A pascal

PROGRAMOZÁSI NYELV

Felhasználói kézikönyv és a nyelv formális leírása

K. Jensen–N. Wirth

Kathleen Jensen
Niklaus Wirth

A PASCAL programozási nyelv

Kathleen Jensen
Niklaus Wirth

A PASCAL programozási nyelv

Felhasználói kézikönyv és a nyelv formális leírása

(az ISO Pascal szabvány szerint)

2., bővített kiadás

Műszaki Könyvkiadó, Budapest, 1988

Az eredeti mű:

Kathleen Jensen – Niklaus Wirth

Pascal User Manual and Report

Third Edition, Prepared by A. B. Mickel – J. F. Miner

Illustrations by William W. Porter

Lektorálta:

BAKOS TAMÁS

ETO: 519.682
681.3.06
ISBN: 963 10 7735 7

Tartalomjegyzék

Jelentés

(A nyelv formális leírása)

Előszó az átdolgozott kiadáshoz

A Jensen–Wirth szerzőpáros Pascal-leírása közel egy évtizede a legfontosabb tankönyve a nyelvvel ismerkedőknek, s egyszersmind kézikönyve a Pascal-programozóknak. A hetvenes évek folyamán a Pascal népszerűsége a legmerészebb várakozásokat is felülmúlva robbanásszerűen nőtt, s napjainkra világszerte az egyik legáltalánosabban használt programnyelvvé vált. Míg régebben – legalábbis az Egyesült Államokban – a gyakorlati szakemberek gyakran nagyobb érdeklődést mutattak a nyelv iránt, mint a tudományos és oktatási intézmények, ma az egyetemek többségén a Pascal a programozástanítás nyelvi eszköze. A Pascal mindinkább kiszorítja a PL/1-et és az ALGOL-60-at, sőt egyes újításai még a FORTRAN korszerűsített változataiba is beépültek.

A Pascal User Group (Pascal Felhasználói Csoport) és a Pascal News (Pascal Hírek) munkatársaiként tanúi lehettünk a nyelv elterjedésének. 1971-ben még csak egy számítógéprendszeren futott Pascal fordítóprogram, 1974-ben már tízen, 1979-re több, mint nyolcvanon. Ma valamennyi korszerű gépen rendelkezésre áll a nyelv; a mindenhová bevonult személyi számítógépek, az intelligens munkaterminálok elképzelhetetlenek a Pascal nyelv nélkül.

Az 1977-es southamptoni Pascal-konferencián felmerült kérdések [10] hatására kezdődött meg a szervezett munka egy hivatalosan elfogadott, nemzetközi Pascal-szabvány kidolgozására. A résztvevők igyekeztek összegyűjteni és egységesen megválaszolni mindazokat a kérdéseket, amelyek felmerültek, ha valaki a Jensen–Wirth műben lefektetett definíciók alapján Pascal-fordítót próbált írni. E munka eredményeként született meg az ISO 7185 Pascal Szabvány [11], a Pascal hivatalos definíciója, amely a Jensen–Wirth könyv átdolgozását is szükségessé tette.

Az átdolgozással nem az volt a célunk, hogy a könyv vegye át a szabvány szerepét. Helyesebbnek láttuk, ha csupán a szabványnak megfelelően módosítjuk az anyagot, s így minél inkább megőrizzük olvasmányosságát és eleganciáját, amely – véleményünk szerint – a szabványtól megkülönbözteti. Korszerűsítettük a szintaxis leírását (Wirth EBNF-rendszerét alkalmaztuk), javítottunk a Felhasználói kézikönyv programpéldáinak stílusán. Akik a könyvet korábbi kiadásból ismerik, különösen hasznosnak találhatják az E. függeléket, amelyben a szabvány által szükségessé tett változtatásokat foglaltuk össze.

Végül úgy érezzük, nem lenne teljes a könyv, ha nem emlékeznénk meg arról, hogy a Pascal programnyelv nevét Blaise Pascalról, a XVII. századi francia matematikusról és filozófusról kapta, aki számológép-építéssel is foglalkozott. Köszönettel tartozunk Roberto Minionak és Niklaus Wirth-nek a könyv átdolgozásának támogatásáért, Henry Ledgard-nak mindig a legjobbkor jött és hasznos tanácsaiért, Elise Oranges-nak a határidők betartásához nyújtott lelkiismeretes segítségéért, William W. Porternek a grafikáért és Linda Strzegowskinak a könyv szedéséért.

Andy Mickel
Jim Miner

Minneapolis, USA,
1984. november

Előszó az első kiadáshoz

A Pascal programozási nyelv első, vázlatos formáját 1968-ban fogalmaztuk meg. Szellemében az új nyelv az ALGOL-60 és az ALGOL-W vonalát követte. 1970-ben, intenzív fejlesztési munka eredményeként, elkészült az első működő fordítóprogram, majd egy évvel ezután megjelentek a nyelvvel kapcsolatos publikációk [1], [8]. Egyre növekedett az igény, hogy más számítógépekre is elkészüljön a nyelv fordítóprogramja. Ez, valamint a Pascal kétéves alkalmazása során szerzett tapasztalatok néhány módosítást tettek szükségessé. Így került sor 1973-ban a Revised Report (Átdolgozott Jelentés) kiadására, ill. a nyelv ISO karakterkészlettel való leírására.

Az Olvasó két részből álló könyvet tart a kezében. Az első rész a Felhasználói kézikönyv, a második a nyelv formális leírása, az átdolgozott Jelentés. A kézikönyvet azoknak szánjuk, akik foglalkoztak már számítógép-programozással és most a Pascal programnyelvvel szeretnének megismerkedni. Éppen ezért e kézikönyvet tankönyvnek szántuk, amely a Pascal különböző tulajdonságait számos példán keresztül szemlélteti. Függelékként csatoltuk az összefoglaló táblázatokat és a szintaxis leírását.

A könyv második részének az a célja, hogy tömör összefoglaló leírást adjon a programozók, a fordítóprogram-készítők és a nyelv egyéb felhasználói részére. Az átdolgozott Jelentés (Revised Report) az ún. standard Pascalt definiálja, amely a nyelv különböző változatainak közös alapját képezi.

Azt ajánljuk, amennyiben az Olvasó kézikönyvünket tankönyvként használja, tartsa magát a kézikönyv szerkezetéhez, és olvasás közben fordítson különös figyelmet a példaprogramokra. Inkább olvassa el újra azokat a részeket, amelyek nehézséget okoznak. Ha a beviteli és a kiviteli eljárásokkal kapcsolatban kérdések merülnek fel, különösen nagy szükség lehet a 13. fejezet ismételt átolvasására.

A könyv a standard Pascalt írja le. A standard Pascal feldolgozása az elsődleges követelmény, amit a fordítóprogram-készítő támaszt az általa megvalósított rendszerrel szemben. Csak a standard Pascalban leírt nyelvi lehetőségekkel élhet az a programozó, aki egyik számítógépről a másikra átvihető portábilis programot akar írni. A nyelv különböző megvalósított változatai természetesen ehhez képest további szolgáltatásokat is tartalmazhatnak, ezeket azonban mindenképpen kiterjesztésként kell kezelni.

A könyv sokak munkájának eredménye. Külön köszönettel tartozunk a Zürichi Műszaki Egyetem Informatikai Intézete (Institut für Informatik, ETH Zürich) munkatársainak, valamint John Larmouth-nak, Rudy Schildnek, Olivier Lecarme-nak és Pierre Desjardins-nek bírálataikért, javaslataikért, bátorításukért.

A Pascal-megvalósítás, amely könyvünk megjelentetését lehetségessé és ugyanakkor szükségessé tette, Urs Ammann és munkatársa, Helmut Sandmayr munkája.

Kathleen Jensen
Niklaus Wirth

ETH Zürich,
Svájc, 1974. november

Kathleen Jensen
Niklaus Wirth

Felhasználói kézikönyv

1. Bevezetés

1.1. A Pascal programok felépítése

A következőkben abból indulunk ki, hogy az Olvasónak, ha csak minimálisan is, vannak számítástechnikai ismeretei, és nagyjából tudja, milyen felépítésű egy számítógépprogram. Ebben a fejezetben ezeket a többé-kevésbé hézagos ismereteket szeretnénk felfrissíteni, tudatosítani.

Egy *algoritmus* vagy számítógépprogram mindig két fő részből áll: az elvégzendő *tevékenységekből* (műveletek) és az ezek által kezelt *adatok leírásából.* A tevékenységeket ún. *utasításokkal,* az adatokat ún. *deklarációkkal* és *definíciókkal* adjuk meg.

A program *programfejre* és *blokknak* nevezett törzsre tagolódik. A fej nevet ad a programnak, és felsorolja a program paramétereit. A paraméterek (állomány típusú) változók, és a számítás argumentumait, ill. eredményeit reprezentálják. A blokk hat részből áll, amelyek közül – az utolsót kivéve – bármelyik üres is lehet. A hat rész sorrendje kötelezően a következő:

Blokk = CímkedeklarációsRész
KonstansdefiníciósRész
TípusdefiníciósRész
VáltozódeklarációsRész
EljárásÉsFüggvénydeklarációsRész
Utasításrész

Mutatja ezt az alábbi program is:

```
program Inflacio (Output);
{Turbo Pascal}
{Legyen az inflacio gyorsasaga evi 7,8, ill. 10%!
 Kerdes, hogy milyen mertekben devalvalodik valamely
 valuta (frank, dollar,font, marka, rubel, jen vagy
 holland forint) 1, 2, ..., n ev alatt. }

const MaxEv =10;

var   Ev: 0..MaxEv;
      Tenyezo1, Tenyezo2, Tenyezo3: Real;

begin Ev := 0 ;
      Tenyezo1 :=1.0; Tenyezo2 :=1.0; Tenyezo3 :=1.0;
      Writeln(' Ev     7%      8%     10% '); Writeln;
      repeat Ev := Ev+1;
             Tenyezo1 := Tenyezo1 * 1.07;
             Tenyezo2 := Tenyezo2 * 1.08;
             Tenyezo3 := Tenyezo3 * 1.10;
             Writeln(Ev:5,Tenyezo1:7:3,Tenyezo2:7:3,
                     Tenyezo3:7:3 )
      until Ev = MaxEv
end.
```

A program eredménye:

```
Ev     7%      8%      10%
 1   1.070   1.080   1.100
 2   1.145   1.166   1.210
 3   1.225   1.260   1.331
 4   1.311   1.360   1.464
 5   1.403   1.469   1.611
 6   1.501   1.587   1.772
 7   1.606   1.714   1.949
 8   1.718   1.851   2.144
 9   1.838   1.999   2.358
10   1.967   2.159   2.594
```

Az első rész felsorolja az adott blokkban definiált összes címkét. A második rész a konstansok „szinonimáit" definiálja, azaz azonosítókat vezet be, amelyek aztán az illető konstansok helyett írhatók. A harmadik típusdefiníciókat tartalmaz, a negyedik pedig a változók definícióit. Az ötödik kijelöli az alárendelt programrészeket (vagyis az eljárásokat és a függvényeket). Az utasításrész az elvégzendő tevékenységeket adja meg.

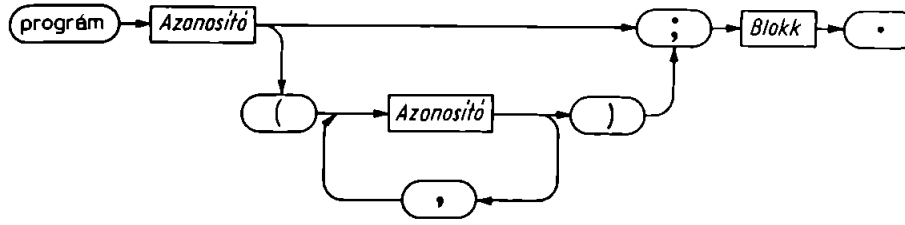
1.2. Szintaxisdiagramok

A program előbb leírt vázlatos szerkezetét pontosabban fejezi ki az 1.1. ábrán látható *szintaxisdiagram*. A *Program* szótól elindulva, ha az ábrán a nyilakat követve végighaladunk, minden lehetséges út egy szintaktikusan helyes programot ad. Az ábra téglalapjai a beírttal azonos nevű szintaxisdiagramokat helyettesítenek. A téglalapok jelentését tehát ezek a részletes szintaxisdiagramok definiálják (1.2. ábra). A Pascal programban ténylegesen előforduló ún. terminális szimbólumokat ovális alakú mezőbe írjuk. (A Pascal teljes szintaxisának leírása a D. függelékben található.)

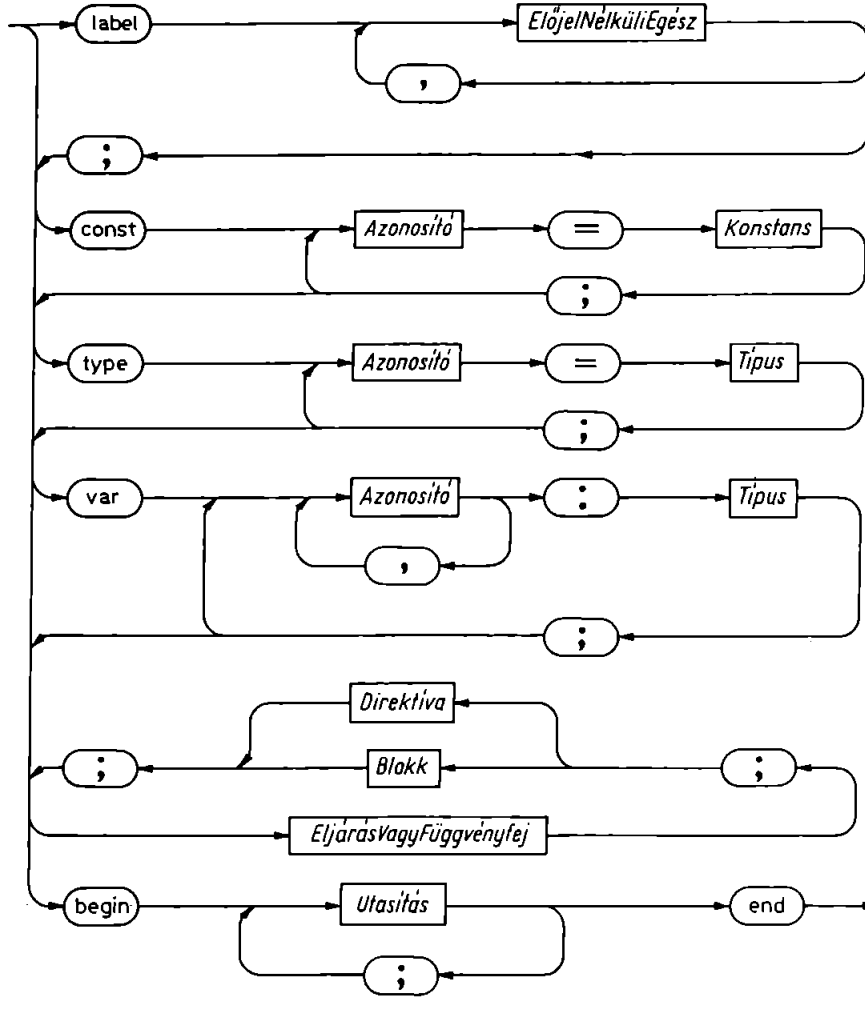
1.3. Az EBNF

A szintaxisleírás másik lehetséges módját a kiterjesztett (Extended) *Backus–Naur-forma* (EBNF) alkalmazása jelenti, ahol a szintaktikai egységeket szavak és füzérek, más szóhasználattal literálok vagy karakterláncok jelölik. A szavak jelentése az egyes egységek jellegére, tartalmára utal, míg a füzérek a nyelvben ténylegesen használt szimbólumok. A füzéreket idézőjelek határolják.

Ha egy szintaktikai egységekből álló sorozatot kapcsos zárójelek ({ és }) közé írunk, ez a sorozat nullaszori vagy többszöri ismétlését jelenti. Az alternatívákat függőleges vonallal (|) választjuk el egymástól. A gömbölyű zárójelek ((és)) csoportosításra szolgálnak, a szögletes zárójelek ([és]) pedig azt fejezik ki, hogy a közéjük írt szintaktikai egységek, ill. füzérek elhagyhatók. (A Pascal teljes EBNF-leírása a D. függelékben található.)



1.1. ábra. *Program* szintaxisdiagramja



1.2. ábra. *Blokk* szintaxisdiagramja

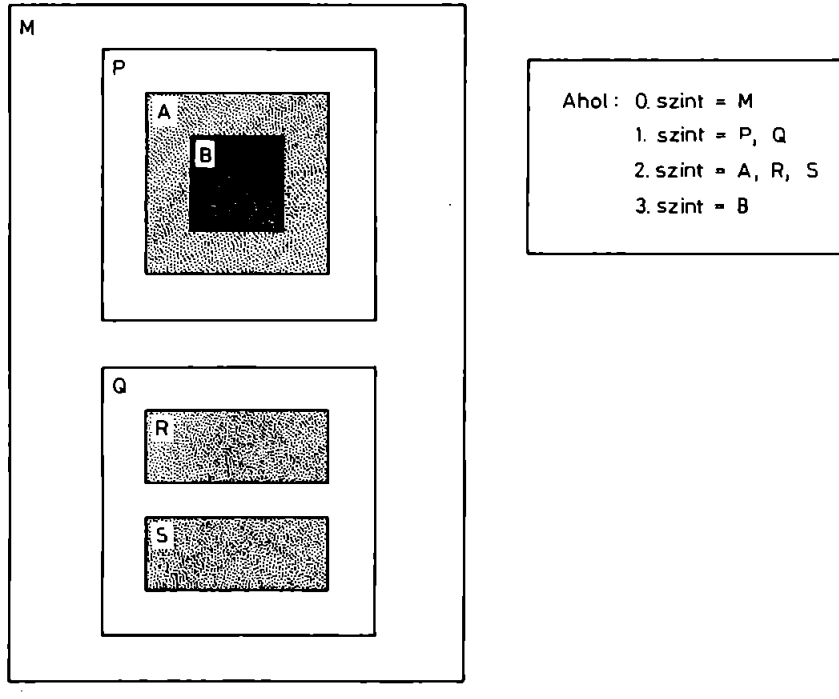
Az 1.1. ábra *Program*-ját pl. a következő formulák, ún. képzési szabályok írják le:

Program = *Programfej* ";" *Blokk* ".".
Programfej = "program" *Azonosító* ["(" *Azonosítólista* ")"].
Azonosítólista = *Azonosító* { "," *Azonosító* }.

1.4. Az azonosítók hatásköre

Minden eljárás- és függvénydeklaráció a programhoz hasonló felépítésű: fejre és blokkra tagolódik. Következésképpen az eljárás- és függvénydeklarációk más eljárásokba vagy függvényekbe ágyazhatók. A címkék, konstansazonosítók, típus-, változó- és eljárásdeklarációk *lokálisak* arra az eljárásra vagy függvényre nézve, amelyben deklarációjuk szerepel. Ez azt jelenti, hogy a nekik megfelelő azonosítók csak az utasításoknak a kérdéses blokkot alkotó részében értelmezettek. Ezt a programrészt az illető azonosító *hatáskörének* nevezzük. A blokkok egymásba ágyazhatóságából következően a hatáskörök is egymásba ágyazhatók. A főprogramban deklarált, azaz egyik eljárásra vagy függvényre nézve sem lokális objektumokat *globális objektumoknak* nevezzük, s ezek a teljes programban mindenütt értelmezettek.

Minthogy eljárás- vagy függvénydeklarációkkal a blokkok egymásba ágyazhatók, mindegyik blokkhoz rendelhetünk egy számot, amely a beágyazás mélységét, szintjét mutatja. Ha a program által definiált legkülső blokkot, azaz a főprogramot 0 szintnek tekintjük, akkor az ebben a blokkban definiált blokkok 1-es szintűek lesznek. Általában minden *i* szintű blokkban definiált blokk $(i + 1)$ szintű lesz. Az 1.3. ábrán egy lehetséges blokkstruktúrát mutatunk be.



1.3. ábra. Blokkszerkezet

Ez a blokkstruktúra a következő programsémához tartozhat:

```
program M;
      procedure P;
            procedure A;
                  procedure B;
                  begin
                  end {B};
            begin
            end {A};
      begin
      end {P};
      procedure Q;
            procedure R;
            begin
            end {R};
            procedure S;
            begin
            end {S};
      begin
      end {Q};
begin
end {M}.
```

Ebből a szemszögből nézve úgy is mondhatjuk, hogy valamely X azonosító hatásköre vagy értelmezési tartománya az a teljes blokk, amelyben X-et definiáltuk, ideértve tehát azokat a blokkokat is, amelyeket X-szel azonos blokkban definiáltunk. (Vegyük észre, hogy példánk esetében minden azonosító különböző kell, hogy legyen. A 4.7. szakaszban foglalkozunk majd azzal az esettel, amikor nem kell feltétlenül minden azonosítónak különbözőnek lennie.)

Az alábbi blokkok	a következő blokkokban található objektumokat érhetik el:
M	M, P, A, B, Q, R, S
P	P, A, B
A	A, B
B	B
Q	Q, R, S
R	R
S	S

1.5. Összehasonlítás más programnyelvekkel

ALGOL, PL/1 vagy FORTRAN ismeretekkel rendelkező programozók talán hasznosnak találják, ha a Pascal tulajdonságait ezekkel a nyelvekkel összehasonlítva is megfogalmazzuk. Ilyen szempontból a Pascal következő jellemzőit sorolhatjuk fel:

(1) A változók deklarációja kötelező.
(2) Bizonyos alapszavak (pl. begin, end, repeat) „foglaltak", azonosítóként nem használhatók.
(3) A pontosvesszőt (;) utasításelválasztónak tekintjük.
(4) A nyelv standard típusai az egész és a valós számok, a logikai értékek és a (kinyomtatható) karakterek. Az alapvető adatstruktúrák a következők: a tömb, a rekord (a COBOL és a PL/1 „structure"-énck megfelelője), a halmaz és a (szekvenciális) állomány. Ezek a struktúrák kombinálhatók, egymásba ágyazhatók, s így halmazokból képzett tömbök, rekordokból álló állományok stb. állíthatók elő. Mód van a dinamikus helyfoglalásra, és az adatok mutatók segítségével is elérhetők. Ezek a mutatók teljesen általános listafeldolgozást tesznek lehetővé. Mód van arra, hogy szimbolikus konstansokkal új, alap-adattípusokat deklaráljunk.
(5) A halmaz (set) adatstruktúra a PL/1 „bitstring"-jéhez hasonló lehetőségeket biztosít.
(6) A tömbök tetszőleges dimenziójúak és méretűek lehetnek, de a tömbhatárok állandóak. (Dinamikus tömbök tehát nincsenek.)
(7) A FORTRAN-hoz, az ALGOL-hoz és a PL/1-hez hasonlóan itt is van goto utasítás. A címkék előjel nélküli egészek, és deklarációjuk kötelező.
(8) Az összetett utasítás ugyanaz, mint az ALGOL-ban, és megfelel a PL/1 DO utasításának.
(9) Az ALGOL kapcsolójának (switch) és a FORTRAN számított goto-jának megfelelő funkciót a Pascalban a case utasítás látja el.
(10) A for utasítás a FORTRAN DO ciklus megfelelője, csak 1 (to) vagy −1 (downto) lépésközzel rendelkezhet. Csak addig fut, míg a ciklus vezérlőváltozója az előírt határok között van. Előfordulhat tehát, hogy a ciklustörzs egyszer sem hajtódik végre.
(11) Nincsenek feltételes kifejezések és nincs többszörös értékadás.
(12) Az eljárások és a függvények rekurzívan is hívhatók.
(13) Az ALGOL-tól eltérően a változók nem lehetnek „own" tulajdonságúak.
(14) A paramétereket értékükkel vagy hivatkozással adjuk át. Név szerinti hívás nincs.
(15) A blokkstruktúra bizonyos fokig eltér az ALGOL-ban vagy a PL/1-ben alkalmazottól, ugyanis a Pascalban nincsenek név nélküli blokkok, más szóval minden blokknak nevet kell adni, így minden blokk eljárássá vagy függvénnyé válik.
(16) Minden objektumot – konstansot, változót stb. – még *azelőtt* kell deklarálni, hogy rá hivatkozás történne. Megengedett azonban két kivétel:
(a) a mutatótípus definíciójában álló típusazonosító (l. a 11. fejezetet);
(b) az eljárás- és függvényazonosítók, ha van előzetes deklaráció (l. a 12.3. szakaszt).

Azok, akik először találkoznak a Pascallal és még nem volt alkalmuk mélyebben megismerni a nyelvet, gyakran nehezményezik, hogy bizonyos „népszerű" eszközök nincsenek meg benne. Példaként említik a többi között a hatványozó operátort, a füzérek konkatenációját, a dinamikus tömböket, a logikai értékekkel végzett aritmetikai műveleteket, az automatikus típusátalakítást, az automatikus deklarációt. Ezek nem tévedésből, hanem szándékosan maradtak ki a nyelv eszköztárából. Egyes esetekben engedélyezésük kevésbé hatékony programozási megoldásokra csábítana, más esetekben pedig úgy éreztük, hogy ellentétben állna céljainkkal: az áttekinthetőséggel, a megbízhatósággal, a „jó programozási stílussal". Nem utolsósorban pedig szigorúan meg kellett rostálnunk a rengeteg rendelkezésre álló programozási eszközt, hiszen azt akartuk, hogy a fordítóprogram viszonylag tömör és hatékony legyen, hogy minden felhasználó hatékonyan és gazdaságosan alkalmazhassa – az is, aki kis programokat ír, s a nyelvnek csak kevés eszközét használja, de az is, aki nagy programokat készít, s a nyelv minden lehetőségét igyekszik kiaknázni.

2. Jelölések: szimbólumok és elválasztók

A Pascal programok szimbólumokból és szimbólum-elválasztókból (röviden elválasztókból) állnak. A szimbólumok a következő csoportokba sorolhatók: *speciális szimbólumok, szószimbólumok* (alapszavak), *azonosítók, számok, füzérek, címkék* és *direktívák*. A következő szakaszban az *elválasztókkal* foglalkozunk.

2.1. Elválasztók

A szóközt, a sor végét és a magyarázatot (megjegyzést, commentet) elválasztóként használjuk. A Pascal szimbólumok belsejében elválasztó (sem annak része) nem szerepelhet. Bármely két azonosító, szám vagy alapszó között azonban legalább egy elválasztót kell alkalmazni.

A *magyarázat* (comment) { vagy (* jellel kezdődik (nem füzér belsejében), és } vagy *) zárja le. A magyarázat tetszőleges, karakterekből és sorvégekből álló jelsorozat lehet, de {-et vagy *)-ot nem tartalmazhat. Ha a programszövegben a magyarázat helyére szóközt írunk, a szöveg jelentése nem változik.

A szóközök, üres sorok (sorvégek) és magyarázatok közbeiktatása gyakran olvashatóbbá teszi a Pascal programot.

2.2. Speciális szimbólumok és alapszavak

A következőkben a Pascal programokban használatos speciális szimbólumokat és alapszavakat tekintjük át. Fontos, hogy a kétkarakteres speciális szimbólumok között nincs elválasztó! A speciális szimbólumok:

```
+    -    *    /
.    ,    :    ;
=    <>   <    <=   >    >=
:=   ..   ↑
(    )    [    ]
```

Egyes speciális szimbólumokat más jelekkel pótolhatunk:

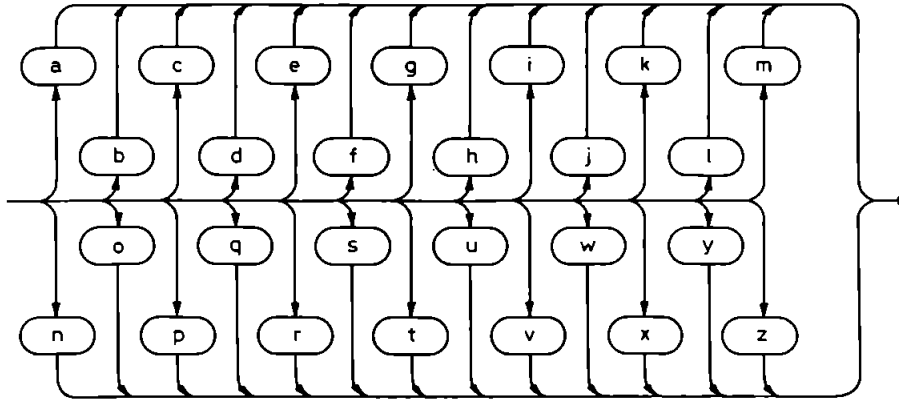
[helyett (.
] helyett .)
↑ helyett @ vagy ^

Az *alapszavakat* a kézzel írt programban általában aláhúzzuk: így jelezzük, hogy karaktereik egyetlen, rögzített jelentésű szimbólumot alkotnak. Ezeket a szavakat kizárólag a Pascal definíciójában lefektetett összefüggésben szabad használni. Alapszó sosem lehet azonosító! Az alapszavak egyszerűen kis- vagy nagybetűk sorozatai; kezdetük vagy végük jelölésére nincs szükség speciális karakterekre. Az alapszavak:

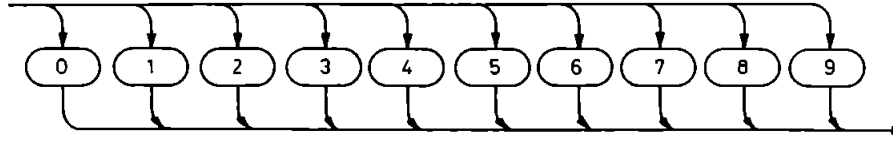
and	end	nil	set
array	file	not	then
begin	for	of	to
case	function	or	type
const	goto	packed	until
div	if	procedure	var
do	in	program	while
downto	label	record	with
else	mod	repeat	

2.3. Azonosítók

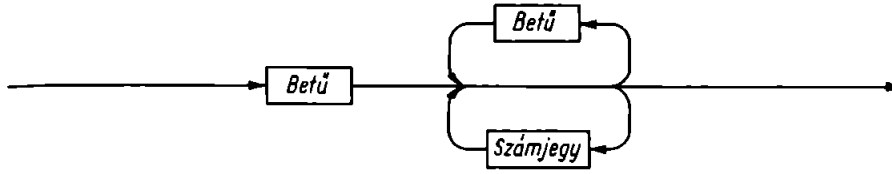
Az *azonosítók* a konstansok, típusok, tartományhatárok, változók, eljárások és függvények jelölésére szolgáló nevek. Betűvel kell kezdődniük, amely után viszont betűk és számok tetszőleges kombinációja állhat. Az azonosítók hossza nincs korlátozva, a fordítóprogram valamennyi karakterüket értékesnek tekinti. Az összetartozó kis- és nagybetűk azonosnak számítanak.



2.1. ábra. *Betű* szintaxisdiagramja



2.2. ábra. *Számjegy* szintaxisdiagramja



2.3. ábra. *Azonosító* szintaxisdiagramja

Példák azonosítókra:

Telefonkönyv* Gyök3 Pi h4g_X
EzEgyNagyonHosszúDeAzértHelyesAzonosító
EzEgyNagyonHosszúDeAFentitőlKülönbözőAzonosító

A BetűkÉsSzámok azonosító megegyezik a betűkésszámok azonosítóval.

Hibás azonosítók:

7Törpe array Szint.4 Gyök-3 Tizedik_Bolygó

Bizonyos ún. *standard azonosítók* előredefiniáltak (pl. sin, cos). Az alapszavakkal (pl. array) ellentétben nem kell az így megadott definíciókhoz ragaszkodnunk, tehát bármelyik standard azonosító újradefiniálható. Ennek az az oka, hogy a standard azonosítók a programban úgy szerepelnek, mintha egy, az egész programot körülvevő hipotetikus blokkban deklaráltuk volna őket. A Pascal összes standard azonosítóját megadtuk a C. függelékben.

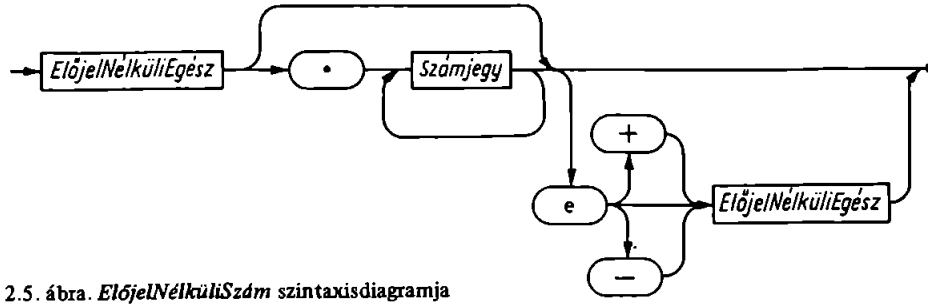
2.4. Számok

A *számokat* – egész és valós értékeket – decimális írásmódban írjuk. A számok előtt előjel (+ vagy −) állhat; az *előjel nélküli számok* esetén nem szabad előjelet használni!

A számok vesszőt nem tartalmazhatnak. A valós számokat tizedestörtként és/vagy 10 hatványaival írjuk le. A kitevő előtt egy E (vagy e) betű áll; jelentése: „...-szer 10 a ... hatványon". Figyeljük meg, hogy ha a szám tizedespontot tartalmaz, a pont előtt és után is legalább egy-egy számjegynek kell állnia!

2.4. ábra. *ElőjelNélküliEgész* szintaxisdiagramja

*A közölt programpéldáknál és részleteknél, amelyek nem nyomtatón készültek, meghagytuk az angol jelkészletben nem szereplő ékezetes karaktereket. *(A szerk.)*



2.5. ábra. *ElőjelNélküliSzám* szintaxisdiagramja

Előjel nélküli számok:

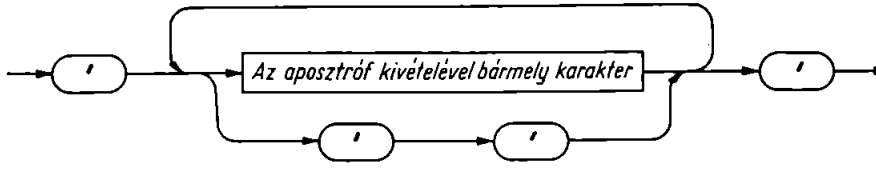
3 03 6272844 0.6 5E-8 49.22E+08 1E10

Hibásan felírt számok:

3,487,159 XII .6 E10 5.E-16 öt 3.487.159

2.5. Füzérek

Az egyszeres idézőjelek (aposztrófok) közé zárt karaktersort *füzérnek* (string) nevezzük. Ha aposztrófot tartalmazó füzérre van szükségünk, a megfelelő helyen két aposztrófot kell írni.



2.6. ábra. *Füzér* szintaxisdiagramja

Példák:

'a' ':' '3' 'begin' 'BNV''86'
' Ez 33 karakteres füzér.'

2.6. Címkék

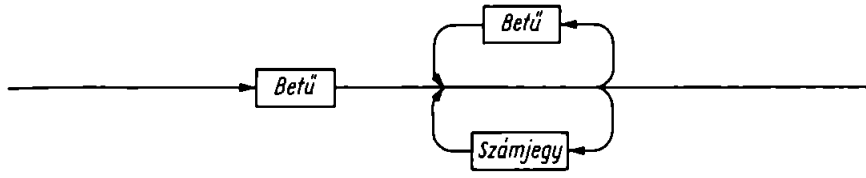
A címkék előjel nélküli egész számok, amelyeket Pascal utasítások megjelölésére használunk. Értékük 0 és 9999 közé kell, hogy essen.

Példák:

13 00100 9999

2.7. Direktívák

A direktívák eljárás- és függvényblokkokat helyettesítő nevek. Szintaxisuk megegyezik az azonosítókéval (l. a 12. fejezetet).



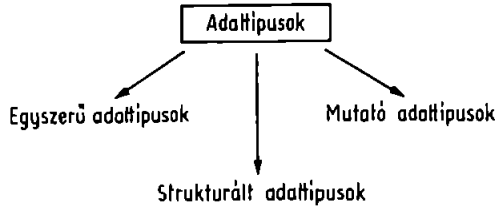
2.7. ábra. *Direktíva* szintaxisdiagramja

3. Az adat fogalma: egyszerű adattípusok

Az *adat* szó gyűjtőfogalom. Adatnak tekintünk minden objektumot, amivel (amin) a számítógép műveleteket végez. A hardver és a gépi kód szintjén minden adatot valamilyen bináris számsor (bitsorozat) ábrázol. A magas szintű programozási nyelvek lehetővé teszik, hogy elvonatkoztassunk a konkrét ábrázolásmód részleteitől. Mindebben igen jelentős szerepet játszik az *adattípus* fogalma.

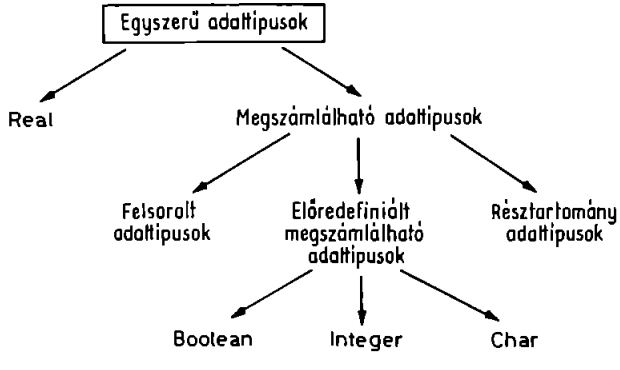
Az adattípus határozza meg, hogy egy változó milyen értékeket vehet fel és milyen műveletek végezhetők rajta. A program minden változójának valamilyen (de csakis egyetlen) típushoz kell tartoznia. Bár a Pascalban meglehetősen összetett adattípusok is előfordulhatnak, minden összetett (strukturált) adattípus végső soron egyszerű, strukturálatlan adattípusokból épül fel.

A Pascal több, egymástól független, de az alkalmazás szempontjából célszerűen egyszerre megadható adattípus megadására is lehetőséget ad. Ennek eszközei a strukturált és a mutató típusok, amelyekkel a 7–11. fejezetben foglalkozunk.

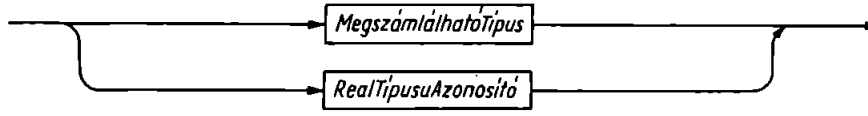


3.1. ábra. Az adattípusok osztályozása

A Pascal egyszerű típusait tovább bonthatjuk a megszámlálható (ordinal) típusok csoportjára és a valós (real) típusra. A megszámlálható típust vagy a programozó definiálja – ekkor felsorolt (enumerated), ill. résztartomány (subrange) típusról van szó –, vagy előredefiniált, standard típusazonosító jelöli: ez a Boolean (logikai), az Integer (egész) vagy a Char (karakter) azonosítók valamelyike lehet. A valós típust a Real standard típusazonosítóval jelöljük.



3.2. ábra. Az egyszerű adattípusok osztályozása

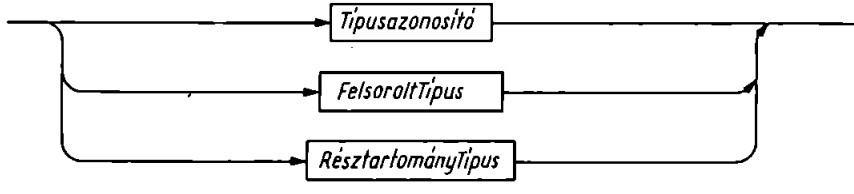


3.3. ábra. *EgyszerűTípus* szintaxisdiagramja

A felsorolt típusokat a típushoz tartozó (különböző) értékek halmaza és az azon értelmezett lineáris rendezés jellemzi. A típus definíciójában az egyes értékeket azonosítókkal jelöljük. A résztartomány típusok olyan megszámlálható típusok, amelyek egy korábban deklarált megszámlálható típus értéktartományának korlátozásával: egy legkisebb és egy legnagyobb érték megadásával jönnek létre. A felsorolt és a résztartomány típusokkal a 6. fejezetben foglalkozunk.

3.1. Megszámlálható típusok

A megszámlálható adattípusok valamilyen véges és rendezett értékhalmazt írnak le. (Matematikai értelemben a megszámlálhatóság fogalmának gyakorlatilag csak végtelen halmazok esetében van jelentősége, hiszen a véges halmazok triviálisan megszámlálhatók. Könyvünkben ugyan kizárólag véges halmazokról van szó, ennek ellenére a definíciók hasonlósága miatt a most bevezetett típuscsoportot megszámlálhatónak nevezzük.) A *megszámlálható* típushoz tartozó értékeket egy-egyértelműen megfeleltetjük a 0, 1, 2, ... természetes számoknak, tehát minden értéknek sorszáma van. (Ez a leképezés az egész típus esetében triviális.) Minden megszámlálható típusnak van egy legkisebb és egy legnagyobb értéke. A legkisebb érték kivételével mindegyik értékhez tartozik egy őt *megelőző* érték. A legnagyobb érték kivételével mindegyik értékhez tartozik egy őt *követő* érték.



3.4. ábra. *MegszámlálhatóTípus* szintaxisdiagramja

Tetszőleges megszámlálható típusú argumentumra alkalmazhatjuk a következő standard függvényeket:

succ (X) eredménye az X-et követő érték,
pred (X) eredménye az X-et megelőző érték,
ord (X) eredménye az X érték sorszáma.

Az =, <>, <, <=, >= és > relációs operátorok minden megszámlálható típusra alkalmazhatók, feltéve, hogy mindkét operandusuk azonos típusú. A rendezést az értékek sorszáma határozza meg.

3.2. A logikai (boolean) típus

Boolean érték csak a két logikai igazságérték, a hamis, ill. az igaz lehet. Ezeket a false *(hamis)* ill. a true *(igaz)* azonosítóval jelöljük.

Logikai operandusokra alkalmazva logikai értéket szolgáltatnak a következő logikai operátorok (a logikai operátorok felsorolását l. a B. függelékben):

and	logikai konjunkció (ÉS művelet),
or	logikai diszjunkció (VAGY művelet),
not	logikai negálás.

Az összes relációs operátor (=, <>, <=, <, >, >=, in) logikai értéket szolgáltat. A "<>" szimbólum jelentése: „nem egyenlő". Lényeges, hogy a logikai típus definíciója értelmében false < true, s így a fenti logikai és relációs operátorok segítségével mind a 16 Boole-algebrai művelet kijelölhető. Ha pl. P és Q logikai értékek,

az implikáció:	P <= Q
az ekvivalencia:	P = Q
a kizáró VAGY:	P <> Q

alakban fejezhető ki.

Standard (beépített vagy előredeklarált) logikai függvények – azaz logikai értéket szolgáltató standard függvények – pl. a következők (a standard függvények összefoglalását l. az A. függelékben):

odd(I)	értéke igaz, ha az I egész szám páratlan, egyébként hamis,
eoln(F)	sor vége (end of line), részletesebben l. a 10. fejezetet,
eof(F)	állomány vége (end of file), részletesebben l. a 10. fejezetet.

3.3. Az egész (integer) típus

Az egész típusú értékek az egész számoknak – az adott Pascal-megvalósításban definiált – részhalmazához tartoznak.

Az alábbi aritmetikai operátorok egész operandusokra alkalmazva egész értéket szolgáltatnak:

*	szorzás,		
div	osztás a törtrész elhagyásával (Az eredmény tehát nem kerekített!),		
mod	maradékképzés:		
	legyen	Maradék = A–(A div B)*B;	
	ekkor	ha Maradék < 0, akkor	A mod B = Maradék + B
		egyébként	A mod B = Maradék
+	összeadás,		
–	kivonás.		

Minden megvalósításban megvan a MaxInt standard konstansazonosító, amely azt a legnagyobb egész értéket jelöli, amellyel az adott rendszerben még valamennyi egészekre értelmezett művelet végrehajtható. Ha A és B két egész típusú kifejezés, az

A op B

művelet helyes végrehajtása csak akkor biztosított, ha

abs(A op B) <= MaxInt
abs(A) <= MaxInt és
abs(B) <= MaxInt

Négy fontos standard függvény szolgáltat egész eredményt:

abs(I) az eredmény: I abszolút értéke,
sqr(I) az eredmény: I négyzet, feltéve, hogy I <= MaxInt div I.
trunc(R) R valós érték, az eredmény: R egészrésze. (A törtrészt a művelet levágja. Így pl. trunc(3.7)=3 és trunc(−3.7) = −3.)
round(R) R valós érték, az eredmény az R-nek megfelelő kerekített egész.
Ha R >= 0, akkor round(R) = trunc(R+0,5),
ha R < 0, akkor round(R) = trunc(R−0,5).

Ha I egész típusú változó, akkor

succ(I) a „következő" egészt, (I+1)-et,
pred(I) a megelőző egészt, (I−1)-et szolgáltatja.

3.4. A karakter (char) típus

A karakter típusú értékek valamilyen véges és rendezett karakterhalmaz (jelkészlet) elemei. Ilyen halmazt kommunikációs célokból minden számítógépes rendszerben definiálnak. Ezek a karakterek állnak rendelkezésünkre a be-, ill. kiviteli berendezéseken. Sajnos szabványos karakterkészlet, amit egységesen alkalmaznának, nem létezik, ezért az elemek definíciója és rendezése szigorúan rendszerfüggő (l. a G. függeléket).

Az aposztrófok (egyszeres idézőjelek) közé zárt jel karakter típusú értéket jelöl.

Példák:

'*' 'G' '3' '''' 'X'

(Az aposztróf karaktert két, aposztrófok közé zárt aposztrófffal jelöljük.)

A karakter típus definiálásakor azonban a rendszertől függetlenül feltételezzük, hogy:

(1) a '0', ..., '9' decimális számjegyek a megszokott módon rendezett és összefüggő,
(2) az 'A', ..., 'Z' nagybetűk (ha szerepelnek a karakterkészletben) alfabetikusan rendezett, de nem feltétlenül összefüggő,
(3) az 'a', ..., 'z' kisbetűk (ha szerepelnek a karakterkészletben) alfabetikusan rendezett, de nem feltétlenül összefüggő sorozatot alkotnak.

Példák:

succ('5') = '6' 'A' < 'B' 'a' < 'b'

Az adott jelkészlet és a természetes számok egy részhalmaza (a karakterek sorszámai) között két standard függvény, az ord és a chr segítségével kölcsönös és egyértelmű leképzés létesíthető.

ord(C) a C karakter sorszáma az adott rendezett karakterkészletben;
chr(I) az I sorszámú karakter.

Azonnal látszik, hogy ord és chr egymás inverz függvényei, tehát:

chr (ord (C)) = C és ord (chr (I)) = I

Az előbbi függvényekkel egy adott karakterkészlet rendezését a következőképpen értelmezhetjük:

C1 < C2 akkor és csak akkor, ha ord(C1) < ord(C2).

Ezt a definíciót az összes relációs operátorra (=, <>, <, <=, >=, >) általánosíthatjuk. Ha R egy ilyen operátor, akkor

C1 R C2 akkor és csak akkor, ha ord(C1) R ord(C2).

Ha a pred, ill. a succ standard függvény argumentuma karakter típusú, a függvényeket a következőképpen értelmezzük:

pred(C) = chr(ord(C)−1)
succ(C) = chr(ord(C)+1)

Megjegyzés: A karaktert megelőző, ill. követő karakter az adott jelkészlettől függ. A két értéknek csak akkor van értelme, ha a készletben létezik az adottat megelőző, ill. követő karakter.

3.5. A valós (real) típus

A valós típusú értékek a valós számok – adott Pascal-megvalósításban definiált – részhalmazának az elemei.

A valós típusú operandusokkal végzett műveletek mindig közelítő eredményt szolgáltatnak: az eredmény pontossága attól függ, milyen fordítóprogrammal (géppel) dolgozunk. A Real az egyetlen nemmegszámlálható egyszerű típus. A valós értékekhez nem rendelhetünk sorszámot, sem megelőző vagy követő értéket.

Abban az esetben, ha az alábbi operátoroknak legalább az egyik operandusa valós (a másik egész típusú is lehet), a következő műveletek valós értéket szolgáltatnak:

* szorzás,
/ osztás (mindkét operandus lehet egész, de az eredmény mindig valós),
+ összeadás,
− kivonás.

Valós argumentumokra alkalmazva valós eredményt ad az

abs(R) és az sqr(R)

standard függvény. Az előbbi eredménye R abszolút értéke, az utóbbié R négyzete, feltéve, hogy az nem esik már kívül a valós számoknak az adott megvalósításban megengedett tartományán.

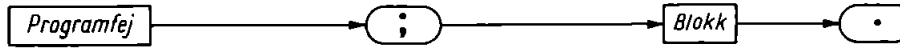
Az alábbi standard függvények valós és egész argumentumokon egyaránt valós eredményt szolgáltatnak:

sin(X)	az X radiánban megadott érték szinusza,
cos(X)	az X radiánban megadott érték koszinusza,
arctan(X)	X arkusz tangense radiánban,
ln(X)	X természetes (*e* alapú) logaritmusa, X > 0,
exp(X)	exponenciális függvény (*e* az X-ediken),
sqrt(X)	X négyzetgyöke, X >= 0

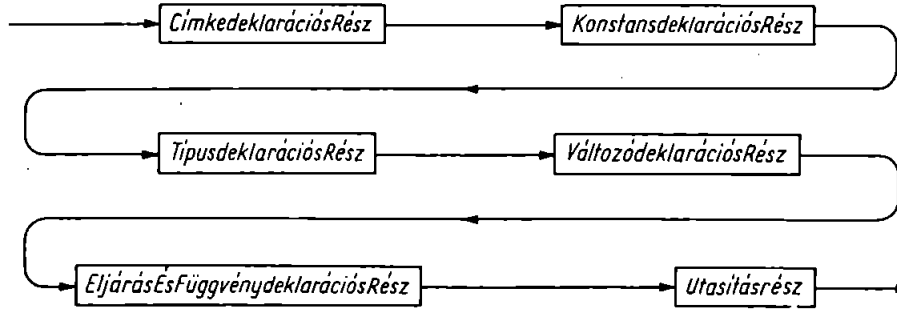
Figyelmeztetés! A valós típus ugyan az egyszerű típusok között van, de nem mindig használható ugyanúgy, mint a többi egyszerű (megszámlálható) típus. Külön ki kell emelnünk, hogy a pred és a succ függvény valós argumentumra nem alkalmazható, továbbá tilos valós értéket tömb és case utasítás indexelésére, for utasítás vezérlésére, halmaz alaptípusának definiálására, ill. résztartomány típusban használni!

4. A programfej és a deklarációs rész

Mint már említettük, minden program fejből és blokkból (törzsből) áll. A blokk két részre tagolódik: a deklarációs részre, amely az összes, a programra nézve lokális objektum definícióját tartalmazza, és az utasításrészre, amely az ezen objektumokon elvégzendő műveleteket, tevékenységeket adja meg.



4.1. ábra. *Program* szintaxisdiagramja

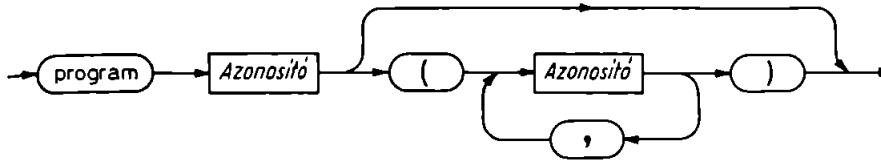


4.2. ábra. *Blokk* szintaxisdiagramja

4.3. ábra. *Utasításrész* szintaxisdiagramja

4.1. A programfej

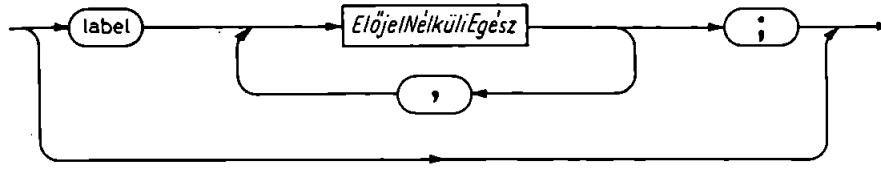
A fej nevet ad a programnak (a név egyébként magában a programban nem lényeges), és felsorolja a program paramétereit, amelyek a programon kívüli objektumokat jelölnek, és amelyeken keresztül a program a környezettel kapcsolatot tart. Ezeket az objektumokat (többnyire állományokról van szó, l. a 10. fejezetet) *külső* (external) objektumoknak nevezzük. A paramétereket ugyanúgy kell deklarálni, mint a közönséges lokális változókat, de deklarációjuknak a programot alkotó blokkban kell állnia (l. a 4.5. szakaszt).



4.4. ábra. *Programfej* szintaxisdiagramja

4.2. A címkedeklarációs rész

A címkék segítségével a program tetszőleges utasítását jelzéssel láthatjuk el. A címkét az utasítás elé kell írni, a címke után kettőspont áll. (Címkével jelölhetjük ki pl. egy goto utasítás célpontját.) Felhasználása előtt azonban minden címkét deklarálnunk kell a *címkedeklarációs részben*. A deklarációt a programban a label *(címke) alapszó* vezeti be. A címkedeklarációs rész általános alakja:



4.5. ábra. *CímkedeklarációsRész* szintaxisdiagramja

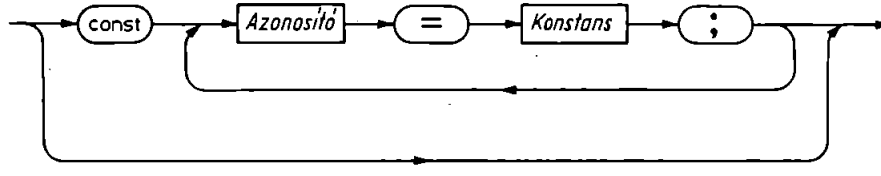
A címke előjel nélküli, 0 és 9999 közé eső egész szám.

Példa:

```
label 13,   00100,   99;
```

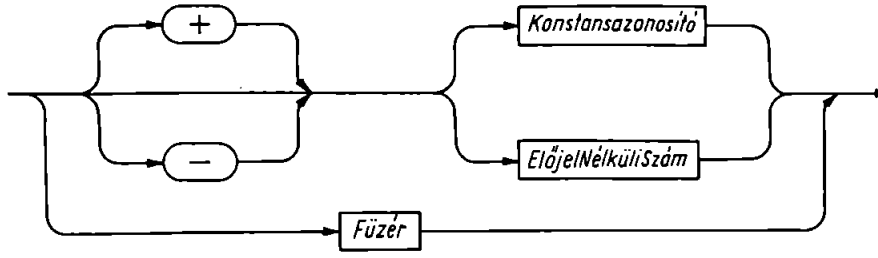
4.3. A konstansdefiníciós rész

A *konstansdefinícióban* egy konstans helyett, annak szinonimájaként, egy azonosítót vezetünk be. A konstansdefiníciós rész a const szimbólummal kezdődik, s általános alakja:



4.6. ábra. *KonstansdefiníciósRész* szintaxisdiagramja

ahol a konstans szám (esetleg előjeles) konstansazonosító, karakter vagy füzér lehet.



4.7. ábra. *Konstans* szintaxisdiagramja

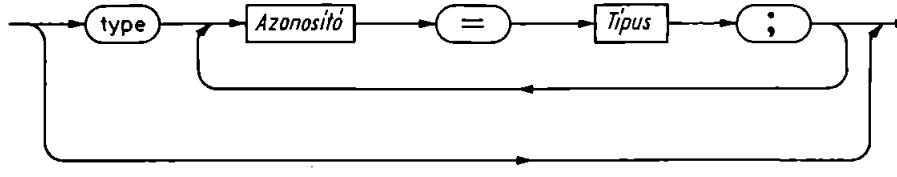
A konstansazonosítók használata általában olvashatóbbá teszi a programot, és megkönnyíti a dokumentációt. Lehetővé teszi, hogy a programozó kigyűjtse a gép- vagy feladatfüggő mennyiségeket a program elejére, ahol azok könnyen figyelemmel kísérhetők, módosíthatók. (Ezáltal a konstansazonosítók alkalmazása javítja a program portábilitását és modularitását.)

Példa:

```
const
    Avogadro = 6.023E23;
    SorokSzáma = 60;
    Határ = '# *';
    ÉnLépek = True;
```

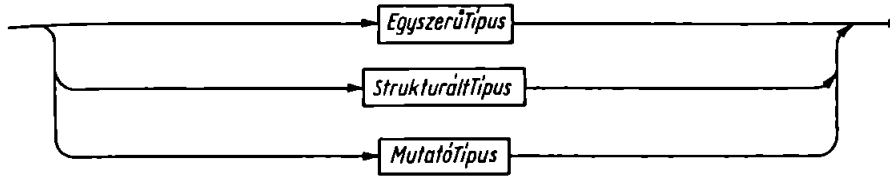
4.4. A típusdefiníciós rész

A Pascalban az adattípust a változódeklaráció vagy *közvetlen leírással* (l. a 4.5. szakaszt), vagy *típusazonosítóval* adhatja meg. A programban azonban vannak helyek, ahol a típust csak típusazonosító jelezheti. Több standard típusazonosító alkalmazható, de a nyelv lehetőséget ad új típusazonosítók bevezetésére (új típusok létrehozására) is. Az erre szolgáló mechanizmust *típusdefiníciónak* nevezzük. A típusdefiníciókat tartalmazó programrészt a type szimbólum vezeti be. Általános alakja a 4.8. ábrán látható.



4.8. ábra. *TípusdefiníciósRész* szintaxisdiagramja

A *típus* lehet egyszerű, strukturált vagy mutató típus, és egy már létező típus azonosítójából vagy egy új típus leírásából áll (4.9. ábra).



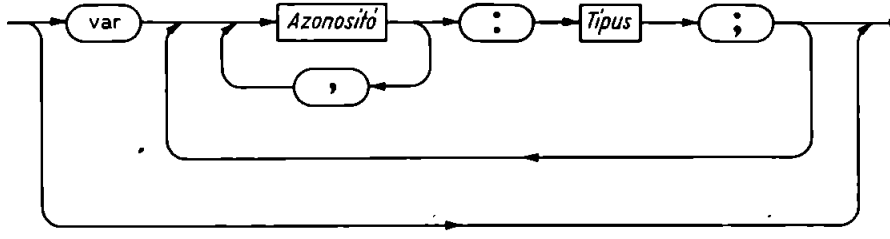
4.9. ábra. *Típus* szintaxisdiagramja

Típusazonosítókra több példát is talál az Olvasó a következő fejezetekben.

4.5. A változódeklarációs rész

Az utasításokban szereplő összes változót ***változódeklarációban*** kell megadni. A deklaráció, hacsak nem programparaméterről van szó, meg kell, hogy előzze a változó felhasználását.

A változódeklarációban az új változóhoz azonosítót és adattípust rendelünk, mégpedig oly módon, hogy egyszerűen leírjuk az azonosítót, majd a változó típusát. A változódeklarációs részt a var szimbólum vezeti be. Általános alakja a 4.10. ábrán látható.



4.10. ábra. *VáltozódeklarációsRész* szintaxisdiagramja

Példa:

```
var
      Gyök1, Gyök2, Gyök3: Real;
      Darab, I: Integer;
      Megvan: Boolean;
      Betű: Char;
```

A programfej paraméterlistájában felsorolt (külső objektumokat, rendszerint állományokat jelölő) azonosítókat – az Input és az Output kivételével – a program változódeklarációs részében deklarálni kell. Az Input és az Output azonosítókat, ha szerepelnek, a fordítóprogram automatikusan szövegállományként deklaráltnak tekinti (l. a 10. fejezetet).

```
program AtszamitasCelsiusbolFahrenheitbe (Output);
{Turbo Pascal}
{4.1. program - A konstans- es tipusdefinicios,
III. a valtozodeklaracios reszt szemlelteto program}
```

```
const Eltolas = 32;
      Szorzo = 1.8;
      AlsoHatar = -20;
      FelsoHatar = 39;
      Elvalaszto = '   ___'; Terkoz ='          ';
type  Celsiustartomany = AlsoHatar .. FelsoHatar;
      {resztartomany tipus, l. 6. fejezet}
var Fok : Celsiustartomany;
begin for Fok := AlsoHatar to FelsoHatar do
          begin Write(Output, Fok:5, 'C');
                Write(Output, Elvalaszto,
                      Round(Fok*Szorzo+Eltolas):5,'F');
                if Odd(Fok)
                then Writeln (Output)
                else Write (Output, Terkoz)
          end;
      Writeln (Output)
end.
```

A program eredménye:

```
-20C   ___   -4F          -19C   ___   -2F
-18C   ___    0F          -17C   ___    1F
-16C   ___    3F          -15C   ___    5F
-14C   ___    7F          -13C   ___    9F
-12C   ___   10F          -11C   ___   12F
-10C   ___   14F           -9C   ___   16F
 -8C   ___   18F           -7C   ___   19F
 -6C   ___   21F           -5C   ___   23F
 -4C   ___   25F           -3C   ___   27F
 -2C   ___   28F           -1C   ___   30F
  0C   ___   32F            1C   ___   34F
  2C   ___   36F            3C   ___   37F
  4C   ___   39F            5C   ___   41F
  6C   ___   43F            7C   ___   45F
  8C   ___   46F            9C   ___   48F
 10C   ___   50F           11C   ___   52F
 12C   ___   54F           13C   ___   55F
 14C   ___   57F           15C   ___   59F
 16C   ___   61F           17C   ___   63F
 18C   ___   64F           19C   ___   66F
 20C   ___   68F           21C   ___   70F
 22C   ___   72F           23C   ___   73F
 24C   ___   75F           25C   ___   77F
 26C   ___   79F           27C   ___   81F
 28C   ___   82F           29C   ___   84F
 30C   ___   86F           31C   ___   88F
 32C   ___   90F           33C   ___   91F
 34C   ___   93F           35C   ___   95F
 36C   ___   97F           37C   ___   99F
 38C   ___  100F           39C   ___  102F
```

4.6. Az eljárás- és függvénydeklarációs rész

Felhasználása előtt minden eljárás- vagy függvényazonosítót deklarálni kell. Az eljárás- és függvénydeklaráció ugyanolyan alakú, mint a program – fejből és blokkból áll. A témával részletesen (példákkal) a 12. fejezetben foglalkozunk. Az eljárás olyan szubrutin, amelyet eljárásutasítással hívhatunk be. A függvény olyan szubrutin, amely eredmény értéket szolgáltat, s így kifejezésekben is előfordulhat.

4.7. Az azonosítók és a címkék hatásköre

Egy (konstans-, típus-, változó-, eljárás- vagy függvény-) azonosító vagy címke deklarációja, ill. definíciója mindaddig érvényes, míg a deklarációt, ill. definíciót tartalmazó blokkban vagyunk, hacsak az azonosítót vagy címkét egy alárendelt blokkban újra nem deklaráljuk, ill. definiáljuk. Ekkor a belső blokkban az új deklaráció (definíció) lesz érvényes. Azt a programterületet, amelyen egy változó vagy címke deklarációja, ill. definíciója hatályos, a változó vagy címke *hatáskörének* nevezzük.

