LETRE(MIT:FIZNEVT);
BEGIN ASSIGN(NEVSOR,MIT);
  REWRITE(NEVSOR);
  CLOSE(NEVSOR);
END; {LETRE}
```

A gyakorlatból tudjuk, hogy a programok futtatása során a felhasználó meggondolatlan lépései miatt olyan hibák keletkezhetnek, amelyek jóvátétele sok munkát igényel. Ezért javasoljuk az olvasónak, hogy az új fájl létrehozásának alkalmazását nehezítse meg a felhasználónak, hogy átgondolatlan döntése miatt nehogy letörölje a meglévő állományt. Célszerű az üres fájl létrehozását egy külön programban megvalósítani.

Az állomány bővítése történhet egyszerűen úgy, hogy az új rekordot a fájl végére írjuk. Most a keletkezett fájl rekordjainak rendezettsége nem követelmény. A bővítendő fájl mutatóját vigyük a fájl végére, és az új fájlt most írjuk a fájlba. A mutató fájl végére való állítását a következő eljárással tehetjük: Seek(adatfile, FileSize(adatfile)).

```
PROCEDURE BOVIT(MIT:FIZNEVT);
BEGIN CLRSCR;
  ASSIGN(NEVSOR);
  WITH SZEMELY DO BEGIN
    WRITE('NÉV='); READLN(NEV);
    WRITE('CÍM='); REDLN(CIM);
                END;
  RESET(NEVSOR);
  SEEK(NEVSOR, FILESIZE(NEVSOR));
  WRITE(NEVSOR,SZEMELY);
  CLOSE(NEVSOR);
END; {BOVIT}
```

A biztonságos használat érdekében érdemes lenne úgy megírni az eljárást, hogy ha még nem létezik a nevezett állomány, akkor azt most hozza léte. Ha ezt a megoldást választjuk, akkor a létrehozás menüpont önállóan nem is szükséges.

Keresés most is csak lineárisan történhet, mert a fájl rekordjai nem rendezettek. A következő menüpont előkészítése érdekében fogalmazzuk meg a

keresést függvényként. A függvény logikai típusú legyen, és visszatérési értéke akkor legyen TRUE, ha megtaláltuk a rekordot. A függvény cím szerint hívott egyik paraméterében jegyezzük meg a megtalált rekord mutatóját is.

```
FUNCTION KERES(HOL:FIZNEVT; MIT:NEVT;VAR HOLVAN:LONGINT):BOOLEAN;
BEGIN
  ASSIGN(NEVSOR,HOL); RESET(NEVSOR);
  SEEK(NEVSOR,0);
  SZEMELY.NEV:=''; HOLVAN:=-1;
  WHILE NOT EOF(NEVSOR) AND (SZEMELY.NEV<>MIT) DO
    BEGIN
      HOLVAN:=HOLVAN+1; HOLVAN:=FILEPOS(NEVSOR)
      READ(NEVSOR,SZEMELY);
    END;
  CLOSE(NEVSOR);
  IF SZEMELY.NEV=MIT THEN KERES:=TRUE
  ELSE
    BEGIN KERES:=FALSE;
          HOLVAN:=-1
    END;
END;{KERES}
```

A keresett KNEV beolvasása után a függvény hívható. A keresett rekord megtalálása esetén a rekord mezői is kiírathatók. A HOLVAN aktuális paraméterében (SZAM) a rekord mutatója lesz. Ezt a későbbiekben használni is fogjuk. A hívás a következő módon történhet:

```
READLN(KNEV);
IF KERES(FIZNEV,KNEV,SZAM) THEN
  BEGIN WRITELN('A KERESETT REKORD MEZOI:');
  ...
  END
ELSE WRITELN(KNEV,'NINCS AZ ÁLLOMÁNYBAN');
```

Törlés a fájlból úgy történhet, hogy a törlendő rekordot megkeressük a keres függvénnyel. A függvény megjegyezte a rekord fájlbeli mutatóját is. Ezután a fájl utolsó rekordját olvassuk be, és írjuk ki a törölni kívánt rekord helyére. A másolt rekord még a fájl végén továbbra is megvan, ezért vágjuk le a fájlból.

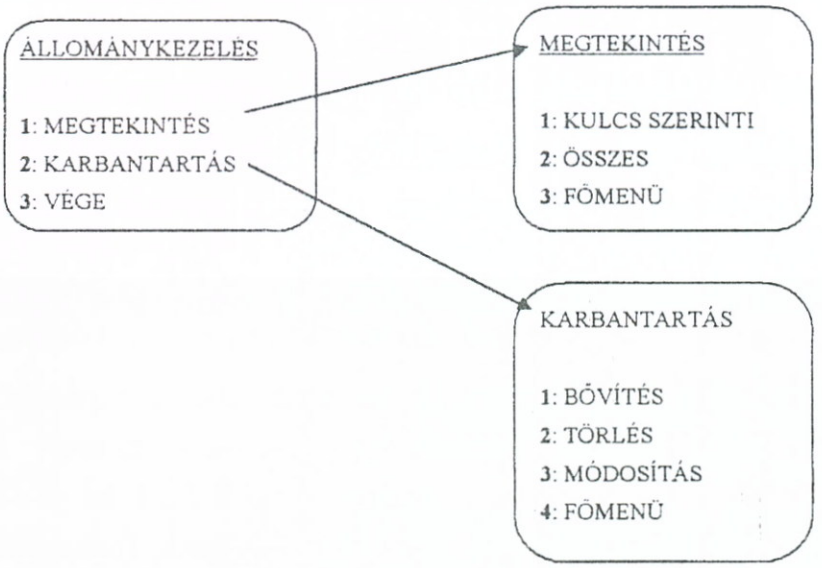
```
PROCEDURE TOROL(HONNAN:FIZNEVT; MIT:NEVT);
VAR MUTAT: LONGINT;
BEGIN IF KERES(HONNAN,MIT,MUTAT) THEN
BEGIN ASSIGN(NEVSOR,HONNAN); RESET(NEVSOR);
```

