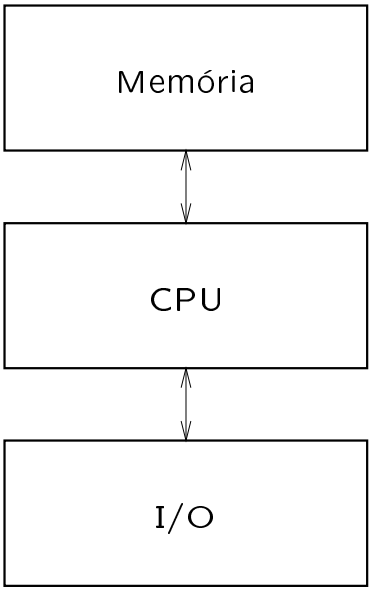
Programozás

Tematika

- Számítógép felépítése
- Gépi kód, assembly
- Magas szintű programozási nyelvek
- Szintaxis(diagram), szemantika
- PASCAL nyelv elemei
- Programszerkezet
- Egyszerű utasítások
- Változók és elemi típusok
- Szelekciós utasítások
- Ciklusutasítások
- Programtervezés
- Tömbváltozók, intervallumtípus
- Eljárások, blokkstruktúra
- Azonosítók hatásköre és élettartama

A tárolt programvezérlésű számítógép

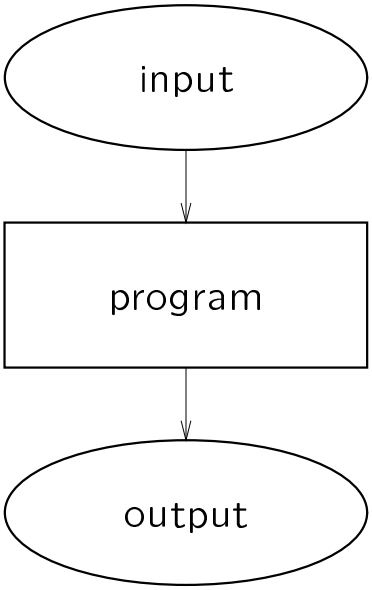


Program és *adatok* a memóriában

A CPU vezérli a gép működését

Az I/O eszközök tartják a kapcsolatot a külvilággal

Adatfeldolgozás



off-line feldolgozás: az adatok előre rendelkezésre állnak

interaktiv program: futás közben is olvas be adatokat

Perifériák

- *input*: billentyűzet, egér, joystick, ...
- *output*: monitor, nyomtató, ...
- *input* és *output*: mágneses és optikai háttértárolók

Memória

0	53
1	−165
2	0
⋮	
N	237

*rekesz*ekből áll, ezeknek van

címe : egy szám

tartalma : egy szám, ez lehet egy másik rekesz címe is

Két művelet végezhető a rekeszekkel

kiolvasás : másolatot kapunk a tartalmáról

beírás : az új érték felülírja a régit

CPU

Regiszterek (névvel címzett rekeszek)

- Akkumulátor (`AC`): részeredmények tárolására
- Utasításmutató (`IP`): soron következő utasítás címét tartalmazza
- Verem mutató (`SP`): ...
- Állapot regiszter (`FLAG`)
  - `ZERO` bit: Az `AC` értéke 0
  - `NEG` bit: Az `AC` értéke negatív
  - `REL` bit: Az utolsó relációs utasítás eredménye

Jelölés

- $[Z]$: a Z rekesz tartalmát kiolvassuk
- $X \mapsto Y$: az X értéket beírjuk az Y rekeszbe

példák: $5 \mapsto 175$, $[5] \mapsto 175$, $5 \mapsto$ `AC`,
$[5] \mapsto$ `AC`, `[AC]` $\mapsto 175$, $5 \mapsto [175]$, `[AC]` $\mapsto [175]$

Utasítások

Elemi tevékenységek, a programok építőkövei

- Utasítás kód: mit kell csinálni
- Utasítás argumentum (opcionális): mivel kell csinálni

⋮		
K	5	kód
$K+1$	-165	argumentum
$K+2$	25	kód
$K+3$	4	kód
$K+4$	237	argumentum
⋮		

A gép *állapot*a a memória és a regiszterek értékeinek összessége. Az utasításokat végrehajtva megváltozik a számítógép állapota.