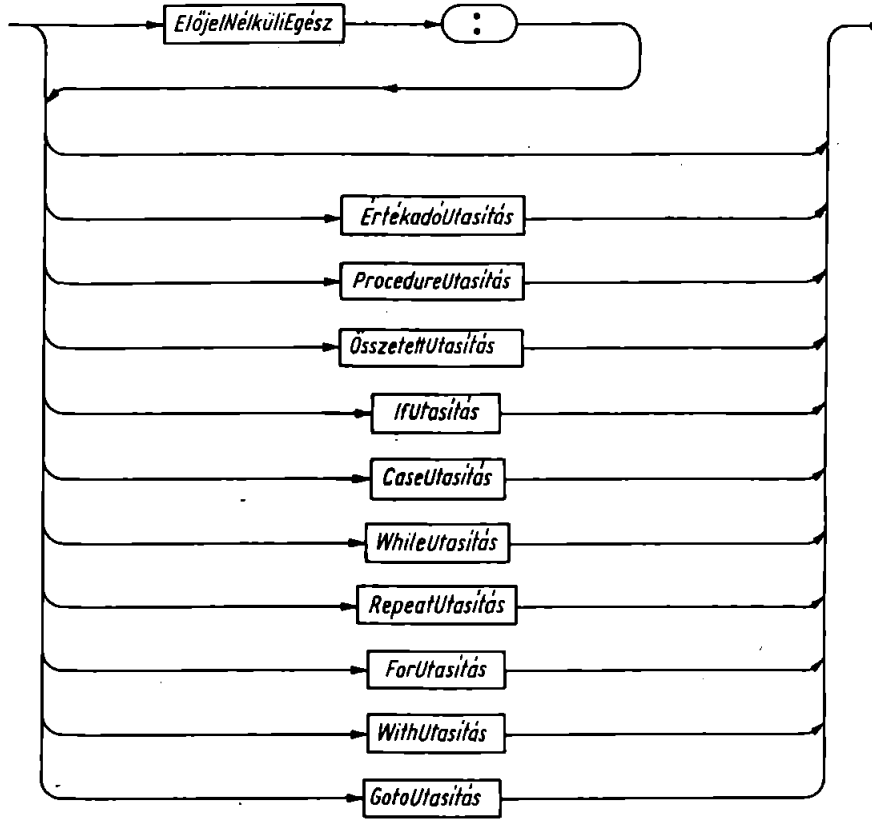
A program blokkban deklarált, ill. definiált azonosítókat *globálisnak* mondjuk. Az azonosítók (címkék) *lokálisak* arra a blokkra nézve, amelyben deklaráltuk (definiáltuk) őket, és *globálisak* azokra a blokkokra nézve, amelyek a deklarációjukat tartalmazó blokkba ágyazottak. (A példákat l. az 1.4. szakaszban.)

Tilos egy azonosítót egyazon szinten és hatáskörön belül egynél többször deklarálni! Hibás tehát a következő deklaráció:

```
var X: Integer;
    X: Real;
```

5. A tevékenység fogalma

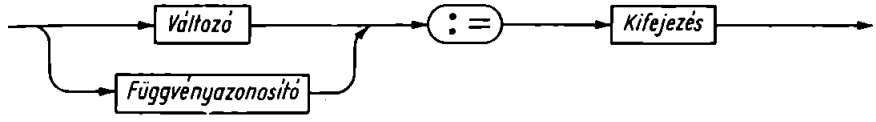
A számítógépes program alapvető tulajdonsága a tevékenység. Más szóval, a programnak mindenképpen csinálnia kell valamit az adataival, még akkor is, ha ez a tevékenysége abból áll, hogy nem csinál semmit! A tevékenységeket *utasítások* írják le. Az utasítások lehetnek *egyszerűek* (mint pl. az értékadó utasítás) vagy *strukturáltak*. Az utasítástípusokat az 5.1. szintaxisdiagramon tekinthetjük át.



5.1. ábra. *Utasítás* szintaxisdiagramja

5.1. Az értékadó utasítás és a kifejezések

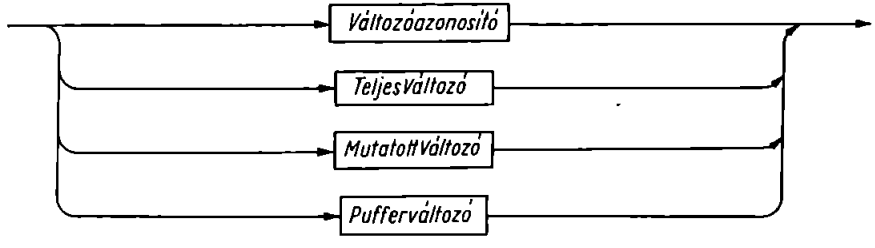
A legalapvetőbb utasítás az *értékadó utasítás*. Hatására egy újonnan kiszámított – egy *kifejezés* által meghatározott – érték rendelődik valamilyen változóhoz. Az értékadó utasítás alakját az 5.2. ábra szintaxisdiagramja mutatja.



5.2. ábra. *ÉrtékadóUtasítás* szintaxisdiagramja

A := szimbólum az értékadás jele, és nem szabad összetévesztenünk az = relációs operátorral. Az "A:=5" utasítás jelentése: "A aktuális értékének helyére az 5 érték kerül", vagy egyszerűen "A legyen egyenlő 5-tel!

A *változó* (5.3. ábra) lehet *teljes változó*, amely egy egyszerű, strukturált vagy mutató típus adatainak tárolásához szükséges teljes tárterületet lefoglalja. Strukturált típusok esetében (l. a 7...10. fejezetet) azonban lehet *elemi* vagy *pufferváltozó* is, amely a teljes tárterület egy elemét jelöli ki. Végül mutató típusok esetében megkülönböztetünk *mutatott változókat*, amelyek a mutatók által indirekt módon megadott tárrekeszek leírására szolgálnak.

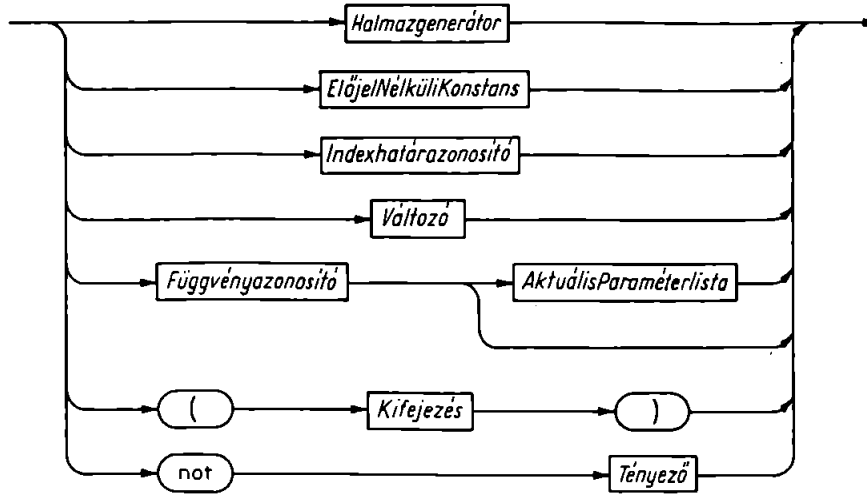


5.3. ábra. *Változó* szintaxisdiagramja

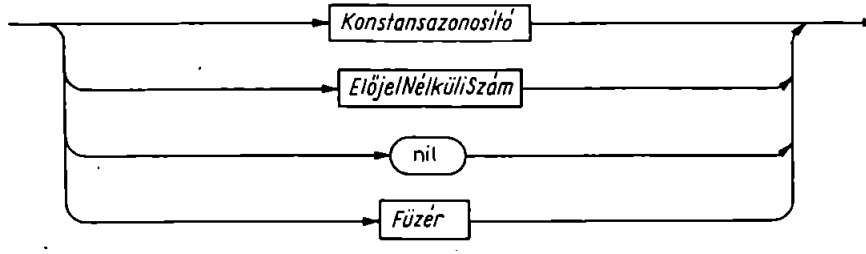
A *kifejezés* operátorokból és operandusokból áll. Az operandusok lehetnek konstansok, változók, tömbparaméter-indexhatárok (l. a 12. fejezetet) vagy függvénykifejezések. (A függvénykifejezés egy függvény kiértékeléséről gondoskodik. A standard függvényeket l. az A. függelékben; a felhasználó által definiált függvényekkel a 12. fejezetben foglalkozunk.)

A kifejezés egy számítás kijelölése: előírja valamilyen érték kiszámításának a módját. A kifejezés kiértékelése a megszokott algebrai szabályoknak megfelelően, balról jobbra haladva, az *operátorprecedencia* figyelembevételével történik. A kifejezések tényezőkből, tagokból és egyszerű kifejezésekből épülnek fel.

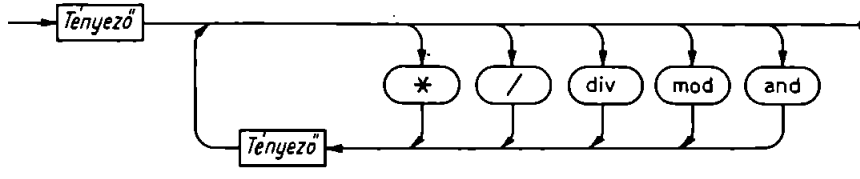
Elsőként a *tényezők* értékelődnek ki. A tényezők lehetnek konstansok, változók, függvénykifejezések, tömbparaméter-indexhatárok vagy halmazgenerátorok (l. a 9. fejezetet). Ha egy logikai értéket leíró tényezőre a not operátort alkalmazzuk, ismét tényezőt kapunk. Tényezőt alkot a zárójelek közé zárt kifejezés is, amely az előtte és mögötte álló operátoroktól függetlenül értékelődik ki.



5.4. ábra. *Tényező* szintaxisdiagramja



5.5. ábra. *ElőjelNélküliKonstans* szintaxisdiagramja

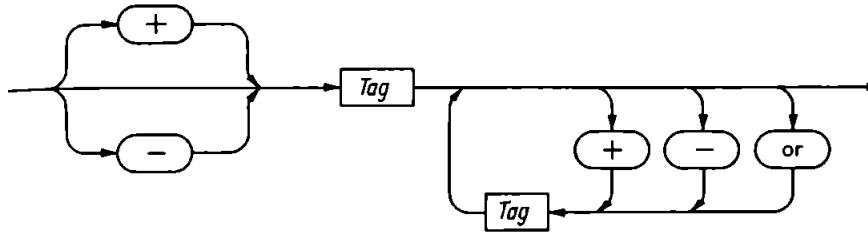


5.6. ábra. *Tag* szintaxisdiagramja

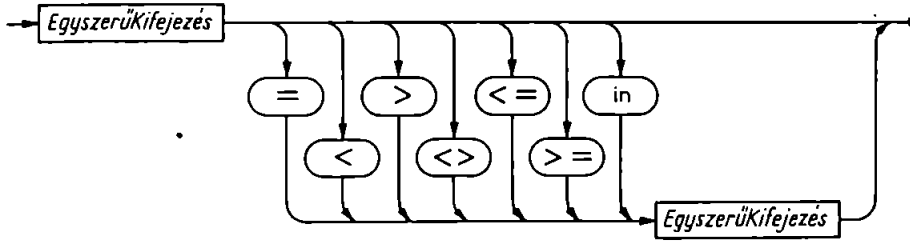
A tényezők után a *tagok* kiértékelése következik. A tag legegyszerűbb esetben egyetlen tényezőből áll, de általában multiplikatív operátorokkal (*, /, div, mod, and) elválasztott tényezők sorozata.

Harmadikként az *egyszerű kifejezések* értékelődnek ki. Ezek legegyszerűbb esetben egyetlen tagból, általában pedig additív operátorokkal (+, –, or) elválasztott tagok sorozatából állnak. Első tagjukat előjel operátor (+, –) előzheti meg, amely azonban nem kötelező.

Mindezek után kerülhet csak sor a kifejezések kiértékelésére. A kifejezések valamilyen relációs operátorral (=, <>, <, <=, >=, >, in) összekapcsolt két egyszerű kifejezésből vagy egyetlen egyszerű kifejezésből állnak (5.7. és 5.8. ábra).



5.7. ábra. *EgyszerűKifejezés* szintaxisdiagramja



5.8. ábra. *Kifejezés* szintaxisdiagramja

Példák:

```
2*3-4*5          = (2*3)-(4*5)     = -14
15 div 4*4       = (15 div 4)*4    =  12
80/5/3           = (80/5)/3        =   5.333
4/2*3            = (4/2)*3         =   6.000
Sqr (Sqr(3) + 11*5)                =   8.000
```

Ha az Olvasó bizonytalannak érzi magát, nézze át a következő táblázatot, amely pontosan megadja az operátorok precedencia-sorrendjét!

Operátor	*Besorolás*
not	Logikai negáció (a legnagyobb precedenciával).
*, /, div, mod, and	Multiplikatív operátorok (a precedenciájuk a második legnagyobb).
+, -, or	Additív operátorok (precedenciájuk sorrendben a harmadik).
=, <>, <, <=, >=, >, in	Relációs operátorok (a legkisebb precedenciával).

(Az operátorok részletes leírása a B. függelékben található.)

A logikai kifejezések érdekes tulajdonsága, hogy értéküket már akkor ismerhetjük, mielőtt az egész kifejezést kiértékeltük volna. Tegyük fel pl., hogy X = 0. Ekkor az

```
(X > 0)   and   (X < 10)
```

kifejezésről már az első tényező kiértékelése után tudjuk, hogy értéke hamis, s a második tényezővel már nem is kell foglalkoznunk. Ez azt jelenti, hogy az első tényező értékétől függetlenül

a programozónak mindig gondoskodnia kell a második tényező helyes felírásáról is. Ha tehát feltételezzük, hogy az A tömb indexhatárai 1 és 10, a következő programpélda hibás!

```
X:=0;
repeat X := X + 1 until (X > 10) or (A [X] = 0)
```

(Vegyük észre, hogy ha egyik A[I] sem zérus, a program az A[11] elemre mutat!)

Az értékadás – az állományokat kivéve (l. a 10. fejezetet) – tetszőleges típusú változóra vonatkozhat. A változó (vagy függvény) és a kifejezés azonban az értékadás szempontjából kompatibilis kell, hogy legyen.

A következőkben az *értékadás-kompatibilitás* valamennyi lehetséges esetét felsoroljuk.

(1) A változó és a kifejezés azonos típusú (kivéve, ha állomány típusúak – l. 10. fejezet –, vagy olyan strukturált típushoz tartoznak, amelynek állomány típusú eleme is van).
(2) A változó valós, a kifejezés pedig egész típusú.
(3) A változó és a kifejezés ugyanahhoz a megszámlálható típushoz, ill. annak egy-egy résztartományához (l. 6. fejezet) tartozik, és a kifejezés értéke a változó típusa által meghatározott zárt intervallumba esik.
(4) A változó és a kifejezés ugyanahhoz a halmaztípushoz (l. a 9. fejezet), vagy két olyan halmaztípushoz tartozik, amelynek alaptípusa vagy ugyanaz a megszámlálható típus, vagy annak egy-egy résztartománya. Vagy mindkét típus tömörített, vagy egyik sem. A kifejezés értéke eleme kell, hogy legyen a változó típusa értékkészletének.
(5) A változó és a kifejezés azonos elemszámú füzér típushoz (l. a 7.2. szakaszt) tartozik.

Példák:

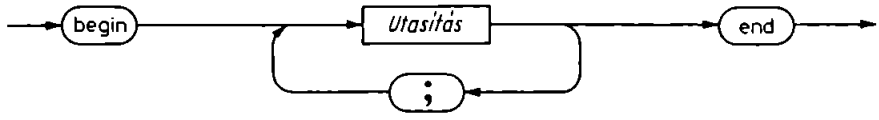
```
Gyök1   :=  Pi * X/Y
Gyök2   :=  -Gyök1
Gyök3   :=  (Gyök1 + Gyök2) * (1.0 + Y)
BajVan  :=  Hőmérséklet > Forrpont
Darab   :=  Darab + 1
Fok     :=  Fok + 10
SqrPr   :=  Sqr(Pr)
Y   :=  Sin(X) + Cos(Y)
```

5.2. Az eljárásutasítás

Az egyszerű utasítások másik fontos fajtája az eljárásutasítás, amely az utasításban megnevezett eljárás végrehajtását idézi elő. Az eljárás olyan alprogram, amelybe valamely, az adatokon végzendő tevékenységcsoportot fogunk össze. Könyvünkben eddig a Read, a Readln, a Write és a Writeln eljárást használtuk: segítségükkel olvastuk be, ill. írtuk ki az adatokat, ill. eredményeket. Az eljárásutasítással részletesen a 12. fejezetben foglalkozunk.

5.3. Az összetett és az üres utasítás

Az *összetett utasítás* feladata annak biztosítása, hogy összetevői adott sorrendben hajtódjanak végre. Az utasítászárójel szerepét a begin és az end alapszó tölti be. Itt jegyezzük meg, hogy az utasításrész, azaz a program törzse mindig összetett utasítás formáját ölti (5.9. ábra).



5.9. ábra. *ÖsszetettUtasítás* szintaxisdiagramja

```
program BeginEndPelda(Output);
{Turbo Pascal}
{5.1. program - Az osszetett utasitas szemleltetese.}

var Osszeg: Integer;

begin Osszeg := 3 + 5;
      Writeln (Output, Osszeg, -Osszeg)
end.
```

A program eredménye:

```
8           -8
```

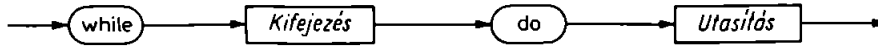
A Pascal a pontosvesszőt nem az utasítás lezárására, hanem az utasítások elválasztására használja; a pontosvessző tehát *nem* tartozik az utasításhoz. A pontosvesszővel kapcsolatos pontos szabályokat a D. függelék szintaxisleírása tartalmazza. Ha előbbi példánkban a második utasítás után pontosvesszőt tettünk volna, a fordítóprogram ezt a pontosvessző és az end szimbólum közötti *üres utasításként* értelmezte volna, melynek hatására semmilyen tevékenység nem történik. Ez nem lett volna baj, mert ezen a ponton megengedhető az üres utasítás. A rossz helyre tett pontosvesszők azonban problémákat okozhatnak (l. az 5.5. szakasz if utasításokkal kapcsolatos példáját).

5.4. Ciklusutasítások

A *ciklusutasítások* bizonyos utasítások ismételt végrehajtását biztosítják. Ha előre (az ismétlések megkezdése előtt) ismerjük a ciklusok számát, általában a for utasítás fejezi ki legalkalmasabban a feladatot, ellenkező esetben a repeat vagy a while utasítást célszerű használni.

5.4.1. A while utasítás

A while utasítás alakja az 5.10. ábrán látható. A do alapszó utáni utasítás lehet, hogy egyszer sem hajtódik végre.



5.10. ábra. *WhileUtasítás* szintaxisdiagramja

A ciklust vezérlő kifejezésnek logikai típusúnak kell lennie. Az utasítás végrehajtása előtt a gép kiértékeli a kifejezést; ha az igaz, végrehajtja az utasítást, egyébként kilép a while utasításból. Mivel a kifejezést a gép minden ciklus előtt kiértékeli, ügyelnünk kell arra, hogy a lehető legegyszerűbb alakban írjuk fel.

```
program WhilePelda(Output);
{Turbo Pascal}
{5.2. program - A H(N) =1 +1/2 +1/3 + ... +1/N
harmonikus sor N-edik reszosszegenek kiszamitasa
while ciklussal.}

var N: Integer;
       H: real;

begin Read(Input,N); Write(Output,N);
      H := 0;
      while N> 0 do
            begin H := H+ 1/N;
                  N := N-1
            end;
      Writeln(Output,H)
end.
```

A program eredménye:

```
10  2.92896825400E+00
```

Az 5.3. program az X valós értéket az Y hatványkitevőre emeli, ahol Y nemnegatív egész. Egyszerűbb, láthatóan helyes megoldást kapunk a belső while utasítás elhagyásával – ekkor az Eredmény változó értékét úgy kapjuk, hogy X-et Y-szor önmagával szorozzuk. Figyeljük meg azonban az Eredmény*hatvány(Alap, Kitevő) = hatvány(X, Y) ciklusinvariánst! A belső while utasítás az Eredmény-t és hatvány(Alap,Kitevő)-t változatlanul hagyja, ezáltal nyilvánvalóan javítja az algoritmus hatékonyságát.

```
program Hatvanyozas(Input, Output);
{Turbo Pascal}
{5.3. program - Hatvany (X,Y) kiszamitasa  termeszetes
 kitevore. Hatvany (X,Y) jelentese: "X az Y-adikon".}

var Kitevo, Y:Integer;
    Alap, Eredmeny, X: Real;
```

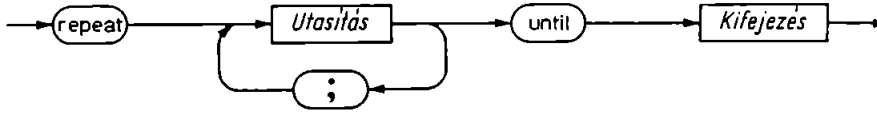
```
begin Read(Input,X,Y);
      Writeln(Output,X,Y:6);
      Eredmeny:=1; Alap:=X; Kitevo:=Y;
      while Kitevo>0 do
            begin while not odd(Kitevo) do
                        begin Kitevo:=Kitevo div 2;
                              Alap:=Sqr(Alap)
                        end;
                  Kitevo:=Kitevo-1;
                  Eredmeny:=Eredmeny*Alap
            end;
      Writeln(Output,Eredmeny)
end.
```

A program eredménye:

```
2.0000000000E+00       7
1.2800000000E+02
```

5.4.2. A repeat utasítás

A repeat utasítás általános alakja az 5.11. ábrán látható.



5.11. ábra. *RepeatUtasítás* szintaxisdiagramja

A repeat és az until szimbólum közötti utasítássorozat legalább egyszer végrehajtódik. Minden ciklus után kiértékelődik a logikai kifejezés. Az ismételt végrehajtások mindaddig folytatódnak, amíg a kifejezés igaz (true) értékűvé nem válik. Minthogy a kifejezés minden ciklusban kiértékelődik, fontos, hogy a lehető legegyszerűbb alakban írjuk fel.

```
program RepeatPelda(Input, Output);
{Turbo Pascal}
{5.4. program - A H(N)=1+1/2+1/3+...+1/N harmonikus sor
 N-edik reszosszegenek kiszamitasa repeat ciklussal.}

var N:Integer;
    H:Real;

begin Read(Input,N); Write(Output,N);
      H:=0;
      repeat H:=H+1/N;
             N:=N-1
      until N=0;
      Writeln(Output,H)
end.
```

A program eredménye:

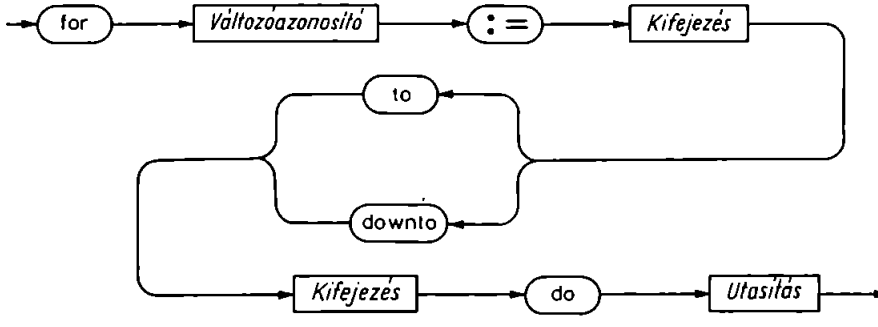
```
10  2.9289682540E+00
```

A fenti program akkor hajtódik helyesen végre, ha N > 0. Az Olvasóra bízzuk annak vizsgálatát, mi történik, ha N < = 0. Ugyanennek a programnak a while utasítással megoldott változata minden N-re (N = 0-ra is) helyes eredményt ad.

Figyeljük meg, hogy bár a repeat utasítás utasítássorozatokat hajt végre, a begin...end zárójelpár alkalmazása felesleges (de nem lenne hiba)!

5.4.3. A for utasítás

A for utasítás egy utasítás ismételt végrehajtását biztosítja, miközben az ún. *ciklusváltozójának* folyamatosan növekvő vagy csökkenő értéket ad. Általános alakja az 5.12. ábrán látható.



5.12. ábra. *ForUtasítás* szintaxisdiagramja

```
program ForPelda(Input,Output);
{Turbo Pascal}
{5.5. program - A H/N=1+1/2+1./3+...+1/N harmonikus sor
 N-edik reszosszegenek kiszamitasa for ciklussal.}

var I,N:Integer;
    H:Real;

begin Read(Input,N); Write(Output,N);
      H:=0;
      for I:=N downto 1 do
          H:=H+1/I;
      Writeln(Output,H)
end.
```

A program eredménye:

```
10  2.9289682540E+00
```

A ciklus változója a for alapszó után áll. Megszámlálható típusúnak kell lennie, és ugyanabban a blokkban kell deklarálni, ahol a for utasítás szerepel. A kezdő- és a végérték a ciklus-

változó típusával kompatibilis megszámlálható típushoz kell, hogy tartozzon. A ciklusváltozót a ciklusmag utasítása nem módosíthatja. Sem magában a for utasításban, sem pedig az azt tartalmazó blokkban deklarált eljárásokban vagy függvényekben nem szerepelhet tehát értékadás bal oldalán, Read vagy Readln eljárásban vagy másik for utasítás ciklusváltozójaként.

A kezdő- és a végértéket a gép csak egyszer határozza meg. A to (downto) használata esetén, ha a kezdőérték a végértéknél nagyobb (kisebb), az utasítás nem hajtódik végre. Ha viszont végrehajtására sor kerül, hiba, ha a ciklusváltozó nem veheti fel a kezdő- vagy végértéket. Normális esetben a for utasításból kilépve a ciklusváltozó értéke határozatlan lesz.

```
program Koszinusz(Input,Output);
{Turbo Pascal}
{5.6. program - A koszinuszfuggveny ertekenek ki-
 szamitasa sorbafejtessel:
 Cos(X)=1-Sqr(X)/(2*1)+Sqr(X)*Sqr(X)/(4*3*2*1)-...}

const Epszilon=1e-7;

var  Szog: Real {Radian};
     Szog2: Real {Szog a negyzeten};
     Sor: Real {A cos sora};
     Tag: Real {A sor kovetkezo tagja};
     I,N: Integer {A kiszamitando cos-ertekek szama};
     Ktv: Integer {A kovetkezo tag kitevoje};

begin Readln(Input,N);
      for i:=1 to N do
          begin Readln(Input,Szog);
                Tag:=1; Ktv:=0; Sor:=1;
                Szog2:=Sqr(Szog);
                while Abs(Tag)>Epszilon*Abs(Sor) do
                    begin Ktv:=Ktv+2;
                          Tag:=-Tag*Szog2/(Ktv*(Ktv-1));
                          Sor:=Sor+Tag
                    end;
                Writeln(Output,Szog,Sor,(Ktv div 2):5
                   {= Konvergenciakriterium altal
                      megkovetelt tagszam})
          end
end.
```

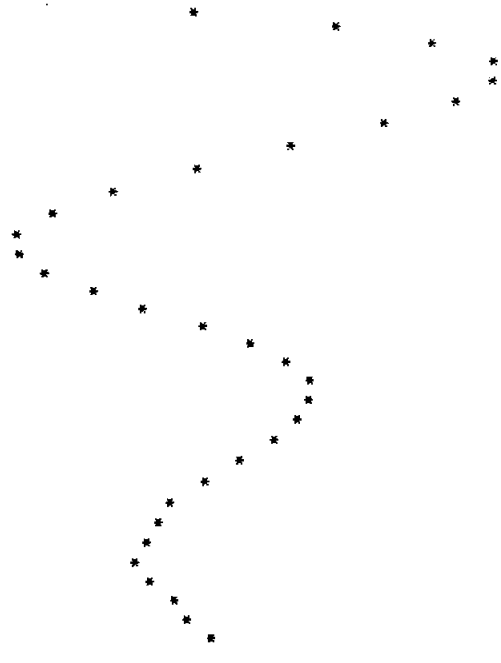
A program eredménye:

```
1.5346220000E-01  9.8824776815E-01     3
3.3333330000E-01  9.4495695723E-01     4
5.0000000000E-01  8.7758256189E-01     5
1.0000000000E+00  5.4030230588E-01     6
3.1415930000E+00 -9.9999999992E-01    10
```

A következő program az f(X) valós értékű függvényt ábrázolja oly módon, hogy a függőlegesen futó X tengely mentén a megfelelő koordinátájú pontokba egy-egy csillagot nyomtat. A csillagok helyzetét a következőképpen határozzuk meg: kiszámítjuk Y = f(X) értékét, megszorozzuk egy léptéktényezővel, a szorzatot a legközelebbi egész számra kerekítjük, végül hozzáadunk egy konstanst, hogy a csillag előtt ennyi hely üresen maradjon.

```
program Rajzolo1(Output);
{Turbo Pascal}
{5.7. program - Az f(X)=exp(-X)*sin(2*Pi*X)
 fuggveny abrazolasa.}

const XLeptek = 16 {az 1 abszcisszaegysegre juto
                    soremelesek szama};
      YLeptek = 32 {az 1 ordinataegysegre juto
                    karakterpoziciok szama};
      ZerusY = 34 {az x-tengely helye};
      XHatar=32 {a grafikon sorokban kifejezett
                 hossza};

var Delta: Real {lepeskoz az abszcisszan};
    KetPi:Real {2*Pi=8*ArcTan(1.0)};
    X,Y:Real;
    Pont:Integer;
    YPozicio:Integer;

begin {a konstansok ertekeinek beallitasa}
      Delta:=1/XLeptek;
      KetPi:=8*ArcTan(1.0);
      for Pont:=0 to XHatar do
          begin X:=Delta*Pont;Y:=Exp(-X)*Sin(KetPi*X);
                YPozicio:=Round(YLeptek*Y)+ZerusY;
                repeat Write(Output, ' ');
                       YPozicio:=YPozicio-1
                until YPozicio=0;
                Writeln(Output, '*')
          end
end.
```

A program eredménye:



Végül tekintsük a következő példaprogramot:

```
program Sorosszegek(Output);
{Turbo Pascal}
{5.8. program - Szamitsuk ki negyfelekeppen az
 1-1/2+1/3-...+1/9999-1/10000 sorosszeget:
 (1) sorrenben, balrol jobbra (SBJ),
 (2) a pozitiv es negativ tagokat balrol jobbra
    osszeadva, majd a ket osszeget egymasbol kivonva
    (BJPoz-BJNeg),
 (3) sorrendben, jobbrol balra (SJB),
 (4) a pozitiv es negativ tagokat jobbrol balra
    osszeadva, majd a ket osszeget egymasbol kivonva
    (JBPoz-JBNeg).}

var SBJ, BJPoz, BJNeg, SJB, JBPoz, JBNeg,
    KovPozBJ, {balrol jobbra haladva a kovetkezo
               pozitiv tag}
    KovNegBJ, {balrol jobbra haladva a kovetkezo
               negativ tag}
    KovPozJB, {jobbrol balra haladva a kovetkezo
               pozitiv tag}
    KovNegJB: Real {balrol jobbra haladva a kovetkezo
                    negativ tag};
    Tagpar: Integer {a tagparok szama};

begin SBJ:=0; BJPoz:=0; BJNeg:=0;
      SJB:=0; JBPoz:=0; JBNeg:=0;
      for Tagpar:=1 to 5000 do
          begin KovPozBJ:=1/(2*Tagpar-1);
                KovNegBJ:=1/(2*Tagpar);
                KovPozJB:=1/(10001-2*Tagpar);
                KovNegJB:=1/(10002-2*Tagpar);
                SBJ:=SBJ+KovPozBJ-KovNegBJ;
                BJPoz:=BJPoz+KovPozBJ;
                BJNeg:=BJNeg+KovNegBJ;
                SJB:=SJB+KovPozJB-KovNegJB;
                JBPoz:=JBPoz+KovPozJB;
                JBNeg:=JBNeg+KovNegJB
          end;
      Writeln(Output,SBJ);
      Writeln(Output,BJPoz-BJNeg);
      Writeln(Output,SJB);
      Writeln(Output,JBPoz-JBNeg)
end.
```

A program eredménye:

```
6.9309718302E-01
6.9309718053E-01
6.9309718306E-01
6.9309718300E-01
```

Miért tér el a négy „azonos" összeg?

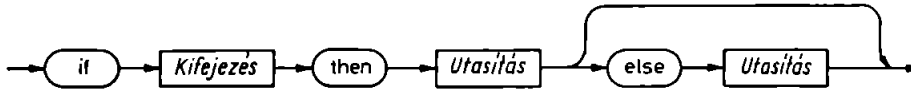
5.5. Feltételes utasítások

A *feltételes utasítás* több utasítást tartalmaz, s ezek közül egyet jelöl ki végrehajtásra. A Pascal kétféle feltételes utasítást biztosít: az if és a case utasítást.

5.5.1. Az if utasítás

Az *if utasítás* csak akkor engedélyezi egy utasítás végrehajtását, ha egy bizonyos feltétel teljesül (tehát ha egy logikai kifejezés igaz értékű). Ha a kifejezés értéke hamis, akkor vagy a következő utasításra lépünk, vagy az else szimbólum utáni utasítás hajtódik végre.

Az if utasítás alakja az 5.13. ábrán látható.



5.13. ábra. *IfUtasítás* szintaxisdiagramja

Az if és a then között álló kifejezés Boolean típusú kell, hogy legyen. Figyeljük meg, hogy az első forma nem más, mint a második alak speciális, rövidebb változata, ahol az alternatív utasítás az üres utasítás. Vigyázat: az else előtt sohasem állhat pontosvessző! Az

```
if  P  then  begin  S1;  S2;  S3  end;  else  S4
```

programrészlet tehát hibás. Talán még inkább csalóka a következő szöveg:

```
if  P  then;  begin  S1;  S2;  S3  end
```

Itt az if a then és a pontosvessző közötti üres utasítás végrehajtását vezérli, így az if utasítást követő összetett utasítás mindig végrehajtódik. Az

```
if kifejezés1 then if kifejezés2 then utasítás1 else utasítás2
```

konstrukció szintaktikai kétértelműségét úgy oldhatjuk fel, hogy hatását az alábbi feltételes utasítással tekintjük egyenértékűnek:

```
if kifejezés1 then
    begin if kifejezés2 then utasítás1
          else utasítás2
    end
```

Ismét figyelmeztetjük az Olvasót, hogy nagy árat fizethet, ha az if utasítást nem kellő körültekintéssel fogalmazza meg. Tegyük fel pl., hogy n db egymást *kölcsönösen kizáró* feltétel n számú különféle tevékenységet vált ki. Jelölje a feltételeket rendre C1, ..., Cn, a tevékenységeket Si, ..., Sn. Legyen P(Ci) annak valószínűsége, hogy Ci értéke igaz, és legyen mondjuk $P(Ci) >= P(Cj)$ ha $i < j$. Ekkor a leghatékonyabb if utasítássorozat:

```
if C1 then S1
    else if C2 then S2
        else ...
            else if C(n-1) then S(n-1) else Sn
```

Ha valamelyik feltétel teljesül, a hozzá tartozó utasítás végrehajtásával az if utasítás befejeződik, s így a hátralevő feltételvizsgálatokat átugorhatjuk.

Ha Found logikai típusú változó, akkor az if utasítás következő példánkban felesleges. (Ez szintén gyakori hiba!)

```
if Kulcs = KeresettÉrték then Found := True else Found := False
```

Sokkal egyszerűbb ezt így leírni:

```
Found := (Kulcs = KeresettÉrték)
```

```
program ArabRomai(Output);
{Turbo Pascal}
{5.9. program - ketto hatvanyainak kiirasa arab es
 romai szamokkal}

var Maradek, Szam: Integer;

begin Szam:=1;
      repeat Write(Output,Szam,  '  ' );
             Maradek:=Szam;
             while Maradek>=1000 do
                   begin Write(Output, 'M');
                         Maradek:=Maradek-1000
                   end;
             if Maradek>=900
             then begin Write(Output, 'CM');
                        Maradek:=Maradek-900
                  end
             else if Maradek>=500
                  then begin Write(Output, 'D');
                             Maradek:=Maradek-500
                       end
                  else if Maradek>=400
                       then begin Write(Output, 'CD');
                                  Maradek:=Maradek-400
                            end;
             while Maradek>=100 do
                   begin Write(Output, 'C');
                         Maradek:=Maradek-100
                   end;
             if Maradek>=90
             then begin Write(Output, 'XC');
                        Maradek:=Maradek-90
                  end
             else if Maradek>=50
                  then begin Write(Output, 'L');
                             Maradek:=Maradek-50
                       end
```

```
                else if Maradek>=40
                     then begin Write(Output, 'XL');
                                Maradek:=Maradek-40
                          end;
               while Maradek>=10 do
                     begin Write(Output, 'X');
                           Maradek:=Maradek-10
                     end;
               if Maradek=9
               then begin Write(Output, 'IX');
                          Maradek:=Maradek-9
                    end
               else if Maradek>=5
                    then begin Write(Output, 'V');
                               Maradek:=Maradek-5
                         end
                    else if Maradek=4
                         then begin Write(Output, 'IV');
                                    Maradek:=Maradek-4
                              end;
               while Maradek>=1 do
                     begin Write(Output, 'I');
                           Maradek:=Maradek-1
                     end;
               Writeln(Output);
               Szam:=Szam*2
          until Szam>5000
end.
```

A program eredménye:

```
1   I
2   II
4   IV
8   VIII
16   XVI
32   XXXII
64   LXIV
128   CXXVIII
256   CCLVI
512   DXII
1024   MXXIV
2048   MMXLVIII
4096   MMMMXCVI
```

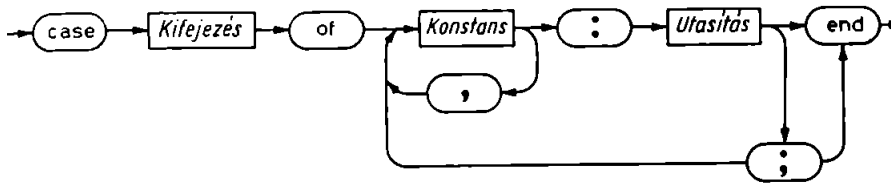
Ismételten felhívjuk a figyelmet arra, hogy az if utasítás egy-egy ága csak egyetlen utasítás végrehajtását vezérelheti. Ezért, ha egy feltételhez több utasítást kötünk, összetett utasítást kell használnunk.

5.5.2. A case utasítás

A case utasítás egy szelektornak nevezett kifejezésből és egy utasításlistából áll. Mindegyik utasításhoz egy, a szelektorral azonos típusú konstans tartozik. A szelektor csak megszámlálható típusú lehet. A case utasítás a szelektor aktuális értékéhez tartozó utasítást jelöli

ki végrehajtásra. Hibát jelent, ha az utasításlista egyik tagja előtt álló konstans sem egyezik meg a szelektor aktuális értékével. A kijelölt utasítás elvégzése után a vezérlés a case utasítás utánra kerül.

Az utasítás alakja az 5.14. ábrán látható.



5.14. ábra. *CaseUtasítás* szintaxisdiagramja

Példák:

(Tegyük fel, hogy var i: Integer; ch: Char;)

```
case i of
     0: x  := 0;
     1: x  := Sin(x);
     2: x  := Cos(x);
     3: x  := Exp(x);
     4: x  := Ln(x)
end

case Ch of
     'a', 'A', 'e', 'E', 'i',
     'I', 'o', 'O', 'u', 'U':

          AngolMagánhangzó  := AngolMagánhangzó + 1;

     '+', '–', '*', '/', '=', '>', '<',
     '.', ',', '''', '?', '!', ':', ';', '"':

          Írásjel  := Írásjel + 1
end
```

Megjegyzések: A case utasításlista tagjai előtt álló konstansok *nem* címkék (l. a 4.2. és 5.7. szakaszt), goto utasítással rájuk ugrani nem lehet, sorrendjük tetszőleges.

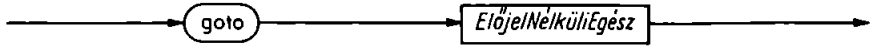
Bár a case utasítás hatékonysága a megvalósítástól függ, általánosságban azt mondhatjuk, hogy akkor célszerű használni, ha több, egymást kölcsönösen kizáró utasításunk van, s mindegyiket ugyanolyan valószínűséggel választhatjuk ki.

5.6. A with utasítás

A *with utasítást* rekord (strukturált) típusú változókkal kapcsolatban alkalmazzuk, így a 8.3. szakaszban tárgyaljuk.

5.7. A goto utasítás

A *goto* olyan egyszerű *utasítás*, amely azt jelzi, hogy a további programvégrehajtást a programszöveg valamely más pontján, mégpedig az adott címke helyén kell folytatni.



5.15. ábra. *GotoUtasítás* szintaxisdiagramja

Minden címke

(1) meg kell, hogy jelenjen a címkedeklarációban, *mielőtt* a blokkban alkalmazhatnánk;
(2) *egy és csak egy* – a blokk utasításrészében álló – utasítást jelölhet;
(3) hatásköre kiterjed a blokk *teljes szövegére*, kivéve, ha valamely beágyazott blokkban a címkét újradeklaráljuk.

A címkének és a rá hivatkozó goto utasításnak az alábbi feltételek közül legalább egynek meg kell felelnie:

(1) A címke olyan utasítás előtt áll, amely magát a goto utasítást (is) tartalmazza.
(2) A címke egy utasítássor (összetett utasítás vagy repeat utasítás) egyik eleme előtt áll; az utasítássor valamelyik eleme a goto utasítást (is) tartalmazza.
(3) A címke egy utasítássor egyik eleme előtt áll; az utasítássor egy blokk utasításrésze; a goto utasítást a blokkban található eljárás vagy függvénydeklaráció tartalmazza.

Példaként tekintsük a következő programrészletet:

```
label 1; {A blokk}
    ...
    procedure B; {B blokk}
        label 3, 5;
    begin
        goto 3;
    3:  Writeln('Jó napot!');
    5:  if P then begin S; goto 5 end; {while P do S}
        goto 1;
        {emiatt korán kilépünk B-ből}
        Writeln('Viszonthallásra!')
    end; {B blokk}

begin
    B;
1:  Writeln('Még beszélnénk!')
    {Az A blokkban "goto 3" nem megengedett}
end {A blokk}
```

Tilos kívülről egy strukturált utasítás belsejébe ugratni. Éppen ezért a következő példák hibásak.

Hibás példák:

(a)
```
for I := 1 to 10 do
      begin S1;
      3:  S2
      end;
goto 3
```

(b)
```
if B  then goto 3;
      . . .
if B1 then 3; S
```

(c)
```
procedure P:
      procedure Q;
      begin . . .
            3: S
      end;
begin . . .
      goto 3
end.
```

A goto utasítást csak olyan szokatlan, rendkívüli esetekben alkalmazzuk, amikor meg kell bontanunk egy algoritmus természetes felépítését. Hasznos szabály: kerüljük az ugrási utasítás alkalmazását, ha utasítások ismételt vagy feltételes végrehajtását akarjuk kifejezni! Az ilyen ugrások ugyanis nem engedik, hogy a program szövegszerkezete (statikus struktúrája) a számítás tényleges menetét, struktúráját tükrözze. A program szöveges és számítási (statikus és dinamikus) szerkezete közötti megfelelő kapcsolat hiánya pedig nagymértékben rontja a program áttekinthetőségét, és jelentősen megnehezíti a program helyességének ellenőrzését. Ha egy Pascal programban goto utasítások vannak, ez gyakran annak a jele, hogy a programozó még nem tanult meg eléggé „Pascalban gondolkodni". (Más programozási nyelvekben a goto-ra szükség van.)

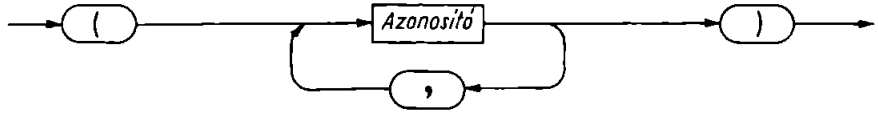
6. Felsorolt és résztartomány típusok

Az előzőekben megismerkedtünk a Boolean, a Char, az Integer és a Real standard típus-azonosítókkal, amelyekkel az általuk jelölt előredefiniált (standard) típusokra hivatkozhatunk. Most két olyan módszert mutatunk be, amellyel új megszámlálható típusokat hozhatunk létre. Az első eredményeként egy olyan új – ún. *felsorolt* (enumerated) – típust kapunk, amely egyetlen más típushoz sem kötődik, míg a második egy meglévő megszámlálható típus érték-készletének valamely részhalmazát definiálja önálló – ún. résztartomány (subrange) – típusként.

6.1. Felsorolt típusok

A felsorolt típusok definíciójában értékek valamely rendezett halmazát adjuk meg oly módon, hogy felsoroljuk az értékeket jelölő konstansazonosítókat.

Az elsőként felsorolt konstans sorszáma 0, a másodiké 1 stb.



6.1. ábra. *FelsoroltTípus* szintaxisdiagramja

Példa:

```
type Szín = (Fehér, Vörös, Kék, Sárga, Lila, Zöld, Narancs, Fekete);
     Nem = (Férfi, Nő);
     Nap = (Hétfő, Kedd, Szerda, Csütörtök, Péntek, Szombat, Vasárnap);
     Operátorok = (Plusz, Mínusz, Szor, Per);
     Földrész = (Afrika, Antarktisz, Ausztrália, Ázsia, Európa, DélAmerika,
                 ÉszakAmerika)
```

Hibás példa:

```
type Munkanap = (Hétfő, Kedd, Szerda, Csütörtök, Péntek, Szombat);
     Szabadnap = (Szombat, Vasárnap);
```

(mert Szombat típusa kétértelmű).

Az Olvasó már ismeri a logikai standard típust, amelynek definíciója:

```
type Boolean = (False, True);
```

Ez automatikusan definiálja a false és a true konstansazonosítót, ill. hogy false < true.

Az =, <>, <, <=, >=, > relációs operátorok minden felsorolt típusra alkalmazhatók, feltéve, hogy mindkét oldal azonos típusú. Az értékek rendezését felsorolásuk sorrendje határozza meg.

Megszámlálható (és így felsorolt) típusú argumentumon értelmezett standard függvények az alábbiak:

succ (X) pl.	succ(Kék) = Sárga		az X utáni érték,
pred (X)	pred(Kék) = Vörös		az X előtti érték,
ord (X)	ord(Kék) = 2		X sorszáma.

Ha C és C1 Szin típusúak (l. fent), B logikai típusú és S1, ..., Sn tetszőleges utasítások, akkor van értelmük a következő utasításoknak:

```
for C  := Fekete downto Vörös do S1;
while (C1 <> C) and B do S1;
if C > Fehér then  C  := pred(C);
case C of
     Vörös, Kék, Sárga: S1;
     Lila: S2;
     Zöld, Narancs:  S3;
     Fehér, Fekete:  S4
end;
```

A 6.1. program felsorolt típusú adatokon végzett műveletekre mutat néhány példát.

```
program TegnapMaHolnap(Output);
{Turbo Pascal}
{6.1. program - A felsorolt tipusok szemleltetese.}

type AHetNapjai = (H,K,Sze,Cs,P,Szo,V);
     Mikor = (Mult,Jelen,Jovo);

var Nap: AHetNapjai;
    Tegnap, Ma, Holnap: AHetNapjai;
    Ido:Mikor;
begin Ma:=V {A Pascalban felsorolt tipusu erteket nem
             lehet az Inputrol beolvasni.};
      Ido:=Jelen;
      repeat if Ido = Jelen
             then {Milyen nap volt tegnap?}
                  begin Ido:=Mult;
                        if Ma = H
                        then Tegnap:=V
                        else Tegnap:=Pred(Ma);
                        Nap:=Tegnap
                  end
```

```
            else if Ido = Mult
                 then {Milyen nap lesz holnap?}
                      begin Ido:=Jovo;
                            if Ma=V
                            then Holnap:=H
                            else Holnap:=Succ(Ma);
                            Nap:=Holnap
                      end
                 else {Ido=Jovo; visszaallitas Jelen
                       -re}
                      begin Ido:=Jelen;
                            Nap:=Ma
                      end;
            case Nap of
                 H: Write(Output, 'Hetfo');
                 K: Write(Output, 'Kedd');
                 Sze: Write(Output, 'Szerda');
                 Cs: Write(Output, 'Csutortok');
                 P: Write(Output, 'Pentek');
                 Szo: Write(Output, 'Szombat');
                 V: Write(Output, 'Vasarnap')
            end;
            Writeln(Output, (Ord(Ido)-1):3)
      until Ido = Jelen
end.
```

A program eredménye:

```
Szombat -1
Hetfo  1
Vasarnap  0
```

6.2. Résztartomány típusok

Egy típust definiálhatunk úgy is, hogy valamely, már előzőleg definiált megszámlálható típus egy részhalmazát, *résztartományát* adjuk meg. Az eredeti típust a résztartomány típus *alaptípusának* (host type) nevezzük. A résztartományt egyszerűen a tartomány legkisebb és legnagyobb konstansának megadásával definiáljuk. Az alsó határ a felsőnél nagyobb nem lehet. A valós típusból résztartomány *nem* képezhető, mivel a Real nemmegszámlálható típus.

6.2. ábra. *RésztartományTípus* szintaxisdiagramja

Az alaptípus szabja meg, hogy a résztartomány típusok értékein milyen műveletek értelmezhetők. Mint mondtuk, a megszámlálható típusok értékadás-kompatibilitásának feltétele, hogy a változó és a kifejezés ugyanahhoz a megszámlálható típushoz, ill. annak egy-egy résztartományához tartozzon, és a kifejezés értéke a változó típusa által meghatározott zárt intervallumba essen. Induljunk ki pl. az alábbi deklarációból:

```
var A: 1..10; B: 0..30; C: 20..30;
```

Itt A, B és C skalár alaptípusa az egész típus, így az

```
A := B; C := B; B := C;
```

értékadások mind helyes utasítások, bár végrehajtásuk esetleg lehetetlen. Ezért mindenütt, ahol könyvünkben megszámlálható típusokról szólunk, állításaink ezek résztartományaira is vonatkoznak, bár ezt nem mindig hangsúlyozzuk.

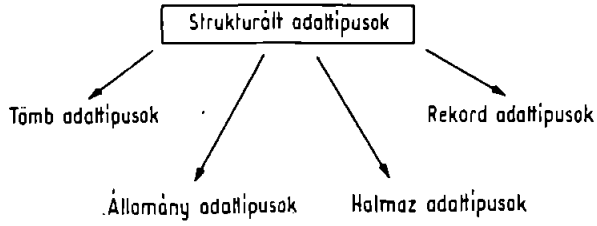
Példa:

```
type Napok = (Hétfő, Kedd, Szerda, Csütörtök, Péntek, Szombat, Vasárnap)
        { felsorolt típus } ;
     Munkanapok = Hétfő..Péntek { napok résztartománya } ;
     Index = 0..63 { az egész típus résztartománya } ;
     Betű = 'A'..'Z' { a karakter típus résztartománya } ;
     TermészetesSzámok = 0..MaxInt;
     PozitívSzámok = 1..MaxInt;
```

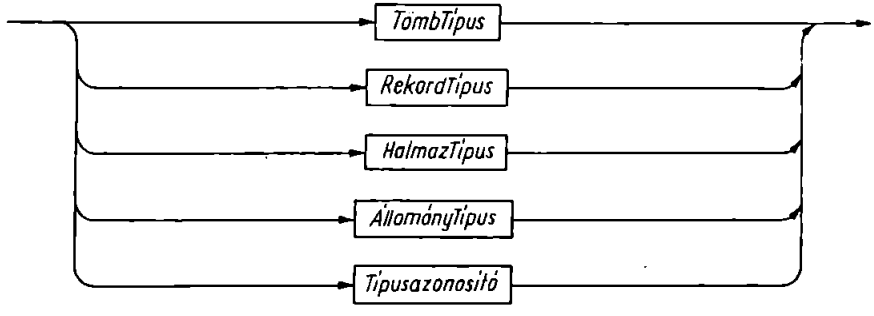
A résztartomány típusok a feladatra orientált problémamegfogalmazás eszközei. A gépi megvalósítást végző szakember szempontjából alkalmazásuk tármegtakarítást tesz lehetővé, és módot ad az értékadások futásidőben történő ellenőrzésére (l. pl. a 7.1. programot). Egy 0..200 típusúnak deklarált változó pl. sok rendszerben lehet, hogy csak egy byte-ot (8 bitet) foglal el, egy Integer változó viszont több byte-ot köthet le.

7. Strukturált típusok – tömbök

Az egyszerű (megszámlálható és valós) típusok strukturálatlan típusok. A Pascal rajtuk kívül strukturált és mutató típusokat különböztet meg. Ahogyan a strukturált utasítások is más utasításokból felépülő egységek, a strukturált típusok is más típusokból felépülő konstrukciók. A strukturált típust az *elemek* (összetevők) típusa(i) és legfőképpen a strukturálás módja jellemzik.



7.1. ábra. A strukturált adattípusok osztályozása



7.2. ábra. *StrukturáltTípus* szintaxisdiagramja

Bármelyik strukturálási módot választjuk, mindig lehetőségünk van a számunkra legkedvezőbb belső adatábrázolás előírására. A típusdefiníció elé a packed (tömörített) szimbólumot írva utasíthatjuk a fordítóprogramot, hogy takarékosan gazdálkodjon a tárkapacitással, még akkor is, ha ez az elvégzendő kifejtési és tömörítési műveletek miatt esetleg a futási idő és a kód hosszabbodását vonja maga után. A felhasználó dolga, hogy él-e ezzel a lehetőséggel, enged-e a hatékonyságból a helyigény csökkentése kedvéért. (A hatékonyságra és a tárigényre gyakorolt hatások tényleges alakulása a gépi megvalósítástól függ, és az is lehet, hogy az eredő hatás végül is zérus.)

7.1. A tömb (array) típus

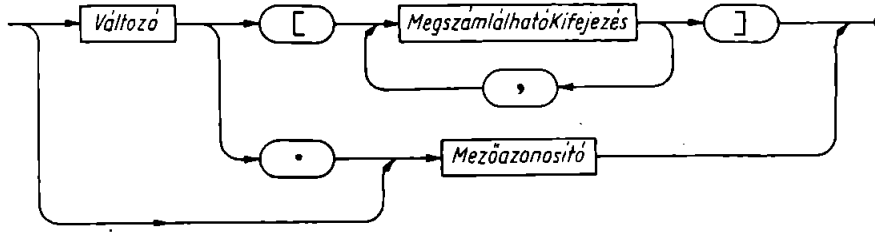
A tömb rögzített számú elemből áll (ezt a tömb létrehozásakor definiáljuk). Az elemek mind azonos típushoz, az ún. *elemtípushoz* tartoznak. Az egyes elemekre a tömbváltozó nevével és az azt követő, szögletes zárójelben álló ún. *indexszel* hivatkozhatunk. Így az elemek explicit módon jelölhetők, és közvetlenül hozzáférhetők. Az index maga is lehet számítási eredmény, típusa az ún. *indextípus*. Mivel az egyes elemek elérési ideje nem függ a szelektor (index) értékétől, a tömböt *közvetlen elérésű struktúrának* nevezzük.

A tömb definíciójában mind az elemtípust, mind az indextípust meg kell adni. A definíció alakja:

```
type A = array [T1] of T2;
```

ahol A egy új típusazonosító, T1 az indextípus, amely csak megszámlálható típus lehet. T2 pedig tetszőleges típus.

A tömbstruktúra segítségével egy névvel több, azonos jellemzőkkel rendelkező változót foghatunk össze. A tömbváltozó deklarációja a teljes struktúrának nevet ad. A teljes tömbváltozókra két művelet megengedett: az értékadás és az elemkijelölés. Az utóbbi úgy történik, hogy a tömbváltozó neve után szögletes zárójelben egy megszámlálható kifejezéssel megadjuk a kívánt elem indexét. Az így kijelölt elemváltozóra mindazok a műveletek megengedettek, amelyek az adott tömb típus elemének a típusához tartozó változókra általában értelmezettek.



7.3. ábra. *ElemTípus* szintaxisdiagramja

Példák változódeklarációra:

```
Tár: array [0..Max] of Integer
Beteg: packed array [Napok] of Boolean
```

Példák értékadásra:

```
Tár[I + J]: = X
Beteg[Hétfő]: = true
```

(A példák természetesen feltételezik, hogy a bennük használt segédazonosítókat korábban már definiáltunk.)

A 7.1. és a 7.2. program a tömbök alkalmazását szemlélteti. Az Olvasó gondolja meg, hogyan kellene kibővíteni a 7.2. programot – tömb alkalmazásával és a nélkül –, ha egynél több függvényt akarnánk ábrázolni!

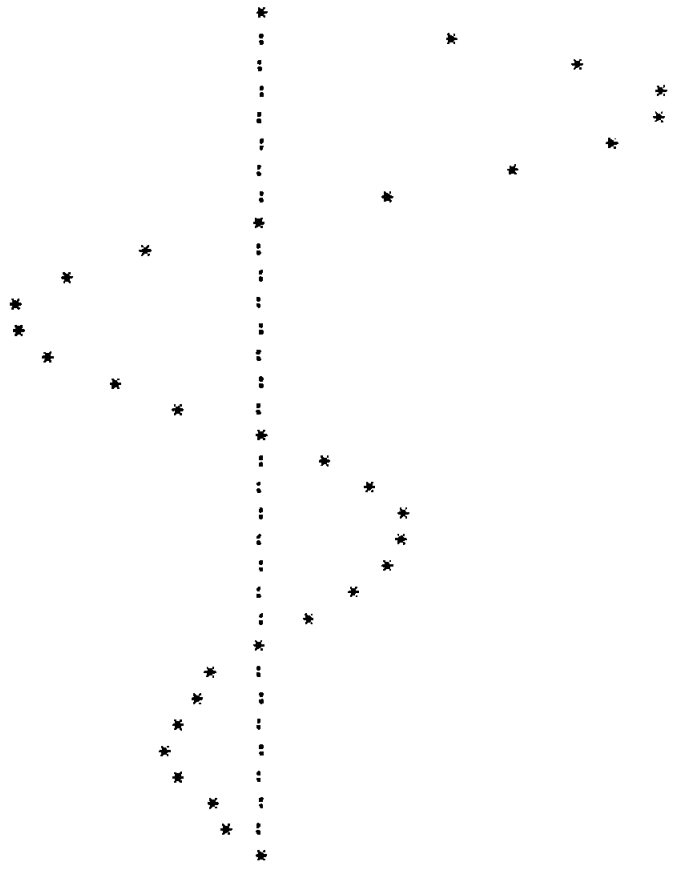
```
program MinMax(Input,Output);
{Turbo Pascal}
{7.1. program - Adott lista legnagyobb es legkisebb
 elemenek kivalasztasa.}

const MaxMeret = 16;

type ListaMeret = 1..MaxMeret;

var Elem: ListaMeret;
    Min, Max, Elso, Masodik: Integer;
    A: array[Listameret] of Integer;

begin for Elem:=1 to MaxMeret do
          begin Read(Input, A[Elem]);
                Write(Output, A[Elem]:4)
          end;
      Writeln(Output);
      Min:=A[1];Max:=Min;Elem:=2;
      while Elem<MaxMeret do
            begin Elso:=A[Elem];
                  Masodik:=A[Elem+1];
                  if Elso>Masodik
                  then begin if Elso>Max
                             then Max:=Elso;
                             if Masodik<Min
                             then Min:=Masodik
                       end
                  else begin if Masodik>Max
                             then Max:=Masodik;
                             if Elso<Min
                             then Min:=Elso
                       end;
                  Elem:=Elem+2
            end;
      if Elem = MaxMeret
      then if A[MaxMeret]>Max
           then Max:=A[MaxMeret]
           else if A[MaxMeret]<Min
                then Min:=A[MaxMeret];
      Writeln(Output, Max, Min:6)
end.
```

A program eredménye:

```
35  68  94   7  88  -5  -3  12  35   9  -6   3   0  -2  74  88
 94    -6
```

```
program Rajzolo2(Output);
{Turbo Pascal}
{7.2. program - Az f(X)=exp(-X)*sin(2*Pi*X) abrazolasa
 az  X-tengely felrajzolasaval.}

const XLeptek=16 {az 1 abszcisszaegysegre juto
                  soremelesek szama};
      YLeptek=32 {az egy ordinataegysegre juto
                  karakterpoziciok szama};
      ZerusY=34 {az X-tengely helye};
      XHatar=32 {a grafikon sorokban kifejezett
                 hossza};
      YHatar=68 {a grafikon karakterpoziciokban
                 kifejezett szelessege};

type Tartomany=1..YHatar;

var Delta: Real {lepeskoz az abszcisszan};
    KetPi: Real {2*Pi=8*ArcTan(1.0)};
    X, Y: Real;
    Pont: Integer;
    Hely, YPozicio, Futashatar: Tartomany;
    Ykar: array[Tartomany] of Char;

begin {a konstansok ertekeinek beallitasa}
      Delta:=1/XLeptek;
      KetPi:=8*ArcTan(1.0);
      for Hely:=1 to YHatar do
          Ykar[Hely]:=' ';
      for Pont:=0 to XHatar do
          begin X:=Delta*Pont;
                Y:=Exp(-X)*Sin(KetPi*X);
                YKar[ZerusY]:=':';
                YPozicio:=Round(YLeptek*Y)+ZerusY;
                YKar[YPozicio]:='*';
                if YPozicio<ZerusY
                then Futashatar:=ZerusY
                else Futashatar:=YPozicio;
                for Hely:=1 to Futashatar do
                    Write(Output,YKar[Hely]);
                Writeln(Output); YKar[YPozicio]:=' '
          end
end.
```

A program eredménye:



Mivel T2 tetszőleges típus lehet, a tömbök elemei strukturáltak is lehetnek. Abban a speciális esetben, amikor T2 is tömb, az eredeti A tömböt *többdimenziósnak* mondjuk. Egy M többdimenziós tömb deklarációját tehát a következőképpen írhatjuk fel:

```
var M : array [A..B] of array [C..D] of T ;
```

és ekkor M[I] [J] M I-edik elemének (T típusú) J-edik komponensét jelöli.