```
SEEK(NEVSOR, FILESIZE(NEVSOR)-1); READ(NEVSOR, SZEMELY);
SEEK(NEVSOR, MUTAT); WRITE(NEVSOR,SZEMELY);
SEEK(NEVSOR,FILESIZE(NEVSOR)-1); TRUNCATE(NEVSOR);
CLOSE(NEVSOR);
END ELSE WRITELN('NEM LÉTEZIK A TÖRÖLNI KÍVÁNT NÉV');
END; {TOROL}
```

Feladatok

1. Bővítse úgy a vázolt programot, hogy a fájl bármelyik rekordját lehessen módosítani is.

2. Valósítsa meg azt a programot, melynek ablakszerkezete a következő:



3. Az előző feladatban az összes rekord megtekintését oldja meg úgy, hogy a fájlban lapozni lehessen a rekordok között.

4. Egy vállalkozás a következő adatokat napra készen tartja nyilván dolgozóiról az adóigazolások elkészítéséhez a DOLGOZOK:DAT fizikai nevű fájlban, floppylemezen:

NÉV	C30
AZONOSÍTÓ	C10
GÖNGYÖLT BRUTTÓ	N8
ADÓELŐLEG	N7

A cégtől év közben kilépők adatait a KILEP.DAT állomány tartalmazza. Rekordszerkezete:

NÉV	C30
AZONOSÍTÓ	C10
KILÉPÉSI DÁTUM	C6

Készítsen egy IGAZOLAS.DAT fájlba listát azokról az emberekről, akik már nem dolgoznak a vállalkozásnál, és most számukra az adóigazoló

lapot kell kitölteni. A volt dolgozók rekordjaiban minden szükséges mező jelenjen meg Az elkészített fájlban a képernyőn lehessen keresni és lapozni is. Készítsen a programhoz felhasználói leírást is.

Rendezett fájlok, közvetlen eléréssel

Az eddig elemzett problémáknál nem követeltük meg, hogy a direkt módon kezelt állomány rekordjai valamelyik mező szerint rendezetten helyezkedjenek el a lemezen. Pedig rendezett állományok esetén algoritmusaink köre is bővülhet, valamint bizonyos problémák algoritmusai sokkal hatékonyabban és egyszerűbben valósíthatók meg.

Rendezett fájlt készíthetünk úgy, hogy

1. egy már meglévő, adatokkal feltöltött (rendezetlen) fájlt fizikailag rendezünk, vagy

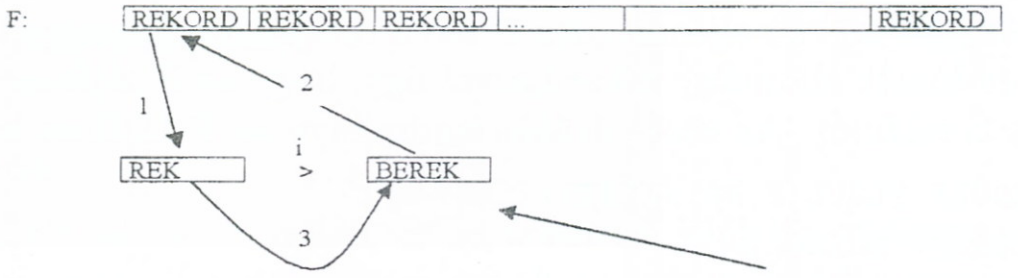
2. a rendezett fájlt mindig úgy bővítjük, hogy rendezett maradjon.

1. Direkt kezelésű fájlokat fizikailag rendezhetünk minden algoritmussal, amelyeket tanultunk. Példánkban a minimumkiválasztás módszerével való rendezést választottuk. Eljárásunkban a fájl rekordszerkezete megegyezik előző példánk rekordszerkezetével. A rendezés kulcsmezője a NEV mező legyen.

```
PROCEDURE RENDEZES(MIT:FIZNEVT);
VAR I,J,INDEX:LONGINT;
  ERTEK,REKORDI,REKORDJ: SZEMELYT;
BEGIN ASSIGN(NEVSOR,MIT); RESET(NEVSOR);
  FOR I:=1 TO FILESIZE(NEVSOR)-1 DO
  BEGIN INDEX:=I
    SEEK(NEVSOR,I-1); READ(NEVSOR,REKORDI);
    ERTEK:=REKORDI;
    FOR J:= I+1 TO FILESIZE(NEVSOR) DO
      BEGIN SEEK(NEVSOR,J-1); READ(NEVSOR,REKORDJ);
      IF ERTEK.NEV>REKORDJ.NEV THEN
        BEGIN ERTEK:= REKORDJ; INDEX:=J;
        END
      END;
      SEEK((NEVSOR,INDEX-1); WRITE(NEVSOR,REKORDI);
      SEEK(NEVSOR,I-1); WRITE(NEVSOR,ERTEK);
  END; CLOSE(NEVSOR);
END; {RENDEZES}
```

2. A rendezett fájl úgy is előállítható, hogy az eddig rendezett fájlt bővítjük úgy, hogy az újabb rekorddal való bővítéskor az új rekordot a

relációnak megfelelő helyre tesszük, és az ez után lévő rekordokat hátrább rakjuk. Ha üres fájllal kezdünk, akkor mindig rendezett lesz a fájl.



A BEREK rekordba olvastuk be azt a rekordot, amellyel a fájlt bővíteni akarjuk.

Ezután végighaladunk a fájl rekordjain a következő lépésekkel:

1. Beolvasunk egy rekordot a REK változóba.

2. Ha a BEREK kulcs mezője kisebb a REK kulcs mezőjénél, akkor a BEREK tartalmát kiírjuk a fájlba arra a helyre, ahonnan az utolsó beolvasást végeztük.

3. A REK tartalmát a BEREK változóba tesszük. Ha a REK>BEREK reláció FALSE, akkor a REK változó tartalmát nem kell visszatenni, hiszen ott van, mert korábban onnan olvastuk be.

A fájl rekordjain végigmenve az új rekord így a helyére került. A BEREK változóban viszont maradt egy rekord, amely még hiányzik a fájlból. A ciklus után ezt is ki kell írni a fájlba.

Ezt az algoritmust most közvetlen kezeléssel kényelmesen megvalósíthatjuk. Az eljárás már feltételezi, hogy a BEREK rekordba benne van az a rekord, amellyel bővíteni akarunk.

```
PROCEDURE BOVITR(F:FIZNEVT; BEREK:SZEMELYT);
VAR REK: SZEMELYT;
    I:LONGINT;
BEGIN ASSIGN(NEVSOR,F); RESET (NEVSOR);
  FOR I:=1 TO FILESIZE(NEVSOR) DO
  BEGIN READ(NEVSOR,REK);
    IF BEREK.NEV<REK.NEV THEN
      BEGIN SEEK(NEVSOR, FILEPOS(NEVSOR)-1;
      WRITE(NEVSOR,BEREK); BEREK:=REK
      END;
  END;
  WRITE(NEVSOR,BEREK); CLOSE(NEVSOR)
END; {BOVITR}
```

Összefésülés (Merge)

Direkt elérésű állományokban két, azonos kulcs szerint rendezett fájl összefésülésekor is felhasználhatjuk az előbb megírt BOVITR(F:FIZNEVT; BEREK:SZEMELYT) eljárást. Az F nevű rendezett állományt bővítjük az F1 nevű rendezett állomány rekordjaival úgy, hogy az F fájlban is megtartja a BEREK a relációt. Az összefésülés eredménye az F fájlban keletkezik.

```
PROCEDURE FESUL(F,F1:FIZNEVT);
VAR BEREK: SZEMELYT;
  NEVSOR1:FAJLT;
  J:LONGINT;
BEGIN ASSIGN(NEVSOR,F); ASSIGN(NEVSOR1,F1);
RESET(NEVSOR1);
FOR J:=1 TO FILESIZE(NEVSOR1) DO
BEGIN READ(NEVSOR1, BEREK);
BOVITR(NEVSOR, BEREK);
END;
CLOSE(NEVSOR1);      {ERASE(NEVSOR1);}
END; {FESUL}
```

Az eljárás helyesen működik. Vegyük azonban észre, hogy a NEVSOR fájl (F nevű) minden egyes bővítésekor túl sok rekordot mozgattunk meg feleslegesen. Ez nagyon rontotta az algoritmus hatékonyságát. Erőltettük a számláló ciklus használatát is. Mi akartunk gondoskodni a mutató állítgatásáról is, pedig ezt a Pascal nyelv magától is tudja. Mentségünkre szolgáljon, hogy azt akartuk bemutatni, hogy direkt fájl kezelése esetén dolgozhatunk a fájl méretével és a fájlmutató állításával is.

Keresés rendezett fájlban direkt eléréssel

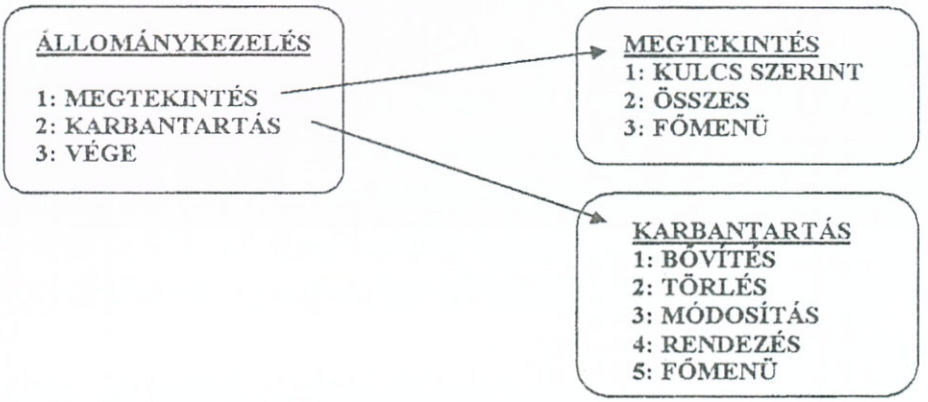
Mivel a direkt kezelés esetén a fájl méretével és a rekordok sorszámával is dolgozhatunk, ezért úgy tekinthetjük a fájlt, mintha tömb lenne. Tehát a logaritmikus keresés alkalmazható. Az algoritmus ismert, így csak az eljárás egy lehetséges változatát közöljük.