Példák

Utasítás:	Összeadás, (0)
Mnemonik:	`ADD` a
Hatása:	$[\texttt{AC}] + [a] \mapsto \texttt{AC}$
Jelzőbitek:	`ZERO`, `NEG`
Hiba:	Túlcsordulás

Utasítás:	Akkumulátor feltöltése, (4)
Mnemonik:	`LDA` a
Hatása:	$[a] \mapsto \texttt{AC}$
Jelzőbitek:	`ZERO`, `NEG`
Hiba:	—

Utasítás:	Akkumulátor tárolása, (7)
Mnemonik:	`STA` a
Hatása:	$[\texttt{AC}] \mapsto a$
Jelzőbitek:	—
Hiba:	—

Utasítás:	Program leállítása, (23)
Mnemonik:	`STOP`
Hatása:	—
Jelzőbitek:	—
Hiba:	—

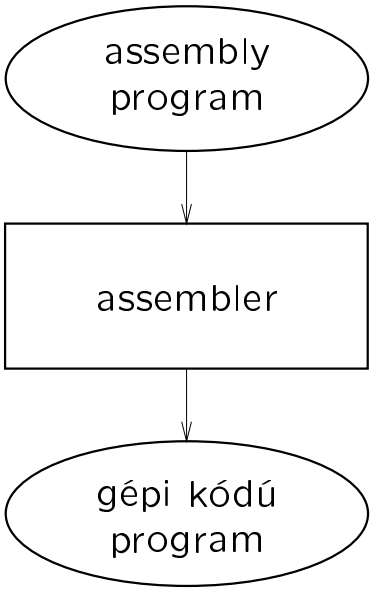
Számítógép működése

- Az adatok és a program betöltése a memóriába
- `IP` beállítása a program első utasítására
- Ismételni amíg a program véget nem ér
  1. utasításkód beolvasása, $[\texttt{IP}] + 1 \mapsto \texttt{IP}$
  2. utasítás értelmezése
  3. paraméter beolvasása, $[\texttt{IP}] + 1 \mapsto \texttt{IP}$
  4. utasítás végrehajtása

Egy gépi kódú program

0	4	`LDA`
1	7	
2	0	`ADD`
3	8	
4	7	`STA`
5	9	
6	23	`STOP`
7	−3	
8	5	
9	?	

Az assembly nyelv



Assembly program

```
CODE 0
        LDA   7    ;első szám AC-ba
        ADD   8    ;másodikat hozzáadjuk
        STA   9    ;tároljuk az eredményt
        STOP       ;program vége
ENDCODE
DATA 7
              -3
              5
        DW    1    ;1 rekeszt lefoglalunk
ENDDATA
```

Assembly program adatcimkékkel

```
CODE
        LDA   x1      ;első szám AC-ba
        ADD   x2      ;másodikat hozzáadjuk
        STA   eredm   ;tároljuk az eredményt
        STOP          ;program vége
ENDCODE
DATA
x1:             -3
x2:             5
eredm:  DW      1     ;1 rekeszt lefoglalunk
ENDDATA
```

- A cimkék betűvel kezdődő, betűkből és számokból álló szavak.
- Minden cimke egyszer fordul elő a baloldalon, de többször is szerepelhet a jobb oldalon paraméterként.
- Az assembler kiszámítja a cimkék értékét és ezeket teszi be a gépi kódú programba.