Többdimenziós tömböknél az alábbi kényelmes, rövidebb írásmódot szokás alkalmazni:

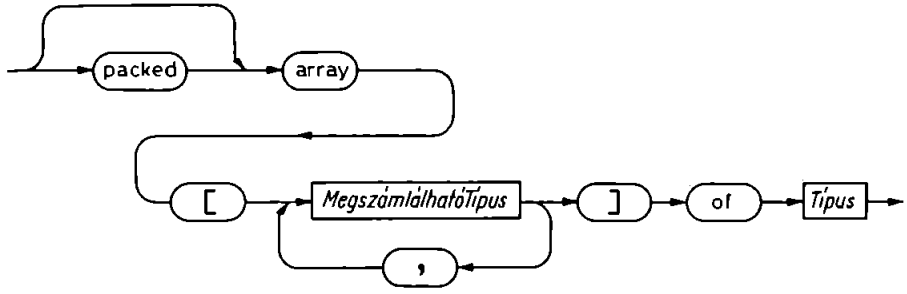
```
var M : array [A..B, C..D] of T;
```

ill.

```
M[I, J]
```

M-et mátrixnak is tekinthetjük, és azt mondhatjuk, hogy M[I,J] az M mátrix I-edik elemének J-edik komponense, vagyis az M matrix I-edik sorának a J-edik oszlopba tartozó eleme.

Mindez nem korlátozódik csupán két dimenzióra, mert T megint lehet strukturált típusú. A típus (rövid) alakja általában:



7.4. ábra. *TömbTípus* szintaxisdiagramja

Ha *n* számú indextípust adunk meg, a tömböt *n-dimenziósnak* mondjuk, a tömb egy elemét pedig *n* darab indexkifejezéssel jelöljük ki.

Ha A és B két azonos típusú tömbváltozó, akkor – feltéve, hogy az egyik tömb elemei rendre felvehetik a másik elemeinek értékét, azaz:

```
A [i] := B [i]
```

minden, az indextípushoz tartozó i-re végrehajtható –, az elemenkénti értékadás helyett a következő, rövidebb alakot használhatjuk:

```
A := B
```

```
program MatrixSzorzas(Input,Output);
{Turbo Pascal}
{7.3. program - Matrixszorzas}

const M = 4;
      P = 3;
      N = 2;

var I: 1..M;
    J: 1..N;
    K: 1..P;
    Osszeg, Elem: Integer;
    A: array[1..M, 1..P] of Integer;
    B: array[1..P, 1..N] of Integer;
    C: array[1..M, 1..N] of Integer;
begin {A es B elemeinek feltoltese}
      for I:=1 to M do
          begin for K:=1 to P do
                    begin Read(Input,Elem);
                          Write(Output,Elem:5);
                          A[I,K]:=Elem
                    end;
                Writeln(Output)
          end;
      Writeln(Output);
```

```
        for K:=1 to P do
            begin for J:=1 to N do
                        begin Read(Input,Elem);
                              Write(Output,Elem:5);
                              B[K,J]:=Elem
                        end;
                  Writeln(Output)
            end;
        Writeln(Output);
           {Az A es B matrix szorzatat tartalmazo
            C matrix eloallitasa}
        for I:=1 to M do
            begin for J:=1 to N do
                        begin Osszeg:=0;
                              for K:=1 to P do
                                  Osszeg:=Osszeg+
                                          A[I,K]*B[K,J];
                              C[I,J]:=Osszeg;
                              Write(Output,Osszeg:5)
                        end;
                  Writeln(Output)
            end;
        Writeln(Output)
end.
```

A program eredménye:

```
  1     2     3
 -2     0     2
  1     0     1
 -1     2    -3

 -1     3
 -2     2
  2     1

  1    10
  6    -4
  1     4
 -9    -2
```

Figyeljük meg, hogy az előbbi programban az A, B és C tömb indextípusa rögzített volt. Hogy általánosított, programkönyvtárba tehető mátrixszorzó alprogramot írhassunk, szükségünk van egy olyan eszközre, amellyel az indextípusokat megfelelően beállíthatjuk. A Pascalban erre a célra az illeszkedőtömb-paraméterek szolgálnak (l. a 12.1.2. pontot); használatukat a 12.4. szakasz (Mátrixszorzás2) szemlélteti.

7.2. Füzér típusok

A *füzért* korábban mint aposztrófok közé tett jelsorozatot definiáltuk (l. a 2.5. szakaszt). Az egy jelből álló füzér a standard karaktertípushoz (l. a 3.4. szakaszt), az N ($N > 1$) jelet tartalmazó füzér pedig a

```
packed array [1..N] of Char
```

definícióval megadott típushoz tartozó konstans. Az ilyen típust *füzér típusnak* nevezzük.

Ha az A tömbváltozó és az E kifejezés azonos elemszámú füzér típushoz tartozik, megengedett az

```
A := E
```

értékadás. Ugyanígy, két azonos elemszámú füzért a relációs operátorokkal (=, <>, <, >, <= és >=) összehasonlíthatunk; a rendezésnél az első elem (A [1]) számít a legértékesebbnek, s a sorrendet a standard Char típus értéksorrendje szabja meg.

7.3. Tömörítés (pack) és kifejtés (unpack)

A tömörített tömbök egyes elemeihez gyakran nehéz hozzáférni. Így a feladattól, ill. az adott nyelvi megvalósítástól függően kívánatos lehet, hogy a programozó a tömb elemeinek tömörítését, ill. kifejtését egyetlen művelettel végezze. Ez a Pack (tömörítés) és az Unpack (kifejtés) standard eljárások segítségével valósítható meg. Legyen U egy

```
array [A..D] of T  { T állomány típust nem tartalmazhat } ;
```

típusú tömörítetlen, P pedig egy

```
packed array [B..C] of T
```

típusú tömörített tömbváltozó, ahol

```
Ord(D) - Ord(A) >= Ord(C) - Ord(B)
```

Ekkor

```
Pack (U, I, P)
```

U-nak az I-edik elemtől kezdődő részét P-be tömöríti, míg

```
Unpack (P, U, I)
```

U-nak az I-edik elemtől kezdődő részébe fejti ki P-t.

8. Rekord (record) típusok

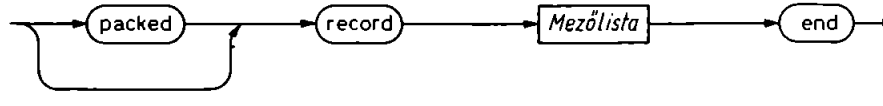
A rekord talán a legrugalmasabb szerkezet. A rekord típus gyűjtőfogalom, ugyanis a rekord olyan szerkezet, amelyben teljesen eltérő tulajdonságú részek fordulhatnak elő. Tegyük fel, hogy pl. egy személy adatait szeretnénk nyilvántartani. Ismert a név, a magasság, a nem, a születési idő, az eltartottak száma és a családi állapot. Ezenkívül ha az illető nős, férjezett vagy özvegy, adott az (utolsó) házasságkötés időpontja; ha elvált, tudjuk a (legutóbbi) válás dátumát, és hogy ez az első válása vagy sem; ha pedig nőtlen vagy hajadon, nincs szükségünk több adatra. Mindezt az információt egyetlen rekordba foglalhatjuk, amelyben minden adatelemhez különkülön hozzáférhetünk.

8.1. Rögzített rekordok

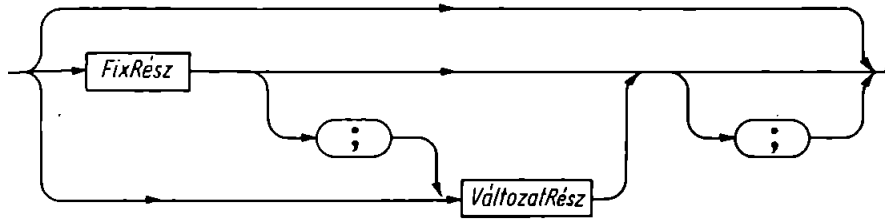
Pontosabban fogalmazva a rekord rögzített számú, *mezőnek* nevezett elemből felépülő adatszerkezet. A tömbtől eltérően az elemeknek nem kell azonos típusúaknak lenniük, és kifejezéssel nem indexelhetők. A rekord típus definíciója megadja az egyes elemek típusát, és minden elemhez egy ún. *mezőazonosítót* rendel. Teljes rekordváltozókra a Pascal két műveletet enged meg: az értékadást és az elemkiválasztást. A mezőazonosító hatásköre az a legbelső rekord, amelyben definiáltuk.

Azért, hogy a kiválasztott elem típusa közvetlenül a programszöveg alapján, a program végrehajtása nélkül is leolvasható legyen, a rekordszelektorban nem valamilyen számított indexérték, hanem rögzített mezőazonosító áll.

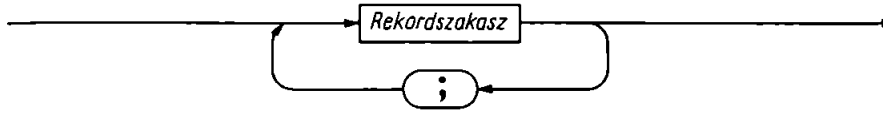
Egyszerű példaként tegyük fel, hogy a + bi alakú komplex számokkal akarunk számításokat végezni, ahol a és b valós szám, i pedig -1 négyzetgyöke. „Komplex" nevű standard típusunk nincs. A programozó azonban könnyen definiálhat a komplex számok ábrázolására alkalmas rekordot. Ebben a rekordban két, Real típusú mezőnek kell lennie, amelyek a valós, ill. a képzetes részt írják le. Ezt az alábbi szintaxisok segítségével fejezhetjük ki:



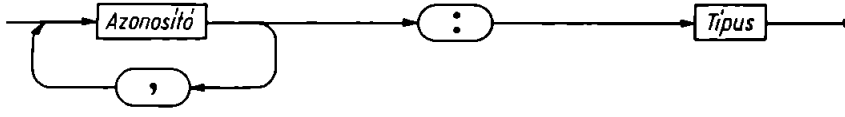
8.1. ábra. *RekordTípus* szintaxisdiagramja



8.2. ábra. *Mezőlista* szintaxisdiagramja



8.3. ábra. *FixRész* szintaxisdiagramja



8.4. ábra. *Rekordszakasz* szintaxisdiagramja

A fenti szabályok alkalmazásával a következő definícióra és deklarációra juthatunk:

```
type Komplex = record Re, Im: Real end;
var Z: Komplex;
```

ahol Komplex a típusazonosító, Re és Im mezőazonosítók, Z pedig egy Komplex típusú változó, azaz egy kételemű, két mezőből álló rekord (l. a 8.1. programot).

```
program KomplexAritmetika(Output);
{Turbo Pascal}
{8.1. pragram - Muveletek komplex szamokkal.}
const Novekmeny=4;

type Komplex = record  Re, Im: Real
                end;

var X,Y: Komplex;
    Par: Integer;

begin X.Re:=2;X.Im:=5; {X kezdoerteke}
      Y:=X; {Y kezdoerteke}
      for Par:=1 to 5 do
          begin Writeln(Output, 'X= ', X.Re:5:1,
                        X.Im:6:1, 'i');
                Writeln(Output, 'Y= ', Y.Re:5:1,
                        Y.Im:6:1, 'i');
```

```
                    {X+Y}
                    Writeln(Output, 'Osszeg= ',
                            X.Re+Y.Re:5:1,
                            X.Im+Y.Im:6:1, 'i');
                    {X*Y}
                    Writeln(Output,'Szorzat= ',
                            X.Re*Y.Re-X.Im*Y.Im:5:1,
                            X.Re*Y.Im+X.Im*Y.Re:6:1, 'i');
                    Writeln(Output);
                    X.Re:=X.Re+Novekmeny;
                    X.Im:=X.Im-Novekmeny
                end
end.
```

A program eredménye:

```
X=    2.0    5.0i
Y=    2.0    5.0i
Osszeg=    4.0   10.0i
Szorzat=  -21.0   20.0i

X=    6.0    1.0i
Y=    2.0    5.0i
Osszeg=    8.0    6.0i
Szorzat=    7.0   32.0i

X=   10.0   -3.0i
Y=    2.0    5.0i
Osszeg=   12.0    2.0i
Szorzat=   35.0   44.0i

X=   14.0   -7.0i
Y=    2.0    5.0i
Osszeg=   16.0   -2.0i
Szorzat=   63.0   56.0i

X=   18.0  -11.0i
Y=    2.0    5.0i
Osszeg=   20.0   -6.0i
Szorzat=   91.0   68.0i
```

A rekord valamely elemére (mezőjére) úgy hivatkozhatunk, hogy leírjuk a rekord nevét, majd egy pont után a mező azonosítóját (l. a 7.3. ábrát). A következő két programsor pl. Z-nek 5 + 3i értéket ad:

```
Z.Re := 5;
Z.Im := 3
```

Ugyanígy definiálhatunk egy dátumot leíró változót is:

```
Dátum = packed record Év: 1900...2100;
                      Hó: (Jan, Feb, Márc, Ápr, Máj, Jún, Júl, Aug, Szept, Okt, Nov,
                           Dec);
                      Nap: 1..31
                end;
```

Megjegyzés: A Dátum típus megengedi pl. az április 31-et is!

Egy játék megadása:

```
Játék = record Milyen: (Labda, Búgócsiga, Hajó, Baba, Kocka, Társas, Modell, Könyv);
               Ár: Real;
               Kaptam: Dátum;
               Szeretem: (Nagyon, Eléggé, Kicsit, Nem);
               Eltört, Elveszett: Boolean
        end;
```

Egy házi feladat jellemzőinek leírása:

```
Hf = record Tantárgy: Töri, Nyelv, Irodalom, Matek, Fiz, Kém);
            Kiírva: Dátum;
            Osztályzat: 1..5;
            Nehézség: 1..10
     end;
```

Ha a rekord maga is egy másik szerkezetbe van ágyazva, a rekordváltozó neve tükrözi ezt a szerkezetet. Tegyük fel, hogy azt szeretnénk nyilvántartani, hogy a család egyes tagjai mikor kaptak utoljára himlőoltást. Az egyik lehetőség, hogy a családtagokat felsorolt típusként, az oltási időpontokat pedig rekordokból álló tömbként definiáljuk:

```
type Családtag = (Apa, Anya, Gyerek1, Gyerek2, Gyerek3);
var OltásDátuma: array [Családtag] of Dátum;
```

A legújabb adatokat ekkor az alábbi alakban vehetjük nyilvántartásba:

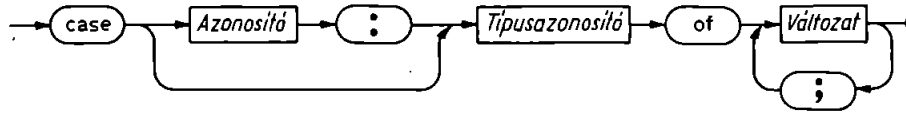
```
OltásDátuma [Gyerek3].Év  := 1986
OltásDátuma [Gyerek3].Hó  := Ápr;
OltásDátuma [Gyerek3].Nap := 23
```

8.2. Változó rekordok

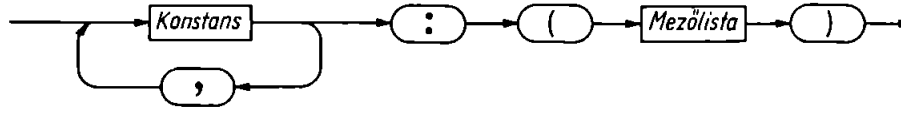
Előfordulhat, hogy olyan információt szeretnénk felvenni egy rekordszerkezetbe, amely más, már a rekordban levő információtól függ. Ilyenkor változórekord-típust adhatunk meg, amelybe bizonyos mezők csak akkor kerülnek be, ha egy másik mező értéke ezt előírja.

A rekord típus szintaxisa lehetőséget biztosít változat rész képzésére is, azaz amint az elnevezés is mutatja, egy rekord típus több *változatot* is tartalmazhat. Ez azt jelenti, hogy különböző, jóllehet azonos típusúnak mondott változók bizonyos szempontból eltérő szerkezetűek lehetnek. Eltérő lehet az elemek száma és típusa.

Az egyes változatokat egy-egy lista jellemzi, amely a változathoz tartozó elemek deklarációját tartalmazza. A listát zárójelek határolják. Minden ilyen deklarációlista előtt egy vagy több konstans áll, az egész listasort pedig egyetlen case utasítás fogja össze, amely megadja a konstansok típusát (vagyis azt a típust, amely szerint a változatokat különválasztottuk).



8.5. ábra. *VáltozatRész* szintaxisdiagramja



8.6. ábra. *Változat* szintaxisdiagramja

Példaként tegyük fel, hogy létezik egy:

```
type CsaládiÁllapot = (Nős, Özvegy, Elvált, NőtlenVagyHajadon);
```

Ekkor egy személy adatait a következő módon írhatjuk le:

```
type Személy =
     record
          {az összes személynél közös mezők};
          case CsaládiÁllapot of
               NősVagyFérjezett: ( {a kizárólag házas személyekre vonatkozó mezők});
               NőtlenVagyHajadon: ( {a kizárólagos egyedülálló személyekre vonatkozó
                  mezők} );
                     ...
     end;
```

Fontos, hogy a változatok megkülönböztetésére használt ún. kijelölő típus valamennyi lehetséges értéke előforduljon valamelyik változat előtt. Fenti példánk tehát csak akkor helyes, ha a változatfelsorolásban a NősVagyFérjezett és a NőtlenVagyHajadon után az Özvegy és az Elvált konstans is megjelenik.

Többnyire maga a rekordelem (mező) jelöli ki, hogy éppen melyik változata „hatályos". A fent definiált személyi rekordban pl. valószínűleg szerepel egy

```
CsÁ: CsaládiÁllapot
```

közös mező. Ebben a gyakran előforduló esetben lerövidíthetjük a programot, ha a kijelölést végző elem, az ún. kijelölő mező deklarációját magába a case utasításba írjuk, az alábbi példa szerint:

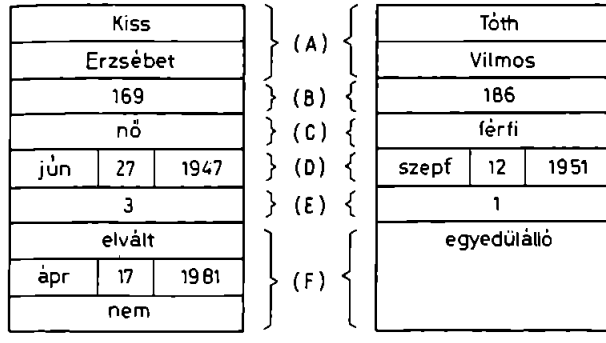
```
case CsÁ: CsaládiÁllapot of
```

Mielőtt a személyi adatokat megpróbálnánk változórekord-szerkezetben megadni, célszerű külön összefoglalni a szükséges információkat.

I. Személy
(A) név (vezetéknév, keresztnév);
(B) magasság (természetesszám);
(C) nem (férfi, nő);
(D) születési idő (év, hó, nap);
(E) eltartottak száma (természetes szám);

(F) családi állapot:
ha nős, férjezett vagy özvegy:
(a) a házasságkötés dátuma (év, hó, nap)
ha elvált:
(a) válás dátuma (év, hó, nap),
(b) első válás, (false, true)
ha nőtlen vagy hajadon.

A 8.7. ábra illusztrációképpen két, különböző adatokkal rendelkező személy „nyilvántartási lapját" mutatja.



8.7. ábra. Két „nyilvántartási lap" minta

A Személy rekord ezek alapján a következőképpen írható fel:

```
type  Füzér15 = packed array [1..15] of Char;
      CsaládiÁllapot = (NősVagyFérjezett, Özvegy, Elvált, NőtlenVagyHajadon);
      Dátum = packed record
                        Év: 1900..2100;
                        Hó: (Jan, Feb, Márc, Ápr, Máj, Jún, Júl, Aug, Szep, Okt,
                            Nov, Dec);
                        Nap: 1..31;
                  end;
      TermészetesSzám = 0..MaxInt;
      Személy = record
                  Név: record Vezetéknév, Utónév: String15 end;
                  Magasság: TermészetesSzám  {cm};
                  Nem: (Férfi, Nő);
                  Születésildő: Dátum;
                  EltartottakSzáma: TermészetesSzám
                  case CsÁ: CsaládiÁllapot of
                        NősVagyFérjezett, Özvegy: (HázasságkötésDátuma:
                           Dátum);
                        Elvált: (VálásDátuma: Dátum;
                              ElsőVálás: Boolean);
                        NőtlenVagyHajadon: ( )
            end {Személy};
```

Megjegyzések:

(1) Minden mezőnév különböző kell, hogy legyen, még akkor is, ha különböző változatokban szereplő nevekről van szó.

(2) Ha valamelyik változat üres (tehát egy mezője sincs), alakja: C: ( )
(3) Egy mezőlistában csak egy változó rész lehet, s ez csak a fix rész után állhat.
(4) Egy változat maga is tartalmazhat változatrészt, tehát a változatrészek egymásba ágyazhatók.
(5) A rekord típusban bevezetett felsorolt típusú konstansazonosítók hatásköre a befoglaló blokkra terjed ki.

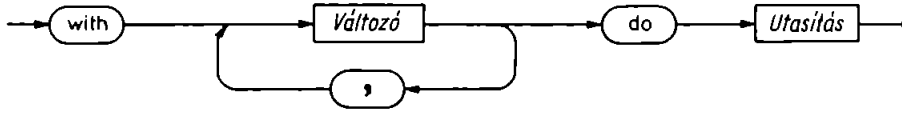
A rekord valamely elemére való hivatkozás lényegében a rekord vázának lineáris rekonstrukciója. Példaként vegyünk egy Személy típusú P változót, és állítsuk elő a 8.7. ábra második személyének adatait:

```
P.Név.Vezeték := 'Macska';
P.Név.Kereszt := 'Márton';
P.Magasság := 186;
P.Nem := férfi;
P.Született.Év := 1951;
P.Született.Hó := Szept;
P.Született.Nap := 12;
P.Eltartottak := 1;
P.CsÁ := NőtlenVagyHajadon;
```

8.3. A with utasítás

A fenti írásmód kissé terjengős lehet, de a felhasználó ugyanezt rövidebben is megfogalmazhatja, mégpedig a *with utasítás* segítségével. A with utasítás felnyitja az adott rekordváltozó mezőazonosítóit tartalmazó hatáskört, s ily módon a mezőazonosítók változóazonosítókként szerepelhetnek. Ezáltal lehetőséget biztosít a fordítóprogramnak, hogy optimalizálja a with utasításban kijelölt utasítást.

A with utasítás általános alakja a 8.8. ábrán látható.



8.8. ábra. *WithUtasítás* szintaxisdiagramja

A with utasítás do utáni utasításában a rekordváltozó kívánt mezőjére egyszerűen a mezőazonosítóval hivatkozunk. Nem tesszük ki tehát a mezőazonosító elé az egész rekord nevét.

Az alábbi with utasítás egyenértékű az előbbi értékadásokkal:

```
with P do
    begin
        with Név do
            begin Utónév := 'Márton  ';
                  Vezetéknév := 'Macska  '
            end;
        Magasság := 186;
```

```
        Nem  := Férfi;
        with Születésildő do
             begin Év  := 1951; Hó  := Szep; Nap  := 12 end;
        EltartottakSzáma  := 1;
        CsÁ  := NőtlenVagyHajadon;
    end
```

Hasonlóképpen, a

```
var MaiDátum: Dátum;
...
with MaiDátum do
    if Hó = Dec then
        begin Hó  := Jan; Év  := Év + 1
        end
    else Hó  :=  Succ(Hó)
```

programrészlet egyenértékű a következővel:

```
var MaiDátum  : Dátum;
...
if MaiDátum.Hó = Dec then
    begin MaiDátum.Hó  := Jan;
        MaiDátum.Év  := MaiDátum.Év + 1
    end
else MaiDátum.Hó  :=  Succ (MaiDátum.Hó)
```

Az alábbi programrészlet pedig a korábbi példánkban látott himlőoltás-nyilvántartást végzi:

```
with OltásDátuma [Gyerek3] do
    begin Év  := 1986; Hó  := Ápr; Nap  := 23 end
```

A with utasítás végrehajtásakor a do utáni utasítás végrehajtása előtt „ráállunk" a kívánt rekordváltozóra. Ha tehát a belső utasításban a rekord változólista valamelyik elemére vonatkozó értékadás áll, az nem változtatja meg a rekordváltozót.

Például:

```
var Kicsoda: Családtag;
...
Kicsoda  := Apa;
with OltásDátuma [Kicsoda] do
    begin
        Kicsoda  := Anya;
        Év  :=  1947; Hó  := Júl; Nap  :=  7;
    end
```

Itt a with utasítás OltásDátuma [Apa] mezőit állítja be.

Az egymásba ágyazott with utasításokat rövidebben is felírhatjuk. A

```
with R1, R2, ..., RN do S
```

alak a

```
with R1 do
    with R2 do
    ...
        with RN do S
```

utasításnak felel meg.

A P személy adatait megadó előbbi példánkat tehát a következőképpen írhatnánk át:

```
with P, Név, Születésildő do
    begin Vezetéknév := 'Macska    ';
        Utónév := 'Márton    ';
        Magasság := 186;
        Nem := férfi;
        Év := 1951;
        Hó := Szep;
        Nap := 12;
        EltartottakSzáma := 1;
        CsÁ := NőtlenVagyHajadon;
    end  { with }
```

Nézzünk most egy példát a mezőazonosítók hatáskörére! Míg

```
var A: array [2..8] of Integer;
    A: 2..8;
```

hibás, mivel A definíciója nem egyértelmű, a

```
var A: Integer;
    B: record A: real; B: Boolean
       end;
```

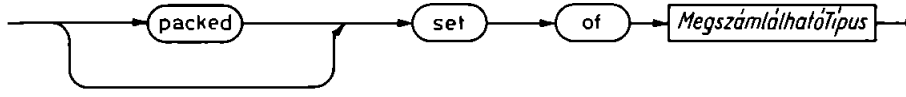
szöveg helyes, minthogy az egész A és a valós B.A könnyen megkülönböztethető. Ugyanígy a B rekordváltozó is megkülönböztethető a Boolean B.B-től. A

```
with B do S
```

utasításban szereplő S utasításban az A, ill. B azonosító a B.A, ill. a B.B elemet jelöli, az A egész változó nem hozzáférhető.

9. Halmaz (set) típusok

A halmaz (set) típus alkalmazásával tömör alakban dolgozhatunk az egyazon megszámlálható típushoz tartozó értékek együttesével. Pontosabban a halmaz típus alaptípusa hatványhalmazát definiálja. Ez az alaptípusbeli értékek összes részhalmazának halmaza (amely az üres halmazt is tartalmazza). A halmaz is közvetlen elérésű struktúra. Jellemzője, hogy elemei mind ugyanazon – megszámlálható – típushoz tartoznak.



9.1. ábra. *HalmazTípus* szintaxisdiagramja

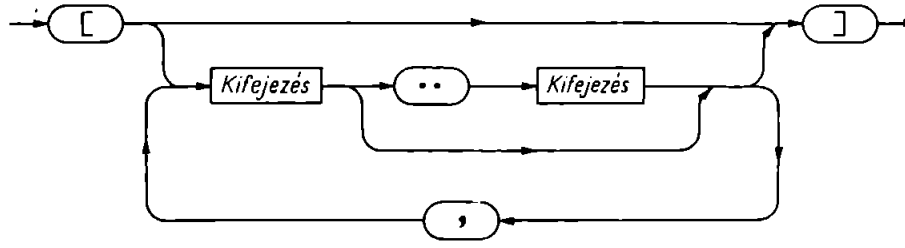
Teljes halmazértékekre a következő műveletek megengedettek: az értékadás, az ismert halmazműveletek (pl. egyesítés = unió), az egyenlőség- és részhalmazvizsgálat, valamint az elem-(tartalmazás-) vizsgálat (l. később). A halmazokat elemeikből a halmazgenerálásnak nevezett művelettel hozhatjuk létre.

A Pascal gépi megvalósításai általában korlátozzák a halmazok méretét. Ez a korlát egész kicsiny is lehet (pl. az egy számítógépi szóban levő bitek száma), és közvetlenül behatárolja a halmaz alaptípusok elemszámát.

9.1. Halmazgenerátorok

A halmazokat elemeikből *halmazgenerátorok* segítségével állíthatjuk elő. A halmazgenerátor a halmazelemek felsorolásából áll. A listát szögletes zárójelek közé írjuk, az egyes elemeket vesszővel választjuk el. Az elemeket megadhatjuk kifejezés formájában – ekkor az elem nem más, mint a kifejezés értéke, de írhatunk a szögletes zárójelek közé AlsóHatár..FelsőHatár alakú tartományt is: ekkor az AlsóHatár, ill. a FelsőHatár kifejezés értéke rögzíti a halmaz értékkészletét. Ha AlsóHatár > FelsőHatár, a halmaznak egy eleme sincs.

A kifejezéseknek egyazon megszámlálható típushoz, a halmazgenerátor típus *alaptípusához* kell tartozniuk. A [ ] halmazgenerátor *minden* halmaztípus esetén az üres halmazt jelöli. A halmazgenerátorok nem hordoznak teljes típusinformációt [10], nem mondják meg például, hogy a halmaz tömörített-e vagy sem. Hogy a halmazkifejezésekben a más halmazokkal való kompatibilitást biztosítsuk, a halmazgenerátor típusát egyidejűleg tömörítettnek és kifejtettnek tekintjük.



9.2. ábra. *Halmazgenerátor* szintaxisdiagramja

Példák:

```
[13]
[i+j, i-j]
['0'..'9']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
'x', 'y', 'z']
```

9.2. Halmazműveletek

Ha X halmazváltozó és E halmazkifejezés, megengedett az

X := E

értékadás, feltéve, hogy E minden tagja X alaptípusához tartozik, és X, ill. E vagy egyaránt tömörített, vagy egyik sem az. A következő operátorok tetszőleges halmazstruktúrájú operandusra alkalmazhatók. Legyen A és B két azonos típusú halmazkifejezés. Ekkor:

A + B az A-hoz és B-hez tartozó elemek halmaza (a két halmaz egyesítése, uniója);
A * B A és B közös elemeinek halmaza (a két halmaz metszete);
A – B A azon elemeinek halmaza, amelyek nem tartoznak egyidejűleg B-hez (a két halmaz különbsége).

Halmazoperandusokra öt relációs operátor alkalmazható. Legyen A és B azonos típusú halmazkifejezés, e pedig az alaptípushoz tartozó, megszámlálható típusú kifejezés. Ekkor:

e in A elem- (tartalmazás-) vizsgálat. Eredménye True, ha e eleme A-nak, egyébként False.
A = B egyenlőségvizsgálat.
A <> B egyenlőtlenség-vizsgálat.
A <= B részhalmazvizsgálat. Eredménye True, ha A valódi vagy nem valódi részhalmaza B-nek.
A >= B részhalmazvizsgálat. Eredménye True, ha B valódi vagy nem valódi részhalmaza A-nak.

Példák deklarációra:

```
type  Alapszín = (Vörös, Sárga, Kék);
      Szín = set of Alapszín;
```

```
var   Árnyalat1, Árnyalat2: Szín;
      Magánhangzók, Mássalhangzók, Hangok: set of Char;
      Műveletikód: set of 0..7;
            Összeadás: Boolean;
            K: Char;
```

Példák értékadásra:

```
Árnyalat1  := [Vörös]; Árnyalat2 := [ ];
Árnyalat2  := Árnyalat2 + [Succ(Vörös)]
Hangok := ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q',
           'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'];
Magánhangzók := ['A', 'E', 'I', 'O', 'U'];
Mássalhangzók := Hangok – Magánhangzók
Összeadás := [2, 3] <= Műveletikód
```

A halmazműveletek viszonylag gyorsak, s más bonyolultabb vizsgálatok helyett jól alkalmazhatók. Az

```
if (Ch = 'A') or (Ch = 'E') or (Ch = 'I') or (Ch = 'O') or (Ch = 'U') then S
```

feltételvizsgálatot pl. az alábbi formára egyszerűsíthetjük:

```
if Ch in ['A', 'E', 'I', 'O', 'U'] then S
```

```
program EgesszeAlakitas(Input,Output);
{Turbo Pascal}
{9.1. program - Szamjegysor beolvasasa es az altaluk
     kifejezett egesz szamma alakitasa.
     Elojel nelkuli szamokat tetelezunk fel.}

var K:Char;
    Szamjegyek: set of '0'..'9';
    Szam:Integer;

begin Szamjegyek:=['0'..'9'] {halmaz inicializalasa};
      Read(Input,K);
      Szam:=0;
      while K in Szamjegyek do
            begin Szam:=Szam*10+Ord(K)-Ord('0');
                  Writeln(Output,Szam);
                  Read(Input,K)
            end
      {K az egesz szam utani karaktert tartalmazza}
end.
```

A program eredménye:

```
4
43
432
4321
```

```
program Halmazmuveletek(Output);
{Turbo Pascal}
{9.2. program - A halmazmuveletek bemutatasa}

type AHetNapjai = (H,K,Sze,Cs,p,Szo,V);
     Het = set of AHetNapjai;

var EgeszHet,Munka,Szabad: Het;
    Nap: AHetNapjai;

procedure Vizsgalat(X: Het);
{az eljarasokkal a 12. fejezetben foglalkozunk}
var N:AHetNapjai;
begin for N:=H to V do
          if N in X
          then Write(Output, 'X')
          else Write(Output, 'O');
      Writeln(Output)
end {vizsgalat};

begin Munka:=[]; Szabad:=[]; EgeszHet:=[H..V];
      Nap:=Szo;
      Szabad:=[Nap]+Szabad+[V];
      Vizsgalat(Szabad);
      Munka:=Egeszhet-Szabad;
      Vizsgalat(Munka);
      if Szabad<=EgeszHet
      then Write(Output, 'O');
      if EgeszHet>=Munka
      then Write(Output, 'K');
      if not (Munka>=Szabad)
      then Write(Output, 'Zsoke');
      if [Szo]<=Munka
      then Write(Output, 'Na ne');
      Writeln(Output)
end.
```

A program eredménye:

```
OOOOOXX
XXXXXOO
OKZsoke
```

9.3. A programfejlesztésről

A programozás – ha ezen algoritmusok és adatstruktúrák tervezését és megfogalmazását értjük – általában bonyolult folyamat, amely számos részletkérdés és speciális módszer biztos ismeretét követeli meg. Csak kivételes esetben fordul elő, hogy egy feladatnak csak egyetlen jó megoldása van. Többnyire annyi a megoldás, hogy az optimális program kiválasztásához a rendelkezésre álló algoritmusok és számítógépek beható tanulmányozása mellett még azt is részletesen meg kell vizsgálnunk, milyen módon használják majd leggyakrabban a kérdéses programot.

A program tehát sok megfontolás, vizsgálat és tervezői döntés eredményeként jöhet csak létre. A kezdeti fázisban legjobb, ha figyelmünket a globális problémákra összpontosítjuk, s a

megoldás első megfogalmazásában, vázlatában csak kevés figyelmet fordítunk a részletekre. A tervezési folyamat előrehaladtával a problémát részfeladatokra bonthatjuk, s fokozatosan tekintetbe vehetjük a feladatspecifikáció részleteit, a rendelkezésre álló eszközök jellemzőit. A probléma ilyen megközelítése szorosan kapcsolódik a *lépésenkénti finomítás* [2] és a *struktúrált programozás* [4] fogalmához.

Fejezetünk hátralevő részében egy algoritmus kidolgozásának menetét mutatjuk be: egy C. A. R. Hoare által a [4]-ben közölt példát „hangszereltünk át" Pascalra.

A feladat a 2..N tartományba eső prímszámok előállítása, ahol $N >= 2$. A különböző algoritmusok összehasonlítása után egyszerűsége miatt (mivel sem szorzást, sem osztást nem tartalmaz) az eratosztenészi szita mellett döntünk.

Először szóban fogalmazzuk meg a feladatot:

(1) Minden 2 és N közötti számot a szitába teszünk.
(2) Megkeressük és kivesszük a szitából a benne levő legkisebb számot.
(3) Ezt a számot a prímszámok közé írjuk.
(4) Kiszitáljuk ennek a számnak minden többszörösét.
(5) Ha a szita nem üres, a 2–5. lépést megismételjük.

Bár minden program végrehajtásának első lépése a változók kezdeti értékének beállítása (az inicializálás), a programfejlesztés során gyakran ezt hagyjuk utoljára. A helyes inicializálás előfeltétele, hogy pontosan ismerjük az algoritmus működését. Ha a programot módosítjuk, egyidejűleg a kezdeti értékeket is megfelelően át kell állítani, különben nem futhat a program. (Sajnos az átállítás önmagában nem mindig elegendő!)

Hoare mind a szita, mind a prímszámok ábrázolására 2..N elemű halmaz típust választ. A most következő szöveg – kis módosítástól eltekintve – megegyezik az általa közölt programvázlattal.

```
program Primszamok1;
{Turbo Pascal}
{9.3. program - Eratosztenesz szitajanak
                megvalositasa halmazokkal}
const N=255;
type PozitivSzam = 1..MaxInt;
var Szita, Primek: set of 2..N;
    KovPrim, Tobbsz: PozitivSzam;
begin {inicializalas}
      Szita:=[2..N]; Primek:=[];
      KovPrim:=2;
      repeat {a kovetkezo primszam keresese}
             while not (KovPrim in Szita) do
                   KovPrim:=Succ(KovPrim);
             Primek:=Primek+[KovPrim];
             Tobbsz:=KovPrim;
             while Tobbsz<= N do {kiejtes}
                   begin Szita:=Szita-[Tobbsz];
                         Tobbsz:=Tobbsz+KovPrim
                   end
      until Szita=[]
end.
```

Hoare gyakorlásképpen azt a feladatot adja az olvasónak, hogy írja át ezt a programot úgy, hogy a halmazok csak a páratlan számokat tartalmazzák. Az egyik lehetséges megoldás az alábbi. Figyeljük meg, milyen szoros a kapcsolat az első változattal!

```
program Primszamok2;
{Turbo Pascal}
{9.4. program - Eratosztenesz szitajanak megvalositasa
                halmazokkal, csak a paratlan szamok
                figyelembevetelevel}
const N=255 {N'=N div 2};
type PozitivSzam=1..MaxInt;
var Szita, Primek: set of 2..N;
    KovetkezoPrim, Tobbszoros, UjPrim:PozitivSzam;
begin {inicializalas}
      Szita:=[2..N]; Primek:=[];
      KovetkezoPrim:=2;
      repeat {a kovetkezo primszam keresese}
             while not (KovetkezoPrim in Szita) do
                   KovetkezoPrim:=Succ(KovetkezoPrim);
             Primek:=Primek+[KovetkezoPrim];
             UjPrim:=2*KovetkezoPrim-1;
             Tobbszoros:=KovetkezoPrim;
             while Tobbszoros<=N do {kifejtes}
                   begin Szita:=Szita-[Tobbszoros];
                         Tobbszoros:=Tobbszoros+UjPrim
                   end
      until Szita=[]
end.
```

A Pascal-megvalósításokkal szemben támasztott egyik tervezési követelmény, hogy a halmazokon végzett alapműveletek viszonylag gyorsak legyenek. Egyes gépi megvalósításokban a halmazok maximális méretét a számítógép szóhosszának megfelelően korlátozzák, s ily módon az alaphalmaz minden elemét egy bit reprezentálja (0 a hiányt, 1 a meglétet jelenti). A legtöbb megvalósítás nem engedne meg pl. 10 000 elemű halmazt. Ezek a megfontolások az adatábrázolás módosítását vonják maguk után (l. a 9.5. programot).

A nagy halmazokat kisebb halmazokból alkotott tömbökként ábrázolhatjuk. Az elemi halmazok méretét (a megvalósítástól függően) úgy választjuk meg, hogy egy halmaz éppen egy számítógépi szónak feleljen meg. A következő programban a második programvázlatot használtuk fel az algoritmus absztrakt modelljeként. A Szita és a Prímek halmazokból alkotott tömb, a KövetkezőPrím pedig rekord lesz.

```
program Primszamok3(Output);
{Turbo Pascal}
{9.5. program - A 3..10000 kozotti primszamok
     meghatarozasa  az ebbe a tartomanyba eso
     paratlan egeszeket tartalmazo szitaval}

const HMer = 128 {megvalositasfuggo};
      MaxElem = 127;
      HalmResz = 39 { =10000 div HMer div 2};

type TermeszetesSzam = 0..MaxInt;

var Szita, Primek: array [0..HalmResz] of
                   set of 0..MaxElem;
    KovetkezoPrim: record Resz, Elem: TermeszetesSzam
                   end;
    T, UjPrim: TermeszetesSzam;
    R, N, Szamlalo: TermeszetesSzam;
    Ures: Boolean;
```

```
begin {inicializalas}
      for R:=0 to HalmResz do
          begin Szita[R]:=[0..MaxElem];
                Primek[R]:=[]
          end;
      Szita[0]:=Szita[0]-[0]; Ures:=False;
      KovetkezoPrim.Resz:=0;KovetkezoPrim.Elem:=1;
      with KovetkezoPrim do
           repeat { a kovetkezo primszam keresese}
                  while not (Elem in Szita[Resz]) do
                  Elem:=Succ(Elem);
                  Primek[Resz]:=Primek[Resz]+[Elem];
                  UjPrim:=2*Elem+1;
                  T:=Elem; R:=Resz;
                  while R<=HalmResz do {kiejtes}
                        begin Szita[R]:=Szita[R]-[T];
                              R:=R+Resz*2;
                              T:=T+UjPrim;
                              while T>MaxElem do
                                    begin R:=R+1;
                                          T:=T-HMer
                                    end
                        end;
                  if Szita[Resz]=[]
                  then begin Ures:=True;
                             Elem:=0
                       end;
                  while Ures and (Resz<HalmResz) do
                       begin Resz:=Resz+1;
                             Ures:=Szita[Resz]=[]
                       end
          until Ures;
          Szamlalo:=0;
          for R:=0 to HalmResz do
              for N:=0 to MaxElem do
                  if N in Primek[R]
                  then begin Write(Output,
                                   2*N+1+R*HMer*2:6);
                             Szamlalo:=Szamlalo+1;
                             if (Szamlalo mod 8)=0
                             then Writeln(Output)
                       end
end.
```

A program eredménye:

```
     3     5     7    11    13    17    19    23
    29    31    37    41    43    47    53    59
    61    67    71    73    79    83    89    97
   101   103   107   109   113   127   131   137
     .
     .
     .
  9871  9883  9887  9901  9907  9923  9929  9931
  9941  9949  9967  9973 10007 10009 10037 10039
 10061 10067 10069 10079 10091 10093 10099 10103
 10111 10133 10139 10141 10151 10159 10163 10169
```

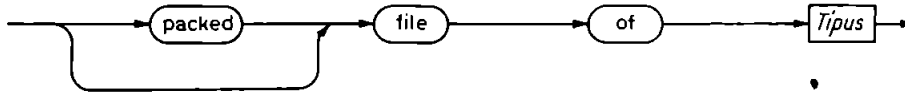
10. Állomány (file) típusok

Sok szempontból a legegyszerűbb strukturálási mód a sorbarendezés. Az adatfeldolgozás területén a sorozatot leggyakrabban egy *szekvenciális állománnyal* (sequential file) írják le. A Pascal egyszerűen a file alapszót használja az olyan adatszerkezet jelölésére, amely azonos típusú elemek sorozatából áll.

Az állományok különleges csoportját alkotják a *szöveg- (text) állományok*, amelyek különböző hosszúságú karaktersorok sorozatából állnak. Segítségükkel valósul meg az ember és a számítógéprendszerek közötti írásos kommunikáció.

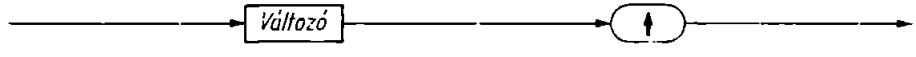
10.1. Az állományok szerkezete

A sorozat maga definiálja az elemek természetes rendezését. Mindig csak egyetlen elem lehet közvetlenül hozzáférhető. A többi elem az állománybeli sorrendben, az állományon végighaladva érhető el. Az elemek számára – az *állomány* ún. *hosszára* – az állomány típusdefiníciója nem tesz megkötést. Ez olyan tulajdonság, amely egyértelműen megkülönbözteti az állományt a tömbtől. Azt az állományt, amely egyetlen elemet sem tartalmaz, *üresnek* nevezzük. Az állomány típus tehát abban tér el a tömb, a rekord és a halmaz típustól, hogy soros elérésű, azonos típusú elemekből felépülő szerkezet.



10.1. ábra. *ÁllományTípus* szintaxisdiagramja

Minden F állományváltozó deklarálásakor automatikusan létrejön egy F↑ jelű, az elemek típusához tartozó ún. *pufferváltozó* is. Ezt olyan ablaknak tekinthetjük, amelyen át megvizsgálhatjuk, kiolvashatjuk a már meglevő elemek értékét, ill. amelyen keresztül az állományhoz új elemeket csatolhatunk (írhatunk). A pufferváltozót bizonyos állományműveletek automatikusan pozicionálják. A teljes állományra vonatkozó értékadás tiltott. Ehelyett a pufferváltozón keresztül egyenként írhatunk új elemeket az állományba. Ha az állomány utolsó elemén túllépünk, a pufferváltozó értéke határozatlanná válik.



10.2. ábra. *Pufferváltozó* szintaxisdiagramja

A soros feldolgozás, a változó hossz és a pufferváltozó jelenléte már jelzi, hogy az állományok a háttértárral és a perifériákkal hozhatók kapcsolatba. A megvalósítástól függ, pontosan hol és hogyan tároljuk az egyes elemeket, de feltételezzük, hogy az operatív tárban egyidejűleg csak néhány elem van, és csak az az elem érhető el közvetlenül, amelyre F↑ mutat.

Ha az F↑ ablak az F állomány végén (*end of file*) túlra kerül, az eof (F) standard logikai függvény True értéket szolgáltat; értéke egyébként False. Az alapvető állománykezelő eljárások:

Reset (F) az állományablakot olvasás céljából az állomány elejére állítja vissza. Ha F nemüres, F↑-nek F első elemének az értékét adja; eof (F) értéke False lesz.

Rewrite (F) F létrehozása (írása) előtt alkalmazzuk. F aktuális értéke az üres állomány lesz. Az eof (F) True lesz, és megkezdődhet az új állomány beírása.

Get (F) az állományablakot a következő elemre lépteti, azaz az F↑ pufferváltozónak ezen elem értékét adja. Ha nincs következő elem, eof (F) True értéket vesz fel, F↑ értéke pedig határozatlan lesz. A Get (F) hibát okoz, ha végrehajtása előtt eof (F) True értékű volt, vagy ha F írása közben alkalmazzuk.

Put (F) az F↑ pufferváltozó értékét az F állományhoz illeszti. Hibát okoz, ha végrehajtása előtt az eof (F) logikai függvény nem True értékű, vagy ha F olvasása közben alkalmazzuk. Az eof (F) True marad, és F↑ határozatlan lesz.

Elvileg a fenti négy alapvető állományoperátorral és az eof logikai függvénnyel az összes szekvenciális állományírási és -olvasási műveletet ki tudnánk fejezni. A gyakorlatban azonban az állománypozíció léptetése gyakran természetes módon együttjár a pufferváltozóval végzett műveletekkel. Ezért bevezetünk két új eljárást (Read és Write), amelyek definíciója a következő:

```
Read(F,X)   jelentése (X változó):
      begin
            X := F↑; Get(F)
      end

Write(F,E)  jelentése (E kifejezés):
      begin
            F↑ := E; Put(F)
      end
```

A Read és a Write olyan különleges eljárás, amely változó számú aktuális paramétert is elfogad. Ha V1,...,Vn változók és E1,...,En kifejezések, akkor:

```
Read(F,V1,...,Vn)  jelentése:
      begin Read(F,V1);...;Read(F,Vn) end
```

és

```
Write(F,E1,...,En)  jelentése:
      begin Write(F,E1);...;Write(F,En) end
```

A Read és a Write eljárás bevezetése a rövidebb írásmód mellett azért is előnyös, mert fogalmi egyszerűsítést is jelent, hiszen elhagyható az F↑ pufferváltozó, amelynek értéke bizonyos esetekben határozatlanná válik. A pufferváltozó azonban hasznos „előretekintő" (look-ahead) eszköz lehet.

Példák deklarációra:

```
var Adatok: file of Integer;
      A: Integer;
```

```
var Rajzállomány: file of
                    record
                        C: Szín;
                        Hossz: TermészetesSzám
                    end;
var Klub: file of Személy;
    Sz: Személy;
```

Példák állományokra vonatkozó utasításokra:

```
A := Adat↑; Get(Adatok)
Read(Adatok,A)
Rajzállomány↑.C := Vörös;
Rajzállomány↑.Hossz := 17; Put(Rajzállomány)
Klub↑ := Sz; Put(Klub)
Write(Klub,Sz)
```

Az állományok lehetnek valamely porgramra (eljárásra) nézve lokálisak, de létezhetnek a programtól függetlenül, azon kívül is. Az utóbbi esetben *külső (externa) állományokról* beszélünk. A külső állományok a programfej paramétereiként szerepelnek a programban (l. a 4. fejezetet).

A következő két program az állományok használatát szemlélteti. A 10.1. program (normalizálás) valós számokkal kifejezett mérési adatokból álló állományt dolgoz fel – az adatok közvetlenül a műszertől vagy egy másik programtól származhatnak. A 10.2. program (összefésülés) *két állományt fésül össze.* A két kiinduló állomány vezetéknév szerint rendezett személyi adatokat tartalmaz; ezekből a program egy hasonlóképpen rendezett harmadik állományt hoz létre. Formálisan, ha a kiinduló állományok elemei rendre:

F1,F2,...,Fm és G1,G2,...,Gn

ahol F(I + 1) >= F(I), ill. G(J + 1) >= G(J) minden I-re, ill. J-re, akkor az eredményként előálló H állományra is:

H(K + 1) >= H(K), ahol K = 1,2,...,(M + N−1).

```
program Normalizalas(K_Adatok,E_Adatok);
{MS Pascal}
{10.1. program - Meresi adatok normalizalasa. A valos
 szamokbol allo adatallomany a muszerrol vagy egy masik
 programbol szarmazik.}

type MeresiAdatok = file of real;
     TermeszetesSzam = 0..Maxint;
var K_Adatok, E_Adatok: MeresiAdatok;
    N_Osszeg, Szoras, Osszeg, K_Ertek: real;
    N: TermeszetesSzam;
begin Reset(K_Adatok); N:=0;
      Osszeg:=0.0; N_Osszeg:=0.0;
      while not Eof(K_Adatok) do
            begin N:=N+1;
                  Osszeg:=Osszeg+K_Adatok^;
                  N_Osszeg:=N_Osszeg+Sqr(K_Adatok^);
                  Get(K_Adatok)
            end;
```

```
        K_Ertek:=Osszeg/N;
        Szoras:=Sqrt((N_Osszeg/N)-Sqr(K_Ertek));
        Reset(K_Adatok); Rewrite(E_Adatok);
        while not Eof(K_Adatok) do
              begin E_Adatok^:=(K_Adatok^-K_Ertek)/Szoras;
                    Put(E_Adatok); Get(K_Adatok)
              end
end {Normalizalas}.


program Osszefesules(F, G, H);
{MS Pascal}
{10.2. program - Az F es G vezeteknevek szerint
                 rendezett allomanyok osszefesulese
                 a H allomanyba.}

type TermeszetesSzam = 0.. Maxint;
     Fuzer15 = packed array[1..15] of Char;
     szemely = record
               Nev: record
                    Keresztn, Vezetek: Fuzer15
                    end;
               Magassag: TermeszetesSzam {Centimeter}
               end;
var F, G, H: file of Szemely;
    FVagyGVege: Boolean;
begin Reset(F); Reset(G); Rewrite(H);
      FVagyGVege:=Eof(G) or Eof(G);
      while not FVagyGVege do
            begin if F^.Nev.Vezetek<G^.Nev.Vezetek
                  then begin H^:=F^; Get(F);
                             FVagyGVege:=Eof(F)
                       end
                  else begin H^:=G^; Get(G);
                             FVagyGVege:=Eof(G)
                       end;
                  Put(H)
            end;
      while not Eof(G) do
            begin Write(H, G^);
                  Get(G)
            end;
      while not Eof(F) do
            begin Write(H, F^);
                  Get(F)
            end
end.
```

10.2. Szövegállományok

Azokat az állományokat, amelyek változó hosszúságú, sorokra tagolt *karaktersorozatokból* állnak, *szövegállományoknak (textfile)* nevezzük. A szövegállományokat a Text (szöveg) standard típushoz tartozónak deklaráljuk.

Azt mondhatjuk, hogy a Text típust a (feltételezett) sorlezáró (end-of-line) karakterrel kibővített Char típuson mint alaptípuson definiáljuk. A Text típus tehát *nem* egyenértékű a (Packed) file of Char-ral. A sorlezáró karakter előállítása, ill. felismerése a következő speciális szövegeljárásokkal történik:

Writeln(F)	az F szövegállomány aktuális sorának lezárása.
Readln(F)	ugrás az F szövegállomány következő sorának elejére (F↑ a következő sor első karakterére mutat).
Eoln(F)	logikai függvény, amely azt jelzi, hogy az F szövegállomány aktuális sorának végére értünk. (Ha True értékű, akkor F↑ a sorlezáró – end-of-line – karakterre mutat és tartalma *üres*.)

Ha F szövegállomány és Ch karaktervváltozó, akkor az alábbi rövidebb írásmódot alkalmazhatjuk:

rövid alak:	teljes alak:
Write(F,Ch)	F↑:=Ch; Put(F)
Read(F,Ch)	Ch :=F↑, Get(F)

A Pascalban van két standard szövegállomány típusú változó: az Input és az Output. Ezekkel mint programparaméterek segítségével történik az írásos ember–gép kommunikáció. Az Input és az Output változóval, valamint a Read, a Write, a Readln és a Writeln további változataival a 13. fejezetben részletesen foglalkozunk.

A most következő programvázlatokban, a fenti írásmódot alkalmazva néhány tipikus szövegállomány-műveletet mutatunk be.

(1) Az Y szövegállomány írása. Tegyük fel, hogy P(C) minden végrehajtása során egy (következő) karaktert állít elő, s a C paraméternek ezt a karakterértéket adja. Az aktuális sor végéhez érve a B1 logikai változó True-ra állítódik, az egész szöveg végénél pedig a B2 változó lesz True értékű.

```
Rewrite(Y);
repeat
      repeat P(C); Write(Y,C)
      until B1;
      Writeln(Y)
until B2
```

(2) Az X szövegállomány olvasása. Tegyük fel, hogy Q(C) minden végrehajtása egy (következő) C karakter feldolgozását eredményezi. Az egyes sorok végére érve az R tevékenységet kell elvégeznünk.

```
Reset(X);
while not eof(X) do
      begin
            while not eoln(X) do
                  begin Read(X,C); Q(C)
                  end;
            R; Readln(X)
      end
```

(3) Az X szövegállomány átmásolása az Y szövegállományba, X sorszerkezetének megtartásával.

```
Reset(X); Rewrite(Y);
while not eof(X) do
      begin ⟨ egy sor átmásolása ⟩
            while not eoln(X) do
                  begin Read(X,C); Write(Y,C)
                  end;
            Readln(X); Writeln(Y)
      end
```

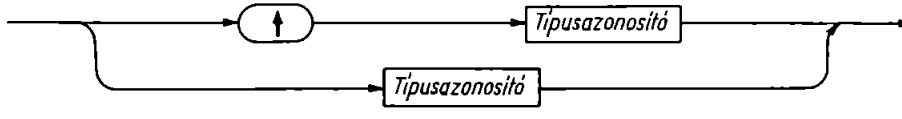
Végül egy megvalósítással kapcsolatos megjegyzés: A sorokat célszerűen vezérlő- (control) karakterekkel zárhatjuk le. Az ASCII karakterkészletben pl. a sor végét két karakterrel, a cr (carriage return : kocsi vissza) és az lf (line feed : soremelés) párossal jelöljük. Egyes számítógépek jelkészletéből azonban hiányoznak ezek a vezérlőkarakterek. Ilyenkor a sorvéget valamilyen más módon kell jelezni.

11. Mutató típusok

Mindeddig olyan típusokról volt szó, amelyek az ún. statikus változók deklarálására szolgáltak. *Statikusnak* mondjuk azt a változót, amelyet a programban deklarálunk, majd ezt követően azonosítójával jelöljük. Az elnevezés onnan ered, hogy a változó mindvégig „él" annak a blokknak (programnak, eljárásnak vagy függvénynek) a végrehajtása során, amelyre nézve lokális (más szóval a változó számára mindvégig helyet foglalunk a tárban). Változók azonban a program statikus szerkezetétől teljesen függetlenül *dinamikusan* – azaz egy blokk végrehajtása során – is előállíthatók. Az ilyen változót következésképpen *dinamikus (dynamic) vagy mutatott (identified) változónak* nevezzük.

11.1. Mutató- (pointer) és dinamikus (dynamic) változók

Ezek a változók nem szerepelnek explicit változódeklarációban, és azonosítókkal közvetlenül nem hivatkozhatunk rájuk: létrehozásukra, ill. törlésükre egy-egy standard eljárás, a New, ill. a Dispose szolgál. A dinamikus változókra mutató- *(pointer-)* értékekkel hivatkozhatunk. A mutató fizikailag nem más, mint az újonnan létrehozott változó tárcíme. A mutatóváltozók csak a megfelelő mutató típushoz tartozó értékeket vehetnek fel.



11.1. ábra. *MutatóTípus* szintaxisdiagramja

A mutató típust egy alaptípuson definiáljuk:

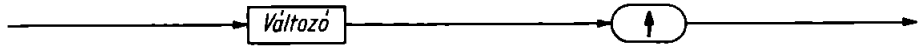
```
type P = ↑T;
```

Egy P mutató típushoz tehát tetszőleges számú érték tartozhat, de ezek az értékek mind egy adott T típus elemeire (T típusú változókra) mutatnak. Emellett mindig eleme P-nek a nil érték, amely semmilyen változóra sem mutat.

Egy dinamikus változót az őt megcímző mutatóértékkel érhetünk el. Ha Ptr-t

```
var Ptr:P;
```

-ként deklaráltunk és értékeket kapott, a dinamikus változót Ptr↑-lal jelöljük. Ha Ptr értéke nil vagy határozatlan, Ptr↑-at észlelve a program hibát jelez.



11.2. ábra. *MutatottVáltozó* szintaxisdiagramja

Ha T típusú dinamikus változóra van szükségünk, a New (Ptr) eljárást használjuk. Ez lefoglalja a tárhelyet, és Ptr-nek a változóra mutató értéket ad. A Ptr értéke által kijelölt változó törlésére (a tárhely felszabadítására) a Dispose (Ptr) eljárás szolgál. A Dispose után Ptr értéke határozatlan lesz.

Mutatók alkalmazásával egyszerűen képezhetünk összetett, rugalmas (sőt rekurzív) adatszerkezeteket. Ha a T típus rekordszerkezetű, és egy vagy több P típusú mezőt tartalmaz, akkor tetszőleges, véges gráfnak megfelelő struktúrát hozhatunk létre, amelyben a dinamikus (mutatott) változók a csomópontokat, a mutatók az éleket reprezentálják.

A 11.1. program sorban álló vásárlók kiszolgálását szimulálja, s a várakozók listájának kezelésén keresztül szemlélteti a mutatók használatát. (Az eljárásokkal a következő fejezetben foglalkozunk.)

```
program Sorbanallas(Input,Output);
{Turbo Pascal}
{11.1. program - Varakozolista szimulacioja;
       az elso harom vasarlo kiszolgalasa}

const Nevhossz = 15;

type NevIndex = 1..Nevhossz;
     NevFuzer = packed array [NevIndex] of Char;
     TermeszetesSzam=0..MaxInt;
     Vasarlomutato=^Vasarlo;
     Vasarlo=record Nev:NevFuzer;
                    Kovetkezo:Vasarlomutato
             end;
var Eleje,Vege:Vasarlomutato;
    Nev:packed array [NevIndex] of Char;

procedure Nevbeolvasas;
var c:NevIndex;
begin for c:=1 to Nevhossz do
           if Eoln(Input)
           then Nev[c]:=' '
           else begin Read(Input,Nev[c]);
                      Write(Output,Nev[c])
                end;
       Readln(Input);
       Writeln(Output)
end {Nevbeolvasas};

procedure UjVasarloListahozIrasa;
var UjVasarlo:Vasarlomutato;
begin New(UjVasarlo);
      if Eleje=nil
       then Eleje:=UjVasarlo
       else Vege^.Kovetkezo:=UjVasarlo;
       UjVasarlo^.Nev:=Nev;
       UjVasarlo^.Kovetkezo:=nil;
       Vege:=UjVasarlo
end {UjVasarloListahozirasa};
```

```
procedure Kiszolgalas(HanyVasarlot:TermeszetesSzam);
var Kiszolgalando:Vasarlomutato;
begin while (HanyVasarlot>0) and (Eleje <> nil) do
            begin Kiszolgalando:=Eleje;
                  Eleje:=Eleje^.Kovetkezo;
                  Writeln(Kiszolgalando^.Nev);
                  Dispose(Kiszolgalando);
                  HanyVasarlot:=HanyVasarlot-1
            end
end {Kiszolgalas};

begin {Sorbanallas}
      Eleje:=nil;
      while not Eof(Input) do
            begin Nevbeolvasas;
                  UjVasarloListahozirasa
            end;
      Writeln(Output);
      Kiszolgalas(3)
end.
```

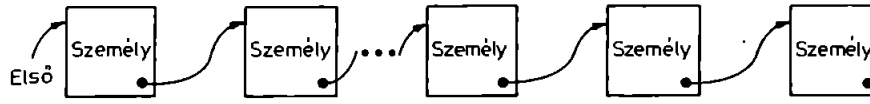
A program eredménye:

```
Peter
Maria
Kristof
Cecilia
Boldizsar
Jozsef
Eniko
Bendeguz

Peter
Maria
Kristof
```

Nézzünk egy másik példát! Tegyük fel, hogy egy adott embercsoportról „adatbankot" szeretnénk felállítani. Írjuk le a személyeket ugyanolyan rekordokkal, mint a 8. fejezetben! Ha mármost – mint azt a következő séma mutatja – mindegyik rekordba egy mutató típusú mezőt is beiktatunk, a rekordokból láncot vagy összeszerkesztett listát állíthatunk össze. Ehhez könnyen fűzhetünk újabb rekordokat, s egyszerű az egyes adatok visszakeresése is.

```
type Kapcsolat = ↑Személy;
     ...
     Személy = record
          ...
          Következő: Kapcsolat;
          ...
               end;
```

Az n személyből álló összeszerkesztett listát a 11.3. ábrával szemléltethetjük. Mindegyik négyzet egy-egy személyt jelképez.



11.3. ábra. Láncolt adatok listája

A lista első elemére az Első nevű, Kapcsolat típusú változó mutat. Az utolsó személy Következő mezője nil. Figyeljük meg, hogy az

Első↑.Következő↑.Következő

kifejezés a lista harmadik személyére mutat.