```
PROCEDURE BINKERES(HOL:FIZNEVT; MIT:NEVT; VAR KOZEPSO:LONGINT;
VAR TALAL:BOOLEAL);
VAR ALSO, FELSO: LONGINT;
    REK: SZEMELYT;
BEGIN ASSIGN(NEVSOR, F); RESET(NEVSOR);
  ALSO:=1; FELSO:=FILESIZE(NEVSOR); TALAL:=FALSE;
  WHILE (ALSO<=FELSO) AND (TALAL=FALSE) DO
  BEGIN KOZEPSO:=(ALSO+FELSO) DIV 2;
      SEEK(NEVSOR, KOZEPSO-1); READ(NEVSOR,REK);
```

```
        IF REK.NEV=MIT THEN TALAL:=TRUE
          ELSE IF REK.NEV<MIT THEN ALSO:=KOZEPSO+1
                  ELSE FELSO:=KOZEPSO-1;
      END; CLOSE(NEVSOR);
END;
```

Feladatok

1. Felvázolt algoritmusaink és eljárásaink alapján dolgozza ki teljes részletességgel modellfeladatunkat úgy, hogy most mindig rendezett állománnyal dolgozzunk. Annak érdekében, hogy korábban elkészített fájljainkat is használni tudjuk, a fizikai rendezést is oldjuk meg. Valósítsa meg azt a programot, amelynek ablakszerkezete a következő:



2. Év végi/félévi statisztikát kell készíteni az iskolában minden osztályfőnöknek az osztály félévi eredményeiről. Az osztályok tanulóinak névsorát kezelje minden osztálynál külön fájlként. A fájl fizikai neve az osztály neve legyen. A félévi osztályozókonferencia után minden tanulónak ismert a tantárgyankénti jegye. Készítsen olyan programot, mely tanulónként és az egész osztályra nézve is kiszámítja a tantárgyak és a hiányzások átlagát. A program kódolása előtt fogalmazza meg a feladatot pontosan, majd tervezze meg a rendszert. A programról készítsen felhasználói leírást is, és futassa élesben is a programot.

3. Írjon olyan programot, amivel egy iskolai verseny teendőit végzi. Fogalmazza meg a verseny lebonyolítását és kiértékelését pontosan. Készítsen a programhoz felhasználói leírást is.

4. A SÜTI Sütőipari Vállalkozás meghatározott termékllistával rendelkezik. Ezt a listát a környék üzleteivel is közölte. A környék üzleteitől mágneslemezen kéri a napi megrendelést a következő rekordszerkezetben:

```
Az üzlet kódja:        C4
Dátum:                 C8
Cikkszám:              C4
Mennyiség:             N6.2
```

A SÜTI pékség — esténként — összesíti az üzletektől kapott állományokból a megrendeléseket cikkszám szerint, és ez alapján történik a rendelés teljesítése. Egy cikkszám egy üzleten belül többször is szerepelhet, és rendezettség sincs. Az is előfordulhat, hogy olyan terméket kérnek, amelyek nem szerepelnek a termékliston. A reggeli áruátvételhez minden megrendelő kapja vissza lemezét azzal a fájllal együtt, amelyben a tényleges teljesítés is szerepel. A nem gyártott termék mellett legyen megjegyzés. Készítse el a programrendszer leírását, a programot, és éles adatokkal is futtassa.

5. Leltározáskor minden cikkről a következő adatokat visszük a számítógépbe:

```
Cikkszám:        C20
Mennyiség:       N6.2
Egységár:        N3.2
```

Készüljön program leltározás lebonyolítására. Ugyanolyan cikkszámú tárgy többször is szerepelhet, és azok sorrendje tetszőleges. Készüljön — egyelőre a képernyőre — olyan lista, mely Cikkszám szerint rendezetten tartalmazza a termékek jegyzékét. Minden előforduló termék csak egyszer szerepeljen, és a rekord mezői után a képviselt érték is szerepeljen. A teljes lista után az üzlet teljes vagyonát is adja meg. A képernyőn látható lista fájlban is legyen meg.

6. Készítse el egy pénztárgép programját. Minden árunál a rekord felépítése a következő:

```
ARU:record
  Kod:string[5];
  EgysAr:real;
  Menny: real;
```

```
KEZDÉS
VÁSÁRLÁS
ZÁRÁS
VÉGE
```

A vevő által vásárolt minden áru esetén be kell ütni a pénztárgépbe a rekord minden mezőjének értékét: az áru kódját, egységárát, mennyiségét. A pénztárgép csináljon (a monitorra) számlát a vevőnek, és összesítse is a forgalmat az esti záráshoz. Az esti forgalmat az áru kódja szerinti rendezettségben adja meg a program. A pénztárgépet kezelő személy a gép bekapcsolása utáni munka során a menüből választhasson.

7. Készítse el most az előző témájú programot úgy, hogy az üzlet árukészlete legyen egy fájlban feltöltve (ugyanezzel a rekordszerkezettel). Az eladás során csak a kód és a mennyiség kerüljön rögzítésre. Ellenőrzést is végezzen a program. Ha valamelyik áru nincs az árukészletben, azt hangjelzéssel is tudassa. A vásárlásnak megfelelően az árukészletet is csökkentse.

8. Végezze el programja üzletünkben azt a karbantartást, amelyet egy árváltozás tesz szükségessé. Készítsen a rendszerről leírást és kezelési útmutatót.

2.14.2.2. Szöveges fájlok

A texttípus

Mint feladatainkból is látszik, egyre jobban ki tudjuk elégíteni a gyakorlat által keletkező igényeket. A fájlok segítségével már biztonságosan tárolhatjuk adatainkat, és amikor szükséges, feldolgozhatjuk azokat. A gyakorlati munka és a tárolás szempontjából fájlok között jelentős szerepet játszanak a szövegfájlok. Ezért a következő fejezetben ezzel a témával foglalkozunk.

Mint azt már jól tudjuk, a fájl meghatározatlan számú — rendszerint azonos típusú — értékek összessége. A szövegfájlokban nem rekordok a fájl alkotóelemei, hanem változó hosszúságú karaktersorozatok. A Pascal szabványos nevet is rendel ehhez a típushoz. Ez a típus a TEXT, amely ugyanúgy használható deklarációkban, mint más standard típus. Pl.:

```
TYPE szovegtip= TEXT;
VAR level: TEXT;
```

A szövegfájl tehát ASCII karakterek sorozatából áll. A texttípus is soros fájl. Minden egyes karakter tárolásához 1 byte tárolóterületre van szükség. A karakterek sorozatában néhány karakter azonban kitüntetett szerepet játszik.

A sor karakterek sorozatából áll, amelyet a **sorvégjel** jelöl. A sorvégjel a kocsi vissza [chr(13)] és a soremelés [chr(10)] karakterekből áll. Ezek az elnevezések az írógép funkcióit idézik. A sorvégjelet a programból is figyelhetjük az EoLn(fajlnev) függvénnyel. A sorok változó hosszóságúak lehetnek. Ezért a szövegfájlokat csak sorosan kezelhetjük, azaz csak sorban, előre léphetünk. A visszalépést csak úgy valósíthatjuk meg, hogy lezárjuk a fájlt, majd ismét megnyitjuk, és az adott helyig haladunk.

A szövegfájl végét a 26-os belső kódú karakter (Ctrl+Z, illetve F6) jelzi. Ez a karakter csak egy jelölő karakter, amely már nem tartozik a szöveghez. A szövegfájl végét is figyelhetjük az Eof(fajlnev) függvénnyel.

Eljárások, függvények

Ezek után foglaljuk össze a szöveges fájlok esetén alkalmazható eljárásokat és függvényeket.

1. Deklaráció: VAR fajlnev : TEXT. Természetesen a fajlnev egyedi azonosító, a fájl logikai nevét jelenti.

2. Összerendelés: ASSIGN(fajlnev, fizikai_nev). A fajlnev az a logikai név, amit a fájl visel a programban. A fizikai_nev egy string típusú értéket képvisel, és ezen a néven van a fájl a lemezen.

3. A szövegfájlnál a különböző megnyitási eljárások egyszerre csak egyirányú adatmozgatást tesznek lehetővé. Ez azt jelenti, hogy a megnyitott szöveges fájl vagy olvasható, vagy írható. Szöveges fájlt — egyszerre — írásra és olvasásra nem lehet megnyitni.

A REWRITE(filenev) eljárás új fájlt hoz létre, és megnyitja a fájlt, de csak írásra. A fájl mutatóját 0-ra állítja. Ha létezett már ilyen fájl, akkor lezárja, majd letörli, és létrehoz egy új fájlt.

A RESET(filenev) már létező fájlt nyit meg csak olvasásra, a mutatót 0-ra állítja.

Az APPEND(filenev) eljárás már létező fájlt nyit meg folytatásra. Ez valójában azt jelenti, hogy a fájl mutatóját a fájl végére állítja. Ha a létező fájl nyitott volt, akkor előbb lezárja azt, majd újra megnyitja írásra, és a fájl mutatóját a fájl végére állítja. A megnyitott fájl csak írható.

4. A feldolgozó eljárások és függvények köre a textfájlok esetén meglehetősen korlátozott. Némelyiknél — jelentős módosulással — csak a következők használhatók: Read, Radln, Write, Writeln, Flush, IOResult, Eof, SeekEof, Eoln, SeekEoln, SetTextBuf.

A READ(Var filenev: Text; V_1[,V_2 ..., V_N]) és READLN(Var filenev: Text; V_1[, V_2 ..., V_N]) eljárások beolvassák a változókba a fájlból a megfelelő értékeket. A két eljárás között azonban különbség van: A READ eljárás egy változóba csak annyi karaktert olvas be, amennyi a változó típusának megfelel. A következő változóba való beolvasást onnan folytatja, ahol azt az előző abbahagyta. A soron következő beolvasó eljárás is erről a helyről folytatja a munkát. Tehát nem maradhat ki egyetlen karakter sem a beolvasásból. A READLN eljárás is hasonlóan dolgozik, de miután feltöltötte a változókat, megkeresi a következő sorvégjelet. A következő beolvasó eljárás innen folytatja a beolvasást. Így adatvesztés is előfordulhat. Az I/O eljárások jobb megértése érdekében nézzük meg, hogy milyen típusú változók szerepelhetnek az eljárások paraméterlistáján.

A változók típusa lehet:

CHAR: Ekkor a beolvasó eljárás ténykedése nem szorul magyarázatra.

STRING: A READ és a READLN eljárás ekkor annyi karaktert olvas be, amekkora a változó deklarált karakterlánc hossza.

Numerikus változó esetén számelválasztó karakterre is szükség van azért, hogy az eljárás eldönthesse, hogy meddig tartoznak a karakterek a numerikus értékhez. Számelválasztó karakterként a következő kódú karakterek szerepelhetnek: 32 (Space), 9 (Tab), 13 (CR), 10 (LF). Ekkor a beolvasó eljárások automatikus számkonverziót is végeznek.

A WRITE(Var filenev: Text; K_1[,K_2... K_N]); WRITELN(Var filenev: Text; K_1[,K_2,... K_N]); eljárások a kifejezések értékeinek a fájlba való írását

jelentik. A $K_1,\ldots K_N$ sorozat kifejezések sorozata, így ezeken a helyeken konstansok, változók, függvények és kifejezések is állhatnak. Minden kifejezés után megadhatunk mezőszélességet is a következő formában: :M1[:M2]. A mezőszélességet a listaelem típusától függően kell megadni. M1 jelöli a kiírandó érték karaktereinek teljes hosszát, M2 pedig azt, hogy a tizedespont után hány karaktert írjon ki az eljárás (ha a kifejezés valós típusú). Értelemszerűen: ha a típus nem valós, akkor M2 értékét nem adható. A WRITELN(filenev); eljárás hatására csak egy újsor-jel kerül a fájlba. A két eljárás között egyértelmű különbség az, hogy a WRITELN sorvégjelet is tesz a kifejezések értékeinek kiírása után, míg a WRITE nem. A kifejezések típusa íráskor is lehet: Karakter, String vagy bármelyik numerikus típus. Numerikus típus esetén konvertálás is történik.

A szövegfájlok I/O műveletei közvetlenül nem a memória és a lemez között valósulnak meg, hanem egy közbülső pufferen keresztül. A Rewrite és Append eljárásokkal megnyitott szöveges fájloknál minden írás (Write vagy Writeln) esetén az Output művelet fizikailag a pufferbe való írást jelent, mindaddig, amíg a puffer meg nem telik. Csak a következő írás hatására kerül a puffer tartalma a lemezre. Input-Outputra megnyitott fájlok esetén a puffer tartalma akkor is kiürül, ha az I/O műveletek iránya megfordul. A puffer használata jelentős mértékben gyorsítja az adatforgalmat.

A puffer méretét a SetTextBuf (var F:Text; var Buf [; Meret:word]) eljárással állíthatjuk (a 4.0 verziótól). Az eljárás az F szövegfájlhoz hozzárendeli az alapértelmezés szerinti 128 byte nagyságú puffer helyett a Buf átmeneti puffert. A Buf tetszőleges típusú lehet. Ha megadjuk az opcionális Meret értékét, akkor az adatforgalomban részt vevő csomag méretét adjuk meg. Ennek szerepeltetése nélkül a puffer mérete a Buf változó méretével egyezik meg. Az eljárás a megnyitott F állományra vonatkozik. Hatása a következő — F fájlra vonatkozó — Assign eljárásig, illetve a program végéig tart.

A Flush(Var F:Text) eljárás üríti az Append vagy Rewrite segítségével megnyitott F szövegfájl pufferének tartalmát, azaz fizikálisan végrehajtja az aktuális outputra a kiírást, akkor is, ha a puffer nem telt még meg. Ennek az eljárásnak az alkalmazása különösen az utolsó írási eljárás után tanácsos, mert nélküle az utolsó csomag a pufferben maradhat, és ekkor adatot veszíthetünk. Az eljárás csak outputra megnyitott fájlokra alkalmazható. Olvasásra megnyitott fájlok esetén hatástalan.

Az EOF(filenev):Boolean logikai függvény értéke True, ha az aktuális fájl pozíció a fájl végén áll, egyébként False. Ezt a függvényt már korábbi tanulmányainkból ismerjük.

Az EOLN(filenev):Boolean nem a fájl végjelét, hanem a sorvégjelet vizsgálja. Értéke akkor TRUE, ha a sor végén van a fájl aktuális mutatója.

A SEEKEOF(filenev):Boolean hasonlít az EOF függvényhez, de ez a függvény „átnéz" a vessző és a szóköz karaktereken, és így vizsgálja, hogy vége van-e a fájlnak.

A SEEKEOLN(filenev):BOOLEAN kihagyja a vessző, a szóköz és TAB karaktereket, amikor a sor végét vizsgálja.

5. Fájl lezárása: CLOSE(filenev).

Jegyezzük meg, hogy csak ezek az eljárások és függvények használhatók a szöveges fájloknál, és a FILESIZE, FILEPOS stb. eljárások, függvények nem alkalmazhatók. Néhány eljárás nem egészen úgy dolgozik, mint azt a nem szöveges állományoknál megszoktuk, ezért érdemes pontosan megtanulni a hasonlóságokat és az eltéréseket. Nos ezen ismeretek birtokában nézzünk meg egy példát.

Kidolgozott feladat

Írjunk olyan programot, mely tud szövegfájlt:

— létrehozni,
— bővíteni,
— sort kihagyni,
— beszúrni,
— módosítani,
— fájlt kiíratni (sorai számozásával).

Minden sor tartalmazzon 70 karaktert. Minden kis részletet nem írtunk le. A program részletezését az olvasóra bízzuk. Ötleteket így is mutatunk közben.