További aritmetikai utasítások

Utasítás:	Kivonás, (1)
Mnemonik:	`SUB` a
Hatása:	[`AC`] $-$ [a] $\mapsto$ `AC`
Jelzőbitek:	`ZERO`, `NEG`
Hiba:	Túlcsordulás

Utasítás:	Szorzás, (2)
Mnemonik:	`MUL` a
Hatása:	[`AC`] $*$ [a] $\mapsto$ `AC`
Jelzőbitek:	`ZERO`, `NEG`
Hiba:	Túlcsordulás

Utasítás:	Osztás, (3)
Mnemonik:	`DIV` a
Hatása:	[`AC`] $\div$ [a] $\mapsto$ `AC`
Jelzőbitek:	`ZERO`, `NEG`
Hiba:	0-val osztás

További adatmozgató utasítás

Utasítás:	Konstans betöltése, (5)
Mnemonik:	`LDC` c
Hatása:	$c \mapsto$ `AC`
Jelzőbitek:	`ZERO`, `NEG`
Hiba:	—

I/O utasítások

Utasítás:	Szám beolvasása, (24)
Mnemonik:	`INP`
Hatása:	<beolvasott szám> $\mapsto$ `AC`
Jelzőbitek:	`ZERO`, `NEG`
Hiba:	Túlcsordulás

Utasítás:	Szám kiírása, (25)
Mnemonik:	`OUT`
Hatása:	[`AC`] $\mapsto$ <monitor>
Jelzőbitek:	—
Hiba:	—

Összeadás eredményének megjelenítése

```
CODE
        LDA   x1  ;első szám AC-ba
        ADD   x2  ;másodikat hozzáadjuk
        OUT       ;kiírjuk az eredményt
        STOP      ;program vége
ENDCODE
DATA
x1:             -3
x2:             5
ENDDATA
```

Beolvasott számok összeadása

```
CODE
        INP       ;első szám AC-ba
        STA   x1  ;tároljuk
        INP       ;második szám AC-ba
        ADD   x1  ;elsőt hozzáadjuk
        OUT       ;kiírjuk az eredményt
        STOP      ;program vége
ENDCODE
DATA
x1:     DW    1
ENDDATA
```

Vezérlésátadó utasítások

- Alapértelmezésben az utasítások *szekvenciális*an kerülnek végrehajtásra
- A vezérlésátadó utasításokkal befolyásolhatjuk az utasítások végrehajtási sorrendjét

Utasítás:	Feltétel nélküli ugrás, (14)
Mnemonik:	`JP` a
Hatása:	$a \mapsto$ `IP`
Jelzőbitek:	—
Hiba:	Érvénytelen cím

Utasítás:	Feltételes ugrás, (15+)
Mnemonik:	`J<feltétel>` a
Hatása:	$a \mapsto$ `IP`, ha <feltétel> teljesül
Jelzőbitek:	—
Hiba:	Érvénytelen cím

Feltételek a jelzőbitek értékét vizsgálják:

- R, NR: `REL` 1 vagy 0
- Z, NZ: `ZERO` 1 vagy 0
- N, NN: `NEG` 1 vagy 0

Két szám közül a nagyobbik kiválasztása

```
CODE
        INP             ;első szám beolvasása
        STA   x1        ;tároljuk
        INP             ;második szám beolvasása
        STA   x2        ;tároljuk
        SUB   x1        ;kivonjuk belőle az elsőt
        JN    nagy1     ;első nagyobb
        LDA   x2        ;itt a második nagyobb
        OUT             ;kiírjuk
        STOP            ;program vége
nagy1:  LDA   x1        ;itt az első a nagyobb
        OUT             ;kiírjuk
        STOP            ;program vége
ENDCODE
DATA
x1:             DW      1
x2:             DW      1
ENDDATA
```

Fontos:

- `JN` utasítás
- `nagy1` utasítás cimke
- `STOP` a program közepén

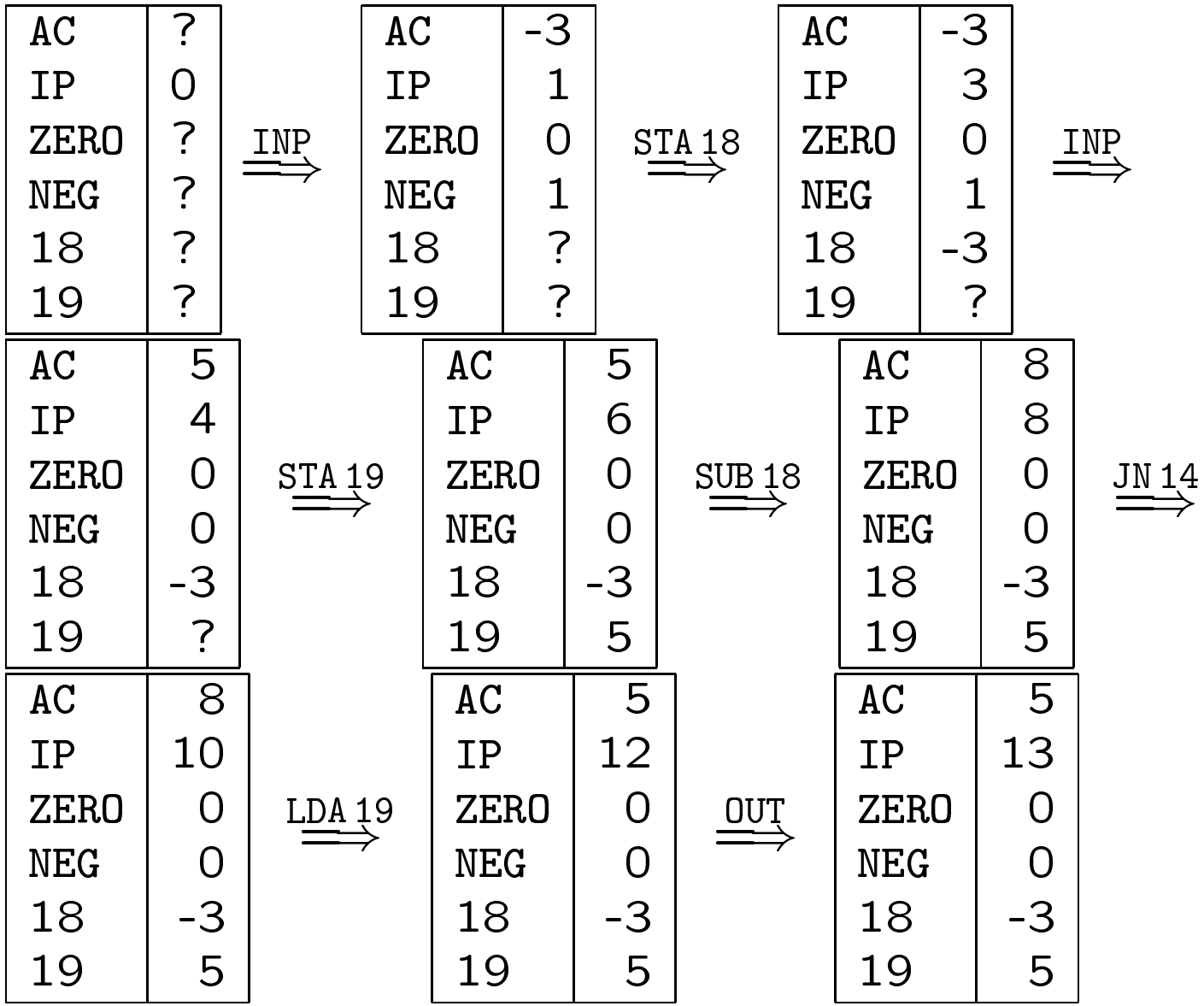
Program elhelyezkedése a memóriában

```
 0:     INP        ;első beolvasása
 1:     STA   18   ;x1, tároljuk
 3:     INP        ;második beolvasása
 4:     STA   19   ;x2, tároljuk
 6:     SUB   18   ;x1, kivonjuk az elsőt
 8:     JN    14   ;nagy1, első nagyobb
10:     LDA   19   ;x2, második nagyobb
12:     OUT        ;kiírjuk
13:     STOP       ;program vége
14:     LDA   18   ;x1, első nagyobb
16:     OUT        ;kiírjuk
17:     STOP       ;program vége
18:     DW    1    ;x1
19:     DW    1    ;x2
```