Ha feltesszük, hogy egész számokat – pl. testmagasságadatokat – olvasunk be, a fenti láncot a következő kóddal hozhatnánk létre:

```
var Első, P: Kapcsolat; Hl:Integer;
        ...
    Első := nil;
    for I: = 1 to N do
        begin Read (H); New(P);
            P↑. Következő := Első;
            P↑. Magasság := H; ATöbbiMezőInicializálása(P↑);
            Első := P
        end
```

Figyeljük meg, hogy a lista hátulról előrefelé nő! Az elemek elérését úgy biztosíthatjuk, hogy bevezetünk egy újabb Kapcsolat típusú változót – nevezzük ezt mondjuk Pt-nek –, amely szabadon mozoghat a listában. Hogyan érhetjük el ezzel a lánc valamelyik elemét? Példaként tegyük fel, hogy van a listán egy olyan Személy, akinek a Magasság-a 175, és ezt a személyt szeretnénk elérni (megkeresni). A stratégia a következő: Pt-t a Kapcsolatok-on át addig léptetjük, míg a keresett listaelemet meg nem találjuk.

```
Pt := Első;
while Pt↑.Magasság <> 175 do Pt := Pt↑.Következő
```

Ez a programrészlet a következőt jelenti: „Mutasson Pt az első Személy-re. Ha a vizsgált Személy Magasság-a nem 175, adjuk Pt-nek az általa éppen kijelölt rekord Következő mezőjében (ebben a másik mutató változóban) tárolt értéket! Folytassuk ezt mindaddig, míg Pt arra a Személy-re nem mutat, akinek a Magasság-a éppen 175."

Vegyük észre, hogy ez az egyszerű keresőutasítás csak akkor működik, ha biztosak vagyunk benne, hogy van legalább egy olyan személy a listán, akinek a Magasság-a 175. De jogos-e ez a feltevés? Ha nem, feltétlenül meg kell vizsgálnunk, nem értük-e már el a lista végét! Elsőként a következő megoldással próbálkozhatnánk:

```
Pt := Első;
while (Pt <> nil) and (Pt↑.Magasság <> 175) do
    Pt := Pt↑.Következő
```

Idézzük fel az 5.1. szakaszban mondottakat! Ha Pt = nil, akkor a Pt↑ változó, amelyre a leállási feltétel második tényezőjében hivatkozunk, egyáltalán nem is létezik, programunk tehát hibás. Így erre a helyzetre pl. az alábbi két programrészlet adhat helyes megoldást:

```
(1) Pt := Első; B := true;
    while (Pt<>nil) and B do
    if Pt↑.Magasság = 175 then B := false else Pt := Pt↑.Következő

(2) Pt := Első;
    while Pt<>nil do
        begin if Pt↑.Magasság = 175 then goto 13;
              Pt := Pt↑.Következő
        end;
        13
```

11.2. A New és a Dispose eljárás

Vessünk fel most egy másik feladatot, és nézzük meg, mi a teendő, ha mondjuk egy újabb személy adatait szeretnénk felvenni a listára! Ilyenkor először létre kell hoznunk egy új (dinamikus) változót: le kell foglalnunk a szükséges tárterületet, s elő kell állítanunk a rá mutató értéket. Ez a New standard eljárással történik.

New (P)	helyet foglal egy új P↑ dinamikus változónak (P↑ típusa P alaptípusa lesz) létrehoz egy új – P-ével egyező típusú – mutatóértéket, s azt P-hez rendeli. Ha P↑ egy változatrekord-típusú változó, akkor New (P) az összes változat számára elegendő helyet biztosít.
New (P,C1,...,Cn)	helyet foglal egy új, P-ével egyező változatrekord-típusú P↑ dinamikus változónak (a rekord n egymásba ágyazott változat részt tartalmaz, az ezekhez tartozó kijelölőmező értékek rendre C1,...,Cn), létrehoz egy új – P-ével azonos típusú – mutatóértéket, s azt P-hez rendeli.

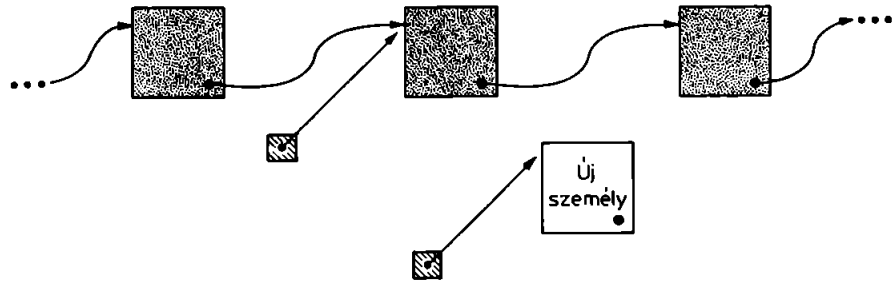
Figyelmeztetés! Ha valamely P↑ rekordváltozót a New eljárás második alakjával állítunk elő, akkor ennek a változónak a programvégrehajtás során mindvégig azonos változatban kell szerepelnie. A teljes változóra vonatkozó értékadás tiltott, P↑ elemeire azonban megengedett!

A felvetett feladat megoldásának programozásakor az első lépés egy mutatóváltozó bevezetése. Nevezzük ezt NewP-nek. Ekkor a

```
New(NewP)
```

utasítással helyet foglalunk egy új Személy típusú változónak.

A következő lépésben az új változót – amelyre NewP mutat – be kell iktatnunk a láncba, mégpedig a Pt által kijelölt listaelem utáni helyre (11.4. ábra).

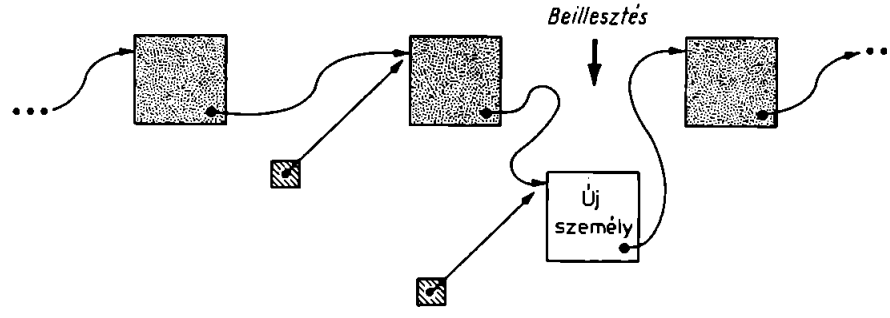


11.4. ábra. Láncolt adatok listája beillesztés előtt

A beiktatás egyszerűen a mutatók átállításából áll:

```
NewP↑.Következő := Pt↑.Következő;
Pt↑.Következő := NewP
```

Az eredményt a 11.5. ábra szemlélteti.



11.5. ábra. Láncolt adatok listája beillesztés után

A Pt segédmutatót követő elemet az alábbi utasítással törölhetjük (hagyhatjuk el):

```
Pt↑.Következő := Pt↑.Következő↑.Következő
```

Gyakran célszerű egy listát két, egymást követő mutató segítségével feldolgozni. A törlést ilyenkor úgy érdemes megoldani, hogy az egyik, mondjuk P1 jelű mutató az elhagyandó listaelem előtti pozícióra, a másik, P2 mutató magára az elemre mutat. Ekkor a törlést az alábbi utasítással írhatjuk le:

```
P1↑.Következő := P2↑.Következő
```

Fel kell hívnunk a figyelmet arra, hogy az így megoldott törlés bizonyos esetekben csökkenti a felhasználható (szabad) tárterületet! Ezen a helyzeten például úgy segíthetünk, hogy pontosan nyilvántartjuk a „kitörölt" (elhagyott) listaelemeket. Az új változókat ekkor nem a New eljárás hívásával hozzuk létre, hanem ebből a – mondjuk Szabad nevű mutatóváltozó segítségével vezetett – nyilvántartásból vesszük (kivéve persze, ha a nyilvántartás üres). Így egy listaelem elhagyása azt jelenti, hogy a kérdéses elemet kiiktatjuk a láncból (listából), és a szabad elemek nyilvántartásába tesszük.

```
P1↑.Következő :=*P2↑.Következő;
P2↑.Következő := Szabad;
Szabad := P2
```

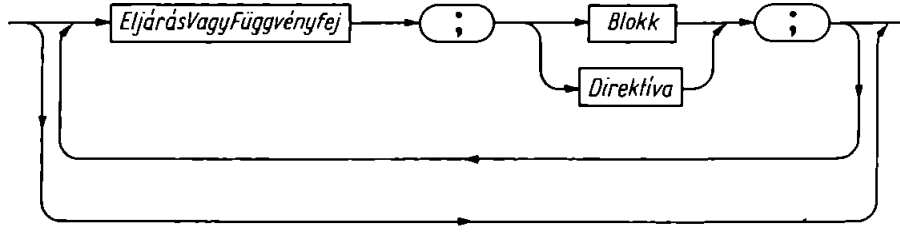
Végül a Dispose standard eljárás használatával a törölt tagok kezelését a gépi megvalósításra bízhatjuk.

Dispose(Q)	felszabadítja a Q↑ dinamikus változó által elfoglalt területet, és törli a Q mutatóértéket. Hívása hibát okoz, ha Q nil, vagy határozatlan. Csak akkor használható, ha Q értékét a New eljárás első alakjával hoztuk létre.
Dispose(Q, K1,...,Kn)	felszabadítja a Q↑ dinamikus változó rekord (amelynek aktív változatait K1,...,Kn jelöli ki) által elfoglalt területet, és törli a Q mutatóértéket. Hibát okoz, ha Q nil vagy határozatlan. Csak akkor használható, ha Q értékét a New eljárás második alakjával hoztuk létre, és K1,...,Kn ugyanazokat a változatokat jelöli ki, mint Q létrehozásakor.

A 12. fejezetben két olyan példaprogramot közlünk, amelyben a feladat egy mutató típusok segítségével felépített fastruktúra bejárása (12.6. és 12.7. program).

12. Eljárások és függvények

Amikor valaki már magas fokon elsajátította a számítógép-programozás mesterfogásait, programját egymást követő *finomítási lépésekben* készíti el. Minden egyes szinten, mindegyik lépésben részfeladatokra bontja a problémát, s ezáltal egy sor részprogramot hoz létre. Meg lehet úgy is írni a programot, hogy ez a tagoltság rejtett maradjon, ez azonban nem kívánatos. Az *eljárás* és a *függvény* fogalom lehetővé teszi, hogy a részfeladatokat ténylegesen részprogramokként írjuk meg.



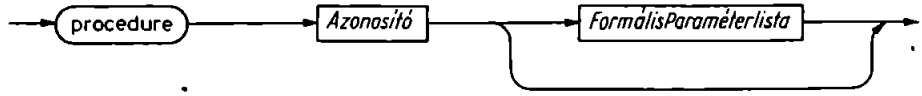
12.1. ábra. *EljárásÉsFüggvénydeklarációsRész* szintaxisdiagramja

12.2. ábra. *EljárásVagyFüggvényfej* szintaxisdiagramja

12.1. Eljárások

Könyvünk példaprogramjaiban sokszor találkozhatott az Olvasó a Read, a Readln, a Write és a Writeln standard eljárással. Ebben a szakaszban bemutatjuk, hogyan dolgozhatunk magunk deklarálta (tehát nem standard) eljárásokkal. (Valójában erre már láthattunk példát a 9.2. és a 11.1. programban.)

Az *eljárásdeklaráció* egy részprogram definiálására, azonosítóval való ellátására szolgál. Ez utóbbi révén a részprogram *eljárásutasítással* hívható. A deklaráció pontosan olyan alakú mint egy program, csak nem programfej, hanem eljárásfej vezeti be.



12.3. ábra. *Eljárásfej* szintaxisdiagramja

Példaként tekintsük ismét a 7.1. programot, amely egész számok listájából a legkisebb és a legnagyobb értéket kereste ki! Egészítsük ki a programot azzal, hogy A[1]...A[n]-et rendre megnöveljük valamilyen növekménnyel, majd ismét megkeressük a legkisebb (Min) és a legnagyobb (Max) értéket! A feladatot megoldó következő programban Min-t és Max-ot egy eljárás segítségével határozzuk meg.

```
program MinMax2 (Input,Output);
{Turbo Pascal}
{12.1. program - A 7.1 program eljarast alkalmazo
      valtozata}

const MaxMeret = 12;

type Listameret = 1..MaxMeret;

var Novekmeny: Integer;
    Elem: ListaMeret;
    A: array [Listameret] of Integer;

procedure MinMax;
var Elem: Listameret;
    Min,Max,Elso,Masodik: Integer;
begin Min:=A[1];
      Max:=Min;
      Elem:=2;
      while Elem<MaxMeret do
            begin Elso:=A[Elem];
                  Masodik:=A[Elem+1];
                  if Elso>Masodik
                  then begin if Elso>Max
                             then Max:=Elso;
                             if Masodik<Min
                             then Min:=Masodik
                       end
                  else begin if Masodik>Max
                             then Max:=Masodik;
                             if Elso<Min
                             then Min:=Elso
                       end;
                  Elem:=Elem+2
            end;
      if Elem=MaxMeret
      then if A[MaxMeret]>Max
           then Max:=A[MaxMeret]
           else if A[MaxMeret]<Min
                then Min:=A[MaxMeret];
           Writeln(Output,Max:4,Min:4);
           Writeln(Output)
end {MinMax};
```

```
begin for Elem:=1 to MaxMeret do
          begin Read(Input,A[Elem]);
                Write(Output,A[Elem]:4)
          end;
      Writeln(Output);
      MinMax;
      for Elem:=1 to MaxMeret do
          begin Read(Input,Novekmeny);
                A[Elem]:=A[Elem]+Novekmeny;
                Write(Output,A[Elem]:4)
          end;
      Writeln(Output);
      MinMax
end.
```

A program eredménye:

```
 2  -4   6  -8  10 -12  14 -16  18 -20  22 -24
22 -24

14   7  16   1  18  -5  20 -11  22 -17  24 -23
24 -23
```

Egyszerűsége ellenére ez a program sok fogalom alkalmazását szemlélteti. Ezek:

(1) Az *eljárásfej* legegyszerűbb alakja, nevezetesen:

procedure Azonosító;

(2) *Blokkok.* Az eljárás nem más, mint egy névvel ellátott blokk. Esetünkben a programblokk neve MinMax2, az eljárásblokké pedig MinMax. Mint látható, a 7.1. programnak azt a részét, amely kizárólag a legkisebb és a legnagyobb érték keresésére szolgált, leválasztottuk és külön nevet adtunk neki. Ez a MinMax. A programblokkhoz hasonlóan az eljárást alkotó blokknak is van deklarációs része: itt adjuk meg az eljárásra nézve lokális objektumokat.

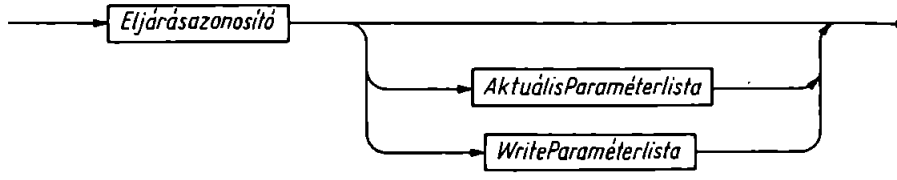
(3) *Lokális változók.* A MinMax eljárásra nézve lokális változók Elem, az Első, a Második, a Min és a Max. A MinMax hatáskörén kívül az ezen változókra vonatkozó értékadásoknak semmiféle hatásuk nincs. Az eljárás hívásakor az utasításrész elején a lokális változók értéke még határozatlan.

(4) *Globális változók.* Az A, az Elem és a Növekmény a főprogramban deklarált globális változók. A programban mindenütt hivatkozhatunk rájuk. (A MinMax első értékadása pl. Min := A[1].)

(5) *Hatáskör.* Figyeljük meg, hogy a programban egy Elem elnevezésű globális és egy ugyanolyan nevű lokális változó is van. Ez a két változó nem azonos egymással. Az eljárás hivatkozhat bármely, számára nem lokális változóra, de lehetőség van a nevek újradefiniálására is. Ha egy nevet újradefiniálunk, a definiáló eljárás hatáskörében az új név-, ill. típus-hozzárendelés lesz érvényben, s a kérdéses nevet viselő globális változó (hacsak paraméterként át nem adtuk) az eljárás hatáskörében nem lesz elérhető. A lokális Elem-re vonatkozó értékadásnak (pl. Elem := Elem + 2) a globális Elem-re semmiféle hatása sincs; és minthogy MinMax-on belül a lokális Elem-nek van elsőbbsége (precedenciája), az Elem globális változót semmiképpen sem érhetjük el. Általában célszerű minden olyan azonosítót, amelyre egy eljáráson kívül máshol nincs hivatkozás, szigorúan az illető eljárásra nézve lokális azonosítóként deklarálni.

Ez nemcsak a dokumentációt segíti elő, hanem nagyobb biztonságot is ad. Az előbbiekben Elem pl. maradhatott volna globális változó, de ebben az esetben, ha a programot egy olyan Elem-mel vezérelt ciklussal egészítenénk ki, amelyben MinMax-ot is hívjuk, hibás eredményt kapnánk.

(6) *Eljárásutasítás.* Példánkban a főprogramban a MinMax utasítás hívja be az eljárást.



12.4. ábra. *ProcedureUtasítás* szintaxisdiagramja

Ha a 12.1. programot részletesen megvizsgáljuk, kitűnik, hogy MinMax-ot kétszer hívtuk. Azzal, hogy a részprogramot eljárásként írtuk meg, azaz nem írtuk le kétszer ezt a programrészletet, nemcsak gépelési időt takarítottunk meg, hanem tárterületet is. A statikus kódot csak egyszer tároljuk, és a lokális változók területe csak az eljárás végrehajtása során, dinamikusan aktiválódik (a rendszer az eljárásba belépve lefoglalja, majd kilépéskor felszabadítja a tárhelyeket).

Azonban ha ezzel olvashatóbbá válik a program, úgy feltétlenül érdemes eljárásként megírni az olyan programrészeket is, amelyek hívására csak egyszer kerül sor. A rövidebb blokkok mindig áttekinthetőbbek, mint a hosszúak. Érthetőbb és könnyebben ellenőrizhető az a program, amelynek fejlesztési lépéseit eljárások formájában fogalmazzuk meg.

12.1.1. Paraméterlisták

Amikor alprogramokra bontunk fel egy feladatot, gyakran van szükség olyan új változók bevezetésére, amelyek az alprogramok argumentumait, ill. eredményeit tartalmazzák. A programszövegből világosan ki kell derülnie annak, hogy milyen célt szolgálnak ezek a változók.

A 12.2. program az előbbi példa kiterjesztése: szintén egy tömb legkisebb és legnagyobb értékét határozza meg, de az előbbinél általánosabb értelemben, s ezáltal számos újabb, eljárásokkal kapcsolatos fogalom jelentését is bemutatja.

```
program MinMax3 (Input,Output);
{Turbo Pascal}
{12.2. program - A 12.1 program ketlistas valtozata}

const MaxMeret = 12;

type Listameret = 1..MaxMeret;
     Lista = array [Listameret] of Integer;

var Elem: Listameret;
    A, B: Lista;
    MinA, MinB, MaxA, MaxB: Integer;

procedure MinMax(var L: Lista; var Min,Max: Integer);
var Elem: Listameret;
    Elso, Masodik: Integer;
```

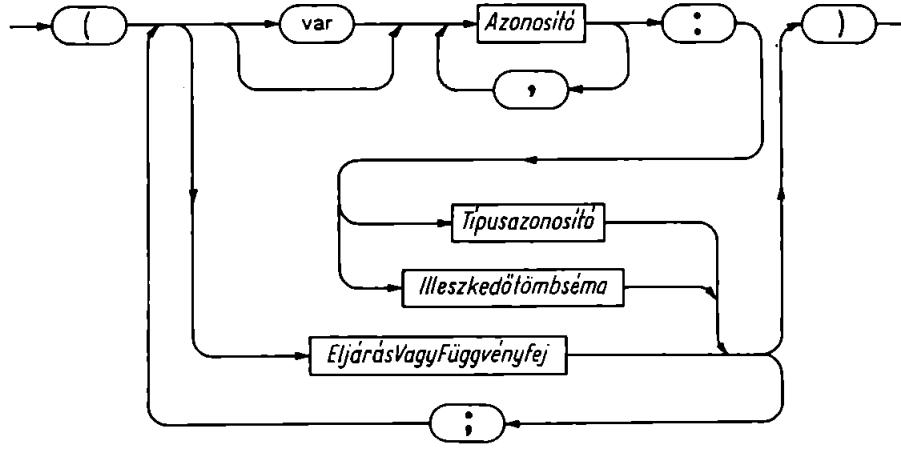
```
begin Min:=L[1];
      Max:=Min;
      Elem:=2;
      while Elem<MaxMeret do
            begin Elso:=L[Elem];
                  Masodik:=L[Elem+1];
                  if Elso>Masodik
                  then begin if Elso>Max
                             then Max:=Elso;
                             if Masodik<Min
                             then Min:=Masodik
                       end
                  else begin if Masodik>Max
                             then Max:=Masodik;
                             if Elso<Min
                             then Min:=Elso
                       end;
                  Elem:=Elem+2
            end;
      if Elem=MaxMeret
      then if L[MaxMeret]>Max
           then Max:=L[MaxMeret]
           else if L[MaxMeret]<Min
                then Min:=L[MaxMeret]
end {MinMax};

procedure ReadWrite(var L:Lista);
begin for Elem:=1 to MaxMeret do
          begin Read(Input,L[Elem]);
                Write(Output,L[Elem]:4)
          end;
      Write(Output)
end {ReadWrite};

begin {foprogram}
      ReadWrite(A);
      MinMax(A,MinA,MaxA); Writeln(Output);
      Writeln(Output,MinA:4,MaxA:4,MaxA-MinA:4);
      Writeln(Output);
      ReadWrite(B);
      MinMax(b,MinB,MaxB); Writeln(Output);
      Writeln(Output,MinB:4,MaxB:4,MaxB-MinB:4);
      Writeln(Output);
      Writeln(Output);
      for Elem:=1 to MaxMeret do
          begin A[Elem]:=A[Elem]+B[Elem];
                Write(Output,A[Elem]:4)
          end;
      Writeln(Output);
      MinMax(A,MinA,MaxA);
      Writeln(Output,MinA:4,MaxA:4,MaxA-MinA:4)
end.
```

A program eredménye:

```
  1   -2    3   -4    5   -6    7   -8    9  -10   11  -12
-12   11   23
  2    4    6    8   10   12   14   16   18   20   22   24
  2   24   22
  3    2    9    4   15    6   21    8   27   10   33   12
  2   33   31
```

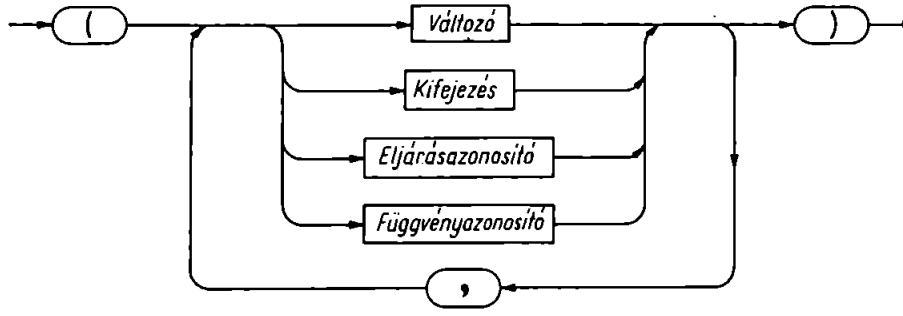
(1) Az *eljárásfej* másik alakjával találkozunk: most a fej paraméterlistát is tartalmaz.

(2) *Formális paraméterek:* A paraméterlista felsorolja valamennyi formális paraméter nevét és típusát. A MinMax formális paraméterei: L, Min és Max. A formális paraméterlista nyitja meg a paraméterek új hatáskörét.



12.5. ábra. *FormálisParaméterlista* szintaxisdiagramja

(3) Aktuális paraméterek. Figyeljük meg, hogy az eljárásfej és az eljárásutasítás között megfeleltetés van! Az utóbbi ugyanis tartalmazza azon aktuális paraméterek listáját, amelyekkel az eljárásdeklarációban definiált megfelelő formális paramétereket helyettesítjük. A megfeleltetés a paramétereknek az aktuális, ill. a formális paraméterlistában elfoglalt pozíciója alapján történik. A paraméterek olyan helyettesítési mechanizmust biztosítanak, amely lehetővé teszi, hogy egyazon műveletsort különböző argumentumokkal megismételhessünk. (Így pl. a MinMax-ot kétszer hívjuk: egyszer az A, egyszer a B tömböt vizsgáljuk.) Négyféle paramétert különböztetünk meg: érték-, változó-, eljárás- és függvényparamétert. Az utóbbi két paramétertípussal a 12.1.4., ill. a 12.2.1. pontban foglalkozunk.



12.6. ábra. *AktuálisParaméterlista* szintaxisdiagramja

(4) *Változóparaméterek.* A MinMax eljárásban ilyenekre láthatunk példát. Az aktuális paraméter *változó* kell, hogy legyen; a megfelelő formális paraméter előtt kötelező a var szimbólum. Az aktuális változót az eljárás végrehajtása során mindvégig a hozzá tartozó formális paraméter reprezentálja. Minden, a formális paraméterre kijelölt művelet tehát közvetlenül az aktuális paraméterrel hajtódik végre. Ha egy paraméter (mint a 12.2. programban Min és Max) az eljárás eredményét szolgáltatja, változóparaméterként definiáljuk. Továbbá, ha a V1,...,Vn formális változóparaméterekhez tartozó aktuális változók X1,...,Xn, akkor X1,...,Xn mind *különböző* változó kell, hogy legyen. Az összes címszámítás az eljáráshíváskor történik. Ha tehát egy változó egy tömb egyik eleme, indexkifejezése csak az eljárás hívásakor értékelődik ki. Megjegyezzük, hogy éppen a címszámítással kapcsolatos megvalósítási problémák elkerülése érdekében tömörített szerkezet eleme vagy változó rekord kijelölőmezője aktuális változóparaméterként nem szerepelhet.

Ha a paraméterrész előtt nem áll szimbólum, akkor a benne szereplő paraméter(eke)t *értékparaméter(ek)nek* nevezzük. Ilyenkor az aktuális paraméter szükségképpen egy *kifejezés* (ami a legegyszerűbb esetben egyetlen változó is lehet). A megfelelő formális paraméter a meghívott eljárás valamely lokális változója. Ennek a változónak a kezdeti értékét a hozzá tartozó aktuális paraméter pillanatnyi értéke (tehát a kifejezésnek az eljáráshívás pillanatában felvett értéke) adja. Az eljárás ezután értékadással megváltoztathatja ennek a változónak az értékét; eközben azonban az aktuális paraméter értéke nem változhat. Éppen ezért számítási eredmény sosem lehet értékparaméter. *Vigyázat!* Állományok vagy állományokból felépülő strukturált változók nem adhatók meg aktuális értékparaméterként, hiszen ez értékadást jelentene!

A 12.3. program azt mutatja, miben különbözik az érték- és a változóparaméter hatása.

```
program Parameterek(Output);
{Turbo Pascal}
{12.3. program - Ertek- es valtozoparameterek}

var A,B: Integer;

procedure Osszeadas1 (X: Integer; var Y: Integer);
begin X:=X+1;
      Y:=Y+1;
      Writeln(Output,X:4,Y:4)
end {Osszeadas};
```

```
begin A:=0;
      B:=0;
      Osszeadas1(A,B);
      Writeln(Output,A:4,B:4)
end (Parameterek).
```

A program eredménye:

```
1    1
0    1
```

A következő táblázatban pontosan összefoglaltuk, hogy a különböző fajtájú paraméterek hogyan jelennek meg a formális, ill. az aktuális paraméterlistában.

	formális paraméter	**aktuális paraméter**
értékparaméter	változóazonosító	kifejezés
változóparaméter	változóazonosító	változó
eljárásparaméter	eljárásfej	eljárásazonosító
függvényparaméter	függvényfej	függvényazonosító

A 12.2. program MinMax eljárásában az L tömb egyik elemének értéke sem változott: L nem volt számítási eredmény. Következésképpen ugyanarra az eredményre jutottunk volna, ha L-t értékparaméterként definiáljuk. Hogy megértsük, miért nem ezt a megoldást választottuk, foglalkozzunk egy kicsit a gépi megvalósítással!

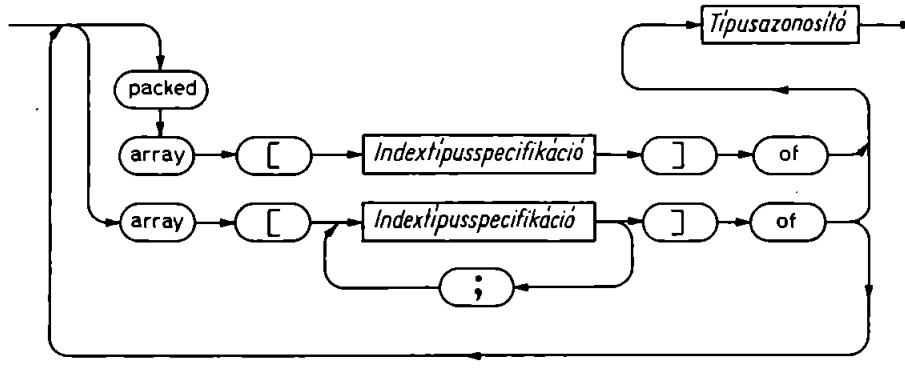
Az *eljáráshívás* valamennyi értékparaméternek új helyet foglal a tárban; ez reprezentálja a lokális változót. Az aktuális paraméter pillanatnyi értéke erre a helyre „másolódik át"; ha kilépünk az eljárásból, ez a tárfelület felszabadul.

Olyan paraméter esetén, amely nem az eljárás eredményét adja át, általában célszerűbb értékparaméterrel dolgozni. Gyorsabb a hozzáférés, és védve vagyunk a téves adatmódosítás veszélyétől. Körültekintéssel kell azonban kezelnünk azt az esetet, amikor valamilyen strukturált (pl. tömb) típusú paraméterrel van dolgunk. Ilyenkor ugyanis az átmásolási művelet meglehetősen időigényes, az átmásolt értékek tárolására lefoglalt terület pedig elég nagy lehet. Példánkban, minthogy az L tömb egyes elemeire csak egyszer hivatkozunk, változóparamétert célszerű használni.

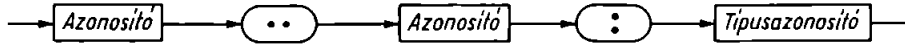
A tömb dimenziószámát egyszerűen MaxMéret újradefiniálásával változtathatjuk meg. Ha a programot valós értékekből álló tömbre akarjuk alkalmazni, csak a típus- és változódefiníciókat kell módosítanunk; az utasítások nem egész értékekkel is helyesen működnek.

12.1.2. Illeszkedőtömb-paraméterek

Van azonban egy másik mód is a különböző méretű tömbök eljárásoknak vagy függvényeknek való átadására: ha – változó- vagy értékparaméterként – illeszkedőtömb-paramétert alkalmazunk a formális paraméterlistában. Vigyázat! Az illeszkedőtömb-paraméterek az ISO Pascal-szabványban csak opcionális lehetőségként szerepelnek; egyes megvalósítások nem támogatják őket!



12.7. ábra. *Illeszkedőtömbséma* szintaxisdiagramja



12.8. ábra. *Indextípusspecifikáció* szintaxisdiagramja

Az illeszkedőtömb a tömb aktuális méreteit (az egyes dimenziók indexhatárát) indexhatár-azonosítókkal adja meg. Ezek tulajdonképpen csak olvasható (read-only) változók. Az aktuális tömbparaméter indextípusa kompatibilis kell, hogy legyen az illeszkedőtömb indextípus-specifikációjában megadott típussal. Az indextípus legkisebb és legnagyobb értékének is belül kell maradnia az indextípus-specifikációban álló típus zárt intervallumán. Az elemtípusoknak meg kell egyezniük; ha pedig az illeszkedőtömb-paraméter elemtípusa egy másik illeszkedőtömb-paraméter, akkor az aktuális tömbparaméter elemtípusának ehhez illeszkedőnek kell lennie.

Illeszkedőtömb-paraméter csak az utolsó dimenziójában tömöríthető. Az értékparaméterként használt illeszkedőtömb-paraméterekhez az aktuális paraméterlistában változók vagy füzérek tartozhatnak.

A most következő 12.4. program a 7. fejezetben látott *MátrixSzorzás* program illeszkedőtömb-paramétert alkalmazó változata. A későbbi 12.7. programban pedig különböző hosszúságú füzéreket adunk át egy formális illeszkedőtömb-paraméternek.

```
program MatrixSzorzas2(Input,Output);

{12.4. program - A 7.3. program modositott valtozata,
      illeszkedotomb-parametert hasznalo eljarassal}

{*** Az illeszkedotomb-parameterek itt kozolt modon
     valo kezeleset  sem a Turbo, sem az MS  Pascal
     nem fogadja el, ezert a program futasi eredme-
     nyeit nem kozoljuk.
 ***}

const M = 4;
      P = 3;
      N = 2;

type Poz = 1..MaxInt;

var A: array [1..M, 1..P] of Integer;
    B: array [1..P, 1..N] of Integer;
    C: array [1..M, 1..N] of Integer;
```

```
procedure MatrixBe(var X: array[KezdS..VegS:Poz;
                           KezdO..VegO:Poz] of Integer);
var Sor, Oszlop: Poz;
begin for Sor:=1 to VegS do
          for Oszlop:=1 to VegO do
              Read(Input, X[Sor,Oszlop])
end {MatrixBe};

procedure Matrixkiiras(var X: array[KezdS..VegS:Poz;
                               KezdO..VegO:Poz] of Integer);
var Sor,Oszlop:Poz;
begin for Sor:=1 to VegS do
      begin for Oszlop:=1 to VegO do
            Write(Output, X[Sor,Oszlop]);
            Writeln(Output)
      end
end {Matrixkiiras};

procedure Szorzas (var A: array[AKezdS..Avegsor:Poz;
                           AKezdO..AVegO:Poz] of Integer;
                   var B: array[BKezdS..BVegS:Poz;
                           BKezdO..BVegO:Poz] of Integer;
                   var C: array[CKezdS..CVegS:Poz;
                           CKezdO..CVegO:Poz] of Integer);

var S: Integer;
    I, J, K: Poz;

begin if (AKezdS<>1) or (AKezdO<>1) or (BKezdS<>1) or
         (BKezdO<>1) or (CKezdS<>1) or (CKezdO<>1)
         or (AVegS<>CVegS) or (AVegO<>BVegS) or
         (BVegO<>CVegO)
      then {hiba}
      else for I:=1 to CVegS do
                begin for J:=1 to CVegO do
                          begin S:=0;
                                for K:=1 to AVegO do
                                    S:=S+A[I,K]*B[K,J];
                                    C[I,J]:=S
                          end
                end
end {Szorzas};

begin MatrixBe(A);
      Matrixkiiras(A);
      MatrixBe(B);
      Matrixkiiras(B);
      Szorzas(A,B,C);
      Matrixkiiras(C)
end.
```

A program eredménye:

```
 1        2        3
-2        0        2
 1        0        1
-1        2       -3

-1        3
-2        2
 2        1

 1       10
 6       -4
 1        4
-9       -2
```

12.1.3. Rekurzív eljárások

Ha az eljárás azonosítóját az eljáráson belül használjuk, az eljárást rekurzív módon hajtjuk végre. Vannak feladatok, amelyek megfogalmazása a probléma természetéből adódóan rekurzív, s ezek gyakran szinte kínálják a rekurzív megoldást. Ilyen esetet példáz a 12.5. program.

Az a feladat, hogy írjunk olyan programot, amely a hagyományos (infix) alakban megadott kifejezéseket lengyel (postfix) alakra hozza. Ezt úgy oldjuk meg, hogy mindegyik szintaktikai egységre (kifejezésre, tagra, tényezőre) külön-külön egy-egy konverziós eljárást készítünk. Minthogy ezeket a szintaktikai egységeket rekurzív definícióval adjuk meg, a megfelelő eljárások is rekurzívak lehetnek. Adatként rendelkezésre állnak az

```
(a + b)*(c-d)
a + b*c-d
(a*b)*c-d

a + b*(c-d)
a*a*a*a
b + c*(d + c*a*a)*b + a.
```

szimbolikus kifejezések, amelyeket a következő EBNF-leírás alapján képeztünk. A bevitel végét pont jelzi.

Kifejezés = *Tag* { („+" | „–") *Tag*} .
Tag = *Tényező* {„*" *Tényező*} .
Tényező = *Azonosító* | „(„*Kifejezés*")".
Azonosító = *Betű*.

```
program LengyelAlak(Input,Output);
{Turbo Pascal}
{12.5. program - Hagyomanyos kifejezes lengyel
 (postfix) alakra hozatala}

var C: Char;

procedure Kereses;
label 13;
begin if Eof(Input)
      then goto 13;
      repeat Read(Input,C)
      until (C<>' ') or Eof(Input);
      13:
end {Kereses};

procedure Kifejezes;
var Op: Char;

procedure Tag;

procedure Tenyezo;
begin if C='('
      then begin Kereses;
                 Kifejezes;
                 {C=')'}
           end
      else Write(Output,C);
      Kereses
end {Tenyezo};

begin {Tag}
      Tenyezo;
      while C='*' do
            begin Kereses;
                  Tenyezo;
                  Write(Output,'*')
            end
end {Tag};

begin {Kifejezes}
      Tag;
      while (C='+') or (C='-') do
            begin Op:=C;
                  Kereses;
                  Tag;
                  Write(Output,Op)
            end
end {Kifejezes};

begin {LengyelAlak}
      Kereses;
      repeat Kifejezes;
             Writeln(Output)
      until C='.'
end {LengyelAlak}.
```

A program eredménye:

```
abc*+
ab+ab-*
sa-sb-*sc-*
bb*4a*c*-
ab*cd*-ab-*ab+*
```

A *bináris fa* olyan adatstruktúra, amelynél kézenfekvően adódik a rekurzív definíció, s amelyet így szintén rekurzív eljárásokkal dolgozhatunk fel. A fa véges számú csomópontból áll. Ezek halmaza vagy üres, vagy egy csomópontot (gyökeret) tartalmaz, amely két, közös rész nélküli bináris fa – a bal-, ill. jobb oldali részfa – kiindulópontja [6]. Természetes, hogy a bináris fák előállítását, feldolgozását végző rekurzív eljárások is ezt a fajta definíciót tükrözik.

A 12.6. program egy bináris fát épít fel, majd gyökérkezdő, gyökérközepű, és gyökérvégző sorrendben (pre-, in-, ill. postorder) bejárja a fa csomópontjait*. A fát gyökérkezdő alakban adjuk meg, ami azt jelenti, hogy a csomópontokat (melyeket esetünkben egy-egy betű jelöl) a gyökértől kiindulva előbb a bal és csak aztán a jobb oldali részfákon végighaladva soroljuk fel. A 12.9. ábrának megfelelő bemenő betűsor tehát:

```
abc..de..fg...hi..jkl..m..n..
```

ahol minden pont egy-egy üres részfát jelöl.

12.1.4. Eljárásparaméterek

A 12.6. programot úgy is átírhatjuk, hogy paraméterként eljárásokat adunk át. Az eljárásparaméterek eljárásfejként jelennek meg az eljárások és függvények formális paraméterlistájában. A megfelelő aktuális paraméterlistában csak az eljárásazonosítót kell megadni. A 12.7. program az előbbiek mellett azt is bemutatja, hogyan adhatunk át (aktuális) füzérértékeket illeszkedőtömb-paramétereknek.

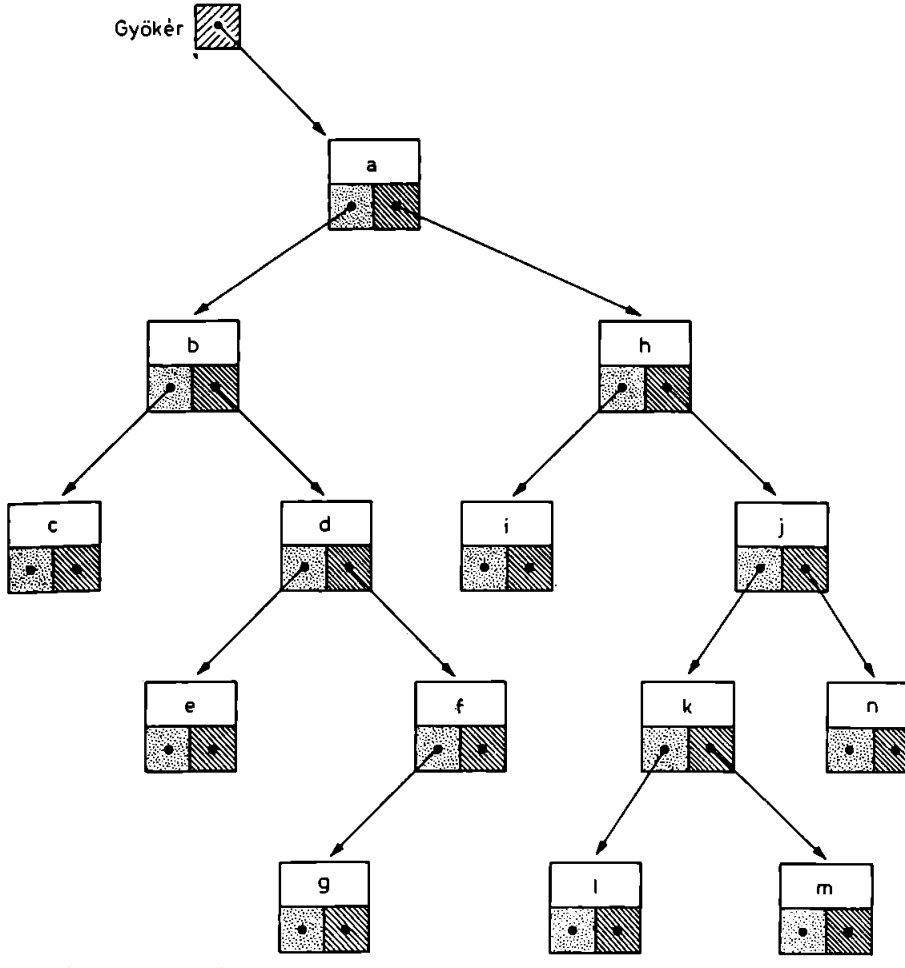
Óva intjük azonban az Olvasót a rekurzív módszerek fenntartás nélküli alkalmazásától, amelyek „ügyesnek" tűnhetnek ugyan, de nem mindig jelentik a leghatékonyabb megoldást!

Ha valamely P eljárás egy Q eljárást hív és viszont, Q is hívja P-t, s egyikük sincs a másikban deklarálva, akkor vagy P-t, vagy Q-t előzetes (forward) deklaráció formájában előre meg kell adni (l. a 12.3. pontot).

Az A. függelékben felsorolt standard (előredeklarált) eljárások minden szabványos Pascal-megvalósításban rendelkezésre állnak. A különféle megvalósításokban ezeken kívül még más előredeklarált eljárások is előfordulhatnak. Minthogy – az összes standard objektumhoz hasonlóan – ezeket is úgy tekintjük, mintha a hatáskörük a felhasználói programot körülvevő, annál tágabb blokkra terjedne ki, nem okoz problémát, ha a programban olyan deklaráció van, amely újradefiniál egy ilyen azonosítót.

Standard eljárások nem adhatók át aktuális eljárásparaméterként.

*Gyökérközepű esetben a bejárást a bal alsó csomóponttal kezdjük, majd balról jobbra, alulról fölfelé haladunk. Először a bal oldali részfát követjük a gyökérig, majd a jobb oldali részfát járjuk be. A gyökér (a) így a felsorolás közepére kerül. Gyökérvégző esetben szintén a bal alsó csomóponttól indulunk el, de a csomópontokat szintenként csoportosítva járjuk be. A gyökérhez a jobb oldali részfa bejárása után, a felsorolás végén jutunk. *(A ford.)*



12.9. ábra. Bináris fa típusú adatszerkezet

```
program Bejaras(Input,Output);
{Turbo Pascal}
{12.6. program - Binaris fa bejarasa}

type Mutato = ^Csomopont;
     Csomopont = record Info: Char;
                        Bal,Jobb: Mutato
                 end;

var Gyoker: Mutato;
    C: Char;

procedure Gyokerkezdo(P:Mutato);
begin if P<>nil
      then begin Write(Output,P^.Info);
                 Gyokerkezdo(P^.Bal);
                 Gyokerkezdo(P^.Jobb)
           end
end {Gyokerkezdo};
```

```
procedure Gyokerkozepu(P:Mutato);
begin if P<>nil
      then begin Gyokerkozepu(P^.Bal);
                 Write(Output,P^.Info);
                 Gyokerkozepu(P^.Jobb)
           end
end {Gyokerkozepu};

procedure Gyokervegzo(P:Mutato);
begin if P<>nil
      then begin Gyokervegzo(P^.Bal);
                 Gyokervegzo(P^.Jobb);
                 Write(Output,P^.Info)
           end
end {Gyokervegzo};

procedure Bevitel(var P:Mutato);
begin Read(Input,C);
      Write(Output,C);
      if C<>'.'
      then begin New(P);
                 P^.Info:=C;
                 Bevitel(P^.Bal);
                 Bevitel(P^.Jobb)
           end
      else P:=nil
end {Bevitel};
begin {Bejaras}
      Bevitel(Gyoker); Writeln(Output);
      Gyokerkezdo(Gyoker); Writeln(Output);
      Gyokerkozepu(Gyoker); Writeln(Output);
      Gyokervegzo(Gyoker); Writeln(Output)
end {Bejaras}.
```

A program eredménye:

```
abc..de..fg...hi..jkl..m..n..
abcdefghijklmn
cbedgfaihlkmjn
cegfdbilmknjha
```

```
program Bejaras2(Input,Output);
{MS Pascal}
{12.7. program - A 12.6. program eljarasparametereket
                 tartalmazo valtozata}

type Mutato = ^Csomopont;
     Csomopont = record Info:Char;
                        Bal,Jobb: Mutato
                 end;
     Pozitiv = 1..MaxInt;
     Szoveg = packed array[1..30] of Char;

var Gyoker: Mutato;
    C: Char;
```

```
procedure Gyokerkezdo(P: Mutato);
begin if P<>nil
      then begin Write(Output,P^.Info);
                 Gyokerkezdo(P^.Bal);
                 Gyokerkezdo(P^.Jobb)
           end
end {Gyokerkezdo};

procedure Gyokerkozepu(P: Mutato);
begin if P<>nil
      then begin Gyokerkozepu(P^.Bal);
                 Write(Output,P^.Info);
                 Gyokerkozepu(P^.Jobb)
           end
end {Gyokerkozepu};

procedure Gyokervegzo(P: Mutato);
begin if P<>nil
      then begin Gyokervegzo(P^.Bal);
                 Gyokervegzo(P^.Jobb);
                 Write(Output,P^.Info)
           end
end {Gyokervegzo};

procedure Bevitel(var P: Mutato);
begin Read(Input,C);
      Write(Output,C);
      if C<>'.'
      then begin New(P);
                 P^.Info:=C;
                 Bevitel(P^.Bal);
                 Bevitel(P^.Jobb)
           end
      else P:=nil
end {Bevitel};

procedure CspontIr(procedure Famuvelet(Start:Mutato);
                   Gyoker: Mutato; Cim: Szoveg);
var K: Pozitiv;
begin Writeln(Output);
      for K:=1 to 38 do
          Write(Output,Cim[K]);
      Writeln(Output);
      Writeln(Output);
      Famuvelet(Gyoker);
      Writeln(Output)
end {CspontIr};

begin {Bejaras2}
      Bevitel(Gyoker); Writeln(Output);
      CspontIr(Gyokerkezdo,Gyoker,
           'A csomopontok gyokerkezdo sorrendben: ');
      CspontIr(Gyokerkozepu,Gyoker,
           'A csomopontok gyokerkozepu sorrendben:');
      CspontIr(Gyokervegzo,Gyoker,
           'A csomopontok gyokervegzo sorrendben: ')
end {Bejaras2}.
```

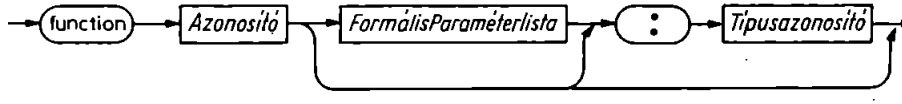
A program eredménye:

```
abc..de..fg...hi..jkl..m..n..

A csomopontok gyokerkezdo sorrendben:

abcdefghijklmn

A csomopontok gyokerkozepu sorrendben:

cbedgfaihlkmjn

A csomopontok gyokervegzo sorrendben:

cegfdbilmknjha
```

12.2. Függvények

A *függvények* az eljárásokhoz hasonló jellegű programrészek, amelyek valamilyen – egy kifejezés kiértékeléséhez szükséges – megszámlálható, valós vagy mutató típusú értékkiszámítást végzik. A *függvény behívásáról* a függvénykifejezés gondoskodik. Ez a függvényt jelölő azonosítóból és az aktuális paraméterek listájából áll. Ezek a paraméterek változók, kifejezések, eljárások vagy függvények lehetnek, és mindig a megfelelő formális paraméterek helyére kerülnek.

A függvénydeklaráció ugyanolyan alakú mint a program, azzal a különbséggel, hogy a *függvényfej* alakja a 12.10. ábra szerinti.



12.10. ábra. *Függvényfej* szintaxisdiagramja

Ugyanúgy, mint az eljárásoknál láttuk, a címkedefiníciós részben szereplő összes címke, a konstansdefiníciós részben bevezetett valamennyi azonosító, a típusdefiníciós rész, a változó-, az eljárás- vagy a függvénydeklarációs rész *lokális* a függvénydeklarációra nézve, amelyet éppen ezért az említett objektumok *hatáskörének* nevezünk. Hatáskörén kívül a fordítóprogram egyik objektumot sem ismeri fel. A lokális változók értéke az utasításrész kezdetén határozatlan.

A függvényfejbe írt azonosító nevet ad a függvénynek. Az eredmény csak egyszerű vagy mutató típusú lehet. A függvénydeklarációban szerepelnie kell egy ténylegesen végrehajtott, a függvényazonosítóra vonatkozó (eredmény típusú) értékadásnak. Ez az értékadás szolgáltatja a függvény eredményét.

A 12.8. programban az 5.3. program hatványozási algoritmusát függvénydeklarációként írjuk fel.

Ha a függvény azonosítója a függvényben szereplő bármelyik kifejezésben is megjelenik, akkor *rekurzív* függvényvégrehajtásról beszélünk. Az F. függelék első példája többek között a rekurzív függvények használatát szemlélteti.

A függvénykifejezés megelőzheti a függvénydeklarációt, ha van *előzetes deklaráció* (l. a 12.3. pontot).

Az A. függelékben felsorolt standard függvények minden szabványos Pascal-megvalósításban rendelkezésre állnak. A különböző megvalósítások további előre deklarált függvényeket is tartalmazhatnak. Standard függvények nem adhatók át aktuális függvényparaméterként.

```
program Hatvanyozas2(Output);
{Turbo Pascal}
{12.8. program - Az 5.3. program fuggvenyt tartalmazo
                 valtozata}

type Termeszetes = 0..MaxInt;

var Pi, PiNegyzet: Real;

function Hatvany(Alap:Real; Kitevo:Termeszetes):Real;
var Eredmeny: Real;
begin Eredmeny:=1;
      while Kitevo>0 do
            begin while not Odd(Kitevo) do
                        begin Kitevo:=Kitevo div 2;
                              Alap:=Sqr(Alap)
                        end;
                  Kitevo:=Kitevo-1;
                  Eredmeny:=Eredmeny*Alap
            end;
      Hatvany:=Eredmeny
end {Hatvany};

begin Pi:=ArcTan(1.0)*4;
      Writeln(Output,2.0:11:6,7:3,Hatvany(2.0,7):11:6);
      PiNegyzet:=Hatvany(Pi,2);
      Writeln(Output,Pi:11:6,2:3,Pinegyzet:11:6);
      Writeln(Output,PiNegyzet:11:6,2:3,
                     Hatvany(PiNegyzet,2):11:6);
      Writeln(Output,Pi:11:6,4:3,Hatvany(Pi,4):11:6)
end {Hatvanyozas2}.
```

A program eredménye:

```
2.000000  7 128.000000
3.141593  2   9.869604
9.869604  2  97.409091
3.141593  4  97.409091
```

12.2.1. Függvényparaméterek

A függvények maguk is átadhatók paraméterként eljárásoknak és függvényeknek. A formális függvényparamétert a függvényfejjel, az aktuálist a függvényazonosítóval adjuk meg. A 12.9. program a megadott függvények értékeiből képezett sor összegét számítja ki.

```
program Sorosszegek2(Output);
{MS Pascal}
{12.9. program - Kulonbozo szamu tagbol allo sorok
                 osszegenek kiszamitasa}

const MaxTagszam=10;

var Tag: 1..MaxTagszam;

function Szigma(function F(X: Real): Real;
                AlsoHatar,FelsoHatar: Integer): Real;
var Index: Integer;
    Osszeg: Real;
begin Osszeg:=0.0;
      for Index:=AlsoHatar to FelsoHatar do
          Osszeg:=Osszeg+F(Index);
          Szigma:=Osszeg
end {Szigma};

function SzinuszXSzerX(X: Real): Real;
begin SzinuszXSzerX:=Sin(X)*X
end {SzinuszXSzerX};

function EgyPerKob(X: Real): Real;
begin EgyPerKob:=1/(Sqr(X)*X)
end {EgyPerKob};

begin {Sorosszegek2}
      for Tag:=1 to MaxTagszam do
          Writeln(Tag,Szigma(SzinuszXSzerX,1,Tag),
                  Szigma(EgyPerKob,1,Tag))
end {Sorosszegek2}.
```

A program eredménye:

```
 1 8.4147100E-01  1.0000000E+00
 2 2.6600660E+00  1.1250000E+00
 3 3.0834250E+00  1.1620370E+00
 4 5.6215050E-02  1.1776620E+00
 5-4.7384060E+00  1.1856620E+00
 6-6.4148970E+00  1.1902920E+00
 7-1.8159870E+00  1.1932070E+00
 8 6.0988790E+00  1.1951600E+00
 9 9.8079450E+00  1.1965320E+00
10 4.3677290E+00  1.1975320E+00
```

12.2.2. Mellékhatások

Ha a függvénydeklarációban nemlokális változóra vagy változóparaméterre vonatkozó értékadás történik, *mellékhatásról (side effect)* beszélünk. Ez gyakran félrevezető, amikor a program szándékolt hatását akarjuk megállapítani, és rendkívül megnehezíti a program ellenőrzését. (Egyes gépi megvalósítások egyenesen megpróbálják megtiltani a mellékhatásokat.) Éppen ezért határozottan óvjuk az olvasót: ne alkalmazzon mellékhatással járó függvényeket! Példaként tekintsük a 12.10. programot.

```
program Mellekhatas(Output);
{Turbo Pascal}
{12.10. program - A fuggvenyek mellekhatasanak
                  szemleltetese}

var A, Z: Integer;

function Becsapas(X: Integer): Integer;
begin Z:=Z-X {mellekhatas Z-re};
      Becsapas:=Sqr(x)
end {Becsapas};

begin Z:=10;
      A:=Becsapas(Z);
      Writeln(Output,A:5,Z:5);
      Z:=10;
      A:=Becsapas(10);
      A:=A*Becsapas(Z);
      Writeln(Output,A:5,Z:5);
      Z:=10;
      A:=Becsapas(Z);
      A:=A*Becsapas(10);
      Writeln(Output,A:5,Z:5)
end {Mellekhatas}.
```

A program eredménye:

```
  100    0
    0    0
10000  -10
```

12.3. Előzetes (forward) deklarációk

Ha van *előzetes deklaráció*, az eljárás- (függvény-) hívás megelőzheti az eljárás- (függvény-) deklarációt. Az előzetes hivatkozás alakja a következő. (Figyeljük meg, hogy a paraméterlista és a tulajdonképpeni eredménytípus csak az előzetes hivatkozásban fordul elő!)

```
procedure Q(X:T); Forward;
procedure P(Y:T);
begin
     Q(A)
end;
procedure Q; { a paraméterek itt már nem szerepelnek }
begin
     P(B)
end;
```

13. Szövegállományok be-, ill. kivitele

A 10. fejezetben már szó esett az ember és a számítógép közötti kapcsolat problémájáról. Az ember és a számítógép egyaránt az *alakfelismerésnek* nevezett folyamat segítségével „tanul", érti meg az információt. Sajnos azok az „alakok", amelyeket az ember a legkönnyebben felismer (elsősorban a kép és a hang) egészen mások, mint amelyeket a számítógép képes felismerni és értelmezni (elektromos impulzusok). Ez olyannyira így van, hogy az adatok fizikai átadása (átvitele) – amelynek során tehát az ember által olvasható jeleket a számítógép által olvasható jelekké kell alakítani, és viszont – ugyanolyan költséges lehet, mint az átadott információ tulajdonképpeni feldolgozása. (Éppen ezért kiterjedt kutatómunka folyik, hogyan lehetne a fordítási folyamat költségeit – minél nagyobb fokú automatizálással – a minimálisra csökkenteni.) A fentiekben vázolt kommunikációs feladatot be-, ill. kivitelnek mondjuk.

Az ember *beviteli perifériák és adathordozók* (pl. billentyűzet, hajlékony mágneslemez, grafikus és rámutató-kiválasztó eszközök, mágnesszalag-kazetta, mágnesszalag, terminál) segítségével adhatja be információit, s az eredményeket *kiviteli perifériákon és adathordozókon* (pl. sornyomtatón, mágnesszalagon, hajlékony mágneslemezen, mágnesszalag-kazettán, rajzgépen, hanggenerátoron, képernyőn) kapja. Ami ezek többségében közös, az az ember számára olvasható – és az adott számítógéprendszer által meghatározott – karakterkészlet (l. a 3. fejezetet). Ez az a karakterkészlet, amelyen a Pascal a Text standard típust definiálja (l. a 10. fejezetet).

Fontos, hogy minden perifériának megvannak a maga sajátosságai: mindegyik egyéni módon értelmez bizonyos karaktereket vagy karaktercsoportokat (füzéreket). A legtöbb nyomtatón pl. korlátozott a sorszélesség. Sok régebbi sornyomtató a sorkezdő karaktereket nemnyomtatandó vezérlőkarakternek tekinti: az egyik hatására lapot dob, a másikra nem emel sort, felülírja az előző karaktereket stb. Ha valamely perifériának szövegállományt feleltetünk meg, ügyeljünk arra, hogy programunk betartsa az illető periféria sajátosságait!

A szövegállományok a Get és a Put standard eljárás segítségével érhetők el. Ez persze nemegyszer hosszadalmas lehet, hiszen ezeket az eljárásokat úgy definiáltuk, hogy egyszerre csak egy karakterrel tudnak dolgozni. Hogy a helyzetet szemléletesebbé tegyük, képzeljük el a következőt. Van egy természetes szám, amelyet valamely X változóban tároltunk, és ezt a számot ki szeretnénk vinni az Output állományra. Gondoljunk csak arra, mennyire más karakterek fordulnak elő egy szám decimális ábrázolásában, mint a római számos írásmódban (l. az 5.9. programot)! Minthogy általában a decimális ábrázolásra vagyunk kíváncsiak, jó, ha vannak a nyelvben olyan „beépített" standard transzformációs eljárások, amelyek az absztrakt számokat (akármilyen belső, számítógépi alakot is használunk) decimális számjegysorozattá alakítják (és viszont).

Ezért, hogy megkönnyítsük a szövegállományok olvasását és írását, több irányban is általánosítjuk a Read és a Write standard eljárást.

13.1. Az Input és az Output standard állomány

Az Input (beviteli) és az Output (kiviteli) szövegállomány általában a számítógépes rendszer szokásos be-, ill. kiviteli eszközeit (pl. a billentyűzetet és a megjelenítőt) reprezentálja. Így ez a két állomány a számítógép és az azt felhasználó ember közötti kapcsolat legfontosabb láncszeme.

Minthogy az Input és az Output állományt nagyon gyakran használjuk, azok a szöveg-állomány-műveletek, ahol az F szövegállomány explicit módon nincs kijelölve, automatikusan e két állomány valamelyikével hajtódnak végre. Részletesen, a táblázat bal oldalán levő műveleteknek a következő műveletek felelnek meg:

Write(Ch)	Write(Output,Ch)
Read(Ch)	Read(Input,Ch)
Writeln	Writeln(Output)
Readln	Readln(Input)
Eof	Eof(Input)
Eoln	Eoln(Input)
Page	Page(Output) (l. a 12.4. szakaszt).

Ha bármelyik felsorolt eljárást állományparaméter megadása nélkül használjuk, akkor megállapodásszerűen a művelet automatikusan az Input, ill. az Output állományra vonatkozik. Ebben az esetben viszont a két állománynak feltétlenül szerepelnie kell a programfej paraméterlistájában.

Megjegyzés: A Reset, ill. Rewrite standard eljárás Input, ill. Output állományra való alkalmazásának hatását a gépi megvalósítás határozza meg.