```
PROGRAM szov;
USES CRT,...;
TYPE nevtip=string[20];
VAR ...
  vanemeg:BOOLEAN;
  fiznev:nevtip;
  f:TEXT;
PROCEDURE nez(nev:nevtip);
VAR ... ;
BEGIN
  IF nev='' THEN
    BEGIN
    ...
    WRITELN(' Nincs filenév megadva!');WRITELN;
    WRITELN(' Filenév:',WRITELN);
    READLN(nev); ASSIGN(f,nev);
```

```
    ...
    END;
END;
PROCEDURE letre(nev:nevtip);
VAR ...
  c:char;
BEGIN
  ...
  WRITE('Fizikai név:');
  READLN(nev);c:='I'; ASSIGN(f,nev);
  $i- RESET(f); $i+
  IF ioresult=0 THEN
    BEGIN
    ...
    WRITELN(' Ilyen file már van!!');
    WRITELN(' Felülírjam? (I/N)');    c:=UPCASE(READKEY);
    ...
    END;
    IF c='I' THEN REWRITE(f);
    ...
END;
PROCEDURE bovit(nev:nevtip);
VAR ...
  sor:STRING[70];
BEGIN
  ...
  nez;
  WRITELN('Beszúrandó sor, kilépés=*');
            WRITE('>');READLN(sor);
  APPEND(f);
  WHILE sor<>'*' DO
    BEGIN
      WRITELN(f,sor);
      WRITE('>');
      READLN(sor);
    END;
  CLOSE(f);
  ...
END;
PROCEDURE kihagy(nev:nevtip);
VAR ...
```

```
  sor:STRING[70];
  sort,i:LONGINT;
  temp:TEXT;
BEGIN
  nez;
  ...
  write('Melyik sort töröljem:');
  readln(sort);
  RESET(f);    ASSIGN(temp,'temp.tmp'); REWRITE(temp);
  i:=0;
  WHILE NOT EOF(f) DO
    BEGIN
      READLN(f,sor);
      INC(i);
      IF i<>sort THEN WRITELN(temp,sor);
    END;
  CLOSE(f);    CLOSE(temp); ERASE(f); RENAME(temp,nev);
  ...
END;
PROCEDURE beszur(nev:nevtip);
VAR ...
  sor,sor1:STRING[70];
  sort,i:LONGINT;
  temp:TEXT;
BEGIN
  nez;
  ...
  WRITE('Beszurando sor:');READLN(sor1);
  WRITE('Hanyadik sor utan:');READLN(sort);
  RESET(f); ASSIGN(temp,'temp.tmp'); REWRITE(temp);
  i:=0;
  WHILE NOT EOF(f) DO
  BEGIN
    READLN(f,sor);
    INC(i);
    WRITELN(temp,sor);
    IF i=sort THEN WRITELN(temp,sor1);
  END;
  CLOSE(f); CLOSE(temp); ERASE(f); RENAME(temp,nev);
  ...
END;
```

```
PROCEDURE kiir(nev:nevtip);
VAR ...
  sor:STRING[70];
  i:LONGINT;
BEGIN
  nez; ...
  RESET(f);i:=0;
  WHILE NOT EOF(f) DO
    BEGIN
      READLN(f,sor);
      INC(i);
      WRITELN(i,'. ',sor);
      ...
    END;
  CLOSE(f);
  ...
END;
PROCEDURE javit;
VAR
  sor:STRING[70];
  sort,i:LONGINT;
  temp:TEXT;
BEGIN
  nez;
  ...
  WRITE('Melyik sort javitsam:');READLN(sort);
  RESET(f);ASSIGN(temp,'temp.tmp');REWRITE(temp);
  i:=0;
  WHILE NOT EOF(f) DO
  BEGIN
    READLN(f,sor); INC(i);
    IF i=sort THEN
      BEGIN
        WRITELN(sor); WRITE('>'); READLN(sor);
      END;
    WRITELN(temp,sor);
  END;
  CLOSE(f);CLOSE(temp);
  ERASE(f);RENAME(temp,fiznev);
  ...
END;
```