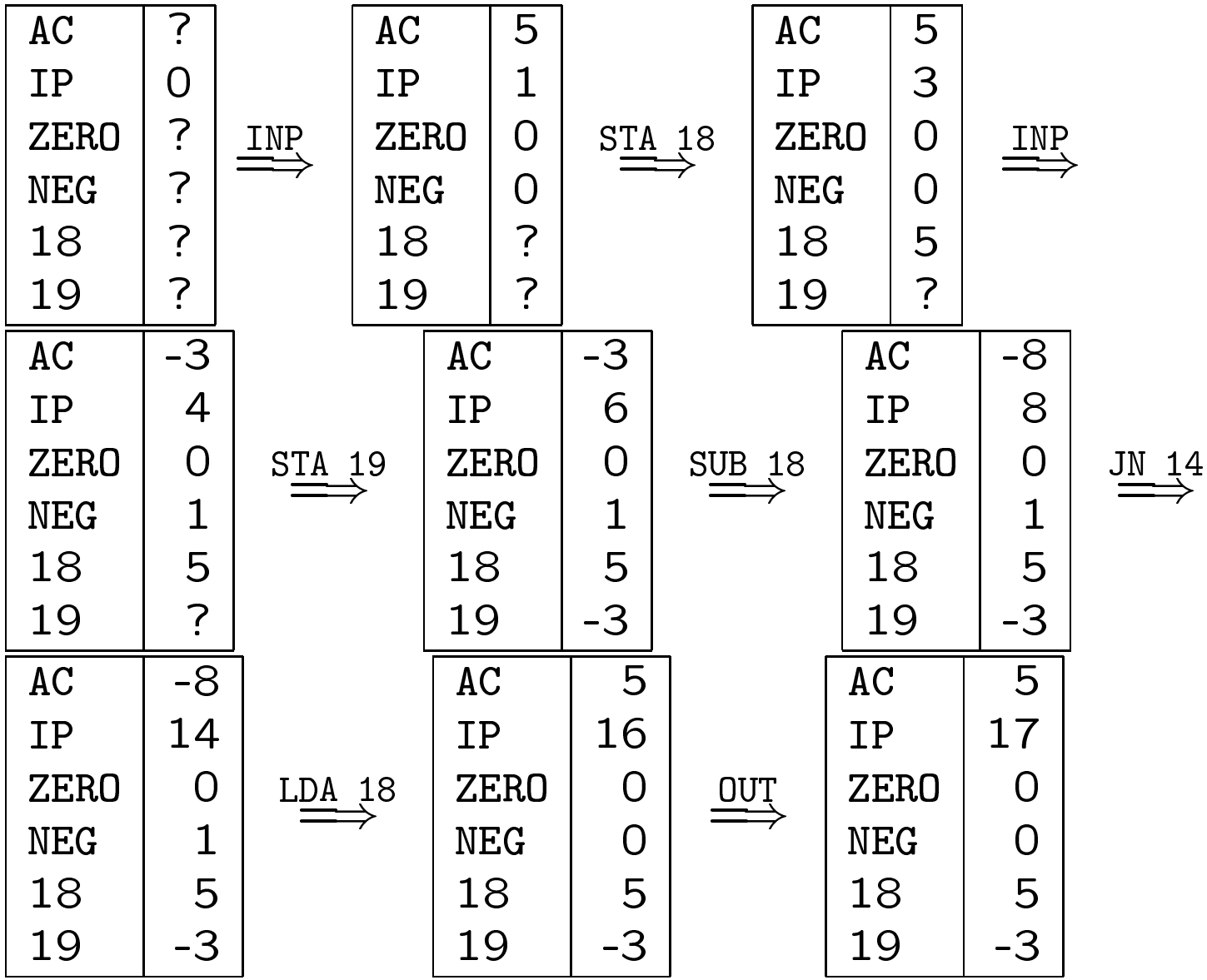
Gép kiindulási állapota

AC	?
IP	0
ZERO	?
NEG	?
18	?
19	?