Ennek megfelelően a szövegállományok írását, ill. olvasását a következőképpen fejezhetjük ki: (legyen var Ch: Char; B1, B2: Boolean; legyen P, Q és R három, a felhasználó által definiált eljárás):

Szöveg írása az Output állományra:

```
repeat
      repeat P(Ch); Write(Ch)
      until B1;
      Writeln
until B2
```

Szöveg olvasása az Input állományról (minden ciklusban egyetlen sort dolgozunk fel):

```
while not eof do
      begin P;
            while not eoln do
                  begin Read(Ch); Q(Ch)
                  end;
            R; Readln
      end
```

A következő két példaprogram az Input és az Output szövegállomány használatát mutatja be. (Gondoljuk meg, hogyan kellene átírnunk a programot, ha a Read, ill. a Write helyett csak a Get, ill. a Put eljárást alkalmazhatnánk!)

```
program Betugyakorisag(Input,Output);
{MS Pascal}
{13.1. program - Az Input allomanyban szereplo betuk
      gyakorisaganak meghatarozasa; az allomany ki-
      irasa. Az angol abece betuivel dolgozunk,ez-
      ert mintaszovegeink ekezet nelkuliek.}

type Termeszetes=0..MaxInt;

var C: Char;
    EfSzam: array [Char] of Termeszetes;
    Betuk,Nagybetuk,Kisbetuk: set of Char;

begin Nagybetuk:=['A','B','C','D','E','F','G','H','I',
                  'J','K','L','M','N','O','P','Q','R',
                  'S','T','U','V','W','X','Y','Z'];
      Kisbetuk:=['a','b','c','d','e','f','g','h','i',
                 'j','k','l','m','n','o','p','q','r',
                 's','t','u','v','w','x','y','z'];
      Betuk:=Kisbetuk+Nagybetuk;
      for C:='A' to 'Z' do
          EfSzam[C]:=0;
      for C:='a' to 'z' do
          EfSzam[C]:=0;
      while not Eof do
            begin while not Eoln do
                        begin Read(C);
                              Write(C);
                              if C in Betuk
                              then EfSzam[C]:=
                                   EfSzam[C]+1
                        end;
                  Readln;
                  Writeln
            end;
      for C:='A' to 'Z' do
          if C in Nagybetuk
          then Writeln(C,EfSzam[C]);
      for C:='a' to 'z' do
          if C in Kisbetuk
          then Writeln(C,EfSzam[C])
end.
```

A program eredménye:

```
Egy patkany Tom hazaban lehet, hogy megette Tom
fagylaltjat! (Aritmetika)
Tegyen a dobozomba ot tucat italos korsot!
A gyors barna roka atugrotta a lusta alvo kutyat!
A        2
B        0
C        0
D        0
E        1
F        0
G        0
H        0
I        0
J        0
K        0
L        0
M        0
N        0
O        0
P        0
Q        0
R        0
S        0
T        3
U        0
V        0
W        0
X        0
Y        0
Z        0
a       22
b        4
c        1
d        1
e        8
f        1
g        7
h        3
i        3
j        1
k        5
l        6
m        5
n        4
o       14
p        1
q        0
r        6
s        4
t       19
u        4
v        1
w        0
x        0
y        7
z        2
```

A következő program az Inputot az Outputra másolja, s minden sor elejére beszúr egy sorszámot.

```
program Sorszamozas(Input,Output);
{MS Pascal}
{13.2. program - Sorszam beirasa szovegallomanyba}

type Termeszetes=0..MaxInt;

var Sorszam: Termeszetes;

begin Sorszam:=0;
      while not Eof do
            begin Sorszam:=Sorszam+1;
                  Write(Output,Sorszam: 4,' ');
                  while not Eoln do
                        begin Write(Output,Input^);
                              Get(Input)
                        end;
                  Readln(Input);
                  Writeln(Output)
            end
end.
```

A program eredménye:

```
1 Egy patkany Tom hazaban lehet, hogy megette Tom
2 fagylaltjat! (Aritmetika)
3 Tegyen a dobozomba ot tucat italos korsot!
4 A gyors barna roka atugrotta a lusta alvo kutyat
```

Ha az Input állományváltozót interaktív terminál beviteli eszközéhez (pl. a billentyűzethez) rendeltük, a legtöbb Pascal-megvalósítás mindaddig nem értékeli ki az Input↑ pufferváltozót, amíg értékére a programban ténylegesen nincs szükség. Ez két esetben fordulhat elő: ha Input↑-at explicit kifejezésben használjuk, vagy ha a Read és a Readln eljárást, ill. az Eof és az Eoln függvényt alkalmazzuk, amelyek implicit igénybe veszik a pufferváltozót. Bár a gép a programvégrehajtás kezdetén egy implicit Reset(Input)-ot hajt végre, mindaddig nem vár adatokat a terminálról, amíg ilyen értelmű utasítást nem kap – tehát pl. amíg Input↑ nem szerepel a programban. Ha a program üzenetet ír ki, s erre választ vár, akkor – ahogy azt általában megszoktuk – a bevitelkérés az üzenet kiírása után következik.

A következő programrészlet ilyen helyzetre mutat példát: kiírunk egy üzenetet, amelyre a felhasználónak válaszolnia kell.

```
program KérdésVálasz(Input,Output);
    var MonddMeg: Integer;
          .
          .
          .
begin{ itt történik az Implicit Reset(Input). }
    Writeln('Kérek egy 1 és 10 közötti egész számot!');
    Read(MonddMeg)
```

Azok a Pascal-megvalósítások, amelyekben nem késleltetett az Input↑ kiértékelése, a program végrehajtás indulásakor végzett implicit Reset(Input) miatt az üzenet kiírása előtt várják (kérik) az adatokat. Azt, hogy a kiértékelés késleltetett-e, a megvalósítás határozza meg.

13.2. A Read és a Readln eljárás

A Read eljárást a 10.2. szakaszban szövegállományokra definiáltuk. Most egyrészt több (változó számú), másrészt egész (egész résztartomány) és valós típusú paraméterekre is kiterjesztjük az értelmezését.

Jelöljön V1, V2, ..., Vn karakter, egész vagy valós típusú változókat (az előbbi kettő résztartományai is megengedettek), és legyen F egy szövegállomány!

Read(F,V) hibát okoz, ha F-et nem definiáltuk, ha nem F olvasása közben vagyunk, vagy ha eof(F) = True.

(1) Read(V1, ..., Vn) jelentése:

Read(Input,V1,...,Vn)

(2) Read(F, V1, ..., Vn) jelentése:

begin Read (F,V1);...;Read (F,Vn) end

(3) Readln(V1,...,Vn) jelentése:

Readln (Input,V1,...,Vn)

(4) Readln(F,V1,...,Vn) jelentése:

begin Read(F,V1);...;Read(F,Vn); Readln(F) end

Readln hatása: miután beolvasta Vn-t az F szövegállományról, az aktuális sor hátralevő részét átugorja. (A V1,...,Vn értékek azonban több sorra is átnyúlhatnak.)

(5) Ha Ch egy karakter vagy karakter-résztartomány típusú változó, akkor:

Read(F,Ch) jelentése:

begin Ch := F↑; Get(F) end

(6) Ha valamelyik V paraméter egész (ill. egész résztartomány) típusú, egy előjeles, szükség szerint szóközökkel kezdődő egész számot leíró karaktersorozat olvasódik be. V ekkor a megadott egész értéket veszi fel.

(7) Ha valamelyik V paraméter valós típusú, egy előjeles – szükség szerint üres karakterrel kezdődő – szám olvasódik be. V ekkor a megadott valós értéket veszi fel.

Ha számokat olvasunk be F-ből, azaz a Read-del számokat keresünk, a szóközöket (üres karaktereket) átugorva sorvégeket is átugorhatunk. F↑ továbbra is a szám utolsó jegyét követő első nem számjegykarakterre mutat. Számsorozatokat csak akkor olvashatunk be helyesen, ha az egymást követő számokat szóközökkel vagy sor vége jelekkel választjuk el. A Read a leghosszabb összefüggő számjegysort olvassa be, és ha két számot nem határolunk el, azokat nemcsak a Read, de egyetlen földi halandó sem tudja különválasztani.

Példák:

Olvassunk be és dolgozzunk fel egy számsort, ahol közvetlenül az utolsó szám után egy csillag áll! Legyen F szövegállomány, X egész (vagy valós) Ch pedig karakter típusú változó!

```
Reset(F);
repeat
      Read(F,X,Ch);
      P(X)
until Ch = '*'
```

Talán még gyakoribb az a helyzet, amikor semmit sem tudunk a beolvasandó adategységek számáról, és nincs olyan külön szimbólum, amely lezárná az adatsort. Két, jól felhasználható sémát mutatunk be. Mindkettőben a SzóközÁtugrás eljárással dolgozunk.

```
procedure SzóközÁtugrás(var F: Text);
      var Kész: Boolean;
begin
      Kész := False;
      repeat
            if eof(F) then Kész := True
            else
                  if F↑ = ' ' then Get(F)
                  else Kész := True
      until Kész
end
```

Az elsőben egyszerre egy adategységet dolgozunk fel:

```
reset (F);
while not eof(F) do
      begin
            Read (F,X); SzóközÁtugrás(F);
            P(X);
      end
```

A második séma n számból álló adatcsoportokat (szám-*n*-eseket) dolgoz fel:

```
Reset(F)
while not eof(F) do
      begin
            Read(F,X1,...,Xn); SzóközÁtugrás(F);
            P(X1,...,Xn);
      end
```

(Ez a program csak akkor működik helyesen, ha az adategységek összes száma n-nek valamilyen többszöröse.)

13.3. A Write és a Writeln eljárás

A Write eljárást a 10.2. szakaszban szövegállományokra definiáltuk. Most több, egész, valós, logikai vagy füzér típusnak megfelelő típusú paraméterre is értelmezzük.

A Write eljárás karakterláncokat (egy vagy több karaktert) illeszt valamilyen szövegállományhoz. Legyen P1, P2, ..., Pn a 13.1. szintaxisdiagrammal megadott alakú paraméter, és legyen F szövegállomány. Ekkor Write(F,P) hibát okoz, ha F-et nem definiáltuk, nem F írása közben vagyunk, vagy ha Eof(F) = False.

(1) Write(P1,...,Pn) jelentése:

Write(Output, P1, ..., Pn)

(2) Write(F,P1,...,Pn) jelentése:

begin Write(F,P1); ...; Write (F,Pn) end

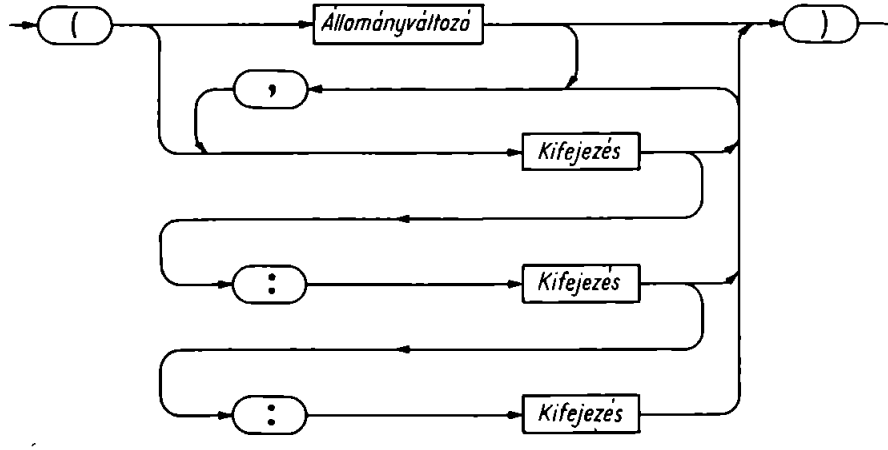
(3) Writeln(P1,...,Pn) jelentése:

Writeln(Output, P1,...,Pn)

(4) Writeln(F,P1,...,Pn) jelentése:

begin Write(F,P1); ...;Write(F,Pn); Writeln(F) end

A Writeln a P1,...,Pn írását, majd az F szövegállomány aktuális sorának lezárását eredményezi.



13.1. ábra. *WriteParaméterlista* szintaxisdiagramja

(5) Mindegyik Pk paraméter a következő alakú kell, hogy legyen:

e vagy
e:w vagy
e:w:f

ahol e, w és f kifejezések. e az írandó érték. Lehet karakter, egész, valós vagy logikai típusú, de lehet füzér is. w az ún. (legkisebb) *mezőszélesség*, egy nem kötelező for-

mázási segédeszköz. Pozitív egész értékű kifejezés kell, hogy legyen. Jelentése: az írásra kerülő karakterek száma. Az e értéket általában w számú karakterrel írjuk (az előtte álló szóközökkel együtt). Ha a mezőszélességet nem adjuk meg, a fordítóprogram egy e típusának megfelelő értéket tételez fel. Az f, az ún. *törtrészhossz* szintén egy választható formázási lehetőséget biztosít, de csak akkor használható, ha e valós típusú. f-nek pozitív egész értékű kifejezésnek kell lennie, és azt mutatja, hogy a tizedespont után még hány számjegy áll.

(6) Ha e karakter típusú, w alapértéke 1. Ezért Write (F,C) f := C; Put(F)-ből áll.

(7) Ha e egész típusú, akkor w alapértékét a megvalósítás határozza meg. Ha w kisebb, mint az egész szám kiírásához szükséges karakterek száma, a szám akkor is teljes egészében (ha e negatív, akkor előjellel együtt) kiíródik.

(8) Ha e füzér típusú, w alapértelmezés szerint a füzér hossza. Ha w ennél kisebb, e-nek az első w karaktere íródik ki.

(9) Ha e logikai típusú, w alapértéke a gépi megvalósítástól függ. w értékétől függően [(8)-nak megfelelően] a 'true' vagy a 'false' karakterlánc íródik az állományba. Az írásmódot, tehát hogy a 'true', ill. a 'false' szót kis-, ill. nagybetűkkel, vagy mindkettőt alkalmazva írja az eljárás, ugyancsak a megvalósítás szabja meg.

(10) Ha e valós típusú, w alapértékét a megvalósítás határozza meg. Ha w kisebb, mint a valós szám leírásához szükséges karakterek száma, a fennmaradó helyekre (ha e negatív, az előjel helyére is) szóköz kerül. Ha f-et (a törtrész hosszát) nem adtuk meg, e értéke *fixpontos*, egyébként pedig *lebegőpontos*, normál alakban íródik ki.

A fixpontos ábrázolás általános alakja a következő karaktersorozat: mínuszjel (csak ha a szám negatív), az egészrészt alkotó számjegyek, egy pont (a tizedespont), végül a törtrészt alkotó számjegyek.

A lebegőpontos ábrázolás általános alakja a következő w karakterből álló füzér: szóköz vagy mínuszjel, egy számjegy, egy pont (a tizedespont), egy számjegysorozat, az E (vagy e) betű, plusz- vagy mínuszjel, végül egy – a megvalósítás által meghatározott hosszúságú –, a kitevőt ábrázoló számjegysorozat. Az első, az E betű előtt álló számjegysorozat hossza w értékétől függ. A decimális lebegőpontos alak előtt bevezető szóközök – az említett előjelpozíció kivételével – nem állhatnak.

A 13.2. ábra mindegyik típus kiírására példát mutat.

Char:	w	Write ('S':w)
	1	$
	3	$

Integer:	w	Write (-1984:w)	Write (1984:w)
	1	-1984	1984
	4	-1984	1984
	5	-1984	1984
	7	-1984	1984

Füzér:

w	Write ('hello' : w)
1	h␣
3	h e l
5	h e l l o
7	␣␣h e l l o

Boolean:

w	Write (false : w)	Write (true : w)
1	f␣	t␣
3	f a l	t r u
5	f a l s e	␣t r u e
7	␣␣f a l s e	␣␣␣t r u e

Real

w	f	Write (123.789 : w : f)	Write (-123.789 : w : f)
1	1	1 2 3 . 8	- 1 2 3 . 8
1	3	1 2 3 . 7 8 9	- 1 2 3 . 7 8 9
1	4	1 2 3 . 7 8 9 0	- 1 2 3 . 7 8 9 0
5	1	1 2 3 . 8	- 1 2 3 . 8
6	1	␣1 2 3 . 8	- 1 2 3 . 8
7	1	␣␣1 2 3 . 8	␣- 1 2 3 . 8

w	Write (987.6 : w)	Write (-987.6 : w)
1	9 . 9 E + 0 2	- 9 . 9 E + 0 2
8	9 . 9 E + 0 2	- 9 . 9 E + 0 2
9	9 . 8 8 E + 0 2	- 9 . 8 8 E + 0 2
10	9 . 8 7 6 E + 0 2	- 9 . 8 7 6 E + 0 2
11	9 . 8 7 6 0 E + 0 2	- 9 . 8 7 6 0 E + 0 2

13.2. ábra. Példák formázott write utasításokra

13.4. A Page eljárás

A szövegállományok kényelmesebb formázása érdekében a Pascal tartalmaz egy Page (oldal) nevű standard eljárást. Page(F) hatására az eljáráshívás utáni szöveg új „oldalra" kerül (az oldal aszerint értelmezendő, hogy F-et nyomtatjuk, képernyőre írjuk, vagy más módon visszük ki).

Page(F) egy, a megvalósításban meghatározott műveletet végez az F állományon. A legtöbb megvalósításban az eljárás a kívánt hatás eléréséhez szükséges vezérlő- (control) karakter(eke)t – pl. az ASCII FF (*F*orm *F*eed : lapdobás) karaktert – iktatja az állományba.

Megjegyzések: Ha Page(F) hívása előtt az F állományon utoljára végzett művelet nem Writeln(F) volt, akkor Page(F) először is egy implicit Writeln(F)-et hajt végre. Hiba, ha Page(F) hívásakor F határozatlan vagy nincs írás módban. A megvalósítástól függ, mi történik, ha Page(F) után olvassuk az F állományt.

Niklaus Wirth

Jelentés

(A nyelv formális leírása)

1. Bevezetés

A Pascal nyelv kifejlesztésének hátterében két célkitűzés állt. Az első, hogy olyan nyelvet alkossunk, amely világosan és kézenfekvő módon tükröz bizonyos alapfogalmakat, s így lehetővé teszi, hogy a programozást rendszerbe foglalt, tudományosan megalapozott tantárgyként oktassuk. A második, hogy az új nyelvnek olyan gépi megvalósításait dolgozzuk ki, amelyek a jelenleg elterjedt számítógéptípusokon megbízhatóan és ugyanakkor hatékonyan alkalmazhatók.

Azért merült fel bennem, hogy a programozásoktatás céljaira új nyelvre lenne szükség, mert elégedetlen voltam a jelenleg használatos ismertebb nyelvekkel, amelyekben sok nyelvi eszköz és konstrukció gyakran nem magyarázható meg logikusan, és amelyek bizony nemegyszer ellentmondanak a következetes gondolkodásnak. Elégedetlenségem mellett ama meggyőződésem is sarkallt, hogy az a nyelv, amelyen a diákok először tanulják meg elképzeléseiket kifejezni, mélyrehatóan befolyásolja gondolkodásmódjukat, találékonyságukat, és hogy az ezeket a nyelveket átható rendszertelenség óhatatlanul rányomja bélyegét a diákok programozási stílusára.

Bőségesen van persze okunk a bizalmatlanságra, amikor már megint egy újabb programozási nyelv bevezetéséről hallunk. Kétségkívül alátámasztható valamelyest – ha mással nem, hát rövidlátó üzleti megfontolásokkal – azoknak az álláspontja, akik ellenzik, hogy olyan nyelven tanítsunk programozni, amely még nem terjedt el széles körben. Ha azonban az elterjedtség és az elfogadottság alapján választjuk meg a nyelvet, amelyet oktatni akarunk, akkor a programozás pedagógiai szempontból oly fontos területét egyhelyben topogásra kárhoztatjuk, hiszen a jövőben nyilván azt a nyelvet fogják a legtöbben használni, amit ma leginkább oktatunk. Úgy érzem tehát, hogy feltétlenül érdemes megkísérelnünk a kitörést ebből az ördögi körből.

Egy új nyelvet természetesen nem alkothatunk meg csupán azért, hogy mindenáron újat hozzunk létre. Ha a meglevő nyelvek eleget tesznek az említett követelményeknek és nem akadályozzák a módszeres struktúra kialakítását, a fejlesztőmunkának mindig ezekből kell kiindulnia. Ilyen értelemben a Pascal alapja az ALGOL-60 volt, ez ugyanis bármely más programozási nyelvnél jobban megfelel az oktatás igényeinek. Így a strukturálás elveit és lényegében a kifejezések alakját is az ALGOL-60-ból vettük át. Nem tartottuk volna ugyanakkor helyesnek, hogy a teljes ALGOL-60-at részhalmazként beépítsük a Pascalba, ez esetben ugyanis bizonyos – főként a deklarációkkal kapcsolatos – konstrukciós elvek nem lettek volna összeegyeztethetők azokkal az elvekkel, amelyek a Pascalban az ALGOL-60-hoz képest többletet jelentő eszközök természetes és kényelmes leírását biztosítják.

A Pascal az ALGOL-60-hoz viszonyítva elsősorban az adatstrukturálási lehetőségek terén gazdagabb, mivel véleményünk szerint ezek hiánya a fő oka az ALGOL-60 viszonylag szűk körű alkalmazhatóságának. A rekord- és az állománystruktúrát azért vezettük be, hogy a Pascalt üzleti, ügyviteli jellegű feladatok megoldására is alkalmassá tegyük, vagy legalábbis programozói tanfolyamokon bemutathassuk, hogy a nyelv ilyen feladatok megoldására is eredményesen alkalmazható.

2. A nyelv rövid leírása

Egy számítógépi program két fő részből áll. Az egyik részben a végrehajtandó *műveleteket* írjuk le, a másikban pedig azokat az *adatokat*, amelyeket e műveletek kezelnek. A műveleteket ún. *utasításokkal*, az adatokat pedig *deklarációkkal* és *definíciókkal* adjuk meg.

Az adatokat *változók* értékei szolgáltatják. Minden, a program valamelyik utasításában előforduló változót előzőleg *változódeklarációval* be kell vezetni, amely ehhez a változóhoz azonosítót és adattípust rendel. A *típus* lényegében az adott változó által fölvehető értékek készletét írja le, és meghatározza, hogy ezekkel az értékekkel milyen műveleteket lehet végezni. A Pascal nyelvben a típust vagy magában a változódeklarációban lehet megadni, vagy egy *típusdefiníció* segítségével típusazonosítót lehet hozzárendelni, azután ezzel a névvel hivatkozni rá.

Egyszerű típusok az eleve definiált Real (valós) típus, továbbá a különféle *diszkrét* (megszámlálható) típusok. Minden egyszerű típus az értékek egy rendezett halmazát határozza meg. A diszkrét típusokat az jellemzi, hogy kölcsönösen egyértelmű hozzárendelés létesíthető értékeik, valamint az egész számok halmaza egy intervallumának elemei között – ez utóbbiakat nevezzük a szóban forgó értékek rendszámainak.

Az alapvető diszkrét típusok a programozó által definiálható *felsorolt* típusok, valamint az eleve definiált Boolean (logikai), Char (karakter) és Integer (egész) típusok. A felsorolt típusok új értékkészletet, és valamennyi, ehhez tartozó érték jelölésére külön azonosítót vezetnek be. A Char típus értékeit idézőjelek közé tett szöveggel, az Integer és Real típuséit pedig számokkal jelöljük; ezek szintaktikailag különböznek az azonosítóktól. A Char típus értékkészlete és értékeinek grafikus ábrázolása esetenként, az alkalmazott számítógépes rendszer karakterkészletének függvényében változik.

Diszkrét típust definiálhatunk valamelyik alapvető diszkrét típus (az ún. törzstípus) *résztartományaként* is, mégpedig úgy, hogy megjelöljük a kívánt résztartományt alkotó értékek intervallumának legkisebb és legnagyobb elemét.

A *strukturált* típusokat elemeik típusának megadásával és a strukturálás módjának megjelölésével definiáljuk. Az egyes strukturálási módokat a strukturált típusú változók elemeinek elérésére szolgáló mechanizmusok különböztetik meg egymástól. A Pascal nyelv a strukturálás négy alapvető módját ismeri: a tömb-, a rekord-, a halmaz- és az állománystruktúrát.

A *tömbstruktúrában* valamennyi elem azonos típusú. Elemeit egy kiszámítható index jelöli ki, amelynek – szükségképpen diszkrét – típusát a tömb típusdeklarációjában kell rögzíteni. Ez általában felsorolt típus, vagy az Integer típus egy résztartománya. Az indextípus egy adott értéke esetén a tömb megfelelő elemét egy *indexelt változó* segítségével érhetjük el. A tömbváltozót ennélfogva tekinthetjük egy, az indextípus és a tömb elemeinek típusa közötti leképzésnek is. A tömb egyes elemeinek eléréséhez szükséges idő független az index értékétől. Ezért nevezik a tömbstruktúrát *véletlen hozzáférésű* (random-access) struktúrának.

A *rekordstruktúra* elemei (amelyeket *mezőknek* nevezünk) nem szükségképpen azonos típusúak. Az egyes mezőket nem valamilyen kiszámítható értékkel, hanem önálló azonosítókkal jelöljük meg, hogy az egyes mezők típusa magából a program szövegéből világosan kiderüljön (anélkül, hogy a programot végre kellene hajtanunk). A mezők azonosítóit a rekord típusának leírásakor deklaráljuk. Az egyes komponensek eléréséhez szükséges idő a rekord esetében sem függ az elérni kívánt mező azonosítójától, tehát a rekord is véletlen hozzáférésű struktúra.

Egy rekord típust úgy is megadhatunk, hogy az több különböző *változatot* tartalmazzon. Ilyenkor az azonos típusúnak deklarált változók szerkezete is eltérhet egymástól bizonyos mértékben: eltérő számú és típusú elemekből állhatnak. A rekordváltozó aktuális értékének megfelelő változatot a rekord egy *kijelölőmezőnek* nevezett, valamennyi változatban azonos szerkezetű elemének értéke is meghatározhatja. Általában több mező is van, amelynek szerkezete valamennyi változatban megegyezik, ezek egyike a kijelölőmező.

A *halmazstruktúra* értékkészlete saját alaptípusa értékkészletének hatványhalmaza, azaz az alaptípus értékkészlete összes részhalmazaiból alkotott halmaz. Az alaptípus csak diszkrét

lehet, mégpedig általában felsorolt vagy Char típus, esetleg az Integer típus résztartománya. A halmazok elemeihez közvetlenül nem férhetünk hozzá, a halmazműveletek azonban (köztük az „eleme" reláció) és a halmazértékű *generátor* lehetővé teszik halmazok létrehozását és kezelését.

Az *állománystruktúra* azonos típusú elemek *sorozatát* definiálja. A sorozat egyúttal az elemek egy természetes sorrendjét is jelenti. Adott pillanatban a sorozatnak mindig csak egy eleméhez lehet hozzáférni. Módunkban áll ekkor ezt az elemet megvizsgálni, vagy az állományba beírni, de sohasem egyszerre a kettőt. A többi elemet úgy érhetjük el, hogy az említett sorrendben végighaladunk az állományon. Az állomány generálása úgy történik, hogy végéhez egyenként újabb és újabb elemeket csatolunk. Az állomány típusleírása ezért az elemek számát nem is szabja meg.

A változók deklarációja során egy azonosítóhoz hozzárendelünk egy típust, és amikor az ezt a változódeklarációt tartalmazó blokkot (l. alább) meghívjuk, az azonosító által megjelölt változó rendelkezésünkre áll. Az ilyen explicit módon deklarált változókat *statikusaknak* is nevezzük. Változókat azonban végrehajtható utasításokkal is generálhatunk; az ilyen *dinamikus* generálás egy (az explicit azonosítót helyettesítő) ún. *mutatót* eredményez, amelynek segítségével később a változóra hivatkozhatunk. Ezeknek a mutatóknak az értékét velük megegyező típusú változókhoz és függvényekhez rendelhetjük hozzá. Minden mutató típushoz tartozik egy rögzített, ún. *főtípus*, és a mutató típus valamely mutató értéke által kijelölt bármely változónak a főtípushoz kell tartoznia. Az ilyen *mutató* értékeken kívül minden mutató típus tartalmazza a *nil* értéket is, amely nem jelöl ki változót. Minthogy a strukturált típusú változók elemei mutató típusúak is lehetnek, egy mutató típus főtípusa pedig maga is lehet strukturált, a mutatók alkalmazása a legáltalánosabb véges gráfok leírását is lehetővé teszi.

A legalapvetőbb utasítás az *értékadó* utasítás. Ez arról intézkedik, hogy egy *kifejezés* kiértékelésével kapott érték egy változóhoz (vagy annak egy eleméhez) rendelődjék hozzá. A kifejezések változókból, konstansokból, tömbparaméterek indexhatáraiból, halmazgenerátorokból, továbbá olyan műveleti jelekből és függvényekből állnak, amelyek az említett mennyiségekre hatva valamilyen értéket adnak eredményül. A változókat, konstansokat és függvényeket vagy magában a programban kell deklarálni, vagy standard („eleve deklarált") objektumokról van szó. A Pascal nyelv a műveleti jelek egy rögzített halmazát értelmezi, e jelek mindegyike úgy tekinthető, mint amely egy, az operandusok típusáról az eredménytípusra való leképezést jelöl. A műveleti jeleket négy csoportra osztjuk.

(1) *Aritmetikai* műveletek az összeadás, a kivonás, az előjelváltás, a szorzás, az osztás és a modulusképzés.
(2) *Logikai* műveletek a negálás, a diszjunkció (logikai vagy) és a konjunkció (logikai és).
(3) *Halmazműveletek* az unió, a metszet és a halmazdifferencia-képzés.
(4) *Relációk* az egyenlőség, a nemegyenlőség, a rendezés, a halmazhoz tartozás és a halmaz tartalmazás. A relációs műveletek eredménytípusa Boolean.

Az *eljárásutasítás* a kijelölt eljárás (l. alább) végrehajtásáról gondoskodik. Az értékadó és eljárásutasítások azok a komponensek, „téglák", amelyekből a *strukturált* utasítások felépülnek. Ez utóbbiak az őket alkotó részutasítások sorrendben, szelektíven vagy ismételten történő végrehajtásáról gondoskodnak. A sorrendben történő végrehajtás az *összetett*, a feltételes vagy szelektív végrehajtás az *if* és *case*, az ismételt végrehajtás pedig a *repeat*, *while* és *for* utasítások eredménye. Az if utasítás egy logikai kifejezés értékétől függően hajtat végre egy utasítást, míg a case utasítás segítségével több utasítás közül választjuk ki a végrehajtandót, egy diszkrét kifejezés értékének megfelelően. A for utasítást arra használjuk, hogy a ciklusmagot egymás után többször hajtassuk végre, miközben egy ún. ciklusváltozó diszkrét értékek egy sorozatán fut végig. A repeat és a while utasításokat másképpen használjuk.

A Pascal nyelvben ezenkívül egy *goto* utasítás használatára is lehetőség van, amely azt jelzi, hogy a végrehajtást a program egy másik pontján kell folytatni, ezt a helyet egy *címke* jelöli, amelyet deklarálni kell.

Az utasításokból, továbbá a címkék, konstansok, típusok, változók, eljárások és függvények deklarációiból *blokkokat* állítunk össze. A címkékre, konstansokra, változókra, típusokra, eljárásokra és függvényekre csak azon a blokkon belül lehet hivatkozni, ahol deklaráltuk őket, ezért ezeket a blokkra nézve *lokálisaknak* nevezzük. Azonosítóiknak csak a blokkot alkotó programszövegen belül van értelmük, ezt nevezzük az említett azonosítók *hatáskörének*. Blokkok az alapegységei a *programok*, *eljárások* és *függvények* deklarációinak is. Ilyen esetekben a blokkhoz egy nevet (azonosítót) rendelünk, amellyel a blokkra hivatkozni lehet. Minthogy az eljárásokat és függvényeket egymásba ágyazhatjuk, a hatáskörök is egymásba ágyazódhatnak.

Az eljárásokhoz és függvényekhez rögzített számú paraméter tartozik. Az eljáráson vagy függvényen belül ezek mindegyikét egy-egy azonosító jelzi, amelyet *formális* paraméternek nevezünk. Amikor egy eljárást vagy függvényt hívunk, minden paraméteréhez ki kell jelölni egy aktuális mennyiséget; az eljárás vagy a függvény belsejéből erre a mennyiségre a formális paraméter segítségével hivatkozhatunk. Az ilyen mennyiségeket aktuális paramétereknek nevezzük. Négyféle paramétert különböztetünk meg: az érték-, a változó-, az eljárás- és a függvényparamétereket. Az első esetben az aktuális paraméter egy kifejezés. A formális paraméter, az eljárás vagy függvény hívásának kezdetén az e kifejezés kiértékelésével kapott értéket veszi föl. A formális paraméter lokális változót jelöl. Változóparaméter esetén az aktuális paraméter egy változót jelöl, és a formális paraméter az eljárás vagy a függvény hívása során mindvégig ugyanezt a változót jelöli. Az eljárás- és a függvényparaméterek esetén az aktuális paraméter eljárás-, ill. függvényazonosító.

A függvényeket az eljárásokhoz hasonlóan deklaráljuk, azzal az egyetlen különbséggel, hogy a függvénynek valamilyen eredményt kell adnia, amelynek a függvény deklarációjában előre megadott típushoz kell tartoznia. Az eredménytípus csak egyszerű vagy mutató típus lehet. A függvényeket kifejezések alkotórészeiként is alkalmazhatjuk. A függvénydeklarációkon belül kerüljük a nemlokális változóknak való értékadást és más olyan műveleteket, amelyek ún. mellékhatást eredményezhetnek!

3. A jelölésmód és a terminológiák

A szintaktikai egységeket dőlt betűkkel írott szavakkal (metaazonosítókkal) jelöljük, és a kiterjesztett Backus–Naur-forma (EBNF) szabályai szerint definiáljuk [13]. Minden szabály egy EBNF-kifejezés segítségével definiál egy metaazonosítót. Az EBNF-kifejezés egy vagy több, egymástól függőleges vonallal (|) elválasztott, alternatív szókapcsolatból áll. A szókapcsolatnak nulla, egy vagy több eleme van, amelyek mindegyike lehet metaazonosító, idézőjelek közé tett, betűkből álló szimbólum, vagy egy újabb, egymásba skatulyázott kapcsos, szögletes, vagy gömbölyű zárójelek közé zárt EBNF-kifejezés. A kapcsos { és } zárójelek a (nulla- vagy többszöri) ismétlést jelzik, a szögletes [és] zárójelek azt, hogy a közéjük zárt tétel opcionális (azaz legfeljebb egyszer szerepelhet, de el is hagyható), végül a gömbölyű (és) zárójelek (amelyeket pontosan egyszer alkalmazunk) a bezárt kifejezés együvé tartozását.

A 4. fejezetben az EBNF-szabályok segítségével leírjuk, hogyan képezhetőek a szótári *szimbólumok* karakterekből; más karakter egyetlen szimbólumban sem fordulhat elő*. Az EBNF-szabályokat az 5-től a 13. fejezetig arra használjuk, hogy a szimbólumok segítségével definiáljuk a programok szintaxisát; ezeket a szimbólumokat – miként azt a 4. fejezetben le fogjuk írni – *elválasztójelek* választhatják el egymástól (vagy előzhetik meg).

*A továbbiakban ennek ellenére a teljes magyar karakterkészletet használni fogjuk; ezek nélkül ugyanis az eredeti szövegben használt beszédes változók lefordítása nehézségekbe ütközne, ill. olvasásuk – akár ékezetek híján, akár a táviratokban szokásos módon, a magánhangzó után írt „e" betűvel jelölve az ékezetes betűt – kényelmetlen volna. *(A ford.)*

A *hiba* kifejezés a program olyan tevékenységét vagy állapotát jelzi, amely az érvényes normákat megsérti, és amelyet valamely adott processzor esetleg elmulaszt jelezni.

A *rendszer által definiált* kifejezés azt jelenti, hogy egy adott Pascal-konstrukció az egyes alkalmazott rendszereken különböző lehet, és hogy minden rendszernek meg kell adnia, hogyan valósítja meg a szóban forgó konstrukciót.

A *rendszerfüggő* kifejezés azt jelenti, hogy egy adott konstrukció az egyes alkalmazott rendszerekben különböző lehet, de nem szükségképpen adják meg, hogy az illető rendszer hogyan valósítja meg a szóbanforgó konstrukciót.

A *bővítés* olyan kiegészítő konstrukció, amely általában nem minden rendszerben hozzáférhető, és amely önmagában nem érinti vagy érvényteleníti a standard Pascal konstrukcióit. Az egyes rendszerek igen gyakran tartalmaznak bővítéseket, például további előredefiniált és deklarált konstansokat, típusokat, változókat, eljárásokat és függvényeket.

Egy standard program nem függhet semmiféle rendszerfüggő konstrukciótól vagy bővítéstől. Ha pedig egy programot több rendszerben is akarunk futtatni, rendkívül óvatosaknak kell lennünk a rendszer által definiált konstrukciók (pl. karakterkészlet vagy az egész számok részére megszabott értékhatárok) alkalmazásával is.

4. Szimbólumok és elválasztók

A program egy, a Pascal szintaktikus szabályainak megfelelően elrendezett szimbólumsorozat. A szomszédos szimbólumokat az olvashatóság kedvéért gyakran elválasztószimbólumokkal határoljuk el. A szimbólumokat a következő csoportokra osztjuk: speciális szimbólumok, azonosítók, direktívák, számok, címkék és karakterfüzérek. Elválasztók a szóköz, a magyarázat és a programszövegben a sorok vége.

SpeciálisSzimbólum = "+" | "−" | "*" | "/" |
"=" | "<>" | "<" | "<=" |
">" | ">=" |
"(" | ")" | "[" | "]" |
":=" | "." | ".." |
":" | ";" | "↑" | *Alapszó*

Alapszó = "div" | "mod" | "nil" | "in" | "or" |
"and" | "not" | "if" | "then" | "else" |
"case" | "of" | "repeat" | "until" |
"while" | "do" | "for" | "to" | "goto" |
"downto" | "begin" | "end" | "with" | "const" |
"var" | "type" | "array" | "record" | "set" | "file" |
"function" | "procedure" | "label" | "packed" | "program" |

A standard Pascalban az alábbi módosításokat engedjük meg:

Eredeti	Módosítás
↑	^ vagy @
[	(.
]	.)

A legtöbb szimbólumot betűkből és számjegyekből állítjuk össze. A füzérek esetétől eltekintve az azonos hangot jelölő kis- és nagybetűk között nem teszünk különbséget.

```
Betű      = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
            "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
            "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
            "y" | "z".

Számjegy  = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Azonosítókkal jelöljük a konstansokat, a típusokat, a változókat, az eljárásokat, a függvényeket, a mezőket és a korlátokat. A direktívákat az eljárások és függvények deklarációiban alkalmazzuk.

```
Azonosító = Betű {Betű | Számjegy}.
Direktíva = Betű {Betű | Számjegy}.
```

Egy alapszó, azonosító vagy direktíva karaktersora az általa tartalmazott betűk és számjegyek teljes sorozata. Az azonosítók és direktívák karaktersora nem egyezhet meg egyetlen alapszóéval sem.

Példák azonosítókra (hat különböző karaktersor):

```
ElsőHely   rend    EljárásVagyFüggvénydeklaráció
Erzsébet   János   EljárásVagyFüggvényfej
```

Az egyes azonosítók karaktersorának deklaráció vagy definíció ad sajátos jelentést, és egy adott azonosító karaktersora nem rendelkezhet más jelentéssel a programnak azon részén belül, amelyet a deklaráció vagy definíció *hatáskörének* nevezünk (l. a 10. fejezetet).

A számokat a szokásos tízes számrendszerbeli írásmóddal ábrázoljuk. Az előjel nélküli egész és valós számok a standard Integer, ill. Real típusba tartozó konstansok (l. a 6.1.2. pontot). Egy előjel nélküli valós számban a karakterisztika előtt álló "e" betű jelentése: „...-szer 10 a ...-ediken". A standard Maxint konstans rendszer által definiált értéke az a maximális érték, amely egy előjel nélküli egész számnak adható.

```
ElőjelNélküliSzám  = ElőjelNélküliEgész | ElőjelNélküliValós.
ElőjelNélküliEgész = SzámjegyekSorozata.
ElőjelNélküliValós = ElőjelNélküliEgész "." SzámjegyekSorozata
                       ["e" Karakterisztika] |
                     ElőjelNélküliEgész "e" Karakterisztika.
Karakterisztika    = [Előjel] ElőjelNélküliEgész.
Előjel             = "+" | "-".
SzámjegyekSorozata = Számjegy {Számjegy}.
```

Példák előjel nélküli egész számra:

```
1    100    00100
```

Példák előjel nélküli valós számra:

```
0.1   0.1e0   87.35e+8   1E2
```

A numerikus értékek szövegállományokból való beolvasása előjeles számok formájában történik (l. a 12. fejezetet).

ElőjelesSzám = *ElőjelesEgész* | *ElőjelesValós*.
ElőjelesEgész = [*Előjel*] *ElőjelNélküliEgész*.
ElőjelesValós = [*Előjel*] *ElőjelNélküliValós*.

A füzérek aposztrófok közé zárt, füzérelemekből álló sorozatok. A füzérelemek a standard Char típusnak a megvalósítás során definiált elemei és vagy két szomszédos aposztrófból állnak, vagy aposztróftól különböző, a megvalósítás során definiált karakterek. Két különböző, füzérelemként előforduló karakternek a Char típus különböző értékeit kell jelölnie. A két aposztrófból álló füzérelem jelöli az aposztróf karaktert.

Karakterfüzér = " ' " *Füzérelem* { *Füzérelem* } " ' ".
Füzérelem = " " " | *BármelyAposztróftólKülönbözőKarakter*.

Egy füzér akkor Char típusú konstans, ha egyetlen füzérelemből áll; máskülönben füzér típusú konstans (l. a 6.2.1. pontot), amelynek annyi eleme van, ahány füzérelemből áll.

Megjegyzés: Egy füzér csak egy programsort foglalhat el.

Példák füzérre:

```
'A'          ';'
'Pascal'     ',,'
'Ez egy karakterfüzér'
```

Elválasztókat a program bármely két szomszédos szimbóluma között vagy a program első szimbóluma előtt elhelyezhetünk. Legalább egy elválasztónak kell állnia két szomszédos azonosító, direktíva, alapszó, címke, vagy szám között. Elválasztó a szóköz, a program egy sorának a vége és a magyarázat. A program jelentése nem változik meg, ha benne egy magyarázatot szóközzel helyettesítünk.

Magyarázat = („ { " | „(*") { *MagyarázatElem* } („ { " | „*)").

A *MagyarázatElem* vagy egy sor vége, vagy egy tetszőleges karaktersorozat, amely nem tartalmazza a „ { " vagy a „*)" jelet.

Megjegyzés: Szintaktikailag helyes a magyarázat { ...*) vagy (*... } módon való jelölése is. A { (*) magyarázat ekvivalens a { } magyarázattal.

5. Konstansok

A konstansdefiníció egy konstansazonosítót vezet be annak az értéknek a jelölésére, amelyet a definícióban szereplő konstans ad meg. Az a konstansazonosító, amelyet éppen definiálunk, nem szerepelhet a definíció „konstans" részében. A konstansdefiníciókat konstansdefiníciós részekbe gyűjtjük össze.

KonstansdefiníciósRész = ["const" *Konstansdefiníció*
";" { *Konstansdefiníció* ";" }].
Konstansdefiníció = *Azonosító* „=" *Konstans*.
Konstans = [*Előjel*] (*ElőjelNélküliSzám* | *Konstansazonosító*) | *Füzér*.
Konstansazonosító = *Azonosító*.

Az olyan konstansazonosítónak, amely elé („+" vagy „–") előjelet illesztettünk, Integer vagy Real típusú értéket kell jelölnie.

Három standard, előredefiniált konstansazonosító létezik: a rendszer által definiált, Integer típusú értéket jelölő Maxint, és a Boolean típusú értékeket jelölő false és true (l. a 6.1.2. pontot).

Példa konstansdefiníciós részre:

```
const
    N = 20;
    Fénysebesség = 2.998e8 { m/s } ;
    Sarkcsillag = 'Polaris';
    epsz = 1E-6;
```

6. Típusok

A *típus* azon értékek halmazát határozza meg, amelyeket az ilyen típusú változók, kifejezések, függvények stb. fölvehetnek. A *típuskompatibilitás* szabályai határozzák meg, hogyan alkalmazhatóak különféle típusok egyazon kifejezésben, értékadásban stb.

A típusdefiníció típusazonosítót vezet be a típus megjelölésére. Az a típusazonosító, amelyet éppen definiálunk, nem szerepelhet másként a definíció „típus" részében, mint egy mutató típus főtípusa (l. a 6.3. szakaszt). A típusdefiníciókat típusdefiníciós részekbe gyűjtjük össze. A 6.4. szakaszban láthatunk példát egy típusdefiníciós részre.

TípusdefiníciósRész = ["type" *TípusDefiníció* ";" { *TípusDefiníció* ";" }].
Típusdefiníció = *Azonosító* "=" *Típus*.
Típusazonosító = *Azonosító*.

A típusokat a *Típus* EBNF metaazonosító jelöli. Ha a típus megjelölése pusztán egy típusazonosítóból áll, akkor ugyanazt a (már létező) típust határozza meg, amelyet a típusazonosító jelöl. Ha a típus megjelölése nem csak egy típusazonosítóból áll, akkor egy egészen új típust határoz meg. A típusokat néhány fontosabb tulajdonságuk alapján csoportosítjuk:

Típus = *EgyszerűTípus* | *StrukturáltTípus* | *MutatóTípus*.

6.1. Egyszerű típusok

Az egyszerű típus értékeknek egy rendezett halmazát határozza meg, és vagy a standard Real típussal egyezik meg, vagy valamilyen megszámlálható típus. Egy valós típusazonosító olyan típusazonosító, amely a valós típust jelöli.

EgyszerűTípus = *MegszámlálhatóTípus* | *ValósTípusAzonosító*.
ValósTípusAzonosító = *Típusazonosító*.

A megszámlálható típusokat az különbözteti meg a *valós* típustól, hogy értékeik és a *rendszámok* halmaza között kölcsönösen egyértelmű leképezés létesíthető. Minden megszámlálható típus rendszámai az egész számok halmazának részintervallumát alkotják.

Egy megszámlálható típus minden X értékére alkalmazható az alábbi három standard függvény:

ord(X) az X-nek megfelelő rendszámot adja meg; az eredmény Integer típusú.
succ(X) az X után következő elemet adja meg. Ez azt jelenti, hogy

succ(X) > X és ord(succ(X)) = ord(X) + 1

föltéve, hogy X nem a legnagyobb ilyen típusú érték; ha igen, akkor succ(X) hibát eredményez.
pred(X) az X-et megelőző elemet adja meg. Ez azt jelenti, hogy

pred(X) < X és ord(pred(X)) = ord(X) − 1

föltéve, hogy X nem a legkisebb ilyen típusú érték: ha igen, akkor pred(X) hibát eredményez.

Egy megszámlálható típus értékeinek rendezését természetesen a hozzájuk tartozó rendszámok rendezésének megfelelően értelmezzük.

Egy megszámlálható típus vagy *felsorolt* típus, vagy a standard Integer, Char és Boolean típusok egyike, esetleg ezek valamelyikének *résztartománya.*

MegszámlálhatóTípus = *FelsoroltTípus* | *RésztartományTípus* | *Típusazonosító*.
MegszámlálhatóTípusAzonosító = *Típusazonosító*.

Egy megszámlálhatótípus-azonosító olyan típusazonosító, amely megszámlálható típust jelöl.

6.1.1. Felsorolt típusok

Egy felsorolt típus teljesen új értékek egy halmazát definiálja és minden egyes érték megjelölésére bevezet egy konstansazonosítót.

FelsoroltTípus = "(" *Azonosítólista* ")".
Azonosítólista = *Azonosító* { "," *Azonosító* } .

Az első azonosító jelöli a legkisebb értéket, amelynek rendszáma zérus. A lista minden további azonosítója az őt megelőző azonosító által megjelölt érték után következő értéket jelöli. A konstansazonosítók tehát értékük növekvő sorrendjében szerepelnek a listán.

Példák felsorolt típusra:

(Vörös, Narancssárga, Sárga, Zöld, Kék)
(Treff, Káró, Kőr, Pikk)
(Hétfő, Kedd, Szerda, Csütörtök, Péntek, Szombat, Vasárnap)

6.1.2. Standard egyszerű típusok

A Pascal nyelvben az alábbi, előredefiniált standard típusazonosítók vannak.

Real a valós számok halmazának egy, a konkrét Pascal rendszer által definiált részhalmazát határozza meg.

Integer azon egész számok halmazát jelöli ki, amelyeknek abszolút értéke nem haladja meg a standard Maxint konstansazonosító rendszer által definiált értékét. Bármely I egész számra ord(I) = I.

Boolean a standard false és true konstansazonosítókkal jelölt igazságértékek halmazát jelöli ki. Megjegyezzük, hogy false < true és ord(false) = 0.

Char egy, a rendszer által definiált karakterkészletet jelöl ki. Az ide tartozó karakterek rendszámát a rendszer definiálja úgy, hogy:

(a) a '0','1',...,'9' számjegyek nagyság szerint legyenek rendezve és egymás után következzenek (tehát pl. succ('0') = '1');

(b) a kisbetűk ('a','b',...,'z'), ha vannak, legyenek ábécésorrendbe rendezve (de nem kell föltétlenül egymás után következniük!); végül

(c) a nagybetűk ('A','B',...,'Z'), ha vannak, legyenek ábécésorrendbe rendezve (de nem kell föltétlenül egymás után következniük!).

6.1.3. Résztartomány típusok

Egy résztartomány típus által meghatározott értékkészlet valamely más, a résztartomány típus törzstípusának nevezett megszámlálható típus értékkészletének részhalmaza. A résztartomány típus a hozzá tartozó legkisebb és legnagyobb értéket adja meg, és minden, e kettő közé eső értéket tartalmaz.

RésztartományTípus = Konstans ".." Konstans.

Mindkét konstansnak a törzstípushoz kell tartoznia. Az elsőként megadott konstans jelzi a legkisebb értéket, és nem haladhatja meg a legnagyobb értéket kijelölő, másodiknak megadott konstanst.

Példák résztartomány típusra:

```
1..N
-10..+10
Hétfő..Péntek
```

6.2. Strukturált típusok

A strukturált típusokat elemeik típusa és strukturálásuk módja jellemzi. Egy strukturált típusban ezenkívül azt is jelezhetjük, milyen módon kívánjuk az adatokat ábrázolni. Egy strukturált típus előtt álló packed szimbólum a program jelentését (két kivételtől eltekintve) nem befolyásolja, csak a fordítóprogramot utasítja, hogy a szóban forgó típus értékeinek tárolásánál takarékoskodjék a hellyel, még ha emiatt kevésbé hatékonyan lehet is majd a struktúra elemeihez hozzáférni; a visszakeresésüket szolgáló kód pedig bonyolultabb lesz. Az ilyen strukturált típust tömörítettnek mondjuk. Az említett két kivétel az, hogy a füzér típusoknak (l. a 6.2.1.

pontot) mindig tömörítetteknek kell lenniük, ill. hogy egy aktuális változóparaméter (l. a 11.3. szakaszt) nem lehet eleme valamely tömörített strukturált változónak. Ha egy tömörített strukturált típus valamely eleme maga is strukturált típusú, úgy az elem típusa csak abban az esetben lesz tömörített, ha típusleírásában szintén szerepel a packed szimbólum.

StrukturáltTípus = ["packed"] *TömörítetlenStrukturáltTípus* |
StrukturáltTípusAzonosító.
TömörítetlenStrukturáltTípus = *TömbTípus* | *RekordTípus* | *HalmazTípus* |
ÁllományTípus.
StrukturáltTípusAzonosító = *Típusazonosító*.

A strukturált típusazonosító olyan típusazonosító, amely strukturált típust jelöl.

6.2.1. Tömb típusok

A tömb típus rögzített számú, azonos típusú elemből álló struktúra. Az elemek típusát a tömb *elemtípusának* nevezzük. Az elemek és az *indextípus* értékei között kölcsönösen egyértelmű megfeleltetés létesíthető.

TömbTípus = "array" "[" *Indextípus* { "," *Indextípus* } "]" "of" *Elemtípus*.
Indextípus = *MegszámlálhatóTípus*.
Elemtípus = *Típus*.

Több indextípus is megadható, például így:

```
packed array [T1, T2, ..., Tn] of C.
```

Ez egyszerűen a

```
packed array [T1] of array [T2, ..., Tn] of C
```

jelölés rövidítése. A fenti két jelölés akkor is ekvivalens, ha egyikük elején sem szerepel a packed szimbólum.

Példák tömb típusra:

```
array [1..100] of Real
array [1..10, 1..20] of 0.99
array [Boolean] of Szín
array [Méret] of packed array ['a'..'z'] of Boolean
```

Minden tömb típusú érték az indexértékek halmazának egyértékű leképezése az elemtípushoz tartozó értékek halmazára.

Egy tömb típust *füzér típusnak* nevezünk, ha tömörített, elemtípusa a standard Char típus, indextípusa pedig az Integer 1-től *n*-ig terjedő résztartománya, ahol *n* nagyobb, mint 1. A karakterfüzérek (l. a 4. fejezetet) füzér típusú konstansok.

Példák:

```
packed array [1..FüzérHossza] of Char
packed array [1..2] of Char
```

6.2.2. Rekord típusok

Egy rekord típus rögzített számú, de egymástól esetleg különböző típusú elemekből áll. Az egyes elemeket, azok típusait, továbbá a rekord típus értékeit a rekord típus *mezőlistája* írja le.

RekordTípus = "record" *Mezőlista* "end".
Mezőlista = [*(RögzítettRész* [";" *VáltozatRész*] | *VáltozatRész)* [";"]].
RögzítettRész = *Rekordszakasz* { ";" *Rekordszakasz* }.
Rekordszakasz = *Azonosítólista* ":" *Típus*.
Mezőazonosító = *Azonosító*.

A mezőlista tartalmazhat egy *rögzített részt*, amely rögzített számú, *mezőnek* nevezett elemet ad meg. A rekordszakasz hatására a benne elhelyezett listán felsorolt azonosítók az ugyancsak a rekordszakaszban megadott típusú mezők azonosítói lesznek. A mezőazonosító hatásköre a saját rekordszakaszára, továbbá azokra a mezőkifejezésekre és with utasításokra terjed ki, ahol esetleg alkalmaztuk (l. a 7.2.2. és 9.2.4. pontokat és a 10.2. szakaszt). Egy rekord típuson belül tehát minden mezőazonosító karaktersorának különbözőnek kell lennie.

Példák olyan rekord típusra, amelynek csak rögzített része van:

```
packed record
      Év: 1900..2100;
      Hónap: 1..12;
      Nap: 1..31
end

record
      Vezetéknév, Keresztnév
            packed array [1..32] of Char;
      Életkor: 0..99;
      Házas: Boolean
end
```

Egy mezőlista *változat részt* is tartalmazhat, amely egy vagy több *változatot* ad meg. Egy változat szerkezetét és értékeit annak mezőlistájában adjuk meg.

VáltozatRész = "case" *Változatszelektor* "of" *Változat* { ";" *Változat* }.
Változat = *Konstans* { "," *Konstans* } ":" "(" *Mezőlista* ")".
Változatszelektor = [*Kijelölőmező* ":"] *KijelölőTípus*.
KijelölőTípus = *MegszámlálhatóTípusAzonosító*.
Kijelölőmező = *Azonosító*.

A kijelölő típus értékét egy, a változatot megelőző konstansnak kell megadnia. Egy adott változó részben minden ilyen értéknek egyszer és csak egyszer kell szerepelnie. Ha egy változatszelektorban előfordul egy kijelölőmező, akkor ennek azonosítója egy, a kijelölő típushoz tartozó mező mezőazonosítója lesz.

Egy adott változat részből egyidejűleg mindig csak egy változat lehet érvényes. Ha van kijelölőmező, akkor az a változat érvényes, amely előtt a kijelölőmező értéke áll. Kijelölőmező hiányában mindig az a változat van érvényben, amelyhez a rekord legutóbb földolgozott eleme is tartozik.

A mezőlista egy értéke meghatározza a rögzített részben megadott mezők értékeit, valamint a változat rész egy értékét. A változat rész egy értéke egy, az érvényes változatot föltüntető jelzésből, a kijelölőmező (ha egyáltalán létezik) értékéből és az érvényes változat egy értékéből áll.

Példák változat részt is tartalmazó rekord típusra:

```
record
case NévIsmert : Boolean of
      false: ( );
      true: (Name: packed array [1..NévMax] of Char)
end

record
      X, Y: Real;
      Terület: Real;
case A: Alakzat of
      Háromszög:        (Oldal: Real;
                         Hajlásszög, Szög1, Szög2: Szög
                         );
      Paralelogramma:   (Oldal1, Oldal2: Real;
                         Dőlésszög, Szög3: Szög
                         );
      Kör:              (Átmérő: Real)
end
```

6.2.3. Halmaz típusok

Egy halmaz típus lehetséges értékeinek halmaza nem más, mint egy *alaptípus* értékeiből álló halmaz hatványhalmaza. Egy halmaz típus minden értéke tehát egy nulla vagy annál több elemet tartalmazó halmaz, és minden ilyen elem az alaptípus valamelyik értéke.

HalmazTípus = "set" "of" *Alaptípus*.
Alaptípus = *MegszámlálhatóTípus*.

Példák halmaz típusra:

```
set of Char
packed set of 0..11
```

6.2.4. Állomány típusok

Az állomány típus egyazon (elemtípusnak nevezett) típushoz tartozó elemek sorozata. A típushoz szervesen hozzátartozik az elemeknek a sorozatban elfoglalt helye, továbbá a mód, amely azt jelzi, hogy az állományt éppen létrehozzuk-e vagy pedig feldolgozzuk. A sorozat elemeinek számát, amelyet az állomány *hosszának* nevezünk, az állomány típusa nem rögzíti. Az állomány *üres*, ha hossza 0.

ÁllományTípus = "file" "of" *Elemtípus*.

Egy állomány típus elemtípusának hozzárendelhető típusnak (l. a 6.5. szakaszt) kell lennie. Ha az állomány *feldolgozási* módban van, a sorozat bármely eleméhez, továbbá az *állomány vége* pozícióhoz egyaránt hozzáférhetünk. A *generálási* módban lévő állományból csak az állomány vége pozícióhoz lehet hozzáférni. Egy állomány tartalmát a standard állománykezelő eljárásokkal és függvényekkel (l. a 11. fejezetet) kezelhetjük.

A standard Text strukturált típusazonosító egy olyan, különleges állomány típust jelöl, amelyben az említett sorozatot *sorokra* tördeljük. A sorok száma lehet 0 vagy annál több. A sorok karakterekből (Char típusú értékekből) épülnek föl, amelyeket egy speciális *sor vége* jel követ. Egy sorba kerülhet 0 vagy annál több karakter. A Text típusú változókat *szöveg típusú állományoknak* nevezzük. Egy nemüres szöveg típusú állományban, ha az feldolgozási módban van, az állomány vége pozíciót közvetlenül megelőző pozíción mindig sor vége jelnek kell állnia. A szöveg típusú állományok kezeléséhez számos további eljárás és függvény áll rendelkezésünkre (l. a 11.5. szakaszt és a 12. fejezetet).

6.3. Mutató típusok

Egy mutató típus abban különbözik az egyszerű és a strukturált típusoktól, hogy értékkészlete dinamikus; a mutató típusú értékek ugyanis a program végrehajtása során keletkeznek és szűnnek meg. Egy mutató típus értékkészlete mindig tartalmaz egy speciális, nil-lel jelölt értéket. Az értékkészlet összes többi elemét a program alkotja meg a standard New eljárás (l. a 11.4.2. pontot) segítségével. Az ilyen értékeket *mutatóértékeknek* nevezzük, mert mindegyik egy változóra, az ún. *dinamikus változóra* mutat (l. a 7.3. szakaszt). A dinamikus változóknak mindig a mutató típus *főtípusához* kell tartozniuk. A mutatóértéket és az általa kijelölt dinamikus változót a standard Dispose eljárás (l. a 11.4.2. pontot) segítségével lehet megszüntetni. A program befejeződésével valamennyi, a program által létrehozott mutatóérték eltűnik.

MutatóTípus = "↑" *Főtípus* | *MutatóTípusAzonosító*.
Főtípus = *Típusazonosító*.
MutatóTípusAzonosító = *Típusazonosító*.

6.4. Példa típusdefiníciós részre

```
type
    TermészetesSzám = 0..Maxint;
    Szín = (Vörös, Sárga, Zöld, Kék);
    Árnyalat = set of Szín;
    Alakzat = (Háromszög, Paralelogramma, Kör);
    Év = 1900..2100;
    Kártya = array [1..80] of Char;
    Füzér18 = packed array [1..18] of Char;
    Komplex = record Re, Im: Real end;
    Személymutató = ↑ Személy;
    Kapcsolat = (Házas; Élettárs, Egyedülálló);
    Személy =
        record
            Vezetéknév, Keresztnév: Füzér18;
            SzületésiÉv: Év;
            Neme: (Férfi, Nő);
            ApjaNeve, AnyjaNeve: Személymutató;
```

```
            Barátai, Gyermekei : file of Személymutató;
            KorábbiKapcsolatokSzáma: TermészetesSzám;
        case Állapot: Kapcsolat of
            Házas, Élettárs: (MásJellemző: Személymutató);
            Egyedülálló: ( )
        end;
    Mátrixindex = 1..N;
    NégyzetesMátrix = array [Mátrixindex, Mátrixindex] of Real;
```

6.5. Típuskompatibilitás

Két típust *kompatibilisnek* mondunk, ha az alábbi négy feltétel valamelyike teljesül:

(a) Azonos típusúak.
(b) Egyik a másik résztartománya, vagy mindketten ugyanannak a törzstípusnak a résztartományai.
(c) Mindkettő halmaz típus, alaptípusaik kompatibilisek, és vagy mindkettő tömörített, vagy egyik sem.
(d) Azonos számú elemből álló füzér típusok.

Egy típust *hozzárendelhetőnek* nevezünk, ha nem állomány típus, és nem is olyan strukturált típus, amelynek valamelyik elemtípusa nem hozzárendelhető.

Egy T2 típusú értékre azt mondjuk, hogy *értékadás-kompatibilis* egy T1 típussal, ha az alábbi négy feltétel valamelyike teljesül.

(a) T1 és T2 ugyanaz a hozzárendelhető típus.
(b) T1 a Real, T2 pedig az Integer típus.
(c) T1 és T2 kompatibilis megszámlálható típusok vagy kompatibilis halmaz típusok, és az említett érték a T1 által meghatározott értékkészlet eleme.
(d) T1 és T2 kompatibilis füzér típusok.

Ha a szintaktikus előírások értékadás-kompatibilitást követelnek meg, T1 és T2 pedig kompatibilis megszámlálható vagy kompatibilis halmaz típusok, mindig hiba keletkezik, ha a T2 típusú érték nem tartozik a T1 által meghatározott értékkészlethez.

7. Változók

Egy változóhoz mindig tartozik egy, a változó deklarációja által meghatározott típus. A változó csak ilyen típusú értékeket vehet föl.

Egy változó határozatlan, ha nincs hozzárendelve típusához tartozó érték. Teljesen határozatlannak mondunk egy változót akkor, ha határozatlan és (amennyiben strukturált) valamennyi eleme teljesen határozatlan. A változók létrejöttükkor mindig teljesen határozatlanok. Egy blokk belsejében deklarált változó a blokk aktiválásakor jön létre és megszűnik, amint a blokk nem aktív többé (l. a 10. fejezetet). A dinamikus változók a standard New és Dispose eljárások segítségével hozhatók létre, ill. számolhatók fel (l. a 6.3. és a 11.4. szakaszt).

Egy változódeklaráció bevezeti egy vagy több változó azonosítóját, továbbá azt a típust, amelyhez e változók mindegyikének értéke tartozik. A változódeklarációkat változódeklarációs részbe gyűjtjük össze.

VáltozódeklarációsRész = ["var" *Változódeklaráció* ";"
{ *Változódeklaráció* ";" }].
Változódeklaráció = *Azonosítólista* ":" *Típus*.
Változóazonosító = *Azonosító*.

Példa változódeklarációs részre:

```
var
    W, X, Y: Real;
    Z: Komplex;
    I, J: Integer;
    K: 0..9;
    P, Q: Boolean;
    MűveletiJel: (Plusz, Mínusz, Szorzásjel);
    Kontrasztfokozat: array [0..63] of Real;
    Láthatóság: array [Szín, Boolean] of Komplex;
    Fény: Szín;
    F: file of Char;
    Árnyalat1, Árnyalat2: set of Árnyalat;
    P1, P2: Személymutató;
    A, B, C; NégyzetesMátrix;
    Minneapolis, Zürich:    packed record
                                Terület: Real;
                                Lakosság: TermészetesSzám;
                                Főváros: Boolean
                            end;
```

A változókat sémáinkban a *Változó* EBNF metaazonosító jelöli.

Változó = *TeljesVáltozó* | *Elemváltozó* | *DinamikusVáltozó* | *Pufferváltozó*.

7.1. Teljes változók

Teljes változók a változóazonosítóval megjelölt változók.

TeljesVáltozó = *Változóazonosító*.

Példák teljes változóra:

```
Input
P1
Láthatóság
```

7.2. Elemváltozók

Egy strukturált változó elemei maguk is változók; az elemváltozóval egy strukturált változó valamelyik elemére hivatkozunk. Az elemváltozó szintaktikus szabályai a strukturált változó típusától függően változnak.

Elemváltozó = *IndexeltVáltozó* | *Mezőkifejezés*.

Egy strukturált változó valamely eleméhez való hozzáférés vagy az arra való hivatkozás egyúttal azt is jelenti, hogy magához a strukturált változóhoz is hozzá kell férnünk, ill. hivatkoznunk kell rá.

7.2.1. Indexelt változók

Az indexelt változók a tömbváltozók elemei. Egy tömbváltozó nem más, mint egy tömb típusú változó.

IndexeltVáltozó = *Tömbváltozó* "[" *Index* {"," *Index*} "]".
Index = *MegszámlálhatóKifejezés*.
Tömváltozó = *Változó*.

Az indexelt változóval a tömbnek azt az elemét érjük el, amelyet az indexkifejezés értéke megjelöl. A hivatkozás idején ennek az értéknek értékadás-kompatibilisnek kell lennie az indextípussal (l. a 6.5. szakaszt). A többszörös indexkifejezések kiértékelésének sorrendje rendszerfüggő.

Példák:

```
Kontrasztfokozat[12]
Kontrasztfokozat[I + J]
Láthatóság[Vörös, True]
```

Ha egynél több index áll a szögletes zárójelben, mint a fenti

```
Láthatóság[Vörös, True]
```

példában is, az egyszerűen a

```
Láthatóság[Vörös] [True]
```

jelölés rövidítése.

7.2.2. Mezőkifejezések

A mezőkifejezések a rekordváltozók mezőit jelölik. Rekordváltozó az olyan változó, amely rekord típusú.

Mezőkifejezés = [*Rekordváltozó* "."] *Mezőazonosító*.
Rekordváltozó = *Változó*.

A mezőkifejezés a mezőazonosítónak megfelelő mezőt jelöli meg; benne csak a rekordváltozó rekord típusához tartozó mezőazonosítók szerepelhetnek. A with utasításon belül (l. a 9.2.4. pontot) amely kilistázza a rekordváltozót, a rekordváltozó és a "." elhagyható.

Példák mezőkifejezésre:

```
Z.Re
Láthatóság[Vörös, True].Im
P2↑.AnyjaNeve
```

Ha egy rekordváltozó valamelyik változata érvénytelenné válik, az adott változat valamennyi eleme teljesen határozatlan lesz. Ha a változat részben nincs kijelölőmező, és az egyik változathoz tartozó valamelyik elemre hivatkozunk, akkor ez a változat válik érvényessé, az összes többi érvényét veszti. Hibát eredményez, ha egy változat érvénytelen, vagy azzá válik, miközben hozzá akarunk fémi vagy hivatkozunk valamelyik elemére. Ha a kijelölőmező határozatlan, a változat rész egyetlen változata sincs érvényben. A kijelölőmező nem lehet aktuális változóparaméter.

7.3. Dinamikus (azonosított) változók

A dinamikus változók jelölik a mutatóváltozók értékeivel megcímzett változókat. A mutatóváltozó egy mutató típusú változó.

DinamikusVáltozó = Mutatóváltozó " ".
Mutatóváltozó = Változó.

Ha egy dinamikus változóhoz hozzá akarunk fémi, ahhoz egyúttal a mutatóváltozóhoz is hozzá kell fémünk. Ilyenkor hibát eredményez, ha a mutatóváltozó határozatlan, vagy aktuális értéke nil. Hibát eredményez az is, ha egy változót kijelölő mutatóértéket megsemmisítünk, miközben érvényben marad az általa megcímzett változóra való hivatkozás.

Példák dinamikus változóra:

```
p1↑
p1↑.ApjaNeve↑
p1↑.Barátai↑↑
```

7.4. Pufferváltozók

Az állományváltozók nem mások, mint az állomány típusú változók. Minden állományváltozóhoz tartozik egy ún. pufferváltozó.

Pufferváltozó = Állományváltozó "↑".
Állományváltozó = Változó.

Ha az állományváltozó Text típusú, akkor a pufferváltozó a Char típusba tartozik; más esetekben a pufferváltozó típusa ugyanaz, mint az állományváltozó állomány típusának elemtípusa. A pufferváltozót az állományváltozó aktuális elemének eléréséhez használjuk. Ha egy állományváltozóhoz tartozó pufferváltozóra hivatkozás van érvényben, hibát okoz minden olyan művelet végrehajtása, amely ennek az állományváltozónak a sorrendjét, pozícióit vagy állapotát megváltoztatja. Ha egy pufferváltozóhoz akarunk hozzáférni, vagy arra hivatkozunk, akkor egyúttal a hozzá tartozó állományváltozót is elérjük, ill. hivatkozunk rá.

Az állományváltozók kezelésére szolgáló standard eljárásokat és függvényeket a 11.4., 11.5. szakaszban és a 12. fejezetben ismertetjük.

Ha egy F szöveg típusú állományban eoln(F) igazzá válik (l. a 11.5.2. pontot), akkor az F↑ pufferváltozó a szóköz (' ') karakter típusú értéket fogja fölvenni. F-ben tehát kizárólag eoln(F) segítségével lehet a sor vége jelet észlelni.

Példák pufferváltozóra:

```
Input↑
P1↑.Barátai↑
P1↑.Barátai↑↑.Gyermekei↑
```

8. Kifejezések

A *kifejezések* olyan számítási előírások, amelyek az illető kifejezés kiértékelésekor egy értéket szolgáltatnak, hacsak az az eset nem áll elő, hogy a kifejezés egy függvényt aktivál, és ezt az aktív állapotot egy goto utasítás megszakítja (l. a 9.1.3. pontot és a 10. fejezetet). Az így kapott érték a kifejezésben előforduló konstansok, határok és változók értékétől, továbbá a kifejezés által behívott műveletektől és függvényektől függ.

Kifejezés = *EgyszerűKifejezés* [*RelációsJel EgyszerűKifejezés*].
EgyszerűKifejezés = [*Előjel*] *Tag* {*AdditívMűveletiJel Tag* }.
Tag = *Tényező* {*MultiplikatívMűveletiJel Tényező* }.
Tényező = *ElőjelNélküliKonstans* | *Határazonosító* | *Változó* |
Halmazgenerátor | *Függvénykifejezés* |
"not" *Tényező* | "(" *Kifejezés* ")".

ElőjelNélküliKonstans = *ElőjelNélküliSzám* | *Karakterfüzér* |
Konstansazonosító | "nil".
Halmazgenerátor = "[" [*Elemleírás* { "," *Elemleírás* }] "]".
Elemleírás = *MegszámlálhatóKifejezés* [".." *MegszámlálhatóKifejezés*].
Függvénykifejezés = *Függvényazonosító* [*AktuálisParaméterlista*].

RelációsJel = "=" | "<>" | "<" | "<=" | ">" | ">=" | "in".
AdditívMűveletiJel = "+" | "−" | "or".
MultiplikatívMűveletiJel = "*" | "/" | "div" | "mod" | "and".

A megszámlálható kifejezés megszámlálható típusú kifejezés. A logikai és az egész kifejezés olyan kifejezés, amelynek típusa Boolean, ill. Integer.

MegszámlálhatóKifejezés = *Kifejezés*.
LogikaiKifejezés = *MegszámlálhatóKifejezés*.
EgészKifejezés = *MegszámlálhatóKifejezés*.

8.1. Operandusok

Egy tagon belül elhelyezkedő multiplikatív műveleti jelnek két operandusa van: a tagnak a műveleti jelet megelőző része és a jelet közvetlenül követő tényező. Egy egyszerű kifejezésben elhelyezkedő additív műveleti jelnek szintén két operandusa van: az egyszerű kifejezésnek az a része, amely műveleti jelet megelőzi, és a jelet közvetlenül követő tag. Egy relációjel két operandusa az azt közvetlenül megelőző és közvetlenül követő egyszerű kifejezés. Egy egyszerű kifejezésben elhelyezett előjel operandusa az előjelet közvetlenül követő tag. Egy tényezőben elhelyezkedő not operandusa a not-ot követő tényező.

Egy műveleti jel operandusai kiértékelésének sorrendje rendszerfüggő. Egy, a szabványhoz illeszkedő programon belül erre nézve semmilyen feltételezést nem kockáztathatunk meg. A rendszer működhet úgy, hogy a két operandus közül először a bal oldalit, de úgy is, hogy először a jobb oldalit értékeli ki, sőt, kiértékelheti a kettőt egyidejűleg is. Néha előfordul, hogy az egyik operandust egyáltalán nem is értékeli ki a másik operandus bizonyos értékei esetén. A (j * (i div j)) kifejezés értékét, ha a j értéke 0, az egyik rendszer tekintheti 0-nak, míg a másik a nullával való osztás miatt hibát jelez.

Egy tényező típusa az őt alkotó részek (pl. változók vagy függvények) típusából állapítható meg. Ha az alkotórészek résztartomány típusúak, akkor a tényező típusa a résztartomány törzstípusa lesz; ha az alkotórészek típusa olyan halmaz típus, amelynek alaptípusa egy résztartomány, akkor a tényező típusa olyan halmaz típus lesz, amelynek alaptípusa ennek a résztartománynak a törzstípusa; minden más esetben viszont a tényező típusa meg fog egyezni alkotórészeinek típusával.

A nil szimbólum minden mutató típushoz hozzátartozik és a nil értéket jelzi.

A halmazgenerátorok halmazértékeket jelölnek. Ha a halmazgenerátor nem tartalmaz elemleírást, akkor az üres halmazt jelöli, amely minden halmaz típushoz hozzá tartozó érték. A halmazérték elemeit minden más esetben le kell írni a halmazgenerátor elemleírásában. Egy halmazgenerátor elemleírásában szereplő valamennyi kifejezésnek ugyanahhoz a típushoz, a halmazgenerátor alaptípusához kell tartoznia. Egy halmazgenerátor típusa lehet tömörített és tömörítetlen és kompatibilis minden olyan halmaztípussal, amelynek alaptípusával a generátor alaptípusa kompatibilis.

Ha egy elemleírás egyetlen kifejezésből áll, akkor azt az elemet írja le, amelynek értéke a kifejezés által megadott érték. Egy a..b alakú elemleírás minden olyan x érték esetére definiál egy elemet, amelyre a <= x <= b. Ha a > b, akkor az a..b leírás nem definiál egyetlen elemet sem. Egy elemleíráson belül a kifejezések kiértékelésének és egy halmazgenerátoron belül az elemleírások kiértékelésének sorrendje rendszerfüggő.

Az egyetlen változóból álló tényező kiértékelése nem jelent mást, mint a változóra való hivatkozást, és eredménye a változó értéke lesz. Ilyen esetben hibát okoz, ha a változó határozatlan.

Az egyetlen függvénykifejezésből álló tényező kiértékelése a függvényazonosító által megjelölt függvény hívását jelenti (l. a 10.3. szakaszt). A formális paraméterek helyébe a megfelelő aktuális paraméterek kerülnek (l. a 11.3. szakaszt). Ha a hívás algoritmusa lefutott, a tényező értéke annak eredménye lesz; hibát eredményez, ha ez az eredmény határozatlan.

8.2. Műveleti jelek

A kifejezések összeállításának szabályai *precedenciasorrendet* határoznak meg a műveleti jelek négy osztálya között. A legmagasabb precedenciájú a not műveleti jel, ezután következnek az ún. multiplikatív műveleti jelek, majd az ún. additív műveleti jelek, végül a legalacsonyabb precedenciájúak, a relációs jelek. Azonos precedenciájú műveleti jelekből álló sorozat feldolgozása balról jobbra történik. A precedenciaszabályok tükröződnek a *Kifejezés*-ekre, *Egyszerű-Kifejezés*-ekre, *Tag*-okra és *Tényező*-kre vonatkozó (fenti) EBNF-szabályokban is.

A műveleti jeleket operandusaik és eredményük típusa szerint aritmetikai, logikai, halmaz- és relációs műveletek csoportjára is feloszthatjuk.

8.2.1. Aritmetikai műveleti jelek

Az aritmetikai műveleti jelek egész vagy valós operandusokra alkalmazhatók, és eredményül egész vagy valós számot adnak.

Az alábbiakban az egy operanduson ható műveleti jeleket, azaz az előjeleket foglaltuk össze

Jel	*Művelet*	*Az operandus típusa*	*Az eredmény típusa*
+	azonosság	Integer vagy Real	ugyanaz, mint az operandusoké
–	előjelváltás	Integer vagy Real	ugyanaz, mint az operandusoké

Az alábbiakban a két operandusra vonatkozó műveleti jeleket foglaltuk össze:

Jel	*Művelet*	*Az operandus típusa*	*Az eredmény típusa*
+	összeadás	Integer vagy Real	Integer vagy Real
–	kivonás	Integer vagy Real	Integer vagy Real
*	szorzás	Integer vagy Real	Integer vagy Real
/	osztás	Integer vagy Real	Real
div	osztás	Integer	Integer
mod	modulo	Integer	Integer

Az összeadás, a kivonás és a szorzás eredménye Integer típusú, ha mindkét operandus Integer típusú. Minden más esetben az eredmény Real típusú.

Ha y zérus, az x/y alakú tag kiértékelése hibát eredményez.

Az x div y alakú tag kiértékelése hibát eredményez, ha y zérus; minden más esetben az alábbi két feltételt kielégítő értéket eredményez:

(a) abs(x) – abs(y) < abs((x div y) * y) <= abs(x)

(b) x div y = 0, ha abs(x) < abs(y);
minden más esetben x div y pozitív, ha x és y előjele megegyezik, és negatív, ha x és y előjele különböző.

Az x mod y alakú tag kiértékelése hibára vezet, ha y nem pozitív; ha y pozitív, az eredmény olyan k egész szám, hogy x mod y eleget tegyen a következő relációnak:

$$0 <= x \bmod y = x - k * y < y$$

Az egészműveletek esetén, ha mindkét operandus és a helyes eredmény egyaránt a –Maxint..Maxint tartományba esik, akkor egy, a szabványhoz illeszkedő nyelvet megvalósító rendszerben meg kell kapnunk a helyes eredményt. Ha azonban akár az operandusok valamelyike, akár az eredmény kívülesik a –Maxint..Maxint tartományon, a nyelvet megvalósító rendszer maga dönt, hogy helyesen végrehajtja-e a műveletet vagy hibaként kezeli.

Minden, valós eredményt adó műveletet és standard függvényt (l. a 11.5. szakaszt) közelítőnek kell tekinteni. A valós műveletek és standard függvények pontosságát a rendszer definiálja.

8.2.2. Logikai műveleti jelek

A logikai műveleti jelek a következők:

Jel	*Művelet*	*Az operandus típusa*	*Az eredmény típusa*
not	logikai NEM	Boolean	Boolean
and	logikai ÉS	Boolean	Boolean
or	logikai VAGY	Boolean	Boolean

8.2.3. Halmazműveleti jelek

Az alábbiakban a halmazműveleti jeleket foglaltuk össze. A két operandus típusának mindig kompatibilisnak kell lennie (l. a 6.5. szakaszt). Az eredmény típusa tömörített, ha mindkét operandus típusa tömörített és tömörítetlen, ha mindkét operandus típusa ilyen.

Jel	*Művelet*	*Az operandus típusa*	*Az eredmény típusa*
+	egyesítés	T alaptípusú halmaz	T alaptípusú halmaz
−	differencia	T alaptípusú halmaz	T alaptípusú halmaz
*	metszet	T alaptípusú halmaz	T alaptípusú halmaz

8.2.4. Relációs jelek

A relációs jeleket az alábbi táblázatban foglaltuk össze. Az in relációjeltől eltekintve az operandusok típusának vagy kompatibilisaknak kell lenniük, vagy egyikük a Real, másikuk az Integer típus kell, hogy legyen. Az in esetében viszont az első (bal oldali) operandusnak olyan megszámlálható típusúnak kell lennie, amely kompatibilis a második operandus halmaztípusának alaptípusával.

Az x <= y kifejezés, ahol x és y halmaz, pontosan akkor lesz igaz, ha x minden eleme eleme y-nak is, azaz, ha x részhalmaza y-nak.

A kompatibilis füzérek közötti rendezést a Char típusú értékek közötti rendezésnek (l. a 6.1.2. pontot) megfelelően értelmezzük.

Jel	*Művelet*	*Az operandus típusa*	*Az eredmény típusa*
=	egyenlőség	egyszerű, mutató, halmaz vagy füzér	Boolean
<>	nemegyenlőség	egyszerű, mutató, halmaz vagy füzér	Boolean
<=	kisebb vagy egyenlő	egyszerű vagy füzér	Boolean
<=	tartalmazás	halmaz	Boolean
>=	nagyobb vagy egyenlő	egyszerű vagy füzér	Boolean
>=	tartalmazás	halmaz	Boolean
<	kisebb	egyszerű vagy füzér	Boolean
>	nagyobb	egyszerű vagy füzér	Boolean
in	eleme	diszkrét és halmaz	Boolean

Példák tényezőre:

```
X
15
(W + X + Y)
sin(X + Y)
[Piros, Sárga, Zöld]
[1, 5, 10..19, 60]
not P
```

Példák tagra:

```
X * Y
I/(1−I)
Q and not P
(X <= Y) and (Y < W)
```

Példák egyszerű kifejezésre:

```
X + Kontrasztfokozat [2 * L]
-X
P or Q
Árnyalat1 + Árnyalat2
I*J + 1
```

Példák kifejezésre:

```
X = 1.5
P <= Q
(I < J) = (J < K)
Sárga in Árnyalat1
```

9. Utasítások

Az utasítások végrehajthatónak nevezett algoritmikus tevékenységet jelölnek. Egy utasítást megelőzhet egy címke, amelyre a goto utasítások hivatkozhatnak. Az utasításokat utasításrészekbe gyűjtjük össze.

Utasítás = [Címke ":"] (EgyszerűUtasítás | StrukturáltUtasítás).
Utasításrész = ÖsszetettUtasítás.

9.1. Egyszerű utasítások

Egyszerűnek nevezzük az olyan utasítást, amelynek egyetlen valódi része sem alkot utasítást. Az üres utasítás semmiféle szimbólumot nem tartalmaz, és nem jelöl semmilyen tevékenységet.

EgyszerűUtasítás = ÜresUtasítás | ÉrtékadóUtasítás | Eljárásutasítás | GotoUtasítás.
ÜresUtasítás = .

9.1.1. Értékadó utasítások

Az értékadó utasítás segítségével hozzáférhetünk egy változóhoz vagy egy függvényhívás eredményéhez, és ennek aktuális értékét egy adott kifejezés kiértékelésével kapott értékkel helyettesíthetjük.

ÉrtékadóUtasítás = (Változó | Függvényazonosító) ":=" Kifejezés.

Fontos, hogy a kifejezés értéke értékadás-kompatibilis legyen a változó vagy a függvényazonosító típusával (l. a 6.5. szakaszt). A rendszertől függ, hogy a változót, ill. a függvényhívás eredményét a kifejezés kiértékelése előtt vagy után hívja-e be. A változó elérése a változóra való hivatkozással történik, amely az új érték hozzárendelése után érvényét veszti.

Példák értékadó utasításra:

```
X  := Y + Kontrasztfokozat [31]
P  := (1 <= I) and (I < 100)
I  := sqr(K) - (I*J)
Árnyalat2 := [Kék, succ(C)]
```

9.1.2. Eljárásutasítások

Az eljárásutasítások az eljárásazonosító által megjelölt eljárás végrehajtását írják elő. Egy eljárásutasítás tartalmazhatja azon *aktuális paraméterek listáját*, amelyek az eljárásdeklarációban (l. a 11.1. szakaszt) definiált, megfelelő *formális paramétereket* fogják helyettesíteni.

Eljárásutasítás = *Eljárásazonosító* [*AktuálisParaméterlista* | *WriteParaméterlista*].

Ha az eljárásazonosító a standard Write és Writeln eljárások valamelyikét jelöli, akkor az aktuális paramétereknek eleget kell tenniük a *WriteParaméterlista* esetére előírt szintaktikus szabályoknak. Bármilyen más standard eljárást jelöl az eljárásazonosító, az aktuális paramétereknek meg kell felelniük a 11.4. szakaszban és a 12. fejezetben felállított szabályoknak.

Példák eljárásutasításra:

```
Következő
Transzponált(A,N,N)
Felezés(Fct, −1.0, +1.0, X)
Writeln(Output, 'Cím')
```

9.1.3. A goto utasítás

A goto utasítás azt jelzi, hogy a feldolgozást a program egy másik pontján, mégpedig a címke által jelzett (l. a 10.1. és 10.3. szakaszt) programpontnál kell folytatni.

GotoUtasítás = "goto" *Címke*.

Egy címkével ellátott utasításnak és minden, erre a címkére hivatkozó goto utasításnak eleget kell tennie az alábbi két szabály egyikének:

(a) Az utasítás vagy tartalmazza a goto utasítást, vagy egy, a goto utasítást tartalmazó utasítássorozathoz tartozik (l. a 9.2. szakaszt).
(b) Az utasítás azon blokk utasításrészének összetett utasítását alkotó utasítássorozathoz tartozik, amely blokkban a címkét deklaráltuk, a goto utasítást pedig ugyanezen blokk eljárás- és függvénydeklarációs része tartalmazza (l. 10.1. szakaszt).

Ezek a szabályok veszik elejét annak, hogy a vezérlést kívülről adjuk át egy strukturált utasítás, eljárás vagy függvény belsejébe. Az első szabály azt is megtiltja, hogy a goto utasítással egy feltételes utasítás egyik „elágazásáról" a másikra adjuk át a vezérlést.

Ha a programpont és a goto utasítás nem ugyanabban az utasításrészben található, minden hívás érvényét veszti (l. a 10.3. szakaszt), amely nem tesz eleget az alábbi két feltétel valamelyikének.

(a) A hívás tartalmazza a programpontot.
(b) A hívás egy másik, olyan hívás hívási pontját tartalmazza, amely nem veszíti érvényét (amely tehát eleget tesz e két feltétel valamelyikének).

9.2. Strukturált utasítások

A strukturált utasítások más utasításokból épülnek fel. Ezeknek a részutasításoknak a végrehajtási módja szerint osztjuk négy csoportra a strukturált utasításokat: az összetett utasításokat alkotó részutasításokat az előre megadott sorrendben kell végrehajtani, a feltételes utasítás részutasításait csak abban az esetben, ha egy adott feltétel teljesül, a ciklusutasításokéit többször, a with utasításéit pedig kiterjesztett hatáskörben.

StrukturáltUtasítás = *ÖsszetettUtasítás* | *FeltételesUtasítás* | *Ciklusutasítás* | *WithUtasítás*.

Utasítássorozatnak utasítások egy olyan sorozatát nevezzük, amelyeket a leírás sorrendjében kell végrehajtani, hacsak egy goto utasítás másképp nem intézkedik.

Utasítássorozat = *Utasítás* { ";" *Utasítás* }.

Utasítássorozatokat az összetett utasításokban (l. a 9.2.1. pontot) és a repeat utasításban (l. a 9.2.3.2. alpontot) alkalmazunk.

9.2.1. Összetett utasítások

Az összetett utasításokat alkotó részutasításokat az előre megadott sorrendben kell végrehajtani. Az utasítást alkotó utasítássorozatot a begin és az end alapszavak zárják közre.

ÖsszetettUtasítás = "begin" *Utasítássorozat* "end".

Példák összetett utasításra:

```
begin   end
begin   W := X; X := Y; Y := W   end
```

9.2.2. Feltételes utasítások

A feltételes utasítások az őket alkotó részutasítások közül választják ki az egyiket, és végrehajtják.

FeltételesUtasítás = *IfUtasítás* | *CaseUtasítás*.

9.2.2.1. Az if utasítás

Az if utasítás arról intézkedik, hogy a then alapszót követő utasítás végrehajtására csak akkor kerüljön sor, ha a logikai kifejezés értéke igaz. Ellenkező esetben az else alapszót követő utasítást kell végrehajtani, ha egyáltalán van ilyen.

IfUtasítás = "if" *LogikaiKifejezés* "then" *Utasítás* ["else" *Utasítás*].

Megjegyzés: Az

```
if e1 then if e2 then s1 else s2
```

utasítás szerkezetéből adódó szintaktikus kétértelműség feloldására a fenti konstrukciót az

```
if e1 then
    begin if e2 then s1 else s2 end
```

utasítással tekintjük egyenértékűnek.

Példák if utasításra:

```
if X < 1.5 then W := X + Y else W := 1.5
if P1 <> nil then P1 = P1↑.ApjaNeve
```

9.2.2.2. A case utasítás

A case utasítás egy megszámlálható típusú kifejezésből (az ún. esetindexből) és egy utasításlistából áll, amelynek minden elemét egy vagy több, az esetindexszel megegyező típusú konstans előzi meg. Hatására annak az utasításnak a végrehajtására kerül sor, amely előtt az esetindex aktuális értéke áll; hibát eredményez, ha egyik utasítás előtt sem áll olyan konstans, amely ezt az aktuális értéket jelöli. Minden érték legfeljebb egy esetkonstanssal egyezhet meg.

CaseUtasítás = "case" *EsetIndex* "of" *Eset* {";" *Eset*} [";"] "end".
Esetindex = *MegszámlálhatóKifejezés.*
Eset = *Konstans* {"," *Konstans*} ":" *Utasítás.*

Példák case utasításra:

```
Case MűveletiJel of
    Plusz:   W := X + Y;
    Mínusz:  W := X - Y;
    Szor:    W := X*Y
end

case I of
    1:   Y := sin(X);
    2:   Y := cos(X);
    3:   Y := exp(X);
    4:   Y := ln(X)
end
```

9.2.3. Ciklusutasítások

A ciklusutasítások bizonyos utasítások ismételt végrehajtásáról gondoskodnak. Ha a szükséges ismétlések számát előre, még az ismétlések megkezdése előtt ismerjük, akkor általában a for utasítás alkalmazása a legcélravezetőbb; más esetekben inkább a while vagy a repeat utasítást használjuk.

Ciklusutasítás = *WhileUtasítás* | *RepeatUtasítás* | *ForUtasítás.*

9.2.3.1. A while utasítás

WhileUtasítás = "while" *LogikaiKifejezés* "do" *Utasítás.*

A program ismételten végrehajtja az utasítást mindaddig, amíg a kifejezés értéke hamis nem lesz. Ha a kifejezés értéke már kezdetben hamis, az utasítás végrehajtására egyáltalán nem kerül sor. A

```
while B do S
```

utasítás ekvivalens az

```
if B then begin S; while B do S end
```

utasítással, hacsak S címkézett utasítást nem tartalmaz.

Példák while utasításra:

```
while Kontrasztfokozat[I] < X do I := succ(I)
```

```
while I > 0 do
    begin
        if odd(I) then Y := Y * X;
        I := I div 2
        X := sqr (X)
    end
```

```
while not eof(F) do
    begin P(F↑); Get(F) end
```

9.2.3.2. A repeat utasítás

RepeatUtasítás = "repeat" *Utasításssorozat* "until" *LogikaiKifejezés.*

A program mindaddig ismételten végrehajtja az utasítássorozatot, amíg a kifejezés igazzá nem válik. A

```
repeat S until B
```

utasítás ekvivalens a

```
begin S; if not B then repeat S until B end
```

utasítással, hacsak S címkézett utasítást nem tartalmaz.

Példák repeat utasításra:

```
repeat K := I mod J; I := J; J := K until J = 0
repeat
    P(F↑);
    Get(F)
until eof(F)
```

9.2.3.3. A for utasítás

A for utasítás azt jelzi, hogy egy utasítást ismételten végre kell hajtani, miközben egy, a for utasítás *ciklusváltozójának* nevezett változó értékek egy növő vagy fogyó sorozatát futja be.

ForUtasítás = "for" *Ciklusváltozó* ":=" *Kezdőérték* ("to" | "downto") *Végérték*
"do" *Utasítás*.
Ciklusváltozó = *Változóazonosító*.
Kezdőérték = *MegszámlálhatóKifejezés*.
Végérték = *MegszámlálhatóKifejezés*.

A ciklusváltozónak lokálisnak kell lennie arra a blokkra nézve, amelynek utasításrésze a for utasítást tartalmazza (l. a 10.2. szakaszt) és a kezdő- és végérték típusával kompatibilis megszámlálható típushoz kell tartoznia.

Az S utasításról azt mondjuk, hogy *veszélyezteti* a V változót, ha az alábbi négy eset valamelyike áll fenn:

(a) S olyan értékadó utasítás, amely V-hez rendel hozzá valamilyen értéket.
(b) S-ben V aktuális változóparaméterként szerepel (l. a 11.3.2.2. alpontot).
(c) S a standard Read vagy Readln eljárást hívó eljárásutasítás, és V ennek aktuális paramétere.
(d) S egy for utasítás és V a ciklusváltozója.

A for utasításon belül egyetlen utasítás sem veszélyeztetheti a ciklusváltozót. Nem tartalmazhat a ciklusváltozót veszélyeztethető utasítást egyetlen olyan eljárás vagy függvény sem, amelyet ugyanabban a blokkban deklaráltunk, mint magát a ciklusváltozót. Ezek a szabályok azt biztosítják, hogy a ciklusmag utasítása ne módosíthassa a ciklusváltozó értékét.

Legyen T1 és T2 a V-vel megegyező típusú, P pedig Boolean típusú új változó (tehát olyan, amihez egyébként nem lehet hozzáférni). Ekkor a magyarázatként szolgáló megjegyzésektől eltekintve a

```
for V := e1 to e2 do S
```

utasítás ekvivalens a

```
begin
    T1 := e1; T2 := e2;
    if T1 <= T2 then
        begin
            { T2-nek értékadás-kompatibilisnak kell }
            { lennie a V típusával }
            V := T1; P := false;
```

```
                repeat
                    S;
                    if V = T2 then P := true else V := succ(V)
                until P
            end
        { V határozatlan }
    end
```

utasítással, a

```
    for V := e1 downto e2 do S
```

utasítás pedig a

```
    begin
        T1 := e1; T2 := e2;
        if T1 >= T2 then
            begin
                { T2-nek értékadás-kompatibilisnak kell }
                { lennie a V típusával }
                V := T1; P := false;
                repeat
                    S;
                    if V = T2 then P := true else V := pred(V)
                until P
            end
        { V határozatlan }
    end
```

utasítással.

Példák for utasításra:

```
    for I := 1 to 63 do
        if Kontrasztfokozat[I] > 0.5 then write ('*')
            else write (' ')

    for I := 1 to n do
        for J := 1 to n do
            begin
                X := 0;
                for K := 1 to n do X := X + A[I,K] * B[K,J];
                C[I,J] := X
            end

    for Fény := Vörös to pred(Fény) do
        if Fény in Árnyalat2 then Q(Fény)
```

9.2.4. A with utasítás

A with utasítás hozzáférést biztosít a listáján szereplő rekordváltozóknak, egy utasítás végrehajtásának erejéig. A hivatkozás addig marad érvényben, amíg ezen utasítás végrehajtása tart.

WithUtasítás = "with" *RekordVáltozólista* "do" *Utasítás*.
RekordVáltozólista = *Rekordváltozó* { "," *Rekordváltozó* }.

A with utasítás hatására a benne elhelyezett listán szereplő minden rekordváltozóra a hozzá tartozó összes mezőazonosító hatásköre úgy bővül, hogy kiterjedjen a belső utasításra is (l. a 10.2. szakaszt). E kibővült hatáskörön belül a mezőazonosítók anélkül szerepelhetnek egy mezőkifejezésben, hogy a rekordváltozót ismét megadnók. Ezek a mezőazonosítók ilyenkor a hivatkozott változó megfelelő mezejét jelzik.

A

```
with r1, r2, ..., rn do S
```

jelölés egyszerűen a

```
with r1 do
    with r2 do
        ...
            with rn do S
```

jelölés rövidítése.

Példa with utasításra:

```
with Dátum do
    if Hónap = 12 then
        begin Hónap := 1; Év := succ (Év) end
    else Hónap := succ (Hónap)
```

Ez az utasítás ekvivalens az

```
if Dátum.Hónap = 12 then
    begin Dátum.Hónap := 1;
        Dátum. Év := succ (Dátum.Év)
    end
else Dátum.Hónap := succ (Dátum.Hónap)
```

utasítással.

10. Blokkok, hatáskör, hívások

A *blokkok* azok az alapvető egységek, amelyekből a programok (l. a 13. fejezetet), továbbá az eljárások és a függvények (l. a 11. fejezetet) fölépülnek. A *hatáskörre* vonatkozó szabályok a program statikus (szöveg szerinti) szerkezete alapján határozzák meg, hogy egy adott

helyen bevezetett azonosító karaktersorát a programon belül hol használhatjuk. A hívási szabályok a program dinamikus (végrehajtás szerinti) szerkezete alapján azt állapítják meg, hogy egy adott azonosító vagy címke milyen objektumot (például változót) jelöl.

10.1. Blokkok

Egy blokk definíciós és deklarációs részekből áll, amelyek mindegyike lehet üres is, továbbá egy utasításrészből.

Blokk = *CímkedeklarációsRész*
KonstansdefiníciósRész
TípusdefiníciósRész
VáltozódeklarációsRész
EljárásÉsFüggvénydeklarációsRész
Utasításrész.

A címkedeklarációs rész egy vagy több címkét vezet be, amelyek mindegyike az utasításrészben szereplő valamelyik utasítás előtt áll.

CímkedeklarációsRész = ["label" *Számjegysorozat* {"," *Számjegysorozat* } ";"].
Címke = *Számjegysorozat*.

Egy címke megjelenési formája az az egész szám, amelyet a címkét alkotó számjegysorozat a szokásos tízes számrendszerbeli ábrázolásmód szerint jelöl. Ez az érték nem lehet nagyobb 9999-nél.

10.2. Hatáskör

Egy definíció vagy egy deklaráció bevezeti egy azonosító vagy egy címke karaktersorát, és ehhez speciális jelentést rendel (például azt, hogy a bevezetett karaktersor egy változó azonosítójáé). A programnak azt a részét, amelyben ennek a karaktersornak, valahányszor előfordul, szükségképpen ez a jelentése, összefoglalóan az adott bevezetés (definíció vagy deklaráció) *hatáskörének* nevezzük. Egy adott karaktersornak – bevezetésének hatáskörén belül – először magában a karaktersor bevezetésében kell előfordulnia. Ez alól a szabály alól egyetlen kivétel van, nevezetesen az, hogy egy típusazonosító karaktersor egy mutató típus főtípusaként (l. a 6.3. szakaszt) az adott karaktersor bevezetését tartalmazó típusdefiníciós rész bármely pontján előfordulhat.

Amint azt ismertetni fogjuk, minden bevezetés a program valamely részére érvényes. Egy bevezetés hatásköre a programnak ez a része, elhagyva belőle azokat a részeket, amelyekre ugyanazon karaktersor egy másik deklarációja érvényes.

Az alábbi deklarációk arra a blokkra érvényesek, amelyben az adott deklaráció előfordul: egy címke a blokk címkedeklarációs részében, egy konstansazonosító a blokk konstansdefiníciós részében vagy egy felsorolt típusban, egy típusazonosító a blokk típusdefiníciós részében, egy változóazonosító a blokk változódeklarációs részében, egy eljárásazonosító egy blokkbeli eljárásdeklarációban (l. a 11.1. szakaszt), és egy függvényazonosító egy blokkbeli függvénydeklarációban (l. a 11.2. szakaszt). Ezeket a címkéket és azonosítókat az adott blokkra nézve *lokálisaknak* nevezzük.

A standard előredefiniált és -deklarált azonosítók implicit bevezetése minden programra érvényes.

Egy mezőazonosítónak egy rekord típuson belüli deklarációja a program alábbi részeinek mindegyikére érvényes:

(a) magára a rekord típusra;
(b) az olyan with utasítások belső utasításaira, amelyek rekordváltozója ehhez a rekord típushoz tartozik; és
(c) az olyan mezőkifejezések mezőazonosító részére, amelyeknek rekordváltozó része ehhez a rekord típushoz tartozik.

A *(c)* esetben a mezőazonosító rész minden más hatáskörön kívül esik.

Egy paraméterazonosító paraméterlistán való deklarációja (l. a 11.3.1. pontot) a paraméterlistára érvényes. Ha pedig a paraméterlista egy eljárásdeklaráció eljárásfejéhez vagy egy függvénydeklaráció függvényfejéhez tartozik, akkor a paraméterazonosítónak megfelelő változóazonosító, határazonosító, eljárásazonosító vagy függvényazonosító bevezetése a szóban forgó eljárás- vagy függvénydeklaráció blokkjára érvényes.

10.3. Hívások

Egy program (l. a 13. fejezetet), eljárás vagy függvény (l. a 11. fejezetet) hívása a program, eljárás vagy függvény blokkjának hívása.

Azt mondjuk, hogy egy blokk hívása a következő objektumok *létezését* feltételezi, ezek mindegyike csak addig létezik, amíg a hívás érvényben van.

(a) Egy *algoritmust*, amelyet a blokk utasításrésze ad meg; az algoritmus a blokk hívásakor veszi kezdetét, és az algoritmus végrehajtásának befejezése érvényteleníti a hívást. (A hívást egy goto utasítás is érvénytelenítheti; l. a 9.1.3. pontot).
(b) Minden olyan címkéhez, amely a blokk utasításrészének valamelyik utasítása előtt áll, az algoritmuson belül egy *programpontot*. Valahányszor egy, a híváson belüli goto utasításban előfordul ez a címke, erre a programpontra utal.
(c) Minden, a blokkra nézve lokális változóazonosítóhoz egy *változót;* az algoritmus beindulásakor ez a változó teljesen határozatlan, hacsak a változóazonosító nem programparaméter. Valahányszor ez a változóazonosító megjelenik a híváson belül, ezt a változót jelöli.
(d) Minden, a blokkra nézve lokális eljárásazonosítóhoz egy *eljárást;* az eljárás blokkja és formális paraméterei azok, amelyek az eljárásdeklarációban szerepelnek. Valahányszor ez az eljárásazonosító előfordul a hívásban, mindig ezt az eljárást jelzi.
(e) Minden, a blokkra nézve lokális függvényazonosítóhoz egy *függvényt;* a függvény blokkja, formális paraméterei és eredménytípusa azok, amelyek a függvényazonosítót bevezető függvénydeklarációban szerepelnek. Valahányszor ez a függvényazonosító előfordul a híváson belül, ezt a függvényt jelenti.
(f) Minden olyan változóazonosítóhoz, amely a blokk egy formális értékparaméter-azonosítója, egy *változót;* az algoritmus beindulásakor a változó fölveszi az eljárást vagy függvényt hívó eljárásutasítás, ill. függvénykifejezés megfelelő aktuális paraméterének értékét. Valahányszor ez a változóazonosító előfordul a híváson belül, ezt a változót jelöli.
(g) Minden olyan változóazonosítóhoz, amely a blokk egy formális változóparaméterének azonosítója, egy *hivatkozást;* a hivatkozás arra a változóra történik, amelyet az algoritmus beindulásakor a megfelelő aktuális paraméter kijelöl. Valahányszor ez a változóazonosító a híváson belül előfordul, a hivatkozott változót jelöli.
(h) A blokk minden formális eljárás- vagy függvényparaméteréhez egy eljárásra, ill. függvényre való *hivatkozást;* a hivatkozás arra az eljárásra vagy függvényre történik, amelyet az algoritmus beindulásakor a megfelelő aktuális paraméter kijelöl. Valahányszor

a híváson belül ez az eljárásazonosító vagy függvényazonosító előfordul, ezt az eljárást ill. függvényt jelöli.

(i) Ha a hívott blokk függvényblokk, egy *eredményt*, amely az algoritmus beindulásakor határozatlan.

Egy eljárás vagy függvény blokkjának hívása esetén azt mondjuk, hogy az az eljárást, ill. függvényt tartalmazó hívás *belsejében* van. Ha egy A hívás egy B hívás belsejében van, akkor A minden más olyan hívásnak is a *belsejében* van, amelynek B a belsejében van.

Egy algoritmusban előforduló eljárásutasítást vagy függvénykifejezést, amely egy blokkot hív, ezen hívás *belépési (hívási) pontjának* nevezünk.

11. Eljárások és függvények

Eljárásoknak és függvényeknek a program azon részeit nevezzük, amelyeket eljárásutasítások (l. a 9.1.2. pontot), ill. függvénykifejezések (l. a 8.1. szakaszt) hívnak. A programozó szükség szerint deklarálhat új eljárásokat és függvényeket. Az eljárásdeklarációkat és a függvénydeklarációkat az eljárás- és függvénydeklarációs részbe gyűjtjük össze.

EljárásÉsFüggvénydeklarációsRész = { *(Eljárásdeklaráció* | *Függvénydeklaráció)* ";" }.

Ezen túlmenően a Pascal nyelvet megvalósító minden rendszernek lehetőséget kell adnia több, „előredeklarált" eljárás és függvény alkalmazására. Mint az ilyen esetekben általában, feltételezzük, hogy ezeknek a deklarált hatásköre teljes egészében tartalmazza a szóban forgó programot. Nem okoz tehát bajt, ha a program valamely deklarációja ugyanezt az azonosítót másként definiálja.

11.1. Eljárásdeklarációk

Az eljárásdeklaráció célja, hogy bevezessen egy eljárásazonosítót és hozzárendeljen ehhez az azonosítóhoz egy blokkot, továbbá esetleg egy formális paraméterlistát. Az eljárásazonosítót és a formális paraméterlistát az eljárásdeklaráció eljárásfeje vezeti be.

Az eljárást deklarálhatjuk csupán egyetlen eljárásdeklarációval, amely az eljárásfejből és a blokkból áll. Az eljárások deklarálásának ez a leggyakoribb formája.

Egy eljárás azonban deklarálható „előzetes deklarációval" is: az első eljárásdeklaráció egy eljárásfejből és a forward direktívákból áll, az ugyanazon eljárás- és függvénydeklarációs részben elhelyezett második deklaráció pedig egy eljárásazonosításból és a blokkból. Az eljárásazonosításban elhelyezkedő eljárásazonosítónak meg kell egyeznie az első deklarációban bevezetett azonosítóval. Megjegyezzük, hogy a formális paraméterlistát, ha egyáltalán van, a második deklarációban nem szerepeltetjük.

Eljárásdeklaráció = *Eljárásfej* ";" *Blokk* | *Eljárásfej* ";" *Direktíva* |
Eljárásazonosítás ";" *Blokk*.
Eljárásfej = "procedure" *Azonosító* [*FormálisParaméterlista*].
Eljárásazonosítás = "procedure" *Eljárásazonosító*
Eljárásazonosító = *Azonosító*.

Ha egy eljárásazonosító azon a blokkon belül, ahol deklaráltuk, egy eljárásutasításban szerepel, ez az eljárás rekurzív végrehajtását eredményezi.

Példa eljárásokat tartalmazó eljárás- és függvénydeklarációs részre:

```
procedure ReadInteger(var F: Text; var X: Integer);
var S: TermeszetesSzam;
begin while F^<>' ' do
            Get(F);
            S:=0;
      while F^ in ['0'..'9'] do
            begin S:=10*S+(ord(F^)-ord('0'));
                  Get(F)
            end;
      X:=S
end {ReadInteger};

procedure Felezes(function F(X: Real): Real;A,B: Real;
                  var Z: Real);
var M: Real;
begin {tegyuk fel,hogy F(A)<0 es F(B)>0}
      while abs(A-N)>1e-10*abs(A) do
            begin M:=(A+B)/2.0;
                  if F(M)<0
                  then A:=M
                  else B:=M
            end;
      Z:=M
end {Felezes};

procedure LNKO(M,N: Integer var X,Y,Z: Integer);
{M es N legnagyobb kozos osztoja X, felteve,
 hogy M>=0 es N>0}
{Altalanositott euklideszi algoritmus}

var A1,A2,B1,B2,C,D,Q,R: Integer;
begin A1:=0; A2:=1;
      B1:=1; B2:=0;
      C:=M; D:=N;
      while D<>0 do
            begin
{A1*M+B1*N=D, A2*M+B2*N=C es LNKO (C,D)=LNKO (M,N)}
                  Q:=C div R; R:=C mod D;
                  A2:=A2-Q*A1; B2:=B2-Q*B1;
                  C:=D; D:=R;
                  R:=A1; A1:=A2; A2:=R;
                  R:=B1; B1:=B2; B2:=R
            end;
      X:=C; Y:=A2; Z:=B2
      {X=LNKO(M,N)=Y*M+Z*N}
end {LNKO};
```

11.2. Függvénydeklarációk

A függvénydeklaráció célja, hogy bevezessen egy függvényazonosítót és hozzárendeljen ehhez az azonosítóhoz egy eredménytípust, egy blokkot és esetleg egy formális paraméterlistát. A függvényazonosítót, az eredmény típusát és a formális paraméterlistát a függvénydeklaráció függvényfeje vezeti be.

A függvényt deklarálhatjuk csupán egyetlen függvénydeklarációval, amely a függvényfejből és a blokkból áll. A függvények deklarálásának ez a leggyakoribb formája.

Egy függvény azonban deklarálható „előzetes deklarációval" is: az első függvénydeklaráció a függvényfejből és a forward derektívákból áll, az ugyanazon eljárás- és függvénydeklarációs részben elhelyezett második deklaráció pedig egy függvényazonosításból és a blokkból. A függvényazonosításban elhelyezkedő függvényazonosítónak meg kell egyeznie az első deklarációban bevezetett azonosítóval. Megjegyezzük, hogy a formális paraméterlistát, ha egyáltalán van, továbbá az eredmény típusát a második deklarációban nem szerepeltetjük.

Függvénydeklaráció = *Függvényfej* ";" *Blokk* | *Függvényfej* ";" *Direktíva* |
Függvényazonosítás ";" *Blokk*.
Függvényfej = "function" *Azonosító* [*FormálisParaméterlista*] ":" *Eredménytípus*.
Eredménytípus = *MegszámlálhatóTípusAzonosító* | *ValósTípusAzonosító* |
MutatóTípusAzonosító.
Függvényazonosítás = "function" *Függvényazonosító*.
Függvényazonosító = *Azonosító*.

Egy függvénydeklaráció blokkjának legalább egy értékadó utasítást kell tartalmaznia, amely a függvényazonosítóhoz értéket rendel. Ha egy függvényazonosító azon a blokkon belül, ahol deklaráltuk egy függvénykifejezésben szerepel, ez a függvény rekurzív végrehajtását eredményezi.

Példa függvényt tartalmazó eljárás- és függvénydeklarációs részre:

```
function Gyok(X: Real): Real;
{Newton-modszer}

var X0,X1: Real;

begin X1:=X; {X>1, Newton-modszer}
      repeat X0:=X1;
             X1:=(X0+X/X0)0.5
      until abs(X1-X0)<Epsz*X1;
      gyok:=X0
end {gyok};
```

```
function Max(A: vektor; N: Integer): Real;
{Az A[1],...,A[N] elemek kozul a maximalis
 erteket keressuk}

var X: Real;
    I: Integer;

begin X:=A[1];
      for I:=2 to N do
          begin {X=Max(A[1],...,A[I-1])}
                if X<A[I]
                then X:=A[I]
          end;
          {X=Max(A[1],...,A[N])}
          Max:=X
end {Max};

function LNKO(M,N: TerneszetesSzam): TermeszetesSzam;
{M es N Legnagyobb kozos osztojanak meghatarozasa.}

begin if N=0
      then LNKO:=M
      else LNKO:=LNKO(N,M mod N)
end;


function Hatvany(X: Real; Y: TermeszetesSzam): Real;
{X**Y kiszamitasa}

var W,Z: Real;

    I: TermeszetesSzam;
begin W:=X;
      Z:=1;
      I:=Y;
      while I>0 do
        \    begin {Z*W**I=X**Y}
                   if odd(I)
                   then Z:=Z*W;
                   I:=I div 2;
                   W:=Sqr(W)
             end;
             {Z=X**Y}
      Hatvany:=Z
end {Hatvany};
```

11.3. Paraméterek

A paraméterek teszik lehetővé, hogy egy eljárás vagy függvény minden hívása azokkal az objektumokkal (értékekkel, változókkal, eljárásokkal és függvényekkel) dolgozzék, amelyeket a belépési (hívási) pontban (l. a 10.3. szakaszt) egy aktuális paraméterlista sorol fel. Az eljárás- vagy függvényfejben elhelyezett formális paraméterlista határozza meg, hogy ezeket az objektumokat az eljárás vagy a függvény blokkjában milyen azonosítók jelzik, továbbá azt, hogy az aktuális paramétereknek milyen tulajdonságúaknak és típusúaknak kell lenniük.

A standard eljárások és függvények aktuális paraméterei nem mindig tesznek eleget a szokásos eljárásokra és függvényekre vonatkozó előírásoknak (l. a 11.4. és 11.5. szakaszt, valamint a 12. fejezetet).

11.3.1. Formális paraméterlisták

FormálisParaméterlista = "(" *FormálisParaméterrész*
{ ";" *FormálisParaméterrész*.} ")".
FormálisParaméterrész = *Értékparaméterspecifikáció* | *Változóparaméterspecifikáció* |
Eljárásparaméterspecifikáció |
Függvényparaméterspecifikáció.

Egy formális paraméterrész érték-, változó-, eljárás- vagy függvényparamétereket ad meg.

11.3.1.1. Formális érték- és változóparaméterek

Egy érték- vagy változóparaméter specifikáció az azonosítólistáján szereplő valamennyi azonosítót változóazonosítóként határozza meg. Ha a specifikációban egy típusazonosító is szerepel, az a változóazonosítók közös típusát jelöli. Ha illeszkedőtömb-sémát is tartalmaz, akkor a változóazonosítókat illeszkedőtömb-paramétereknek nevezzük, és típusuk az aktuális paraméter típusától függ. Egy adott híváson belül az egyazon formális paraméterrészben definiált valamennyi formális paraméter ugyanolyan típusú.

Megjegyzés: Az illeszkedőtömb-sémák alkalmazására nem minden, a Pascal nyelvet megvalósító rendszerben van lehetőség. A Pascal nyelvet 0 szinten megvalósító rendszerekben nem alkalmazhatóak, az 1 szinten megvalósítóakban igen.

Értékparaméterspecifikáció = *Azonosítólista* ":" (*Típusazonosító* |
Illeszkedőtömbséma).

Változóparaméterspecifikáció = "var" *Azonosítólista* ":" (*Típusazonosító* |
Illeszkedőtömbséma).

Illeszkedőtömbséma = *TömörítettIlleszkedőtömbséma* |
TömörítetlenIlleszkedőtömbséma.

TömörítettIlleszkedőtömbséma = "packed" "array" "[" *Indextípusspecifikáció* "]"
"of" *Típusazonosító*.

TömörítetlenIlleszkedőtömbséma = "array" "[" *Indextípusspecifikáció* {";"
Indextípusspecifikáció } "]"
"of" (*Típusazonosító* | *Illeszkedőtömbséma*).

Indextípusspecifikáció = *Azonosító* ".." *Azonosító* ":"
MegszámlálhatóTípusAzonosító.

Határazonosító = *Azonosító*.

Az indextípus-specifikációban megadott két azonosító a megszámlálhatótípus-azonosító által megjelölt típusú határazonosítók. Az

array [Alsó1..Felső1: T1; Alsó2..Felső2: T2] of T

illeszkedőtömb-séma egyszerűen az

array [Alsó1..Felső1: T1] of array [Alsó2..Felső2: T2] of T

séma rövidítése.

Példa illeszkedőtömb-paramétert bemutató függvénydeklarációra:

```
function Max(A: array[L..H: Integer] of Real;
             N: Integer): Real;
{Valasszuk ki az A[L],..,A[N] elemek kozul a legnagyobbat}

var X: Real;
    I: Integer;

begin X:=A[L];
      for I:=succ(L) to N do
          begin {X=Max A[L],...,A[I-1]}
                if X<A[I]
                then X:=A[I]
          end;
          {X=Max A[L],...A[N]}
      Max:=X
end {Max};
```

11.3.1.2. Formális eljárás- és függvényparaméterek

Egy eljárásparaméter-specifikáció az eljárásazonosítót vezeti be valamilyen, az eljárásfejben definiált, hozzá tartozó formális paraméterlistával együtt.

Eljárásparaméterspecifikáció = Eljárásfej.

Egy függvényparaméter-specifikáció a függvényazonosítót jelöli ki a függvényfejben definiált eredménytípussal és valamilyen hozzá tartozó formális paraméterlistával együtt.

Függvényparaméterspecifikáció = Függvényfej.

11.3.2. Aktuális paraméterlisták

Minden hívási ponton, azaz eljárásutasításnál vagy függvénykifejezésnél egy aktuális paraméterlista adja meg azokat az aktuális paramétereket, amelyekkel az adott hívás során az eljárás vagy függvény formális paramétereit helyettesíteni kell. Ha az eljárásnak vagy függvénynek nincs formális paraméterlistája, akkor a hívásban sem lehet aktuális paraméterlista. Az aktuális és a formális paraméterek közötti megfeleltetés a paraméterek saját listájukon elfoglalt pozícióinak megfelelően történik. Rendszerfüggő, hogy egy lista aktuális paramétereinek behelyettesítésére milyen sorrendben kerül sor.

AktuálisParaméterlista = "(" *AktuálisParaméter* {"," *AktuálisParaméter* } ")".
AktuálisParaméter = *Kifejezés* | *Változó* | *Eljárásazonosító* | *Függvényazonosító*.

Egy adott hívási ponton az ugyanazon formális paraméterrészben definiált illeszkedő-tömb-paramétereknek megfelelő valamennyi aktuális paraméter ugyanahhoz a típushoz kell, hogy tartozzék. Ennek a típusnak illeszthetőnek kell lennie (l. a 11.3.4. pontot) a formális paraméterrész illeszkedőtömb-sémájához. Egy híváson belül valamennyi megfelelő formális paraméter ugyanahhoz a típushoz tartozik, ezt a típust az illeszkedőtömb-séma az aktuális paraméter(ek) típusából származtatja (l. a 11.3.4. pontot).

11.3.2.1. Aktuális értékparaméterek

Minden aktuális értékparaméter egy kifejezés. A formális paraméter egy változót jelöl, amelynek kezdőértékként az aktuális paraméter értékét adjuk (l. a 10.3. szakaszt).

Ha a formális paraméter nem illeszkedőtömb-paraméter, akkor az aktuális paraméternek értékadás-kompatibilisnak kell lennie (l. a 6.5. szakaszt) a formális paraméter típusával.

Ha a formális paraméter illeszkedőtömb-paraméter, akkor az aktuális paraméter nem lehet illeszkedő típusú (l. a 11.3.4. pontot).

11.3.2.2. Aktuális változóparaméterek

Az aktuális változóparaméterek változók. A hívás során a formális paraméter azt a változót jelenti, amelyet az aktuális paraméter a hívás kezdetekot kijelöl (l. a 10.3. szakaszt). Az aktuális paraméter nem jelölheti valamely tömörített tömb- vagy rekordváltozó elemét, sem egy kijelölőmezőt.

Ha a formális paraméter nem illeszkedőtömb-paraméter, akkor az aktuális és a formális paraméternek azonos típusúnak kell lennie.

11.3.2.3. Aktuális eljárásparaméterek

Az aktuális eljárásparaméterek eljárásazonosítók. A formális paraméter az aktuális paraméter által megjelölt eljárást jelenti (l. a 10.3. szakaszt). A formális és az aktuális paraméter formális paraméterlistájának, ha egyáltalán van, kongruensnek kell lennie (l. a 11.3.3. pontot).

11.3.2.4. Aktuális függvényparaméterek

Az aktuális függvényparaméterek függvényazonosítók. A formális paraméter az aktuális paraméter által megjelölt függvényt jelenti (l. a 10.3. szakaszt). A formális és az aktuális paraméter eredménytípusának ugyanazt a típust kell jelölnie. A formális és az aktuális paraméter formális paraméterlistájának, ha egyáltalán van, kongruensnek kell lennie (l. a 11.3.3. pontot).

11.3.3. Paraméterlisták kongruenciája

Két formális paraméterlista kongruens, ha elemeik száma megegyezik, és a megfelelő helyen álló formális paraméterek eleget tesznek a következő feltételek valamelyikének:

(a) Mindkettő értékparaméter-specifikáció, azonosítólistáikon ugyanannyi azonosítóval, és vagy mindkettő típusazonosítót tartalmaz, amelyek ugyanazt a típust jelölik, vagy mindkettőben illeszkedőtömb-séma szerepel, és ezek ekvivalensek.
(b) Mindkettő változóparaméter-specifikáció, azonosítólistáikon ugyanannyi azonosítóval, és vagy mindkettő típusazonosítót tartalmaz, amelyek ugyanazt a típust jelölik, vagy mindkettőben illeszkedőtömb-séma szerepel, és ezek ekvivalensek.
(c) Mindkettő eljárásparaméter-specifikáció, és formális paraméterlistáik kongruensek.
(d) Mindkettő függvényparaméter-specifikáció, formális paraméterlistáik kongruensek, és eredménytípusaik ugyanazt a típust jelölik.

Két illeszkedőtömb-séma (amelyek mindketten csak egy indextípus-specifikációt tartalmaznak) *ekvivalens*, ha az alábbi három feltétel mindegyike teljesül:

(a) Az indextípus-specifikáció megszámlálhatótípus-azonosítói ugyanazt a típust jelölik.
(b) Vagy mindkettő illeszkedőtömb-sémából épül föl és az őket alkotó részsémák ekvivalensek, vagy mindkettő típusazonosítóból épül föl, és az őket alkotó típusazonosítók ugyanazt a típust jelölik.
(c) Vagy mindkettő tömörített, vagy mindkettő tömörítetlen illeszkedőtömb-séma.

Példa két ekvivalens illeszkedőtömb-sémára:

```
array [L1..H1: Integer; L2..H2: Szín] of packed array [L3..H3: T2] of T
array [Alsó1..Fölső1: Integer] of array [Alsó2..Fölső2: Szín] of
      packed array [Alsó3..Fölső3: T2] of T
```

11.3.4. Illeszthetőség és illeszkedő típusok

Azt mondjuk, hogy egy T tömb típus (amelynek egyetlen indextípusa van) *illeszthető* egy (egyetlen indextípus-specifikációval ellátott) S illeszkedőtömb-sémához, ha az alábbi feltételek mindegyike teljesül. Jelölje I az S indextípus-specifikációjának megszámlálhatótípus-azonosítóját!

(a) T indextípusa kompatibilis az I által jelzett típussal.
(b) T indextípusának minden értéke az I-vel jelzett típus értékeinek halmazához tartozik.
(c) Ha S nem tartalmaz illeszkedőtömb-sémát, akkor T elemeinek típusa megegyezik az S-beli típusazonosító által jelzett típussal; máskülönben T elemtípusa az S-beli sémához illeszthető.
(d) T pontosan akkor tömörített, ha S tömörített illeszkedőtömb-séma.

Ha a szintaktikus szabályok illeszthetőséget követelnek meg, a *(b)* feltétel megsértése hibát eredményez.

Azt mondjuk, hogy egy típus *a T-ből S által származtatott illeszkedő típus*, ha a T-vel azonos indextípusú tömb típus, pontosan akkor tömörített, ha T is az, és elemtípusa vagy megegyezik T elemtípusával, vagy ha S egy másik illeszkedőtömb-sémából épül föl, akkor a T elemtípusából e részséma által származtatott illeszkedő típus. Az indextípus-specifikációban bevezetett határazonosítók az illeszkedő típus indextípusának legkisebb és legnagyobb értékét jelölik.

11.4. Standard eljárások

11.4.1. Állománykezelő eljárások

Több, kifejezetten a szöveg típusú állományok kezelésére definiált standard eljárás van. Ezeket a 12. fejezetben ismertetjük részletesen. Az alábbi eljárások bármilyen f állományváltozóra vonatkozhatnak (l. a 6.2.4. pontot és a 7.4. szakaszt).

Rewrite(f) eredményeképpen f egy üres sorozatot tartalmazó, generálási módban lévő állomány lesz;

Put(f) hibát eredményez, ha f határozatlan, ha f nemgenerálási módban van, vagy ha az f↑ pufferváltozó határozatlan. Az f-beli elemsorozat végére beilleszti f↑ értékét;

Reset(f) hatására f feldolgozási módba kerül, és a benne elhelyezkedő sorozat első pozíciójára állunk rá. Ha ez a sorozat üres, **eof(f)** értéke igazzá, f↑ pedig teljesen határozatlanná válik; minden más esetben **eof(f)** értéke hamis lesz, és f↑ a sorozat első pozícióján elhelyezkedő értéket veszi föl;

Get(f) hibás, ha f határozatlan, vagy **eof(f)** igaz. Hatására az állományban áttérünk a következő elemre, ha ilyen egyáltalán létezik, és f↑ ennek az értékét veszi fel, ha az állománynak nincs következő eleme, **eof(f)** értéke igazzá válik, f↑ pedig teljesen határozatlan lesz.

Az alábbi definíciókban, valahányszor f előfordul, mindig ugyanazt az állományváltozót jelenti. A v, v1, ..., vn szimbólumok változókat, az e, e1, ..., en szimbólumok pedig kifejezéseket jelölnek. A v, v1, ..., vn változók nem aktuális változóparaméterek, tehát lehetnek tömörített tömbök vagy rekordok elemei is.

Read(f,v1,...,vn) ekvivalens a

```
begin Read(f,v1); ...; Read(f,vn)
      end
```

utasítással.

Read(f,v) ahol f nem szöveg típusú állomány, ekvivalens a

```
begin v := f↑; Get(f) end
```

utasítással,

Write(f,e1,...,en) ekvivalens a

```
begin
      Write(f,e1); ...; Write(f,en)
end
```

utasítással.

Write(f,e) ahol f nem szöveg típusú állomány, ekvivalens a