```
BEGIN
  FIZNEV:='';
  ...
END.
```

Reméljük, hogy vázolt programunk segítséget ad a teljes program elkészítéséhez. A program kódolásakor előbb ne a díszítésen legyen a fő hangsúly, de miután teszteltük a rendszer működését, egészítsük azt ki úgy, hogy a program használata biztonságos és kényelmes legyen.

Feladatok

1. Készítsen programot, mely segítségével levelet írhatunk bekért (fizikai) nevű fájlba. A levél végét az F6 funkcióbillentyű jelentse, melynek ASCII kódja 26.

2. Az előzőekben megírt levelet írja ki a képernyőre. Számolja meg a program, hogy hány karaktert és hány sort tartalmaz a levél.

3. A levelet úgy írja ki a képernyőre, hogy minden sora előtt szerepeljen sorszám, majd szóköz. A levél sorai pontosan 50 karakter hosszúságúak legyenek.

4. Egy lemezen elkészítettük a CIMEK.DAT és a LEVEL.TXT nevű szöveges fájlt. A CIMEK.DAT minden rekordja egy-egy cím, a LEVEL.TXT egy levél szövegét tartalmazza. Készítse el a KORLEV.TXT nevű fájlt, mely valamennyi címhez tartalmazza a levél szövegét. Írja ki a képernyőre a KORLEV.TXT fájlt, de úgy, hogy abban lapozni is lehessen.

Külső eszközök mint textfájlok, nyomtatás

Mint már tisztáztuk, az ASSIGN(lognev, fiznev) eljárás feladata az, hogy egy fájl logikai és fizikai nevét egymáshoz rendelje. A fizikai név egy string, amely a lemezen lévő fájl DOS-beli elnevezését jelenti. Magában foglalhatja a meghajtót és az elérési utat is. A logikai név a fájl programbeli azonosítója. A két elnevezés között az ASSIGN eljárás teremti meg az összerendelést. Amíg az ASSIGN eljárást másik — a nevezett fájlra vonatkozó — ASSIGN nem követi, addig az változatlanul él. Az eljárás redundáns módon is alkalmazható. Korábbi tanulmányainkból jól tudjuk, hogy a DOS tartalmaz bizonyos „foglalt" fájlneveket. Ezek olyan fizikai fájlnevek, amelyek eszközöket jelentenek. Néhány példa:

CON:	Console. Olvasáskor billentyűzet, íráskor képernyő,
COM1 és COM2	soros portok (pl.: egér, modem, fax),
AUX	=COM1, 1-es kommunikációs port,
LPT1 LPT2, LPT3	párhuzamos portok,
PRN	nyomtató az LPT1-en

NUL — ezt az eszközt akkor használjuk, ha az írást csak szimulálni akarjuk, vagyis ténylegesen nem írunk semmilyen fájlba vagy eszközre.

Természetesen a Turbo Pascal programnyelvben is használhatunk eszközöket, nevükre való hivatkozással. A rendszer a különböző külső hardvereszközöket textfájlokként kezeli, és programjainkban logikai fájlnévként használhatjuk azokat. A logikai fájlnevekhez ekkor nem egy létező adathalmaz tartozik, hanem maga a hardvereszköz. A létező standard eljárások és függvények segítségével használhatjuk ezeket a perifériákat.

Azonban a standard unitok valamelyikében logikai nevek deklarációját és az összerendelést is megtaláljuk, így a program Uses részében csak a megfelelő unit megadásáról kell gondoskodni, és máris használhatjuk az eszközt, logikai nevére való hivatkozással.

A System unit tartalmazza a szabványos I/O eszközök deklarációját, és mint tudjuk: ezt az egységet nem kell a Uses alapszó után felsorolni.

Szabványos eszközei:

Input:Text; Szabványos Turbo Pascal beviteli állomány. A Unit iniciáló része az üres stringhez, vagyis a CON perifériához — a billentyűzethez — rendeli, majd megnyitja az állományt olvasásra.

Assign(Input,");Reset(Input); utasításokat hajt végre, ezért ez után a READ(Input,V_1[,V_2... V_N]); eljárás helyett a READ(V_1[,V_2... V_N]); használható.

```
PROGRAM BEOLVAS;
VAR LISTA:TEXT;
    EGYSOR:STRING[70];
BEGIN ASSIGN(LISTA,'CON');
  RESET(LISTA);
  READLN(LISTA, SOR);
  CLOSE(LISTA);
END.
```

Ezt a programot most egyszerűbben így írhatjuk:

```
PROGRAM BEOLVAS;
VAR EGYSOR:STRING[70];
BEGIN
  READLN(EGYSOR);
END.
```

Output:Text; Szabványos Pascal kiviteli állomány. Az egység iniciáló része az üres stringhez, a CON perifériához — a képernyőhöz — rendeli, majd megnyitja az állományt írásra. Assign(Output,"); és Reset(Output);

utasításokat hajt végre. Ezen utasítások után a WRITE(Output,K_1[,K_2... K_N]) eljárás helyett a WRITE(K_1[,K_2... K_N]) használható.

Ha a program használja a **Crt Unitot**, akkor az Input és Output a Turbo Pascalban definiált CRT perifériához rendelődik, és a Crt egység iniciáló része nyitja meg azokat. Amikor a könyv elején a Read, Readln, Write, Writeln eljárásokat ismertettük, a fájlváltozó nevét „elhallgattuk". Az elhagyás jogosságára az imént elemzett anyagrész hatalmazott fel bennünket. Úgy gondoltuk, hogy akkor még korai említést tenni az elméleti háttérről.

A **Printer Unit** az Lst:Text; logikai állományt deklarálja, és az iniciáló rész LPT1 nyomtatót rendeli hozzá, majd írásra meg is nyitja azt: Assign(Lst,'LPT1'); Rewrite(Lst);.

Ezért, ha nyomtatni akarunk a Turbo Pascal programban, akkor a deklarációk előtt a Uses Printer írásáról ne feledkezzünk el. Az írás eljárásai ekkor az Lst szöveges fájlba való írást, azaz nyomtatást jelentenek. Pl.: Write(Lst,'SZÖVEG'); vagy Writeln(Lst,'SZÖVEG');.

```
PROGRAM NYOMTAT;
VAR LISTA:TEXT;
BEGIN ASSIGN(LISTA,'LPT1');
  REWRITE(LISTA);
  WRITELN(LISTA, 'EZ EGY SOR LESZ A NYOMTATÓN!');
  CLOSE(LISTA);
END.
```

Így ez a program a következőképpen egyszerűsödhet:

```
PROGRAM NYOMTAT;
USES PRINTER;
BEGIN
  WRITELN(LST, 'EZ EGY SOR LESZ A NYOMTATÓN!');
END.
```

A nyomtatási kép megtervezésekor és a nyomtatási rész kódolásakor mindig tartsuk szem előtt, hogy a textfájl egy szekvenciális fájl. Ne akarjunk olyan utasításokat, eljárásokat és függvényeket használni, amelyeket soros fájlokban értelmetlenség alkalmazni.

A nyomtatásra szánt sor karaktereit fűzzük össze, és ezután nyomtassuk ki a sort.

Arra minden esetben gondoljunk, hogy a textfájlban visszafelé nem mozoghatunk, és arra is, hogy a Crt unitban lévő pozicionáló eljárások nyomtatáskor nem alkalmazhatók. Helyettük a mezőszélesség pontos megadásával operáljunk.

Többlapos textfájl Lst-be való írása — a valódi nyomtatás — előtt nézzük meg az állományt a képernyőn, vagy deklaráljunk egy texttípusú

logikai nevet — aminek fizikai neve egy string —, és ebbe a fájlba írjuk ki először a TEXT állományt. Ha a TEXT állományt megtekintettük és jónak ítéltük, akkor a logikai név módosítása után a fájlt kinyomtathatjuk.

A nyomtatást szimulálhatjuk is, ha a NUL logikai eszközt használjuk. Ekkor olyan Output műveletet hajtunk végre, amely valójában nem ír semmilyen fájlba vagy eszközre. Ezt a megoldást is alkalmazhatjuk teszteléskor a nyomtató kímélése érdekében.

A nyomtatással kapcsolatosan szólni kell a karakterek nyomtatásáról is. Mivel a karakterek nyomtatása és a nyomtatót vezérlő karakterek függnek az alkalmazott nyomtatótól, ezért azt tanácsoljuk, hogy használt nyomtatójának karakterkészletét és annak alkalmazását a felhasználói kézikönyvből alaposan tanulmányozza át, hogy nyomtatáskor minél kevesebb kellemetlen meglepetés érje. Ezzel a kérdéssel könyvünkben nem foglalkozunk. Biztatásként elmondhatjuk, hogy a hardvereszközök fejlődésével egyre kényelmesebben és egységesebben tudjuk a különböző nyomtatókat használni.

A **kocsi vissza, soremelés** vezérl karakterének kiadását mindig helyettesíthetjük a Writeln(Lst) eljárással, ezért a vezérlő karakter megjegyzésével nem kell foglalkozni. Viszont a nyomtatással kapcsolatosan jelent meg a **lapdobás** (Page) fogalma. Lapdobás végzésére leporelló esetén is és lapok esetén is szükség lehet. A lapdobást is egy vezérlő karakter végzi. A különböző nyomtatóknál ezt a vezérlést a **12 ASCII kódú karakter** végzi. Így a lapdobás eljárását e karakter kiírásával adhatjuk ki. Pl.: Write(Lst, Chr(12)); vagy Write(Lst, #12); A nyomtatás végén a puffer ürítéséről lehetőleg ne feledkezzünk el. Ezt egy Write(Lst, #12); eljárás meghívásával is megtehetjük.

Feladatok

1. Készítse el az előző fejezet körlevelezését úgy, hogy a leveleket nyomtatóra írja.

2. Egy SZOVEG.TXT nevű lemezfájlt nyomtasson ki úgy, hogy egy lapra 60 sor kerüljön a 75 karakter szélességű papíron. A bal margó 5 karakter széles legyen. Minden lap alján a 62. sorban középen lapszámozás is legyen.

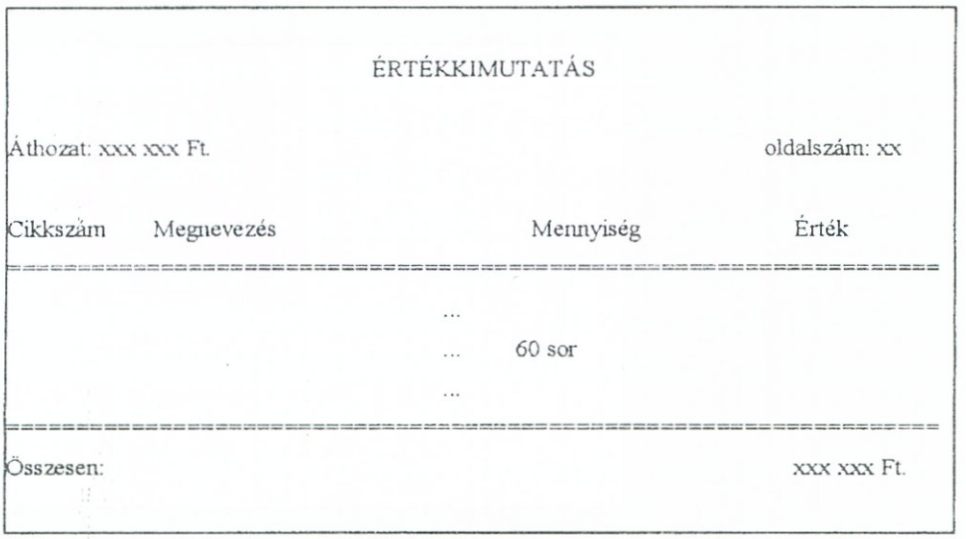
3. A nyomtatóra való munka gyakorlása érdekében fogalmazza meg korábban megoldott feladatait úgy, hogy ahol a feladat gyakorlati alkalmazása elvárhatja, ott nyomtatóra is dolgozzon programja. Mindegyik feladathoz készítsen dokumentációt is, amelyben leírja pontosan a probléma megfogalmazását és a program alkalmazásához szükséges tudnivalókat.

4. A RAKTAR.DAT egy raktárkészlet adatait tartalmazza. Rekordfelépítése:

```
CIKKSZ:        STRING[8];
```

```
MEGN:          STRING[30];
MENNY_E:       STRING[2];
E_AR:          REAL;
MENNY:         REAL;
```

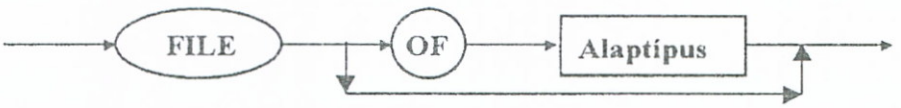
Készítsen kimutatást a raktárkészletről nyomtatón papírra, amely a következőképpen néz ki:



Az áthozat mindig az előző lap Összesen értéke legyen.

2.14.2.3. Nem tipizált fájl

A Turbo Pascalban a fájlokat három csoportba osztottuk. Eddigi fejezeteinkben foglalkoztunk a tipizált fájlokkal és a szöveges fájlokkal. Most a nem tipizált vagy típus nélküli fájlokról lesz szó.



A diagram is mutatja, hogy a komponensek típusát nem mindig jelöljük a fájloknál. Ekkor nem tipizált fájlokról beszélünk.

A típus nélküli állományok esetén a fájl komponenseire, alkotóelemeire nem a típusa, hanem a hossza jellemző. Deklarációja: Var F:FILE

Általában akkor alkalmazzuk a típus nélküli állományokat, amikor nem ismerjük az állomány pontos felépítését, csak azt tudjuk, hogy hány byte-ot kívánunk mozgatni, vagy gyors adatmozgatásra van szükségünk.

Az Assign(Var F; Fiznev:String) eljárással az F logikai fájlt és a Fiznev nevű fizikai állományt összerendeljük. Ezzel a Fiznev nevű fizikai állománnyal tartja az F logikai állomány a kapcsolatot.

Ezután megnyithatjuk az F fájlt a Reset(Var F [:File; Meret:Word]) eljárással írásra és olvasásra is. A megnyitás alkalmával a típus nélküli állományok esetén megadható az F fájl egy komponensének mérete is a Meret opcionális változóval. Ez azt jelenti, hogy Meret byte nagyságú csomagokat fog a program mozgatni a lemez és a memória között. A fájl rekordjának (komponensének) a mérete (hossza) alapértelmezésben 128 byte, ami jól illeszkedik a lemezen való tárolás blokkjaihoz. Ekkor a 128 byte hosszúságú rekordok direkt hozzáférése is biztosított. Ha az eljárást már nyitott állományra alkalmazzuk, akkor azt automatikusan lezárja újranyitás előtt. Ha a fájl nem létezik, és ekkor akarjuk megnyitni, akkor az I/O kapcsoló direktíva kikapcsolásával: {$I-} az I/O hiba lekérdezhető az IOResult függvény segítségével.

A Rewrite(Var F [:File; Meret:Word]) eljárás is hasonlóan működik. Az Assign összerendelő eljárás után létrehozza az új fájlt és megnyitja írásra. Ha az állomány már korábban is létezett, akkor annak tartalmát a Rewrite előbb törli, majd létrehozza és megnyitja az újat. Ennél az eljárásnál is megadható a rekord hossza a Meret értékével. Megállapíthatjuk tehát, hogy a fájlok megnyitási eljárásai a típusos és típus nélküli fájloknál hasonlóan működnek, de a típus nélkülieknél a fájl alkotóelemének a mérete is megadható.

A típus nélküli fájlok nagy előnye, hogy az I/O műveletek esetén nem használnak puffert. Ez a sajátosságuk a műveleti gyorsaságban kamatozik.

A rekordok elérése szekvenciálisan vagy a komponens méretének ismerete miatt — annak sorszáma alapján — direkt módon is történhet. A létező komponensek mutatója most is 0-val kezdődik és lehetséges legnagyobb értéke Filesize(F)−1. Filepoz(F)=Filesize(F) esetén Eof(F) értéke True, vagyis a fájl végén áll a fájlmutatója.

A típus nélküli fájl minden fájllal kompatibilis.

A típus nélküli állomány írható és olvasható is. Minden korábban tanult — a típusos fájlokkal kapcsolatos — eljárás és függvény alkalmazható, csak az I/O eljárások változtak. Az Assign, Close, Eof, Erase, FilePos, FileSize, IOResult, Rename, Reset, Rewrite, Seek, Truncate eljárások most is alkalmazhatók.

A Read, Readln, Write, Writeln eljárások helyére a BlockRead és a BlockWrite eljárások kerültek.

A BlockRead(Var F:File; Var Buf; Db:Word [;VarV:Word]); és a BlockWrite(Var F:File; Var Buf; Db:Word [;V:Word]); eljárások Db, vagy Db-nél kevesebb rekordot mozgatnak a fájl és a memória között. A BlockRead olvas az F fájlból, a BlockWrite pedig ír a fájlba. A mozgatott csomag nagyságát az F fájl megnyitásakor megadott rekordnagyság (Meret: Word) határozza meg. A Buf egy tetszőleges változó, a fájl puffere. Hosszúságának

legalább akkorának kell lennie, mint a fájl rekordmérete, erről a programozónak kell gondoskodni. Az I/O eljárás akkor is végrehajtódik, ha a Buf változó rövidebb, mint a mozgatott rekord, de ekkor a rekord rátelepszik a Buf nevű változó memóriaterület utáni memóriacímekre is, amiből végzetes hibák keletkezhetnek. A maximális rekordméret 65 536 byte=64 Kbyte, alapértelmezésben 128 byte. A V megadása opcionális, és megmutatja az eljárások végrehajtása után a ténylegesen átvitt rekordok számát ([17]).

Ha V=Db, akkor a teljes átvitel megtörtént. Ha V<Db, akkor már a fájl végéhez értünk az előírt átvitel vége előtt. Ebben az esetben I/O hiba lép fel, ha a V opcionális változó nincs specifikálva.

Kidolgozott feladat

Másoljunk át egy 'NEMTIP.PAS' állományt 'UJ.PAS' nevű állományba úgy, hogy a másolás pontos legyen, és közben állapítsa meg a program, hogy hány byte az átmásolt állomány.

A biztonság kedvéért a buffer változót deklaráljuk nagy méretűre, hogy ne töltődjön a fájl olyan területre, amelyre más változónál is szükség lehet. A logikai fájlok rekordjainak a hosszát éppen egy byte hosszra adtuk meg a fájl megnyitásakor. Az igaz, hogy így programunk nem használja ki a lehetséges gyorsaságot, de ekkor pontosan meg tudjuk számolni a lemezes fájl hosszát.