```
begin f↑ := e; Put(f) end
```

utasítással.

11.4.2. Dinamikus helyfoglaló eljárások

A dinamikus helyfoglaló eljárások segítségével hozhatunk létre (New) és szüntethetünk meg (Dispose) új mutatóértékeket és ezek dinamikus változóit. Az alábbi leírásokban p mutatóváltozó, q mutatókifejezés, c1,...,cn, k1,...,kn pedig konstansok. Vegyük észre, hogy p nem aktuális paraméter, tehát tömörített tömbnek vagy rekordnak is lehet eleme.

New(p)	létrehoz egy új, p-vel azonos típusú mutatóértéket, és azt p-hez rendeli. A p↑ dinamikus változó teljesen határozatlan.
New(p,c1,...,cn)	létrehoz egy új, p-évél azonos típusú mutatóértéket, és azt p-hez rendeli. A p↑ dinamikus változó teljesen határozatlan. Az ilyen mutató típus főtípusa szükségképpen változat részt is tartalmazó rekord típus. Az első (c1) konstans kiválaszt egy változatot a változat részből; a következő konstans, ha egyáltalán van ilyen, kiválaszt egy változatot a következő (beágyazott) változat részből, és így tovább. Hibát eredményez, ha e változat részekből a kiválasztottakon kívül bármelyik másik változatot érvényessé tesszük a dinamikus változóban. Hibát eredményez, ha a p↑ dinamikus változót tényezőként, aktuális változóparaméterként vagy egy értékadó utasítás változójaként használjuk föl (noha p↑ elemei használhatóak ilyen értelemben).
Dispose(q)	törli a q mutatóértéket. Hibát okoz, ha q értéke nil. Csak a New eljárás első (rövid) alakjával létrehozott q értékekre alkalmazható, különben hibát okoz.
Dispose(q,k1,...,kn)	törli a q mutatóértéket. Hibát okoz, ha q értéke nil. Csak akkor alkalmazható, ha a q értéket a New eljárás második (hosszú) alakjával hoztuk létre, és a k1,...,kn konstansok ugyanazokat a változatokat jelölik ki, mint a mutatóérték létrehozásakor; különben hibára vezet.

11.4.3. Adatátviteli eljárások

Jelöljön U egy tömörítetlen, S1 indextípusú és T elemtípusú tömbváltozót. Jelöljön P egy tömörített, S2 indextípusú és T elemtípusú tömbváltozót. Jelölje B és C az S2 típus legkisebb és legnagyobb értékét. Jelöljön K egy új, (másképp nem hozzáférhető) S1 típusú, J pedig egy új, S2 típusú változót. Legyen végül I egy S1-gyel kompatibilis kifejezés.

Pack(U,I,P) ekvivalens a

```
begin
    K := I;
    for J := B to C do
        begin
            P[J] := U[K];
            if J <> C then K := succ(K)
        end
end
```

utasítással, Unpack(P,U,I) pedig a

```
begin
    K := I;
    for J := B to C do
        begin
            U[K] := P[J];
            if J <> C then K := succ(K)
        end
end
```

utasítással. Mindkét részletes leírásban az U is és a P is egyetlen változót jelöl a for utasítás valamennyi iterációs lépése során.

11.5. Standard függvények

11.5.1. Aritmetikai függvények

Legyen x tetszőleges valós vagy egész kifejezés. abs és sqr eredménytípusa megegyezik x típusával. A többi aritmetikai függvény eredménytípusa valós.

abs(x)	eredménye x abszolút értéke.
sqr(x)	eredménye x négyzete. Hibát okoz, ha az adott rendszerben az x szám négyzete nem létezik.
sin(x)	eredménye x szinusza, ahol az x értéket radiánban értjük.
cos(x)	eredménye x koszinusza, ahol az x értéket radiánban értjük.
exp(x)	eredménye a természetes logaritmus alapszámának x-edik hatványa.
ln(x)	eredménye x természetes logaritmusa. Hibára vezet, ha x kisebb vagy egyenlő, mint 0.
sqrt(x)	eredménye x négyzetgyöke. Hibát okoz, ha x negatív.
arctan(x)	eredménye x arkusz tangensének radiánban kifejezett főértéke.

11.5.2. Logikai függvények

Legyen i tetszőleges egész kifejezés, f pedig tetszőleges állományváltozó. Valamennyi logikai függvény eredménye Boolean típusú.

odd(i)	ekvivalens az (abs(i) mod 2 = 1) kifejezéssel.
eof(f)	hibát okoz, ha f határozatlan; máskor viszont eof(f) eredménye igaz, ha f generálási módban van, vagy ha túlléptünk az f-beli sorozat utolsó elemén. Ha nincs paraméterlista, az eof függvényt az Input programparaméterre alkalmazzuk.
eoln(f)	hibás, ha f határozatlan, vagy ha eof(f) értéke igaz. f-nek szöveg típusú állománynak kell lennie. Az eoln(f) eredménye igaz, ha az f-beli sorozat aktuális eleme egy sor vége jel. Ha nincs paraméterlista, az eoln függvényt az Input programparaméterre alkalmazzuk.

11.5.3. Konverziós függvények

Jelöljön r egy valós kifejezést. E függvények eredménytípusa mindig Integer.

trunc(r)	eredménye olyan érték, hogy ha r >= 0, akkor 0 <= r − trunc(r) < 1, ha pedig r < 0, akkor −1 < r − trunc(r) <= 0. Hibát okoz, ha ilyen érték nem létezik.
round(r)	eredménye olyan érték, hogy ha r >= 0, akkor round(r) = trunc(r + 0.5), ha pedig r < 0, akkor round(r) = trunc(r−0.5). Hibát okoz, ha ilyen érték nem létezik.

11.5.4. A megszámlálható típusokon értelmezett függvények

Legyen i egész kifejezés, x pedig tetszőleges diszkrét kifejezés.

ord(x)	eredménye x rendszáma.
chr(i)	eredménye az a Char típusú érték, amelynek rendszáma i. Hibát okoz, ha ilyen érték nem létezik. Ha c egy karaktererték, akkor mindig igaz, hogy chr(ord(c)) = = c.

succ(x) eredménye x után következő érték, ha ilyen létezik. Ilyenkor ord(succ(x)) = = ord(x) + 1. Hibát okoz, ha nem létezik a rá következő érték.

pred(x) eredménye az x-et megelőző érték, ha ilyen létezik. Ilyenkor ord(pred(x)) = = ord(x)−1. Hibát okoz, ha nem létezik x-et megelőző érték.

12. Szöveg típusú állományok be- és kivitele

Az olvasható szövegek be- és kivitelének alapegységei azok a szöveg típusú állományok (l. a 6.2.4. pontot), amelyeket egy Pascal programba programparaméterekként (l. a 13. fejezetet) viszünk be, és amelyek a program környezetébe tartozó valamilyen input vagy output egységhez, például terminálhoz, képernyőhöz, mágnesszalaghoz vagy sornyomtatóhoz kapcsolódnak. A szöveg típusú állományok kezelésének megkönnyítésére három további standard eljárást (a Readln, a Writeln és a Page eljárásokat) vezetünk be, két, már tárgyalt standard eljárás (a Read és a Write; l. a 11.4.1. pontot) érvényességét pedig kiterjesztjük. Azoknak a szöveg típusú állományoknak, amelyekre ezeket az eljárásokat alkalmazzuk, nem kell feltétlenül input/output egységekhez kapcsolódniuk, lehetnek lokális állományok is. Az említett eljárások aktuális paraméterlistáira nem a szokásos szabályok vonatkoznak (l. a 11.3. szakaszt), lehetőség van többek között arra is, hogy a paraméterlistán változó számú paraméter álljon. Még a paramétereknek sem kell szükségképpen Char típusúaknak lenniük, tartozhatnak bizonyos más típusokhoz is. Ilyenkor azonban az adatátvitel mellett automatikus adatkonverzióra is sor kerül. Ha az első paraméter egy állományváltozó, akkor ezt az állományt kell beolvasni vagy kiírni. Ha nem, akkor a gép az Input és Output standard állományokat (l. a 13. fejezetet) használja, mégpedig olvasás esetén az előbbit, írás esetén pedig az utóbbit.

12.1. A Read utasítás

Ha a Read eljárást szöveg típusú állományra alkalmazzuk, az alábbi szabályok szerint kell eljárnunk. Jelöljön f egy szöveg típusú állományt, v1,...,vn pedig a Char vagy az Integer típusokhoz (vagy ezek valamely résztartomány-típusához), esetleg a Real típushoz tartozó változókat.

(a) Read(v1,...,vn) ekvivalens a Read(g,v1,...,vn) eljárással, ahol g az Input standard szöveg típusú állományt jelöli.

(b) A Read(f,v1,...,vn) eljárás ekvivalens a

```
begin Read(f,v1); ...; Read(f,vn) end
```

utasítással, ahol f végig ugyanazt a változót jelöli.

(c) Read(f,v) hibát okoz, ha f határozatlan, ha f nem feldolgozási módban van, vagy ha eof(f) igaz.

Read(f,v) hatása attól függ, hogy v milyen típusú.

12.1.1. Karakter beolvasása

Ha v a Char típussal kompatibilis típusú változót jelöl, a Read(f,v) eljárás ekvivalens a

```
begin v := f↑; Get(f) end
```

utasítással, ahol f végig egyetlen változót jelöl. Ha eoln(f) igaz volt a Read(f,v) eljárás végrehajtása előtt, akkor utána igaz lesz, hogy (v = ' ').

12.1.2. Egész típusú szám beolvasása

Ha v az Integer típussal kompatibilis típusú változót jelöl, akkor a Read(f,v) eljárás hatására az f-ből kiolvasott karaktersorozat egy *ElőjelesEgész*-t (l. a 4. fejezetet) alkot, és az így jelölt egész érték lesz v új értéke. Az értéknek értékadás-kompatibilisnek kell lennie v típusával. Az elöl álló szóközöket és a sor vége jeleket átugorjuk. Hiba, ha az előjeles egész számot nem lehet megtalálni.

12.1.3. Valós típusú szám beolvasása

Ha v Real típusú változót jelöl, akkor a Read(f,v) eljárás hatására az f-ből kiolvasott karaktersorozat egy *ElőjelesSzám*-ot (l. a 4. fejezetet) alkot, és az így jelölt érték lesz v új értéke. Az elöl álló szóközöket és a sor vége jeleket átugorjuk. Hiba, ha az előjeles számot nem lehet megtalálni!

12.2. A Readln utasítás

Jelöljön f egy szöveg típusú állományt, v1,...,vn pedig a Char vagy az Integer típusokhoz (vagy ezek valamely résztartomány-típusához), esetleg a Real típushoz tartozó változókat.

Readln(v1,...,vn) ekvivalens a Readln(g,v1,...,vn) eljárással, Readln pedig a Readln(g) eljárással, ahol g az Input standard szöveg típusú állományt jelöli.

Readln(f,v1,...,vn) ekvivalens a

```
begin Read(f,v1,...,vn); Readln(f) end
```

utasítással, ahol f mindvégig egyetlen változót jelöl.

Readln(f) ekvivalens a

```
begin
    while not eoln(f) do Get(f);
    Get(f)
end
```

utasítással, ahol f végig egyetlen változót jelöl.

12.3. A Write utasítás

Ha a Write eljárást szöveg típusú állományra alkalmazzuk, az alábbi szabályok szerint kell eljárnunk. Jelöljön f egy szöveg típusú állományt, p,p1,...,pn a *WriteParaméter*-eket, e egy kifejezést, m és n pedig egész kifejezéseket. A write eljárás aktuális paraméterlistájára a következő szintaktikus előírások érvényesek:

WriteParaméterlista = "(" *(Állományváltozó* | *WriteParaméter)*
("," *WriteParaméter*) ")".
WriteParaméter = *Kifejezés* [":" *EgészKifejezés* [":" *EgészKifejezés*]]

(a) A Write(p1,...,pn) eljárás ekvivalens a Write(g,p1,...,pn) eljárással, ahol g az Output standard szöveg típusú állomány.

(b) A Write(f,p1,...,pn) eljárás ekvivalens a

```
begin Write(f,p1); ...; Write(f,pn) end
```

utasítással, ahol f mindvégig egyetlen változót jelöl.

(c) Write(f,p) hiba, ha f határozatlan, vagy nem generálási módban van.

(d) Egy write paraméter formája az alábbi három forma valamelyike:

```
e    e:m    e:m:n
```

Itt e jelenti az f-be beírandó értéket, m és n pedig az ún. mezőszélesség-paraméterek. Hibát okoz, ha akár m, akár n nem pozitív. e lehet Integer, Real, Char, Boolean vagy füzér típusú. Az n kifejezés csak akkor szerepelhet, ha e Real típusú (l. a 12.3.3. pontot). Ha az m kifejezést elhagyjuk, a gép egy alapértékkel dolgozik. Az alapértéket a rendszer definiálja, ha e Integer, Real vagy Boolean típusú. Char típus esetén ez az alapérték 1, füzér típus esetén pedig a füzér elemeinek száma.

Ha e értékének ábrázolásához m-nél kevesebb pozícióra van szükség, akkor az értéket megfelelő számú szóköz előzi meg úgy, hogy pontosan m számú karakter íródjon ki, e értékének ábrázolása e típusától függ.

12.3.1. A Write utasítás karakter típus esetén

Ha e Char típusú, akkor a Write(f,e:m) eljárás ekvivalens a

```
begin
    for J := 1 to m - 1 do Write(f, ' ');
    f↑ := e; Put (f)
end
```

utasítással, ahol f végig egyetlen változót jelöl, J pedig egy új (másként hozzáférhetetlen) egész változó.

12.3.2. A Write utasítás egész típus esetén

Ha e Integer típusú, a Write(f,e:m) eljárás először egy ' − ' karaktert ír ki, ha e < 0, majd ezt követően kiírja abs(e) értékének tízes számrendszerbeli alakját. e értéke előtt annyi szóközt helyez el, hogy pontosan m számú karakter kiírására kerüljön sor.

12.3.3. A Write utasítás valós típus esetén

Ha e Real típusú a Write(f,e:m:n) eljárás e értékét fixpontos számábrázolással írja ki, n jegyet írva a tizedespont után; a Write(f,e:m) eljárás viszont lebegőpontosan írja ki ugyanezt az értéket. A „**" műveleti jel a hatványozást jelöli.

12.3.3.1. Fixpontos ábrázolás

Legyen w nulla, ha e nulla, egyébként pedig legyen w az e abszolút értéke kerekítve, majd az n-edik tizedesjegynél levágva. $w < 1$ esetén legyen $d = 1$, egyébként pedig legyen olyan, hogy $10**(d-1) <= w < 10**d$ teljesüljön. d a tizedesponttól balra eső jegyek száma. Legyen továbbá s=ord((e < 0) and (w <> 0)). Az érték negatív, ha s=1. Legyen végül k=(s+d+1+n); k a kiírandó nem szóköz karakterek száma.

Ha k < m, akkor a szám előtt m−k darab szóköz áll. e fixpontos ábrázolása az alábbi k karakterből áll:

(a) '−', ha s = 1,
(b) w egészrészének d számú számjegye,
(c) '.',
(d) w törtrészének n legnagyobb helyi értékű tizedesjegye.

12.3.3.2. Lebegőpontos ábrázolás

A Pascal rendszer előírja, hogy a lebegőpontos ábrázolás karakterisztikájában („E részében") hány számjegynek kell állnia; jelölje ezt a számot x. Legyen k az m és az x + 6 számok nagyobbika. A kiírandó értékes jegyek száma k−x−4, éspedig ezek közül egy áll a tizedespont előtt, d pedig utána (és így d = k−x−5). Legyen w és s nulla, ha e értéke nulla. Ha e értéke nullától különböző, akkor legyen s olyan, hogy $10.0**s <= abs(e) < 10.0**(s+1)$ és legyen $w = (abs(e)/10.0**s) + 0.5 * 10.0**(-d)$. Ha ekkor $w >= 10.0$, módosítsuk w-t és s-t úgy, hogy s := s + 1 és w := w/10.0 legyen. Végül vágjuk le w-t a d-edik tizedesjegynél!

Az e lebegőpontos ábrázolása a következőkből áll:

(a) '−', ha ((e < 0) and (w <> 0)), egyébként ' '.
(b) w legnagyobb helyi értékű számjegye.
(c) '.'.
(d) w következő legnagyobb helyi értékű d darab számjegye.
(e) 'e' vagy 'E' (a rendszer határozza meg, hogy melyik).
(f) '−', ha s < 0, egyébként '+'.
(g) s x darab számjegye, ha szükséges, az elején nullákkal feltöltve.

12.3.4. A Write utasítás logikai típus esetén

Ha e típusa Boolean, akkor a Write(f,e:m) eljárás a true és a false szavak valamelyikét írja ki. Az eljárás ekvivalens az

```
if e then Write(f, 'true':m) else Write(f, 'false':m)
```

utasítással, attól eltekintve, hogy a rendszer határozza meg, kis- vagy nagybetűkkel ír-e.

12.3.5. A Write utasítás füzér típus esetén

Ha e k hosszúságú, füzér típusú érték, akkor m > k esetén a Write(f,e:m) eljárás m−k darab szóközt ír. Ezután következnek e elemei, indexeik növekvő sorrendjében. m < k esetén e-ből az utolsó k−m darab karakter nem íródik ki.

12.4. A Writeln utasítás

Jelöljön f egy szöveg típusú állományt, p1,...,pn pedig write paramétereket.

Writeln(p1,...,pn) a Writeln(g,p1,...,pn) eljárással, Writeln pedig a Writeln(g) eljárással ekvivalens, ahol g az Output standard szöveg típusú állományt jelöli.

A Writeln(f,p1,...pn) eljárás ekvivalens a

```
begin Write(f,p1,...,pn); Writeln(f) end
```

utasítással, ahol f mindvégig egyetlen változót jelöl.

Writeln(f) egy sor vége jelet illeszt az f állományban elhelyezkedő sorozat végére. Hibát okoz, ha f határozatlan, vagy f nemgenerálási módban van.

12.5. A Page utasítás

A Page(f) eljárás hatását az f szöveg típusú állományra a rendszer definiálja úgy, hogy az f-be ezután írt szöveg az f nyomtatásakor a következő oldal elejére kerüljön. Ha f nemüres és a benne elhelyezett sorozat utolsó eleme nem sor vége jel, akkor a Page(f) eljárás egy implicit Writeln(f) eljárást is végrehajt. Ha nincs paraméterlista, a gép az Output standard szöveg típusú állománnyal dolgozik. Hiba, ha f határozatlan, vagy nemgenerálási módban van.

Rendszerfüggő, hogy milyen hatással van egy olvasó eljárás egy olyan állományváltozóra, amelyre előzőleg a Page eljárást alkalmaztuk.

13. Programok

Egy Pascal program egy programfejből és egy blokkból áll.

Program = *Programfej* ";" *Blokk* "."
Programfej = "program" *Azonosító* [*ProgramParaméterlista*].
ProgramParaméterlista = "(" *Azonosítólista* ")".

A program alapszó után álló azonosító a program neve; ennek a programon belül már nincs jelentősége. A program paraméterlistáján szereplő azonosítókat programparamétereknek nevezzük. Ezek a programon kívül, attól függetlenül létező – és ezért *külsőnek* (external) nevezett – objektumokat jelölnek. A program ezeken keresztül tart fenn kapcsolatot a környezetével.

Amikor egy programot hívunk, minden programparaméter ahhoz a külső objektumhoz kötődik, amelyet jelöl. Azon programparaméterek esetén, amelyek állományváltozók, ez az összeköttetés a rendszer által definiált; a többi programparaméter esetén rendszerfüggő.

Az Input és az Output kivételével minden programparamétert deklarálni kell a program-

blokk változódeklarációs részében. Az Input és az Output esetében az azonosító felvétele a program-paraméterlistára automatikusan a programblokkbeli szöveg típusú állományként deklarálja az azonosítót, és a program minden hívásakor automatikusan végrehajtja a Reset(Input) vagy a Rewrite(Output) eljárásokat.

A rendszer határozza meg, milyen hatásuk van a Reset és a Rewrite eljárásoknak, Input vagy Output állomány esetén.

Példák programra:

```
program ValosMasolas(F,G);
{Turbo Pascal}

var F,G: file of Real;
    R: Real;

begin Reset(F);
      Rewrite(G);
      while not Eof(F) do
            begin Read(F,R);
                  Write(G,R)
            end
end {ValosMasolas}.
```

```
program Szovegmasolas(Input,Output);
{MS Pascal}

begin while not Eof(Input) do
            begin while not Eoln(Input) do
                        begin Input^:=Output^;
                              Put(Output);
                              Get(Input)
                        end;
                  Readln(Input);
                  Writeln(Output)
            end
end {Szovegmasolas}.
```

14. Illeszkedés az ISO 7185 szabványhoz

Egy *program* összhangban van az ISO Pascal szabvánnyal [11], ha a nyelvnek csak a szabvány által definiált jellemzőit alkalmazza, és nem támaszkodik a rendszerfüggő konstrukciók speciális megvalósítási módjára. Azt mondjuk, hogy a program 0 szintű, ha nem használ illeszkedőtömb-paramétereket, és 1 szintű, ha használ.

A *processzort* a szabvány úgy definiálja, mint „olyan rendszert vagy mechanizmust, amelynek inputja egy program. A processzor ennek végrehajtását készíti elő, és az így meghatározott eljárást adatokkal végrehajtva szolgáltatja az eredményt." Egy processzor akkor van összhangban a szabvánnyal, ha az alábbi feltételek mindegyikének eleget tesz.

(a) A nyelv minden részletét a szabvány által definiáltan fogadja el. 0 szintűnek mondjuk, ha nem fogad el illeszkedőtömb-paramétereket és 1 szintűnek, ha igen.

(b) Ahhoz, hogy a nyelv valamely konstrukcióját megvalósítsa, nincs szükség a nyelv más, helyettesítő elemeire.

(c) Képes a szabvány által leírt szabályok hibának nem minősülő áthágását észlelni, és ezt jelzi a felhasználónak. Ha a processzor nem vizsgálja végig az egész programot, hogy történt-e benne ilyen szabálytalanság, ezt a tényt is jeleznie kell.

(d) Minden, hibának minősülő szabálytalanságot az alábbi négy mód valamelyikén kezel:

1. Dokumentációjában az áll, hogy a hibát nem jelzi.
2. A program előkészítése során jelzi, hogy hiba előfordulhat.
3. A program előkészítése során jelzi, hogy hiba fog előfordulni.
4. A program végrehajtása során jelzi, hogy hibát talált.

(e) Hibaként tud feldolgozni minden olyan esetet, amikor a program a nyelv valamely bővítését vagy rendszerfüggő vonását alkalmazza.

(f) Kísérő dokumentuma az alábbiakat tartalmazza:

1. A rendszer által definiált valamennyi jellemző definícióját.
2. Azon hibák felsorolását, amelyet a processzor nem jelez (l. *(d)* 1.). Ha valamely bővítés egy olyan feltételt használ ki, amelyet a szabvány hibának tekint, és a processzor a hibát emiatt nem jelzi, a dokumentumból ki kell derülnie, hogy ennek a hibának a jelzésére nem fog sor kerülni.
3. A rendszer által alkalmazott valamennyi bővítés leírását.

Irodalom

[1] *Wirth, N.*: The Programming Language Pascal.
(A Pascal programozási nyelv.) *Acta Informatica, 1,* 35–63, 1971.

[2] *Wirth, N.*: Program Development by Stepwise Refinement. (Programfejlesztés – lépésenként.)
Communications of the ACM 14, 221–227, 1971. április.

[3] *Wirth, N.*: Systematic Programming. (Szisztematikus programozás.) Prentice-Hall, Inc. 1973.

[4] *Dahl, O. J.–Dijkstra, E. W.–Hoare, C. A. R.*: Strukturált programozás. Műszaki Könyvkiadó, Budapest, 1978.

[5] *Hoare, C. A. R.–Wirth, N.*: An Axiomatic Definition of the Programming Language Pascal. (A Pascal programozási nyelv axiomatikus definíciója.) *Acta Informatica 2,* 335–355, 1973.

[6] *Knuth, D. E.*: A programozás művészete, 1. köt. Műszaki Könyvkiadó, Budapest, 1987.

[7] *Wirth, N.*: An Assessment of the Programming Language Pascal. (A Pascal programozási nyelv értékelése.)
SIGPLAN Notices *10,* 23–30, 1975. június.

[8] *Wirth, N.*: The Design of a Pascal Compiler. (Fordítóprogram szerkesztése a Pascal nyelvhez.) *SOFTWARE–Practice and Experience 1,* 309–333, 1971.

[9] *Wirth, N.*: Algoritmusok + Adatstruktúrák = Programok.
Műszaki Könyvkiadó, Budapest, 1982.

[10] *Barron, D.*: A Perspective on Pascal. (A Pascal nyelv kilátásai.) Továbbá *Welsh, J.–Sneeringer, W.–Hoare, C. A. R.*: Ambiguities and Insecurities in Pascal.
(Kétértelműségek és bizonytalanságok a Pascal nyelvben.) Pascal – *The Language and its Implementation,* John Wiley, 1981.

[11] International Organization for Standardization, *Specification for Computer Programming Language Pascal.* (A Pascal számítógép-programozási nyelv leírása.) ISO, 7185–1982, 1982.

[12] *Sale, A. H. J.–Wichmann, B.*: The Pascal Validation Suite. *Pascal News 16,* 5–153, 1979.

[13] *Wirth, N.*: What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions? (Mit lehet tenni a szintaktikus definíciók jelölésmódjában mutatkozó szükségtelen különbségek ellen?) *Communications of the ACM 20,* 822–823, 1977. november.

[14] *Wichmann, B.–Ciechanowicz, Z. J.*: Pascal Compiler Validation. (A Pascal fordítóprogram bevizsgálása.) John Wiley, 1983.

A. függelék

Standard eljárások és függvények

Abs(x)
aritmetikai függvény, amely a valós típusú x paraméter valós típusú, vagy az egész típusú x paraméter egész típusú abszolút értékét számítja ki.

ArcTan(x)
aritmetikai függvény, amely a valós vagy egész típusú x paraméter valós típusú, radiánban megadott arkusz tangensét (pontosabban annak főértékét) számítja ki.

Chr(i)
megszámlálható típusú függvény, amely kikeresi azt a karaktert, amelynek rendszáma az i egész típusú paraméter. Chr(i) hibát eredményez, ha ilyen karakterérték nem létezik.

Dispose(q)
dinamikus helyfoglaló eljárás, amely fölszabadítja a q↑ dinamikus változó tárolóterületét és törli a q mutatóértéket. Dispose(q) hiba, ha q értéke nil, vagy határozatlan. A q értéknek a New eljárás rövid alakjával létrehozottnak kell lennie.

Dispose(q,k1,...,kn)
dinamikus helyfoglaló eljárás, amely fölszabadítja a q↑ dinamikus rekordváltozó tárolóterületét és törli a q mutatóértéket. Dispose(q,k1,...,kn) hibát eredményez, ha q értéke nil vagy határozatlan. Az eljárás csak akkor alkalmazható, ha a q értéket a New eljárás hosszú alakjával hoztuk létre, a k1,...,kn konstansok által kijelölt változatoknak pedig meg kell egyezniük azokkal, amelyeket q létrehozásakor alkalmaztunk.

Eof(f)
logikai függvény, amely a true értéket veszi fel, ha az f állományváltozó generálási módban van, vagy ha f feldolgozási módban van ugyan, de túlléptünk utolsó elemén. eof(f) hibás, ha f határozatlan. Eof(f) minden más esetben a false értéket veszi föl. Ha az f paramétert elhagyjuk, a gép az Input standard állománnyal dolgozik.

Eoln(f)
logikai függvény, amely a true értéket veszi fel, ha a feldolgozási módban lévő f szöveg típusú állomány aktuális elemén egy sor vége jel áll. Eoln(f) hibás, ha f határozatlan, vagy ha eof(f) igaz. eoln(f) értéke minden más esetben false. Az f paraméter hiányában a gép az Input standard állománnyal dolgozik.

Exp(x)
aritmetikai függvény, amely az *e* (a természetes logaritmus alapszáma) alapú exponenciális függvény valós típusú értékét számítja ki a valós vagy egész típusú x paraméter által meghatározott helyen.

Get(f)
állománykezelő eljárás, amelynek hatására az f-ben elhelyezkedő sorozat következő eleme (ha egyáltalán van ilyen) válik aktuálissá, f↑ pedig az itt elhelyezett értéket veszi föl; ha az állo-

mányban nincs következő elem, eof(f) igazzá, f↑ pedig határozatlanná válik. Get(f) hibát okoz, ha f határozatlan vagy eof(f) igaz. Az f paraméter hiányában a gép az Input standard állománnyal dolgozik.

Ln(x)

aritmetikai függvény, amely a valós vagy egész típusú x paraméter természetes (*e* alapú) logaritmusának valós típusú értékét számítja ki, föltéve, hogy x > 0. Ha x < = 0, Ln(x) hibát eredményez.

New(p)

dinamikus helyfoglaló eljárás, amely tárolóterületet foglal le az új, p főtípusához tartozó p↑ dinamikus változó számára, továbbá létrehoz egy új, p típusú mutatóértéket és azt p-hez rendeli. Ha p↑ változat részt is tartalmazó rekord, New(p) valamennyi változat tárolására elegendő helyet foglal le.

New(p,c1,...,cn)

dinamikus helyfoglaló eljárás, amely tárolóterületet foglal le az új, p↑ dinamikus változó számára. p↑ típusa a p változat részt is tartalmazó rekord típusa, amelynek *n* számú, egymásba ágyazott változat részéből a kijelölőmező c1,...,cn értékei jelölik ki az aktuális változatot. Az eljárás létrehoz egy új, p típusú mutatóértéket is, és azt p-hez rendeli.

Odd(i)

logikai típusú függvény, amelynek értéke true, ha az egész típusú i paraméter páratlan; a függvény ellenkező esetben a false értéket veszi föl.

Ord(x)

egész típusú függvény, amely az x megszámlálható típusú paraméternek az x típusa által meghatározott értékkészletbeli (egész típusú) rendszámát veszi fel.

Pack(u,i,p)

adatátviteli eljárás, amely az u tömörítetlen tömb tartalmát az i-edik elemtől kezdve a p tömörített tömbben helyezi el.

Page(f)

állománykezelő eljárás, amelynek az f szöveg típusú állományparaméterre való hatását a rendszer definiálja úgy, hogy az f állományba ezután írt szöveg f nyomtatásakor a következő oldal elején kezdődjék. Ha f nemüres, és a benne elhelyezkedő sorozat utolsó eleme nem egy sor vége jel, a Page(f) eljárás implicit módon egy Writeln(f) eljárást is végrehajt. Paraméterlista hiányában a gép az Output standard állománnyal dolgozik. Page(f) hibát okoz, ha f határozatlan vagy nemgenerálási módban van.

Pred(x)

megszámlálható típusú függvény, amely az x megszámlálható típusú paramétert megelőző diszkrét értéket veszi föl, ha az egyáltalán létezik: ord(pred(x)) = ord(x) − 1. Pred(x) hiba, ha x típusának legkisebb értéke.

Put(f)

állománykezelő eljárás, amely f↑ értékét az f végére illeszti. Put(f) hibát okoz, ha f határozatlan vagy nemgenerálási módban van, továbbá ha az f↑ pufferváltozó határozatlan. A Put(f) eljárás végrehajtása után f↑ teljesen határozatlanná válik.

Read(f,v)

l. a Felhasználói Kézikönyv 9. és 12. fejezetét, továbbá a Jelentés 11.4. és 12.1. szakaszát.

Read(f,v1,...,vn)

l. a Felhasználói Kézikönyv 9. és 12. fejezetét, továbbá a Jelentés 11.4. és 12.1. szakaszát.

Readln

l. a Felhasználói Kézikönyv 9. és 12. fejezetét, továbbá a Jelentés 12.2. szakaszát.

Readln(f,v1,...,vn)

l. a Felhasználói Kézikönyv 9. és 12. fejezetét, továbbá a Jelentés 12.2. szakaszát.

Reset(f)

állománykezelő eljárás, amely f-et feldolgozási módba állítja és f első elemét teszi aktuálissá. Ha f üres, eof(f) igazzá, f↑ pedig teljesen határozatlanná válik. Ellenkező esetben eof(f) hamis lesz, f↑ pedig fölveszi az f-ben elhelyezkedő sorozat első elemének értékét.

Rewrite(f)

állománykezelő eljárás, amely f tartalmát törli, és f-et generálási módba állítja. Eof(f) az eljárás hatására igazzá válik.

Round(r)

konverziós függvény, amely a $r >= 0.0$ feltételnek eleget tevő valós típusú r paraméter esetére a trunc(r + 0.5) értéket, az $r < 0.0$ feltételt kielégítő valós típusú paraméter esetére pedig a trunc(r - 0.5) értéket számítja ki, ha ilyen érték az Integer típusban létezik. Ellenkező esetben hibát okoz.

Sin(x)

aritmetikai függvény, amely a valós vagy egész típusú x paraméter radiánban megadott értékének valós típusú szinuszát számítja ki.

Sqr(x)

aritmetikai függvény, amely valós típusú x esetén a valós x * x értéket, egész típusú x-re pedig az egész típusú x * x értéket számítja ki. Hiba, ha ez az érték nem létezik.

Sqrt(x)

aritmetikai függvény, amely az $x >= 0$ feltételnek eleget tevő, egész vagy valós típusú x paraméter valós típusú, nemnegatív négyzetgyökét számítja ki. Hiba, ha $x < 0$.

Succ(x)

egész típusú függvény, amely az x megszámlálható típusú paraméter után következő értéket veszi föl, ha az egyáltalán létezik: ord(succ(x)) = ord(x) + 1. succ(x) hibát okoz, ha x típusának legnagyobb értéke.

Trunc(r)

konverziós függvény, amely $r >= 0.0$ esetén a valós típusú r paraméter értékét meg nem haladó legnagyobb egész típusú értéket, $r < 0.0$ esetén viszont a legkisebb olyan egész típusú értéket számítja ki, amely nem kisebb a valós típusú r paraméternél, feltéve persze, hogy ilyen érték az Integer típusban létezik. Ellenkező esetben hibát okoz.

Unpack(p,u,i)

adatátviteli eljárás, amely a p tömörített tömb tartalmát az u tömörítetlen tömbben, annak i-edik elemétől kezdve helyezi el.

Write(f,v)

l. a Felhasználói Kézikönyv 9. és 12. fejezetét, továbbá a Jelentés 11.4. és 12.3. szakaszát.

Write(f,v1,...,vn)

l. a Felhasználói Kézikönyv 9. és 12. fejezetét, továbbá a Jelentés 11.4. és 12.3. szakaszát.

Writeln

l. a Felhasználói Kézikönyv 9. és 12. fejezetét, továbbá a Jelentés 12.4. szakaszát.

Writeln(f,e1,...,en)

l. a Felhasználói Kézikönyv 9. és 12. fejezetét, továbbá a Jelentés 12.4. szakaszát.

B. függelék

A műveletek összefoglalása

Műveleti jel	*Művelet*	*Az operandusok típusa*	*Az eredmény típusa*
Aritmetikai			
+ (unáris)	azonosság	egész vagy valós	ugyanaz, mint az operandusé
– (unáris)	előjelváltás		
+	összeadás	egész vagy valós	egész vagy valós
–	kivonás		
*	szorzás		
div	egészosztás	egész	egész
/	valósosztás	egész vagy valós	valós
mod	modulus	egész	egész
Reláció			
=	egyenlőség	egyszerű, füzér, halmaz, mutató	logikai
<>	nem egyenlőség		
<	kisebb	egyszerű, füzér	logikai
>	nagyobb		
<=	kisebb egyenlő	egyszerű, füzér	logikai
	vagy halmaz-tartalmazás	halmaz	
<=	nagyobb egyenlő	egyszerű, füzér	logikai
	vagy halmaz-tartalmazás	halmaz	
in	halmazhoz tartozás	az első bármilyen megszámlálható alaptípus, a második ennek halmaztípusa	logikai
Logikai			
not	negálás	logikai	logikai
or	diszjunkció	logikai	logikai
and	konjunkció	logikai	logikai
Halmaz			
+	unió	bármely T halmaz típus	T
–	differencia		
*	metszet		

A műveletek precedenciája kifejezésekben

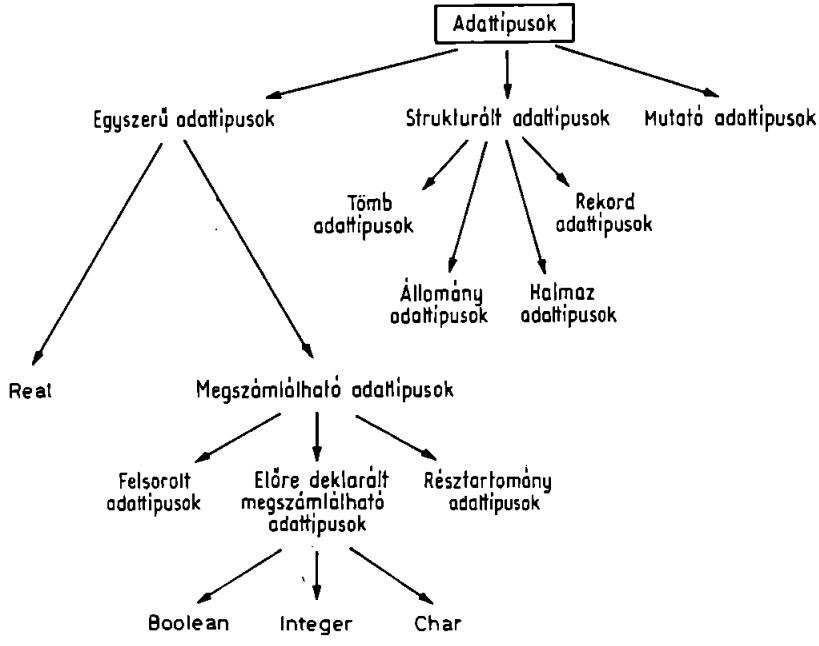
Művelet	*Műveletcsoport*
not	logikai negálás
* / div mod and	multiplikatív műveletek
+ − or	additív műveletek
= <> > < >= <= in	relációk

További műveletek

Jelölés	*Művelet*	*Az operandus típusa*	*Az eredmény típusa*
Értékadás			
:=	értékadás	bármilyen hozzárendelhető típus	nincs
Változók elérése			
[,]	tömbindexelés	tömb	elemtípus
[.]	mezőkijelölés	rekord	mezőtípus
↑	változókijelölés	mutató	főtípus
↑	pufferelérés	állomány	elemtípus
Generálás			
[,]	halmazgenerálás	alaptípus	halmaz
' '	füzérgenerálás	karakter	füzér

C. függelék

Táblák



F.1. ábra. Az adattípusok osztályozása

Standard azonosítók táblája

Konstansok:

False, MaxInt, True

Típusok:

Boolean, Char, Integer, Real, Text

Változók:

Input, Output

Függvények:

Abs, ArcTan, Chr, Cos, Eof, Eoln, Exp, Ln, Odd,
Ord, Pred, Round, Sin, Sqr, Sqrt, Succ, Trunc

Eljárások:

Dispose, Get, New, Pack, Page, Put, Read, Readln,
Reset, Rewrite, Unpack, Write, Writeln

A standard azonosítók ábécésorrendben:

Abs	False	Pack	Sin
ArcTan	Get	Page	Sqr
Boolean	Input	Pred	Sqrt
Char	Integer	Put	Succ
Chr	Ln	Read	Text
Cos	MaxInt	Readln	True
Dispose	New	Real	Trunc
Eof	Odd	Reset	Unpack
Eoln	Ord	Rewrite	Write
Exp	Output	Round	Writeln

Szimbólumok táblája

Speciális szimbólumok:

+	−	*	/	=
<	>	<=	>=	<>
.	,	:	;	:=
(	)	[	]	↑

Alapszavak:

and	end	nil	set
array	file	not	then
begin	for	of	to
case	function	or	type
const	goto	packed	until
div	if	procedure	var
do	in	program	while
downto	label	record	with
else	mod	repeat	

További lehetőségek a szimbólumok ábrázolására:

[	helyett	(.
]	helyett	.)
↑	helyett	@ vagy ^

Direktívák:

forward

D. függelék

Szintaxis

Egy porgramozási nyelv szintaxisának a kiterjesztett Backus–Naur-formában való (EBNF) leírása szabályok sorozatából áll, amelyeket együttesen a nyelv mondatainak alakját leíró „nyelvtannak" nevezünk. Minden szabály egy nemterminális szimbólumból és egy EBNF-kifejezésből áll, amelyeket egy egyenlőségjel választ el. A szabályokat pont zárja le. A nemterminális szimbólum egy „metaazonosító" (egy szóval vagy egybeírt szókapcsolattal jelzett szintaktikus konstans), az EBNF-kifejezés pedig ennek definíciója.

Az EBNF-kifejezés terminális szimbólumokból, nemterminális szimbólumokból és az alábbi táblázatban összefoglalt metaszimbólumokból épül fel. Minden alkotórészből szerepelhet benne nulla vagy annál több.

Metaszimbólum	**Jelentése**
=	definiáljuk úgy, hogy.
|	vagy.
.	szabály vége.
[*X*]	*X*-nek 0 vagy 1 példánya.
{*X*}	*X*-nek 0 vagy több példánya.
(*X* | *Y*)	vagy *X*, vagy *Y*.
"XYZ"	az XYZ terminális szimbólum.
Metaazonosító	a *Metaazonosító* nemterminális szimbólum.

Az EBNF például saját szintaxisának definiálására is felhasználható:

Szintaxis = {*Szabály*}.
Szabály = *Nemterminális* "=" *Kifejezés* ".".
Kifejezés = *Tag* {" |" *Tag*}.
Tag = *Tényező* {*Tényező*}.
Tényező = *Nemterminális* | *Terminális* |"(" *Kifejezés* ")" | "[" *Kifejezés* "]" |
" {" *Kifejezés* "}".
Terminális = " " " " *Karakter* {*Karakter*}" " " ".
Nemterminális = *Betű* {*Betű* |*Számjegy*}.

Megjegyzések:

(1) A terminális szimbólumokat mindig idézőjelek (") közé tesszük; ha magát a " szimbólumot akarjuk idézőjelek közé tenni, akkor kétszer írjuk le. A Pascal nyelv alábbi EBNF-leírásában tehát "[" és "]" a Pascal programban előforduló szögletes zárójeleket jelenti, míg [és] valamely EBNF-kifejezés metaszimbólumai, amelyek azt jelzik, hogy ami közöttük van, az legfeljebb egyszer szerepelhet, de esetleg egyszer sem.

(2) Minden szintaxisban van egy kezdőszimbólum, egy olyan metaazonosító, amelyből kiindulva a nyelv minden mondata generálható. A Pascal szintaxisában ez a kezdőszimbólum a *Program*.

Hierarchikus EBNF-leírás

```
1  Program  = Programfej ";" Blokk ".".
2  Programfej = "program" Azonosító [ProgramParaméterlista].
3  ProgramParaméterlista = "(" Azonosítólista ")".
4
5  ----------.
6
7  Blokk = CímkedeklarációsRész
8              KonstansdefiníciósRész
9              TípusdefiníciósRész
10             VáltozódeklarációsRész
11             EljárásÉsFüggvénydeklarációsRész
12             Utasításrész.
13 CímkedeklarációsRész = [ "label" Számjegysorozat { "," Számjegysorozat } ";" ].
14 KonstansdefiníciósRész = ["const" Konstansdefiníció ";" { Konstansdefiníció ";" } ].
15                             { Konstansdefiníció ";" } ].
16 TípusdefiníciósRész = [ "type" Típusdefiníció ";"
17                          { Típusdefiníció ";" } ].
18 VáltozódeklarációsRész = [ "var" Változódeklaráció ";"
19                             { Változódeklaráció ";" } ].
20 EljárásÉsFüggvénydeklarációsRész = {(Eljárásdeklaráció |
21                                      Függvénydeklaráció) ";" }.
22 Utasításrész = ÖsszetettUtasítás.
23
24 ----------
25
26 Konstansdefiníció = Azonosító "=" Konstans.
27 Típusdefiníció = Azonosító "=" Típus.
28 Változódeklaráció = Azonosítólista ":" Típus.
29 Eljárásdeklaráció = Eljárásfej ";" Blokk |
30                      Eljárásfej ";" Direktíva |
31                      Eljárásazonosítás ";" Blokk.
32 Függvénydeklaráció = Függvényfej ";" Blokk |
33                        Függvényfej ";" Direktíva |
34                        Függvényazonosítás ";" Blokk.
35
36 ----------
37
38 Eljárásfej = "procedure" Azonosító [FormálisParaméterlista].
39 Eljárásazonosítás = "procedure" Eljárásazonosító.
```

```
40 Függvényfej = "function" Azonosító [FormálisParaméterlista]
41                ":" Eredménytípus.
42 Függvényazonosítás = "function" Függvényazonosító.
43 FormálisParaméterlista = "(" FormálisParaméterrész
44                          {";" FormálisParaméterrész }")".
45 FormálisParaméterrész = Értékparaméterspecifikáció |
46                         Változóparaméterspecifikáció |
47                         Eljárásparaméterspecifikáció |
48                         Függvényparaméterspecifikáció.
49
50 ----------
51
52 Értékparaméterspecifikáció = Azonosítólista ":" (Típusazonosító |
53                              Illeszkedőtömbséma).
54
55 Változóparaméterspecifikáció = "var" Azonosítólista ":" (Típusazonosító |
56                                Illeszkedőtömbséma).
57
58 Eljárásparaméterspecifikáció = Eljárásfej.
59
60 Függvényparaméterspecifikáció = Függvényfej.
61
62 Illeszkedőtömbséma = TömörítettIlleszkedőtömbséma |
63                       TömörítetlenIlleszkedőtömbséma.
64 TömörítettIlleszkedőtömbséma = „packed" "array" "[" Indextípusspecifikáció "]" "of"
65                               Típusazonosító.
66
67 TömörítetlenIlleszkedőtömbséma = "array" "[" Indextípusspecifikáció
68                                   {";" Indextípusspecifikáció} "]" "of"
69                                   (Típusazonosító | Illeszkedőtömbséma).
70
71 Indextípusspecifikáció = Azonosító ".." Azonosító ":" MegszámlálhatóTípusAzonosító.
72
73 ----------
74
75 ÖsszetettUtasítás = "begin"
76                     Utasítássorozat
77                     "end".
78 Utasítássorozat = Utasítás {";" Utasítás}.
79 Utasítás = [Címke ":"]
80            (EgyszerűUtasítás |StrukturáltUtasítás).
81 EgyszerűUtasítás = ÜresUtasítás |ÉrtékadóUtasítás | Eljárásutasítás |GotoUtasítás.
82
83 StrukturáltUtasítás = ÖsszetettUtasítás | FeltételesUtasítás |
84                       Ciklusutasítás | WithUtasítás.
85 FeltételesUtasítás = IfUtasítás |CaseUtasítás
86 Ciklusutasítás = WhileUtasítás | RepeatUtasítás | ForUtasítás.
87
88 ----------
89
90 ÜresUtasítás = .
```

```
ÉrtékadóUtasítás = (Változó | Függvényazonosító) ":=" Kifejezés.
Eljárásutasítás = Eljárásazonosító [AktuálisParaméterlista |
                    WriteParaméterlista].
GotoUtasítás = "goto" Címke.
IfUtasítás = "if" LogikaiKifejezés "then" Utasítás
                ["else" Utasítás].
CaseUtasítás = "case" EsetIndex "of"
                  Eset {";" Eset }["";"]
                  "end".
RepeatUtasítás = "repeat"
                    Utasításssorozat
                    "until" LogikaiKifejezés.
WhileUtasítás = "while" LogikaiKifejezés "do"
                   Utasítás.
ForUtasítás = "for" Ciklusváltozó ":=" Kezdőérték
                 ("to" | "downto") Végérték "do" Utasítás.
WithUtasítás = "with" RekordVáltozólista "do"
                  Utasítás.
RekordVáltozólista = Rekordváltozó {"," Rekordváltozó }.
Esetindex = MegszámlálhatóKifejezés.
Eset = Konstans {"," Konstans }":" Utasítás.
Ciklusváltozó = Változóazonosító.
Kezdőérték = MegszámlálhatóKifejezés.
Végérték = MegszámlálhatóKifejezés.

----------

Típus = EgyszerűTípus | StrukturáltTípus | MutatóTípus.
EgyszerűTípus = MegszámlálhatóTípus | ValósTípusAzonosító.
StrukturáltTípus = ["packed"] TömörítetlenStrukturáltTípus |
                      StrukturáltTípusAzonosító.
MutatóTípus = "↑" Főtípus | MutatóTípusAzonosító.
MegszámlálhatóTípus = FelsoroltTípus | RésztartományTípus |
                         MegszámlálhatóTípusAzonosító.
TömörítetlenStrukturáltTípus = TömbTípus | RekordTípus | HalmazTípus |
                                  ÁllományTípus.
Főtípus = Típusazonosító.
FelsoroltTípus = "(" Azonosítólista ")".
RésztartományTípus = Konstans ".." Konstans.


TömbTípus = "array" "[" Indextípus {"," Indextípus } "]" "of"
               Elemtípus.
RekordTípus = "record"
                 Mezőlista
              "end".
HalmazTípus = "set" "of" Alaptípus.
ÁllományTípus = "file" "of" Elemtípus.
Indextípus = MegszámlálhatóTípus.
Elemtípus = Típus.
Alaptípus = MegszámlálhatóTípus.
```

```
141 Eredménytípus = MegszamlálhatóTípusAzonosító | ValósTípusAzonosító |
142                 MutatóTípusAzonosító.
143 Mezőlista = [(RögzítettRész [";" VáltozatRész] | VáltozatRész)
144              [";"]].
145 RögzítettRész = Rekordszakasz {";" Rekordszakasz}.
146 VáltozatRész = "case" Változatszelektor "of"
147                 Változat
148                 {";" Változat}.
149 Rekordszakasz = Azonosítólista ":" Típus.
150 Változatszelektor = [Kijelölőmező ":"] KijelölőTípus.
151 Változat = Konstans {"," Konstans} ":" "(" Mezőlista ")".
152 KijelölőTípus = MegszámlálhatóTípusAzonosító.
153 Kijelölőmező = Azonosító.
154
155 ----------
156
157 Konstans = [Előjel] (ElőjelNélküliSzám | Konstansazonosító) | Karakterfüzér.
158
159
160 ----------
161
162 Kifejezés = EgyszerűKifejezés[RelációsJel |
163             EgyszerűKifejezés].
164 EgyszerűKifejezés = [Előjel] Tag {AdditívMűveletiJel | Tag}.
165 Tag = Tényező {MultiplikatívMűveletiJel Tényező}.
166 Tényező = ElőjelNélküliKonstans | Határazonosító | Változó |
167           Halmazgenerátor | Függvénykifejezés |
168           "not" Tényező | "(" Kifejezés ")".
169 Relációjel = "=" | "<>" | "<" | "<=" | ">" | ">=" | "in".
170 AdditívMűveletiJel = "+" | "-" | "or".
171 MultiplikatívMűveletiJel = "*" | "/" | "div" | "mod" | "and".
172 ElőjelNélküliKonstans = ElőjelNélküliSzám | Karakterfüzér |
173                         Konstansazonosító | "nil".
174 Függvénykifejezés = Függvényazonosító [AktuálisParaméterlista].
175
176 ------------
177
178 Változó = TeljesVáltozó | Elemváltozó | AzonosítottVáltozó | Pufferváltozó.
179
180 TeljesVáltozó = Változóazonosító.
181 Elemváltozó = IndexeltVáltozó | Mezőkifejezés.
182 DinamikusVáltozó = Mutatóváltozó "↑".
183 Pufferváltozó = Állományváltozó "↑".
184 IndexeltVáltozó = Tömbváltozó "[" Index {"," Index} "]".
185 Mezőkifejezés = [Rekordváltozó "."] Mezőazonosító.
186 Halmazgenerátor = "[" [Elemleírás {"," Elemleírás} "]".
187 Elemleírás = MegszámlálhatóKifejezés [".." MegszámlálhatóKifejezés].
188 AktuálisParaméterlista = "(" AktuálisParaméter {"," AktuálisParaméter} ")".
189 AktuálisParaméter = Kifejezés | Változó | Eljárásazonosító |
190                     Függvényazonosító.
```

```
WriteParaméterlista = "(" (Állományváltozó | WriteParaméter)
                          [ {"," WriteParaméter} ")".
WriteParaméter = Kifejezés [ ":" EgészKifejezés
                    [ ":" EgészKifejezés] ].
Tömbváltozó = Változó.
Rekordváltozó = Változó.
Állományváltozó = Változó.
Mutatóváltozó = Változó.
EgészKifejezés = MegszámlálhatóKifejezés.
LogikaiKifejezés = MegszámlálhatóKifejezés.
MegszámlálhatóKifejezés = Kifejezés.

------------------

MutatóTípusAzonosító = Típusazonosító.
StrukturáltTípusAzonosító = Típusazonosító.
MegszámlálhatóTípusAzonosító = Típusazonosító.
ValósTípusAzonosító = Típusazonosító.
Konstansazonosító = Azonosító.
Típusazonosító = Azonosító.
Változóazonosító = Azonosító.
Mezőazonosító = Azonosító.
Eljárásazonosító = Azonosító.
Függvényazonosító = Azonosító.
Határazonosító = Azonosító.


ElőjelNélküliSzám = ElőjelNélküliEgész | ElőjelNélküliValós.
Azonosítólista = Azonosító {"," Azonosító}.

------------------

Azonosító = Betű {Betű | Számjegy}.
Direktíva = Betű {Betű | Számjegy}.
Címke = Számjegysorozat.
ElőjelNélküliEgész = Számjegysorozat.
ElőjelNélküliValós = ElőjelNélküliEgész "." Számjegysorozat ["e" Karakterisztika] |
                        ElőjelNélküliEgész "e" Karakterisztika.
Karakterisztika = [Előjel] ElőjelNélküliEgész.
Előjel = "+" | "-".
Karakterfüzér = "'" Füzérelem {Füzérelem} "'".
Számjegysorozat = Számjegy {Számjegy}.

Betű = "a" | "b" | "c" | "d" | "e" | "f" | "g" |
       "h" | "i" | "j" | "k" | "l" | "m" | "n" |
       "o" | "p" | "q" | "r" | "s" | "t" | "u" |
       "v" | "w" | "x" | "y" | "z".
Számjegy = "0" | "1" | "2" | "3" | "4" | "5" | "6" |
           "7" | "8" | "9".
Füzérelem = "" | BármelyAposztrófTólKülönbözőKarakter.
```

Az EBNF-szimbólumok előfordulási helye a Jelentésben és a hierarchikus leírásban

Előfordulási hely a Jelentésben	Metaazonosító/Alapszó	EBNF kereszthivatkozások
8.	*AdditívMűveletiJel*	160 170
11.3.2.	*AktuálisParaméter*	188 188 189
11.3.2.	*AktuálisParaméterlista*	92 174 188
6.2.3.	*Alaptípus*	136 140
6.2.4.	*ÁllományTípus*	125 137
7.4.	*Állományváltozó*	183 191 197
4.	*Azonosító*	2 26 27 38 40 71 71 153 209 210 211 212 213 214 215 219 219 223
6.1.1.	*Azonosítólista*	3 28 53 56 127 149 219
4.	*BármelyAposztróftólKülönbözőKarakter*	240
4.	*Betű*	223 223 224 224 234
10.1.	*Blokk*	1 7 29 31 32 34
9.2.2.2.	*CaseUtasítás*	85 97
9.2.3.	*CiklusUtasítás*	84 86
9.2.3.3.	*Ciklusváltozó*	105 112
10.1.1.	*Címke*	79 94 225
10.1.1.	*CímkedeklarációsRész*	7 13
7.3.	*DinamikusVáltozó*	179 182
4.	*Direktíva*	30 33 224
8.	*EgészKifejezés*	193 194 199
8.	*EgyszerűKifejezés*	162 163 164
6.1.	*EgyszerűTípus*	118 119
9.1.	*EgyszerűUtasítás*	80 81
8.	*Elemleírás*	186 186 187
6.2.1.	*Elemtípus*	132 137 139
7.2.	*Elemváltozó*	178 181
11.1.	*Eljárásazonosítás*	31 39
11.1.	*Eljárásazonosító*	39 92 189 213
11.1.	*Eljárásdeklaráció*	21 29
11.	*EljárásÉsFüggvénydeklarációsRész*	11 20
11.1.	*Eljárásfej*	29 30 38 59
11.3.1.2.	*Eljárásparaméterspecifikáció*	47 58
9.1.2	*Eljárásutasítás*	82 92
4.	*Előjel*	157 164 229 230

Előfordulási hely a Jelentésben	Metaazonosító/Alapszó	EBNF kereszthivatkozások
6.1.	*MegszámlálhatóTípus*	119 123 138 140
6.1.	*MegszámlálhatóTípusAzonosító*	71 124 141 152 207
6.2.2.	*Mezőazonosító*	185 212
7.2.2.	*Mezőkifejezés*	181 185
6.2.2.	*Mezőlista*	134 143 151
8.	*MultiplikatívMűveletiJel*	165 171
6.3.	*MutatóTípus*	118 122
6.3.	*MutatóTípusAzonosító*	122 142 205
7.3.	*Mutatóváltozó*	182 198
9.2.1.	*ÖsszetettUtasítás*	22 75 83
13.	*Program*	1
13.	*Programfej*	1 2
13.	*ProgramParaméterlista*	2 3
7.4.	*Pufferváltozó*	179 183
6.2.2.	*Rekordszakasz*	145 145 149
6.2.2.	*RekordTípus*	125 133
7.2.2.	*Rekordváltozó*	109 109 185 196
9.2.4.	*RekordVáltozólista*	107 109
8.	*RelációsJel*	162 169
9.2.3.2.	*RepeatUtasítás*	86 100
6.1.3.	*RésztartományTípus*	123 128
6.2.2.	*RögzítettRész*	143 145
6.2.	*StrukturáltTípus*	118 120
6.2.	*StrukturáltTípusAzonosító*	121 206
9.2.	*StrukturáltUtasítás*	80 83
4.	*Számjegy*	223 224 232 232 238
4.	*Számjegysorozat*	13 13 225 226 227 232
8.	*Tag*	164 164 165
7.1.	*TeljesVáltozó*	178 180
8.	*Tényező*	165 165 166 168
6.	*Típus*	27 28 118 139 149
6.	*Típusazonosító*	53 56 66 70 126 205 206 207 208 210
6.	*Típusdefiníció*	16 17 27
6.	*TípusdefiníciósRész*	9 16
6.2.1.	*TömbTípus*	125 131
7.2.1.	*Tömbváltozó*	184 195
11.3.1.1.	*TömörítetlenIlleszkedőtömbséma*	63 67
6.2.	*TömörítetlenStrukturáltTípus*	120 125
11.3.1.1.	*TömörítettIlleszkedőtömbséma*	62 64
9.	*Utasítás*	78 78 79 96 96 104 106 108 111
9.	*Utasításrész*	12 22
9.2.	*Utasítássorozat*	76 78 101
9.1.	*ÜresUtasítás*	81 90
6.1.	*ValósTípusAzonosító*	119 141 208

Előfordulási hely a Jelentésben	Metaazonosító/Alapszó	EBNF kereszthivatkozások
6.2.2.	*Változat*	147 148 151
6.2.2.	*VáltozatRész*	143 143 146
6.2.2.	*Változatszelektor*	146 150
7.	*Változó*	91 166 178 189
7.	*Változóazonosító*	112 180 211
7.	*Változódeklaráció*	18 19 28
7.	*VáltozódeklarációsRész*	10 18
11.3.1.1.	*Változóparaméterspecifikáció*	46 55
9.2.3.3.	*Végérték*	106 114
9.2.3.1.	*WhileUtasítás*	86 103
9.2.4.	*WithUtasítás*	84 107
12.3.	*WriteParaméter*	191 192 193
12.3.	*WriteParaméterlista*	93 191
	and	171
	array	65 68 131
	begin	75
	case	97 146
	const	14
	div	171
	do	103 106 107
	downto	106
	else	96
	end	77 99 135
	file	137
	for	105
	function	40 42
	goto	94
	if	94
	in	169
	label	13
	mod	171
	nil	173
	not	168
	of	65 69 97 131 136 137 146
	or	170
	packed	65 120
	procedure	38 39
	program	2
	record	133
	repeat	100
	set	136
	then	95
	to	106
	type	16
	until	102
	var	18 56
	while	103
	with	107

Az EBNF-szabályok ábécésorrendben

AdditívMűveletiJel = "+" | "−" | "or".
AktuálisParaméter = *Kifejezés* | *Változó* | *Eljárásazonosító* | *Függvényazonosító*.
AktuálisParaméterlista = "(" *AktuálisParaméter* { "," *AktuálisParaméter* } ")".
Alaptípus = *MegszámlálhatóTípus*.
ÁllományTípus = "file" "of" *Elemtípus*.
Állományváltozó = *Változó*.
Azonosító = *Betű* { *Betű* | *Számjegy* }.
Azonosítólista = *Azonosító* { "," *Azonosító* }.
Betű = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
"i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
"q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
"y" | "z".
Blokk = *CímkedeklarációsRész*
KonstansdefiníciósRész
TípusdefiníciósRész
VáltozódeklarációsRész
Utasításrész.
CaseUtasítás = "case" *Esetindex* "of" *Eset* { ";" *Eset* } [";"] "end".
Ciklusutasítás = *WhileUtasítás* | *RepeatUtasítás* | *ForUtasítás*.
Ciklusváltozó = *Változóazonosító*.
Címke = *Számjegysorozat*.
CímkedeklarációsRész = ["label" *Számjegysorozat* { "," *Számjegysorozat* } ";"].
DinamikusVáltozó = *Mutatóváltozó* "↑".
Direktíva = *Betű* { *Betű* | *Számjegy* }.
EgészKifejezés = *MegszámlálhatóKifejezés*.
EgyszerűKifejezés = [*Előjel*] *Tag* { *AdditívMűveletiJel Tag* }.
EgyszerűTípus = *MegszámlálhatóTípus* | *ValósTípusAzonosító*.
EgyszerűUtasítás = *ÜresUtasítás* | *ÉrtékadóUtasítás* | *Eljárásutasítás* | *GotoUtasítás*.
Elemleírás = *MegszámlálhatóKifejezés* [". ." *MegszámlálhatóKifejezés*].
Elemtípus = *Típus*.
Elemváltozó = *IndexeltVáltozó* | *Mezőkifejezés*.
Eljárásazonosítás = "procedure" *Eljárásazonosító*.
Eljárásazonosító = *Azonosító*.
Eljárásdeklaráció = *Eljárásfej* ";" *Blokk* | *Eljárásfej* ";" *Direktíva* | *Eljárásazonosítás* ";" *Blokk*.
EljárásÉsFüggvénydeklarációsRész = { *(Eljárásdeklaráció* | *Függvénydeklaráció)* ":" }.
Eljárásfej = "procedure" *Azonosító* [*FormálisParaméterlista*].
Eljárásparaméterspecifikáció = *Eljárásfej*.

EljárásUtasítás = *Eljárásazonosító* [*AktuálisParaméterlista* | *WriteParaméterlista*].
Előjel = "+" | "−".
ElőjelNélküliEgész = *Számjegysorozat*.
ElőjelNélküliKonstans = *ElőjelNélküliSzám* | *Karakterfüzér* | *Konstansazonosító* | "nil".
ElőjelNélküliValós = *ElőjelNélküliEgész* "." *Számjegysorozat* ["e" *Karakterisztika*] | *ElőjelNélküliEgész* "e" *Karakterisztika*.
Eredménytípus = *MegszámlálhatóTípusAzonosító* | *ValósTípusAzonosító* | *MutatóTípusAzonosító*.
ÉrtékadóUtasítás = *(Változó* | *Függvényazonosító)* ":=" *Kifejezés*.
Értékparaméterspecifikáció = *Azonosítólista* ":" *(Típusazonosító* | *Illeszkedőtömbséma)*.
Eset = *Konstans* {"," *Konstans* } ":" *Utasítás*.
Esetindex = *MegszámlálhatóKifejezés*.
FelsoroltTípus = "(" *Azonosítólista* ")".
FeltételesUtasítás = *IfUtasítás* | *CaseUtasítás*.
FormálisParaméterlista = "(" *FormálisParaméterrész* { ";" *FormálisParaméterrész* } ")".
FormálisParaméterrész = *Változóparaméterspecifikáció* | *Értékparaméterspecifikáció* | *Eljárásparaméterspecifikáció* | *Függvényparaméterspecifikáció*.
ForUtasítás = "for" *ciklusváltozó* ":=" *Kezdőérték* ("to" | "downto") *Végérték* "do" *Utasítás*.
Főtípus = *Típusazonosító*.
Függvényazonosítás = "function" *Függvényazonosító*.
Függvényazonosító = *Azonosító*.
Függvénydeklaráció = *Függvényfej* ";" *Blokk* | *Függvényfej* ";" *Direktíva* | *Függvényazonosítás* ";" *Blokk*.
Függvényfej = "function" *Azonosító* [*FormálisParaméterlista*] ":" *Eredménytípus*.
Függvénykifejezés = *Függvényazonosító* [*AktuálisParaméterlista*].
Függvényparaméterspecifikáció = *Függvényfej*.
Füzérelem = "' " | *BármelyAposztrófotólKülönbözőKarakter*.
GotoUtasítás = "goto" *Címke*.
Halmazgenerátor = "[" [*Elemleírás* { "," *Elemleírás* }] "]".
HalmazTípus = "set" "of" *Alaptípus*.
Határozonosító = *Azonosító*.
IfUtasítás = "if" *LogikaiKifejezés* "then" *Utasítás* ["else" *Utasítás*].
Illeszkedőtömbséma = *TömörítettIlleszkedőtömbséma* | *TömörítetlenIlleszkedőtömbséma*.
IndexeltVáltozó = *Tömbváltozó* "[" *Index* { "," *Index* } "]".
Indextípus = *MegszámlálhatóTípus*.
Indextípusspecifikáció = *Azonosító* ".." *Azonosító* ":" *MegszámlálhatóTípusAzonosító*.
Karakterfüzér = "' " *Füzérelem* { *Füzérelem* } "' ".
Karakterisztika = [*Előjel*] *ElőjelNélküliEgész*.
Kezdőérték = *MegszámlálhatóKifejezés*.
Kifejezés = *EgyszerűKifejezés* [*RelációsJel EgyszerűKifejezés*].
Kijelölőmező = *Azonosító*.
KijelölőTípus = *MegszámlálhatóTípusazonosító*.
Konstans = [*Előjel*] *(ElőjelNélküliSzám* | *Konstansazonosító)* | *Karakterfüzér*.

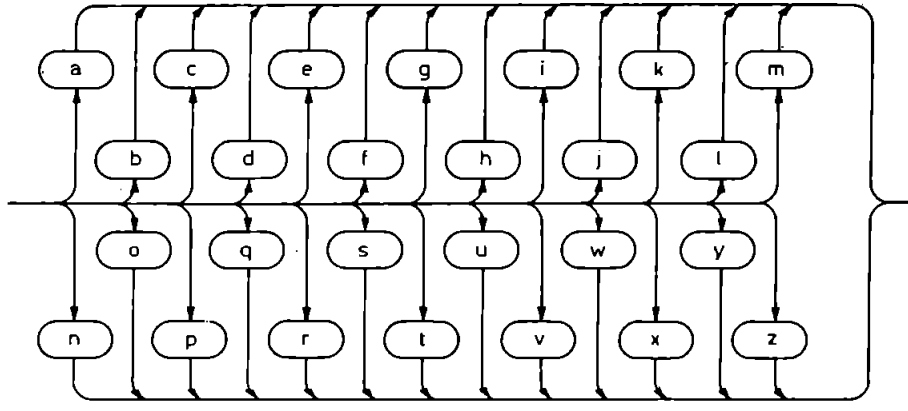
```
Konstansazonosító = Azonosító.
Konstansdefiníció = Azonosító "=" Konstans.
KonstansdefiníciósRész = [ "const" Konstansdefiníció ";" {Konstansdefiníció } ";" ].
LogikaiKifejezés = MegszámlálhatóKifejezés.
MegszámlálhatóKifejezés = Kifejezés.
MegszámlálhatóTípus = FelsoroltTípus | RésztartományTípus |
                      MegszámlálhatóTípusAzonosító.
MegszámlálhatóTípusAzonosító = Típusazonosító.
Mezőazonosító = Azonosító.
Mezőkifejezés = [Rekordváltozó "," ] Mezőazonosító.
Mezőlista = [(RögzítettRész [";" VáltozatRész] | VáltozatRész)] |[";" ].
MultiplikatívMűveletiJel = "*" | "/" | "div" | "mod" | "and".
MutatóTípus = "↑" Főtípus | MutatóTípusAzonosító.
MutatóTípusAzonosító = Típusazonosító.
Mutatóváltozó = Változó.
ÖsszetettUtasítás = "begin" Utasítássorozat "end".
Program = Programfej ";" Blokk ".".
Programfej = "program" Azonosító [ProgramParaméterlista].
Programparaméterlista = "(" Azonosítólista ")".
Pufferváltozó = Állományváltozó "↑".
Rekordszakasz = Azonosítólista ":" Típus.
RekordTípus = "record"
                Mezőlista
              "end".
Rekordváltozó = Változó.
RekordVáltozólista = Rekordváltozó { "," Rekordváltozó}.
RelációsJel = "=" | "<>" | "<" | "<=" | ">" | ">=" | "in".
RepeatUtasítás = "repeat"
                   Utasítássorozat
                 "until" LogikaiKifejezés.
RésztartományTípus = Konstans ".." Konstans.
RögzítettRész = Rekordszakasz { ";" Rekordszakasz }.
StrukturáltTípus = [ "packed" ] TömörítetlenStrukturáltTípus |
                     StrukturáltTípusAzonosító.
StrukturáltTípusAzonosító = Típusazonosító.
StrukturáltUtasítás = ÖsszetettUtasítás | FeltételesUtasítás | Ciklusutasítás |
                      WithUtasítás.
Számjegy = "0" | "1" | "2" | "3" | "4" | "5" |
           "6" | "7" | "8" | "9".
Számjegysorozat = Számjegy { Számjegy }.
Tag = Tényező { MultiplikatívMűveletiJel Tényező}.
TeljesVáltozó = Változóazonosító.
Tényező = ElőjelNélküliKonstans | Határazonosító | Változó | Halmazgenerátor |
          Függvénykifejezés | "not" Tényező | "(" Kifejezés ")".
Típus = EgyszerűTípus | StrukturáltTípus | MutatóTípus.
Típusazonosító = Azonosító.
Típusdefiníció = Azonosító "=" Típus.
TípusdefiníciósRész = [ "type" Típusdefiníció ";" { Típusdefiníció ";" } ].
TömbTípus = "array" "[" Indextípus {"," Indextípus } "]" "of" Elemtípus.
Tömbváltozó = Változó.
```

TömörítetlenIlleszkedőtömbséma = "array" "[" *Indextípusspecifikáció*
{";" *Indextípusspecifikáció*} "]" "of"
(Típusazonosító | *(Illeszkedőtömbséma).*
TömörítetlenStrukturáltTípus = *TömbTípus* | *RekordTípus* | *HalmazTípus* |
ÁllományTípus.
TömörítettIlleszkedőtömbséma = "packed" "array" "[" *Indextípusspecifikáció*
"]" "of" *Típusazonosító.*
Utasítás = [*Címke* ":"]
EgyszerűUtasítás | *StrukturáltUtasítás.*
Utasításrész = *ÖsszetettUtasítás.*
Utasítássorozat = *Utasítás*{";" *Utasítás*}.
ÜresUtasítás = .
ValósTípusAzonosító = *Típusazonosító.*
Változó = *TeljesVáltozó* | *Elemváltozó* | *DinamikusVáltozó* | *Pufferváltozó.*
Változóazonosító = *Azonosító.*
Változódeklaráció = *Azonosítólista* ":" *Típus.*
VáltozódeklarációsRész = ["var" *Változódeklaráció* ";" { *Változódeklaráció* ";" }].
Változóparaméterspecifikáció = "var" *Azonosítólista* ":" *(Típusazonosító* |
Illeszkedőtömbséma).
Változat = *Konstans* {"," *Konstans*} ":" "(" *Mezőlista* ")".
VáltozatRész = "case" *Változatszelektor* "of" *Változat* { ";" *Változat*}.
Változatszelektor = { *Kijelölőmező* ":" } *KijelölőTípus.*
Végérték = *DiszkrétKifejezés.*
WhileUtasítás = "while" *LogikaiKifejezés* "do" *Utasítás.*
WithUtasítás = "with" *RekordVáltozólista* "do" *Utasítás.*
WriteParaméter = *Kifejezés* [":" *EgészKifejezés* [":" *EgészKifejezés*]].
WriteParaméterlista = "(" *(Állományváltozó* | *WriteParaméter)*
{"," *WriteParaméter*} ")".

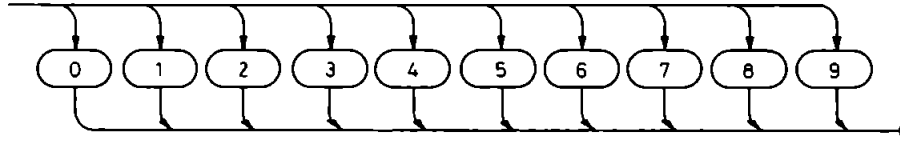
Szintaxisdiagramok

A *Betű*, *Számjegy*, *Azonosító*, *Direktíva*, *ElőjelNélküliEgész*, *ElőjelNélküliSzám* és *Karakterfüzér* feliratú diagramok azt mutatják be, hogyan épülnek föl a szótári szimbólumok a karakterekből. A többi diagramról az olvasható le, hogyan állnak össze a szintaktikus konstrukciók a szimbólumokból.

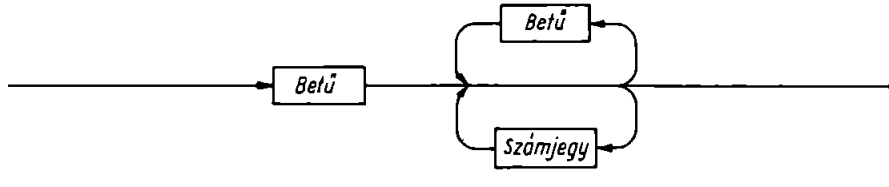
Betű



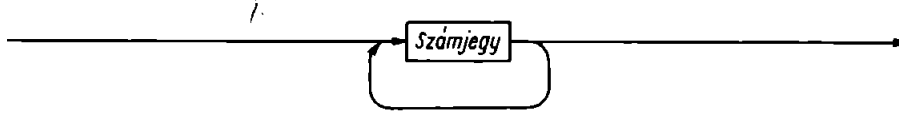
Számjegy



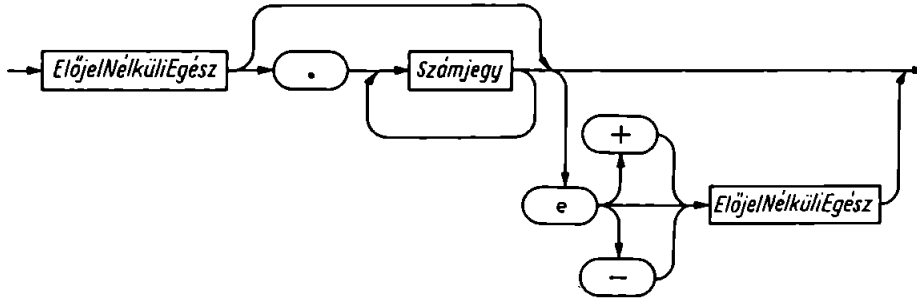
Azonosító és direktíva



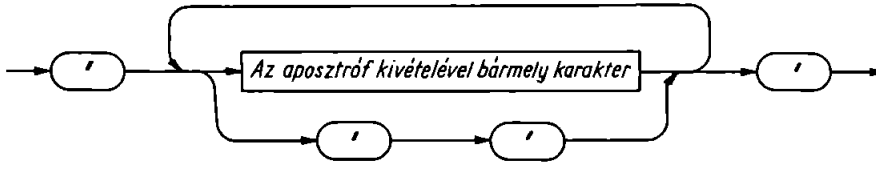
ElőjelNélküliEgész



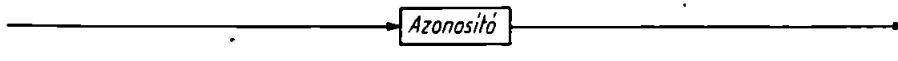
ElőjelNélküliSzám



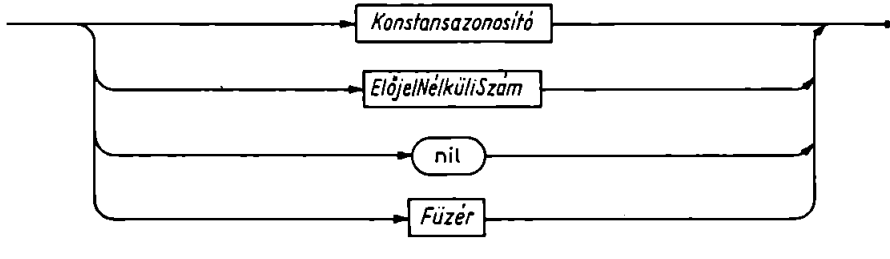
Karakterfüzér



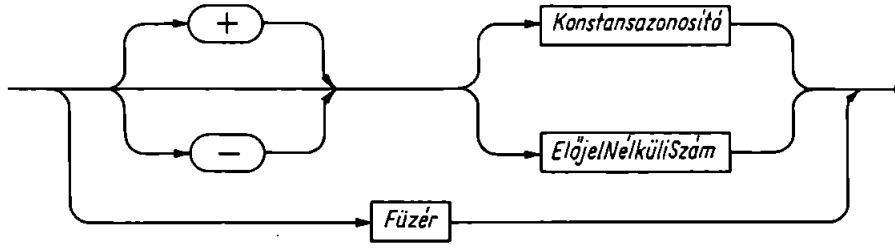
Konstansazonosító, Változóazonosító, Mezőazonosító, Határazonosító, Típusazonosító, Eljárásazonosító, Függvényazonosító



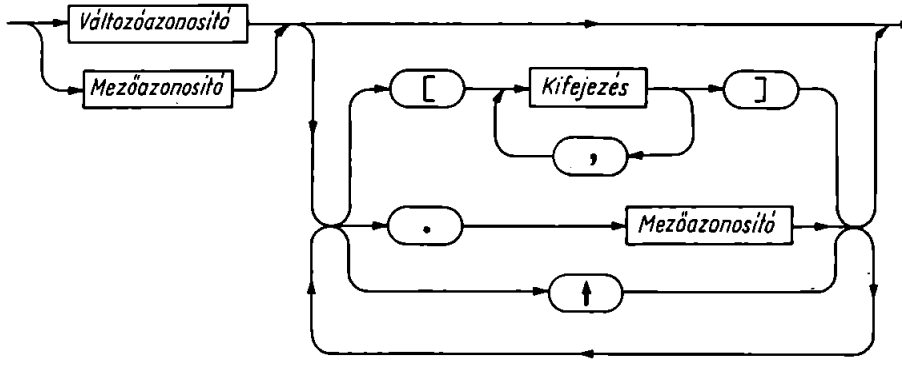
ElőjelNélküliKonstans



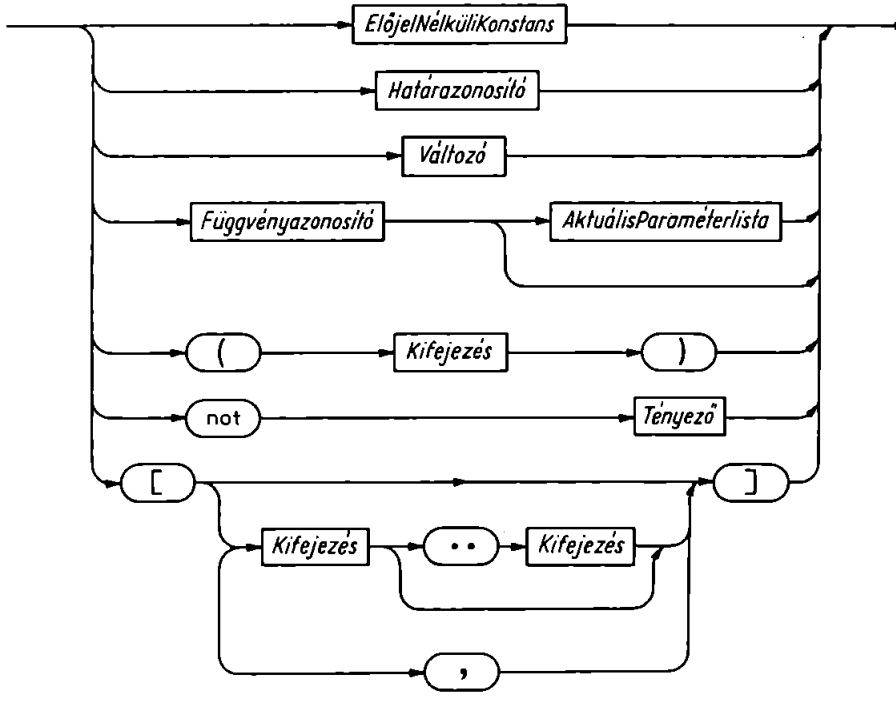
Konstans



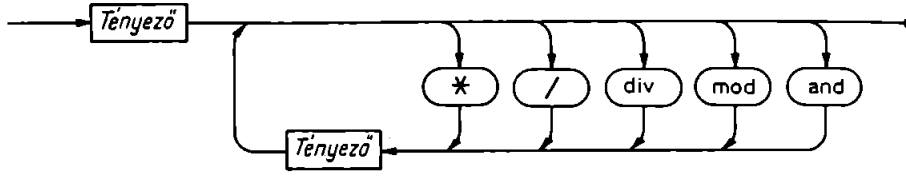
Változó



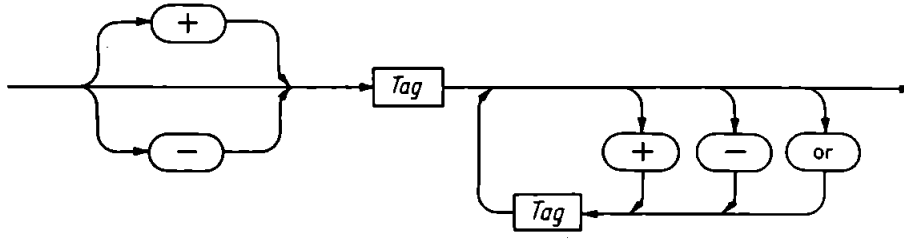
Tényező



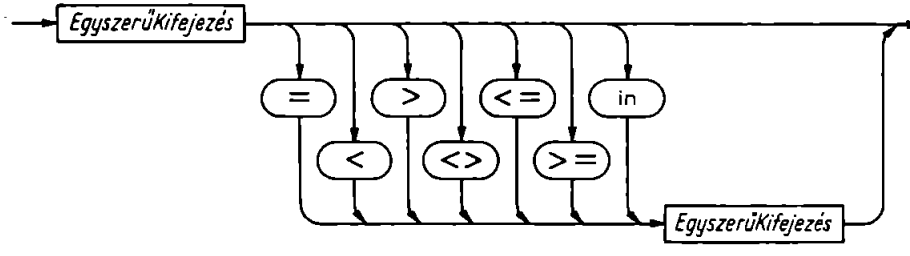
Tag



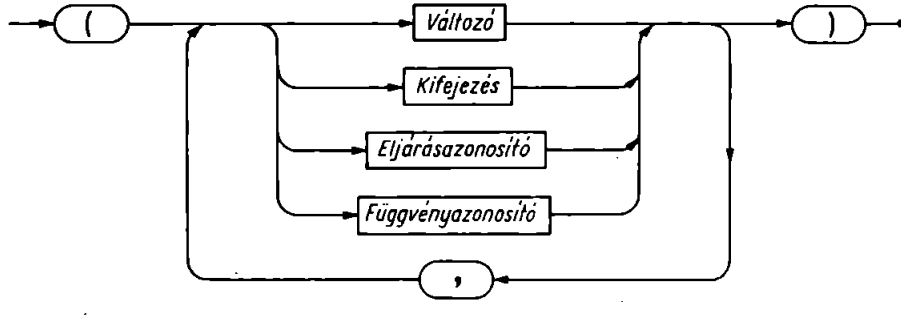
EgyszerűKifejezés



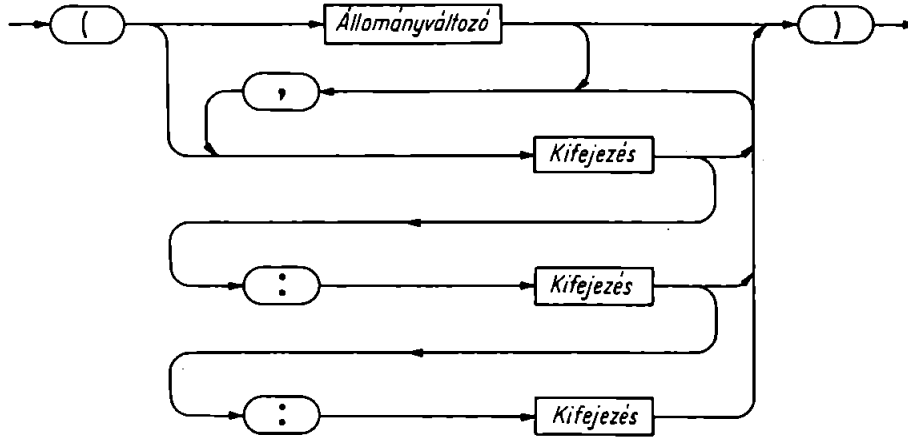
Kifejezés



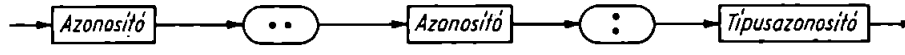
AktuálisParaméterlista



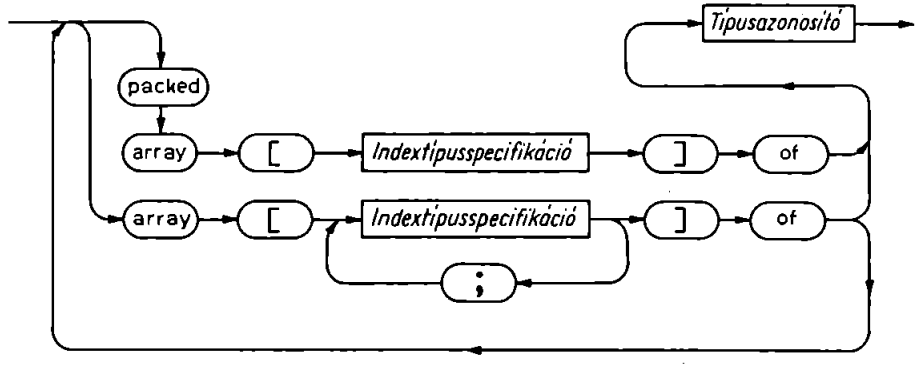
WriteParaméterlista



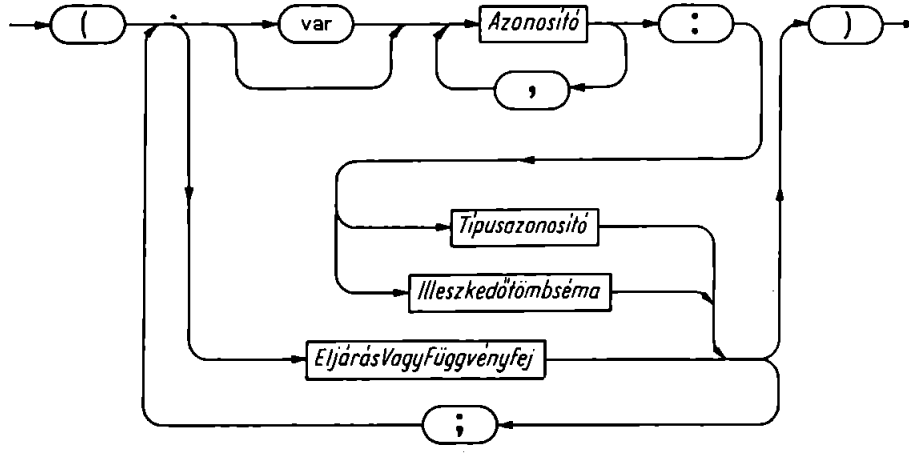
Indextípusspecifikáció



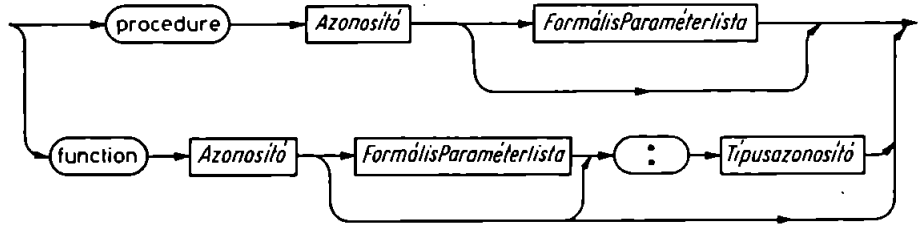
Illeszkedőtömbséma



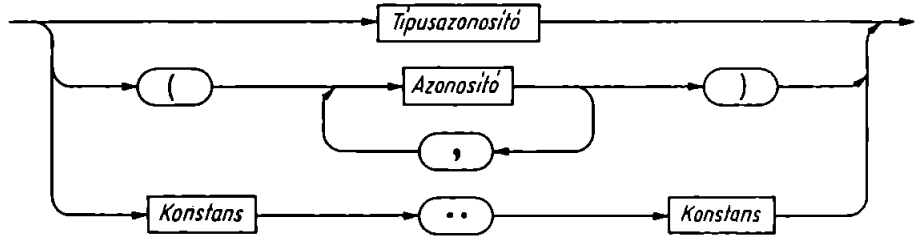
FormálisParaméterlista



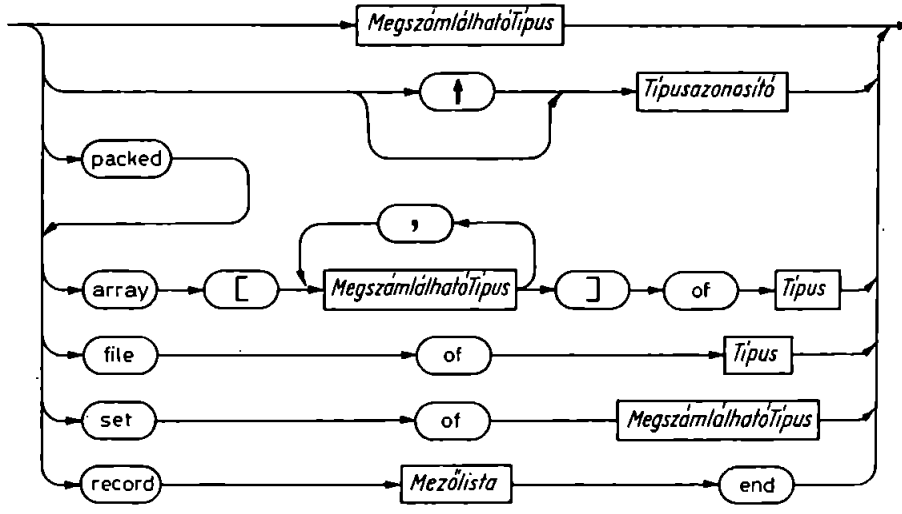
EljárásVagyFüggvényfej



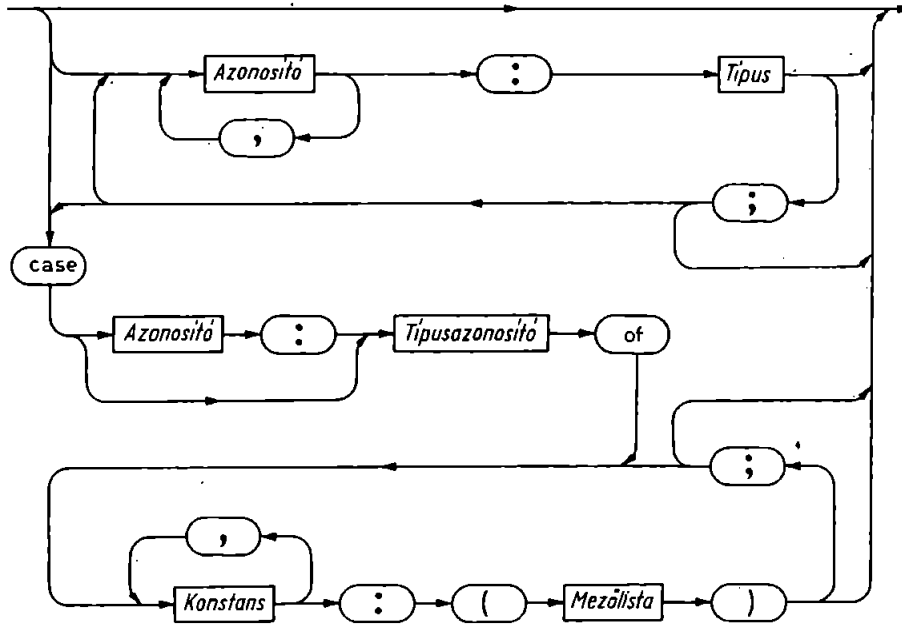
MegszámlálhatóTípus

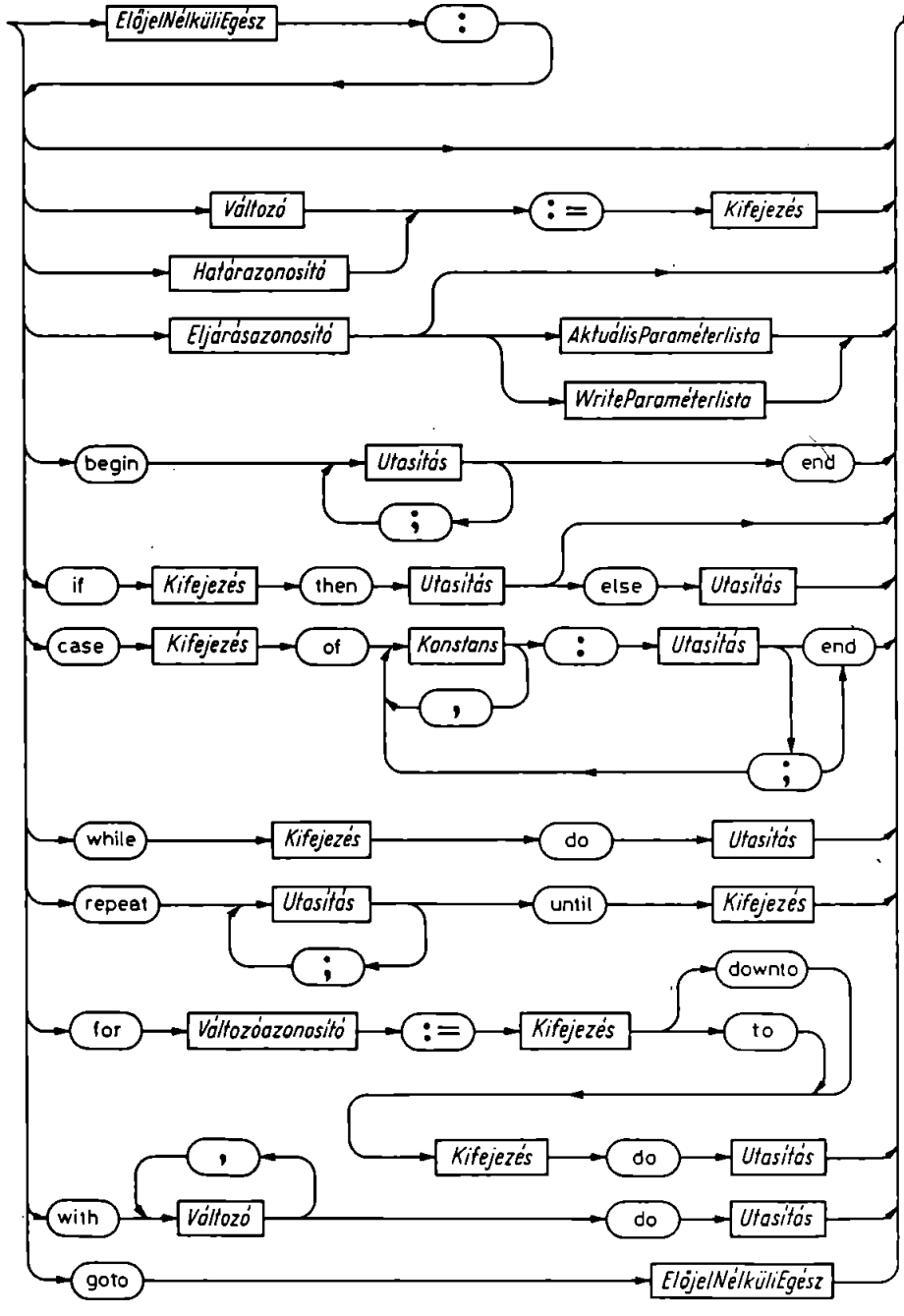


Típus

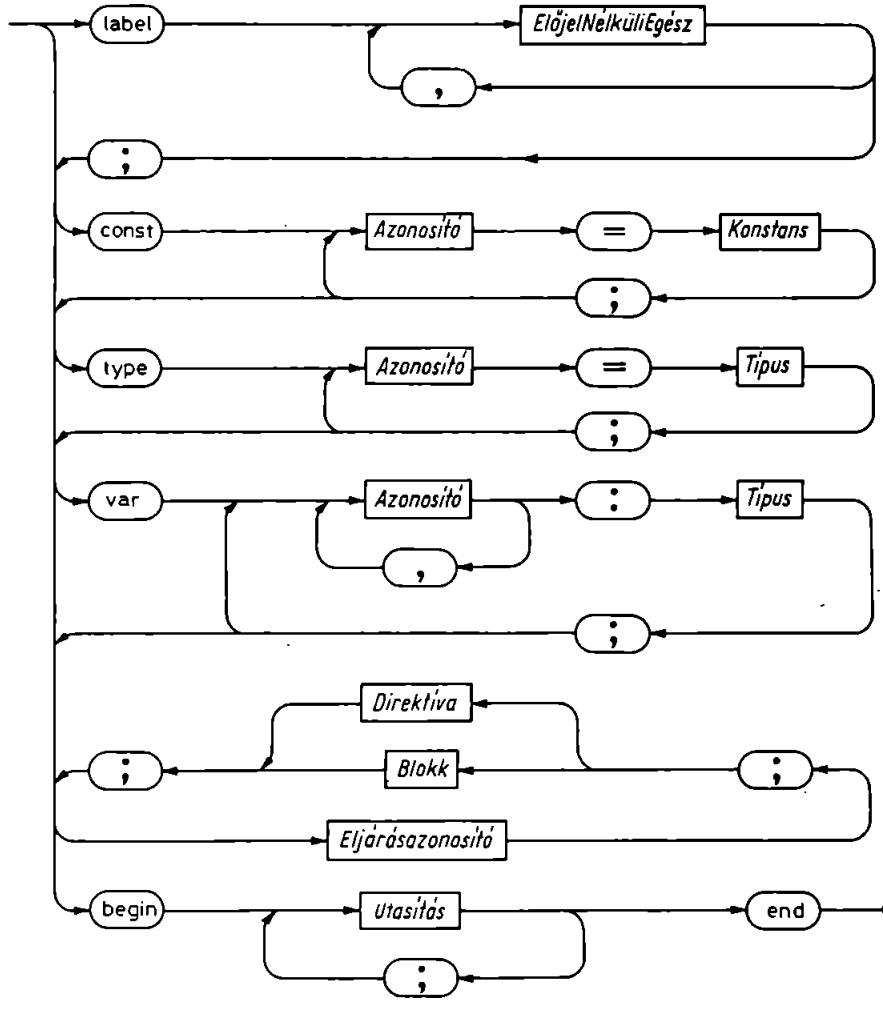


Mezőlista

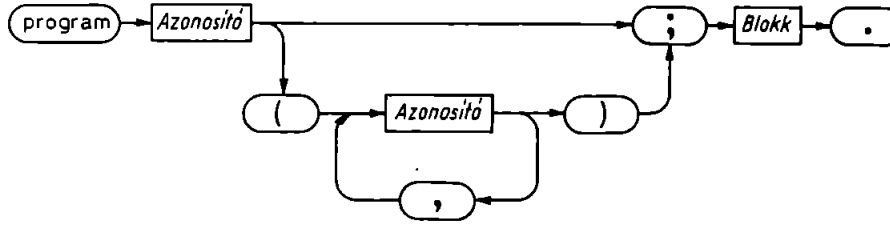



ElőjelNélküliEgész
:
Változó
:=
Kifejezés
Határazonosító
Eljárásazonosító
AktuálisParaméterlista
WriteParaméterlista
begin
Utasítás
;
end
if
Kifejezés
then
Utasítás
else
Utasítás
case
Kifejezés
of
Konstans
,
:
Utasítás
end
;
while
Kifejezés
do
Utasítás
repeat
Utasítás
;
until
Kifejezés
for
Változóazonosító
:=
Kifejezés
downto
to
Kifejezés
do
Utasítás
with
,
Változó
do
Utasítás
goto
ElőjelNélküliEgész

Blokk



Program



E. függelék

A mű első kiadásának eltérései az ISO 7185 szabványtól

Ez a függelék azokról a technikai jellegű módosításokról ad hozzávetőleges, korántsem teljes áttekintést, amelyeket a könyvön a harmadik (ISO szabvány) kiadásra való előkészítés miatt hajtottunk végre. Az összefoglaló azok számára lehet hasznos, akiknek az előző kiadások valamelyike birtokában van.

Jelentés, 3. Jelölésmód és terminológia

BNF helyett az EBNF-et használtuk.

Definiáltuk a *hiba*, a *rendszer által meghatározott* és a *rendszerfüggő* konstrukciók, továbbá a *bővítés* és a *standard Pascal* fogalmát, és a Jelentésben mindvégig következetesen alkalmaztuk őket.

Jelentés, 4. Szimbólumok és elválasztók

Megváltoztattuk a szintaxist leíró szabályokon belül az elválasztó- és határolójeleket.

Bevezettük a ". ." szimbólumot.

A "[,]" és a "↑" speciális szimbólumok jelölésére alternatív lehetőséget engedtünk meg.

Módosítottuk a magyarázatokra vonatkozó szintaktikus előírásokat; tilos a magyarázatok egymásba ágyazása!

Két azonosító különböző, ha karaktersoraik akárhányadik karakterben különböznek.

A szimbólumok új osztálya: a direktívák.

Jelentés, 5. Konstansok

Az új Jelentésben szerepel a MaxInt.

Jelentés, 6. Típusok

A skalár típusok helyett bevezettük a megszámlálható és a valós típusokat; definiáltuk a succ, pred és ord függvényeket és ezáltal egyszerűsítettük a tömbök indexelését, az esetválasztást, a résztartomány típust és a halmazok alaptípusát.

A típusok kompatibilitását neveik kompatibilitásával definiáltuk.

Bevezettük az *értékadás-kompatibilitást* és a *hozzárendelhető típust*.

A strukturált típusok tömörítése más szemantikus következményekkel jár.

A rekord típusok szintaxisát egy opcionális ";" jellel egészítettük ki.

A rekordváltozatokban az egyes eseteket megkülönböztető címkéket most konstansoknak nevezzük.

A rekord típusokban a változat részek teljes specifikálására szükség van.

Az állomány típusok esetében megkülönböztettük a feldolgozási és a generálási módot.

A Text típus nem ekvivalens a Char elemtípusú (tömörített) állománytípussal.

Az állomány típusokat és az állomány típust tartalmazó típusokat (tehát a nem hozzárendelhető típusokat) nem engedjük állomány típusok elemtípusaként alkalmazni.

Bevezettük a mutató típusok főtípusát.

Jelentés, 7. Változók

Bevezettük a *határozatlan* és a *teljesen határozatlan* változók fogalmát.

Ha egy programban az Input és Output azonosítókat használjuk, azok implicit módon deklarált, szöveg típusú állomány-programparaméterek lesznek.

Jelentés, 8. Kifejezések

Tényezőnek tekintjük az illeszkedőtömb-paraméterek határozonosítóit is.

A kifejezések kiértékelésének sorrendjét rendszerfüggőnek tekintjük.

Módosítottuk a mod műveleti jel definícióját.

Egy halmazgenerátor típus lehet tömörített és tömörítetlen is.

Jelentés, 9. Utasítások

A goto utasításokkal elérhető címkékre vonatkozó szabályokat megszigorítottuk.

A case utasítás címkéit konstansoknak neveztük el.

A for utasítás ciklusváltozója csak lokális változó lehet.

Több korlátozást vezettünk be a for utasításra nézve, és pontosan definiáltuk az általa végrehajtott műveleteket.

Jelentés, 10. Blokkok, hatáskör, aktiválás

Definiáltuk a *programpont, aktiválási pont,* továbbá a címkék és azonosítók *definíciójának vagy deklarációjának (bevezetésének) hatásköre* fogalmát.

Pontosan definiáltuk a hatáskörre vonatkozó szabályokat, felszámolva a kétértelműségeket.

A címkéknek megfelelő egész számok értéke nem haladhatja meg a 9999-et.

Jelentés, 11. Eljárások és függvények

Bevezettük az eljárásokra és függvényekre vonatkozó direktívákat; a forward standard direktíva.

A paraméterek közé felvettük az illeszkedőtömböket; bevezettük az *illeszthetőség* fogalmát és az illeszkedő típust.

A formális eljárás- és függvényparaméterek (azaz a paraméterként alkalmazott eljárások és függvények) paraméterlistáit teljes egészükben meg kell adni; bevezettük a *paraméterlisták kongruenciájának* fogalmát.

Megtiltottuk a kijelölőmezők aktuális változóparaméterekként való alkalmazását.

Módosítottuk a pack és unpack eljárások tömbparamétereinek specifikációját.

Pontosan definiáltuk az állománykezelő eljárásokat és függvényeket, továbbá az állományváltozó és a pufferváltozó állapotát.

Jelentés, 12. Szöveg típusú állományok be- és kivitele

A page eljárás standard, állományparamétere opcionális; hatása módosult.

A write és writeln eljárások aktuális paraméterlistájaként bevezettük a *WriteParaméterlistákat,* és speciális szintaktikus szabályokat írtunk elő rájuk.

Az előírt formátummal dolgozó write és writeln eljárások mezőszélesség-paramétereit pontosan definiáltuk.

Jelentés, 13. Programok

A programparaméterek opcionálisak; tulajdonságaikat meghatároztuk.

Jelentés, 14. Összhang az ISO 7185-tel

Definiáltuk a *szabvánnyal összhangban lévő program és processzor* fogalmát.
Kifejtettük, milyen követelményeket jelent az ISO Pascal-szabvánnyal való összhang.

F. függelék

Programozási példák

Két példát mutatunk be az alábbiakban: az első a programok lépésről lépésre történő kifejlesztését [2] illusztrálja, majd egy olyan eljárás következik, amely a többféle rendszeren futtatható programok számára szolgálhat modellként.

1. Példa: Program Palindrome

Olyan programot írunk, amelynek segítségével megkereshető valamennyi 1 és 100 közötti egész szám, amelynek négyzete palindrom. Például: 11 négyzete 121, ami palindrom.

Palindromnak valamely ábécé szimbólumaiból alkotott olyan füzért nevezünk, amelyet elölről hátra vagy hátulról előre olvasva ugyanazt a sorozatot kapjuk. Ismert magyar nyelvű példák (a szóközöktől és az írásjelektől eltekintve):

„goromba rab morog"
„erőszakos kannak sok a szőre"
„ólba rab dobál, de kenguru rúg neked lábodba, rabló"

1. Példa, 1. lépés:

```
program Palindrome (Output);
begin
    KeressükAzon1És100KözöttiEgészeketMelyekNégyzetePalindrom
end {Palindrome}.
```

1. Példa, 2. lépés:

```
program Palindrome (Output);

    {Keressük azon 1 és 100 közötti egészeket, amelyek négyzete palindrom.}

const
    Maximum = 100;

type
    Egésztartomány = 1..Maximum;

var
    N: Egésztartomány;
```

```
begin
      for N := 1 to Maximum do
            if Palindrom(Sqr(N)) then
                  Writeln(N, ' négyzete palindrom. ')
end {Palindrome}.
```

1. Példa, 3. lépés:

```
program Palindrome (Output);

      { Keressük azon 1 és 100 közötti egészeket, amelyek négyzete palindrom. }

      const
            Maximum = 100;

      type
            Egésztartomány = 1..Maximum;

      var
            N: Egésztartomány;

      function Palindrom (Négyzet: Integer): Boolean;

      var
            NJegyei = 1..5  {5 = Log10(Sqr(Maximum)) + 1} ;

      begin  {Palindrom}
            JegyekLevágása;
            Palindrom  := Szimmetriavizsgálat (1, NJegyei)
      end  {Palindrom} ;

begin
      for N  := 1 to Maximum do
            if Palindrom(Sqr(N)) then
                  Writeln(N, ' négyzete palindrom ')
end  {Palindrome}.
```

1. Példa, 4. lépés:

```
program Palindrome(Output);
{Turbo Pascal}
{Keressuk azon 1 es 100 kozotti egeszeket,
 amelyek negyzete palindrom.}

const Maximum = 100;

type Egesztartomany = 1..Maximum;

var N: Egesztartomany;

function Palindrom(Negyzet: Integer): Boolean;
const Jegyek=5 {=Trunc(log10(Sqr(Maximum)))+1};
type NJegyei=1..Jegyek;
     Egyjegyu=0..9;
     JegyVektor=array [NJegyei] of Egyjegyu;
var Szamjegyek: Jegyvektor;
    Meret: NJegyei;

procedure JegyekLevagasa;
begin Meret:=1;
      while Negyzet>9 do
            begin Szamjegyek[Meret]:=Negyzet mod 10;
                  Negyzet:=Negyzet div 10;
                  Meret:=Meret+1
            end;
      Szamjegyek[Meret]:=Negyzet
end {JegyekLevagasa};

function SzimVizsg(Bal,Jobb: NJegyei): Boolean;
begin if Bal>=Jobb
      then SzimVizsg:=true
      else if Szamjegyek[Bal]=Szamjegyek[Jobb]
           then SzimVizsg:=SzimVizsg(Bal+1,Jobb-1)
           else SzimVizsg:=false
end {SzimVizsg};

begin {Palindrom}
      JegyekLevagasa;
      Palindrom:=SzimVizsg(1,Meret)
end {Palindrom};

begin for N:=1 to Maximum do
          if Palindrom(Sqr(N))
          then Writeln(N:2,' negyzete palindrom.')
end {Palindrome}.
```

A program eredménye:

```
 1 negyzete palindrom.
 2 negyzete palindrom.
 3 negyzete palindrom.
11 negyzete palindrom.
22 negyzete palindrom.
26 negyzete palindrom.
```

2. Példa: Procedure MásAlapúSzámrendszerbenÁbrázoltSzám

Az alábbiakban egy olyan általános eljárást mutatunk be, amely bármely, 2 és 16 közé eső alapú számrendszerben ábrázolt egész szám beolvasására alkalmas.

```
type Alapszam = 2..16;

procedure MasAlapuSzamrendszerbenAbrazoltSzam
          (var F: text; {ez tartalmazza a szamot}
           var E: Boolean; {jelzi a hibat}
           var X: Integer; {eredmeny, ha nincs hiba}
               R: Alapszam {a szamrendszer alapja});
{MasAlapuSzamrendszerbenAbrazoltSzam vegrehajtasanal
 feltetelezzuk, hogy az F szovegallomany olyan hely-
 zetben van, hogy altalanositott szamjegyeknek egy o-
 lyan sorozatat lehet rola beolvasni, amely egy egesz
 szam R alapu szamrendszerben abrazolt alakjat adja.
 Az altalanositott szamjegyek, novekvo sorrendben a
 kovetkezok:
           '0','1','2','3','4','5','6','7','8','9'
           'a','b','c','d','e','f'
 A kisbetuk helyett a nekik megfelelo nagybetuket is
 hasznalhatjuk.
 Az E parameter az alabbi hibak egyikenek elofordula-
 sat jelzi:
     (1) Az F allomany nem egy altalanositott szam-
         jegyekbol allo elemre volt beallitva.
     (2) A szamjegysorozat egy Maxint-nel nagyobb
         szamot abrazol.
     (3) Az altalanositott szamjegysorozatban olyan
         elem fordul elo, amely nem R alapu szamrend-
         szerbeli szamjegy.}

type Szamjegytartomany = 0..15;

var D: Szamjegytartomany;
    V: Boolean;
    S: 0..Maxint;

procedure AltalanositottSzamjegyekAtvaltasa
          (C: Char;
           var V: Boolean;
           var D: Szamjegytartomany);

{Az eljaras megallapitja, hogy C altalanositott szam-
 jegy-e. Ezt V ertekevel jelzi, es ha V igaz, D-nek
 az altalanositott szamjegy numerikus erteket adja.}
```

```
begin {AltalanositottSzamjegyekAtvaltasa}
      V:=C in ['0'..'9','a','b','c','d','e','f',
               'A','B','C','E','F'];
      if V
      then case C of
                '0': D:=0; '1': D:=1; '2': D:=2;
                '3': D:=3; '4': D:=4; '5': D:=5;
                '6': D:=6; '7': D:=7; '8': D:=8;
                '9': D:=9;
                'A','a': D:=10; 'B','b': D:=11;
                'C','c': D:=12; 'D','d': D:=13;
                'E','e': D:=14; 'F','f': D:=15
           end
end {AltalanositottSzamjegyekAtvaltasa};

begin {MasAlapuSzamrendszerbenAbrazoltSzam}
      E:=true;
      AltalanositottSzamjegyekAtvaltasa(F^,V,D);
      if V
      then begin E:=false; S:=0;
                 repeat if D<R
                        then if (Maxint-D) div R>=S
                             then begin S:=S*R+D;
                                        Get(F);
           AltalanositottSzamjegyekAtvaltasa(F^,V,D);
                                  end
                             else E:=true
                        else E:=true
                 until E or not V;
                 if not E
                 then X:=S
           end
end {MasAlapuSzamrendszerbenAbrazoltSzam};
```

G. függelék

Az ASCII karakterkészlet

Az ASCII (*A*merican *S*tandard *C*ode for *I*nformation *I*nterchange; az információcsere amerikai szabványkódja) az ISO (*I*nternational *O*rganization for *S*tandardization: nemzetközi szabványügyi szervezet) készletnek nevezett, hivatalosan elfogadott nemzetközi szabvány-karakterkészlet amerikai változata. 128 karakter kódját tartalmazza. Az ISO karakterkód 12 szimbólumban engedélyez nemzeti változatokat (ilyen például a $ jel). A 128 közül 95 valamilyen alakzatot nyomtat ki, 33 pedig a különböző egységek vezérlésére szolgál. A backspace (visszalépés) vezérlőkarakter segítségével egyes kinyomtatott karaktereket lehet felülnyomtatni; így állíthatók elő például az egyes nyelvek számára szükséges ékezetes betűk.

A 33 vezérlőkarakter:

ACK	Acknowledge	visszajelzés.
BEL	Bell	csengő.
BS	Backspace	visszalépés.
CAN	Cancel	törlés.
CR	Carriage Return	kocsi vissza.
DC1	Device Control 1	1. egység vezérlése.
DC2	Device Control 2	2. egység vezérlése.
DC3	Device Control 3	3. egység vezérlése.
DC4	Device Control 4	4. egység vezérlése.
DEL	Delete	törlés.
DLE	Data Link Escape	kilépés az adatsorból.
EM	End of Medium	közeg vége.
ENQ	Enquiry	keresés.
EOT	End of Transmission	átvitel vége.
ESC	Escape	kilépés.
ETB	End of Transmission Block	átviteli blokk vége.
ETX	End of Text	szöveg vége.
FF	Form Feed	lapemelés.
FS	File Separator	állományelválasztó.
GS	Group Separator	csoportelválasztó.
HT	Horizontal Tab	vízszintes tabulálás.
LF	Line Feed	soremelés.
NAK	Negative Acknowledge	negatív visszajelzés.
NUL	Null	nulla.
RS	Record Separator	rekordelválasztó.
SI	Shift In	váltó be.
SO	Shift Out	váltó vissza.
SOH	Start of Heading	fejléc kezdete.
STX	Start of Text	szöveg kezdete.

SUB	Substitute	helyettesítés.
SYN	Synchronous Idle	szinkron üresjárat.
US	Unit Separator	egységelválasztó.
VT	Vertical Tab	függőleges tabulálás.

A teljes, 128 karakterből álló készlet:

	00	16	32	48	64	80	96	112
0	NUL	DLE		0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
10	LF	SUB	*	:	J	Z	j	z
11	VT	ESC	+	;	K	[	k	{
12	FF	FS	,	<	L	\	l	|
13	CR	GS	-	=	M	]	m	}
14	SO	RS	.	>	N	^	n	~
15	SI	US	/	?	O	_	o	DEL

Egy karakter 7 biten ábrázolt kódja a sor- és oszlop elején álló szám összege. A G betű kódja tehát például 7 + 64 = 71.

Tárgymutató

F

G

H

I

K

L

M

N

O, Ö

P

R

S

Sz

T

U, Ü

V

W

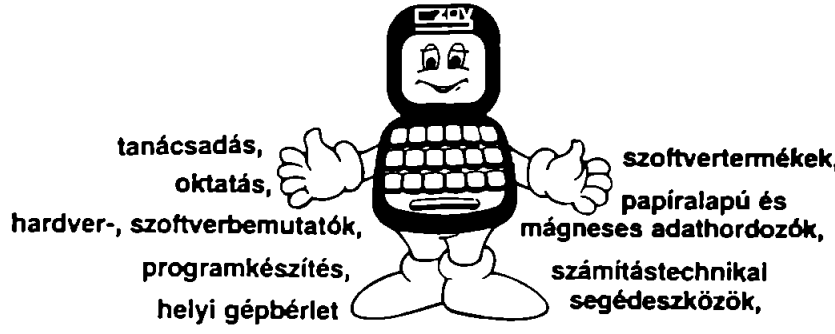
Kiadja a Műszaki Könyvkiadó
Felelős kiadó: Szűcs Péter igazgató
Felelős szerkesztő: Czere Károlyné
Szerkesztő: Donát János
A szedés a Műszaki Könyvkiadóban készült

Formakészítés és nyomtatás:
Révai Nyomda, Eger
Felelős vezető: Horváth Józsefné dr.

Műszaki vezető: Kőrizs Károly
Műszaki szerkesztő: Marcsek Ildikó
A fedelet tervezte: Kováts Tibor
A könyv ábráit rajzolta: Czeglényi Lászlóné
A könyv formátuma: B5
Ívterjedelme: 20,735 (A5)
Ábrák száma: 101
Azonossági szám: 61 330
MŰ: 4147-k-8891
Készült az MSZ 5601 és 5602 szerint
A kézirat lezárva: 1987. szeptember

A KSH Számítástechnikai és Ügyvitelszervező Vállalat folyamatosan bővíti szolgáltatásainak körét a felhasználói igények változásával és a szakmai újdonságok piaci megjelenésével párhuzamosan.

E megújulási folyamat a nagyközönség által is érzékelt látványos eleme a COMPUTER–M ügyfélszolgálati irodahálózat kialakítása.



Az első **COMPUTER–M** 1985-ben nyitotta meg üvegajtóit Kecskeméten. Mire e sorok megjelennek, már minden megyében hasonló, többfunkciójú létesítmények segítik az egyre bővülő számítástechnikai igények kielégítését.

A **COMPUTER–M** persze minden városban, elsősorban komputermarket, egyfajta bolt, kiskereskedelmi egység tehát, ahol – egyebek között – hardverek és szoftverek adásvétele folyik. A kereskedelmi funkció nem titkolt célja, hogy a **SZÜV** alaptevékenységét támogassa. Olyan számítástechnikai eszközöket árusítanak, amelyekhez a **SZÜV** egyéb szolgáltatásokat – például szoftverfejlesztést, szervizt – is képes nyújtani. Csak IBM és azzal kompatibilis gépeket forgalmaznak a bemutatott minta alapján, a már bevizsgált alap- és felhasználói szoftverekkel együtt. A saját szoftvereken kívül mások által készített szoftvereket is kínálnak eladásra. Ezek használhatóságát, kezelhetőségét a **SZÜV** megvizsgálja, a minősítés után veszi át terjesztésre.

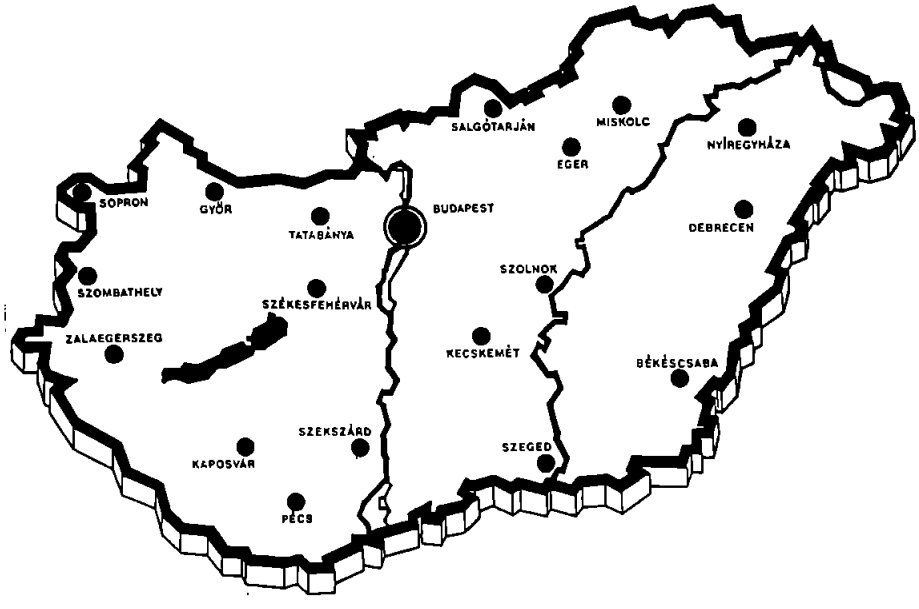
Az üzletekben számítástechnikai segédeszközök, szakkönyvek és számítástechnikai újságok is kaphatók. Korlátozott mennyiségben mikronyomtatóhoz használható leporellót is árusítanak, kifejezetten egyéni felhasználók részére.

Nemcsak a vásárlói igények kielégítése az ügyfélszolgálati irodák feladata, hanem a mikrogépek használatához is hozzásegítik az érdeklődőket. Minden egységnél többféle személyi számítógép használatával ismerkedhetnek a kezdők, ebben őket a **SZÜV** szakemberei segítik.

Szaktanácsért is érdemes a **COMPUTER–M**-ekbe menni, mert az ott dolgozó szakemberek ingyenes szolgáltatásként adnak tanácsot az egyéni és közületi felhasználóknak.

A **COMPUTER–M** ügyfélszolgálati irodák tevékenységét oktatási szolgáltatás teszi teljessé, amely egyéni és közületi igények kielégítésére egyaránt kiterjed.

A kereskedelmi és szolgáltató funkció szervesen kapcsolódik egymáshoz, s a kettő együtt avat minden **COMPUTER–M**-et a mikroszámítógépes komplex vállalkozások egy-egy nélkülözhetetlen láncszemévé.



• BUDAPEST
Ügyfélszolgálati Iroda
1067 Lenin körút 57–59.
Telefon: 224-838 · Telex: 0227610

• KECSKEMÉT
Ügyfélszolgálati Iroda
6000 Horváth Döme utca 12.
Telefon: 76/29-162 · Telex: 026494

• NYÍREGYHÁZA
Ügyfélszolgálati Iroda
4400 Rákóczi út 3.
Telefon: 42/14-481 · Telex: 073337

• KAPOSVÁR
Ügyfélszolgálati Iroda
7400 Rákóczi tér 9–11.
Telefon: 82/13-026 · Telex: 013226

• SZEKSZÁRD
Ügyfélszolgálati Iroda
7100 Wesselényi u. 15–17.
Telefon: 74/16-822 · Telex: 014363

• SZOMBATHELY
Ügyfélszolgálati Iroda
9700 Hunyadi u. 64.
Telefon: 94/14-534 · Telex: 037260

• PÉCS
Ügyfélszolgálati Iroda
7624 Sallai u. 32.
Telefon: 72/19-434 · Telex: 012322

• BÉKÉSCSABA
Ügyfélszolgálati Iroda
5600 Kinizsi u. 4–6.
Telefon: 66/21-155 · Telex: 083418

• SALGÓTARJÁN
Ügyfélszolgálati Iroda
3100 Rákóczi u. 202.
Telefon: 32/11-477 · Telex: 0229223

• SZOLNOK
Ügyfélszolgálati Iroda
5002 József Attila u. 22–24.
Telefon: 56/17-200 · Telex: 023202

• DEBRECEN
Ügyfélszolgálati Iroda
4032 Vöröshadsereg útja 22.
Telefon: 52/17-497 · Telex: 072387

• SZÉKESFEHÉRVÁR
Ügyfélszolgálati Iroda
8000 Schönherz Zoltán u. 36–40.
Telefon: 22/16-330 · Telex: 021230

• ZALAEGERSZEG
Ügyfélszolgálati Iroda
8900 Mártírok útja 42-44.
Telefon: 92/14-390 · Telex: 033304

• TATABÁNYA
Ügyfélszolgálati Iroda
2800 Mártírok útja 81/A
Telefon: 34/10-499 · Telex: 027271

• EGER
Ügyfélszolgálati Iroda
3300 Grónai u. 3.
Telefon: 36/10-522 · Telex: 063405

• SOPRON
Ügyfélszolgálati Iroda
9400 Új u. 30.
Telefon: 99/12-654 · Telex: 0249202

• SZEGED
Ügyfélszolgálati Iroda
6726 Jobb fasor 6–10.
Telefon: 62/11-311 · Telex: 082311

• MISKOLC
Ügyfélszolgálati Iroda
3515 Egyetemváros
Telefon: 46/61-622 · Telex: 062352

138,– Ft