```
program nemtipusos;
var honnan, hova: file;
    buffer: array[1..32767] of byte;
    db,v:word;
begin assign(honnan,'nemtip.pas'); reset(honnan,1);
    db:=filesize(honnan);
    assign(hova,'uj.pas'); rewrite(hova,1);
    blockread(honnan,buffer,db,v); blockwrite(hova, buffer,db,v);
    if db=v then
      write('Rendben, az átmozgatás. Az állomány
                hossza ', db,' byte');
    close(honnan); close(hova);
end.
```

Gépelje be a programot, és nézze meg működését. Változtassa meg az eljárások paramétereit, és nyomkövetéssel futtassa a programot.

Feladatok

1. Mentsük ki egy karakteres képernyő teljes tartalmát egy .TXT kiterjesztésű fájlba.

2. Olvassuk be a karakteres képernyőt a megfelelő memóriaterületre.

3. Készítsen az előző két feladat felhasználásával jól használható karakteres képernyőt mentő és beolvasó programot.

4. Készítsen grafikus ablak elmentéséhez és beolvasásához programot.

5. Készítsen programot, mely egy grafikus képet elhelyez a képernyője adott koordinátájú helyén.

Irodalomjegyzék

[1] Angster E.: Programozás tankönyv I., II. Turbo Pascal. Budapest, 1995.

[2] Angster E.—Kertész L.: Turbo Pascal 5.0-5.5 'A'..'Z'

[3] Angster E.—Kertész L.: Turbo Pascal 6.0. Budapest, 1992.

[4] Angster E.—Kertész L.: Turbo Pascal 6.0 feladatgyűjtemény I., II. Budapest, 1991.

[5] Áts L.: Turbo Pascal kezdőknek. NOVOTRADE Rt., Budapest, 1989.

[6] Benkő T.-né—Benkő L.—Meszéna Zs.—Gyenes K.: Programozási feladatok és algoritmusok Turbo Pascal nyelven. Computerbooks, Budapest, 1996.

[7] Benkő T.-né—Hegedűs A.—Benkő L.: IBM programozása Turbo Pascal nyelven. (Példatár I., II.) BME Mérnöktovábbképző Intézet, Budapest, 1991.

[8] Benkő T.-né—Kiss Z.—Tamás P.—Tóth B.: Programozás Borland Pascal 7.0 rendszerben. Computerbooks, Budapest, 1997.

[9] Boisgontier, J.—Donnay, C.: Turbo Pascal fájlkezelő alkalmazások. Műszaki Könyvkiadó, Budapest, 1990.

[10] Csépai J.: A számítástechnika alapjai. Műszaki Könyvkiadó, Budapest, 1985.

[11] Csőke L.—Garamhegyi G.: A számítógép-programozás logikai alapjai. Nemzeti Tankönyvkiadó Rt., Budapest, 1997.

[12] Gordon E.—Körtvélyesi G.-né—Sós I.—Székely Z.: A Pascal programozási nyelv. Számítástechnikai Alkalmazási Vállalat, Budapest, 1982.

[13] Horváth T.—Kiss M.: Turbo Pascal 7.0. (Könyv és feladatgyűjtemény kezdőknek.) Nemzeti Tankönyvkiadó, Budapest, 1997.

[14] Jensen, K.—Wirth, N.: A Pascal programozási nyelv. Műszaki Könyvkiadó, Budapest, 1988.

[15] Löbel—Müller—Schmid: Számítástechnikai kislexikon. Műszaki Könyvkiadó, Budapest, 1973.

[16] Pirkó J.: Turbo Pascal kezdőknek-haladóknak 4.0 verzióig LSI Oktatóközpont, Budapest, 1989.

[17] Pirkó J.: Turbo Pascal 5.5. LSI Oktatóközpont, Budapest, 1990.

[18] Pirkó J.: Turbo Pascal 7.0. LSI Oktatóközpont, Budapest.

[19] Racskó P.: Bevezetés a számítástechnikába. LSI Oktatóközpont, Budapest.

[20] Sain M.: Matematikatörténeti ABC. Tankönyvkiadó, 1974.

[21] SIMON GY.: Számítástechnika középiskolásoknak. Pedellus Bt., Debrecen, 1995.
[22] SCHUSZTER E.—UGRAI L.—BRECSKA E.: Adatfeldolgozási rendszerek szervezése. Műszaki Könyvkiadó, Budapest, 1978.
[23] Számítástechnika középfokon I., II. Szerk.: HETÉNYI P.-NÉ. OMIKK, Budapest, 1987.
[24] Turbo Pascal Reference Guide Version 5.0, 5.5. Borland International, Inc., 1989.
[25] A Turbo Pascal 6.0 verziójának Help rendszere. Borland International, Inc., 1990.
[26] VARGA L.: Rendszerprogramok elmélete és gyakorlata. Akadémiai Kiadó, Budapest, 1978.
[27] WIRTH, N.: Algoritmusok+Adatstrukturúk=Programok. Műszaki Könyvkiadó, Budapest, 1982.

SZAH-501

Molnár és Társa "2001" Kft. Eger, Tel./fax: 36/422-946