Program futása -3, 5 inputtal



Program futása 5, -3 inputtal



REL flag-et állító utasítások

Utasítás:	"kisebb" vizsgálat, (9)
Mnemonik:	LT a
Hatása:	—
Jelzőbitek:	REL= 1 ha [AC] $<$ [a], 0 különben
Hiba:	—

Utasítás:	"kisebb vagy egyenlő", (10)
Mnemonik:	LTE a
Hatása:	—
Jelzőbitek:	REL= 1 ha [AC] $\leq$ [a], 0 különben
Hiba:	—

Utasítás:	"nagyobb" vizsgálat, (11)
Mnemonik:	GT a
Hatása:	—
Jelzőbitek:	REL= 1 ha [AC] $>$ [a], 0 különben
Hiba:	—

Utasítás:	"nagyobb vagy egyenlő", (12)
Mnemonik:	GTE a
Hatása:	—
Jelzőbitek:	REL= 1 ha [AC] $\geq$ [a], 0 különben
Hiba:	—

Két szám közül a nagyobbik, 2. verzió

```
CODE
        INP             ;első szám beolvasása
        STA   x1        ;tároljuk
        INP             ;második szám beolvasása
        LT    x1        ;összehasonlítás
        JR    nagy1     ;első nagyobb
        OUT             ;kiírjuk a másodikat
        STOP            ;program vége
nagy1:  LDA   x1        ;itt az első a nagyobb
        OUT             ;kiírjuk
        STOP            ;program vége
ENDCODE
DATA
x1:     DW    1
ENDDATA
```

Fontos:

- nincs szükség `x2`-re
- `LT` nem állítja el az `AC` regisztert
- `JR` utasítás

Még relációk

Utasítás:	“egyenlőség” vizsgálat, (13)
Mnemonik:	`EQ` a
Hatása:	—
Jelzőbitek:	`REL`= 1 ha [`AC`] = [a], 0 különben
Hiba:	—

Feltételes részek kifejezése

```
LT a          ≡   GTE a
JR cimke          JNR cimke

GT a          ≡   LTE a
JR cimke          JNR cimke
```

Abszolút érték kiszámítása, 1. verzió

```
CODE
        INP             ;szám beolvasása
        GTE   c0        ;nemnegatív?
        JR    nemneg    ;igen
        STA   x         ;tároljuk
        LDA   c0        ;AC-ba 0
        SUB   x         ;kivonjuk a számot
nemneg: OUT             ;kiírjuk az eredményt
        STOP            ;program vége
ENDCODE
DATA
c0:             0
x:      DW      1
ENDDATA
```

Abszolút érték kiszámítása, 2. verzió

```
CODE
        INP             ;szám beolvasása
        STA   x         ;tároljuk
        LDC   0         ;AC-ba 0
        LTE   x         ;nemnegatív?
        JR    nemneg    ;igen
        SUB   x         ;kivonjuk a számot
nemneg: OUT             ;kiírjuk az eredményt
        STOP            ;program vége
ENDCODE
DATA
x:      DW    1
ENDDATA
```

Írjunk ki 10 db nullát

```
CODE
        LDC   0
        OUT       ;1
        OUT       ;2
        OUT       ;3
        OUT       ;4
        OUT       ;5
        OUT       ;6
        OUT       ;7
        OUT       ;8
        OUT       ;9
        OUT       ;10
        STOP
ENDCODE
DATA
ENDDATA
```

Írjunk ki 10 nullát profi módon

```
CODE
        LDC   10     ;ismétlések száma
        STA   db     ;ciklusváltozó

ujra:   LDC   0
        OUT          ;0 ki

        LDC   -1
        ADD   db     ;ciklusváltozó csökkentése
        STA   db

        JNZ   ujra   ;végeztünk?
        STOP         ;program vége
ENDCODE
DATA
db:     DW    1      ;még kiírandó darabszám
ENDDATA
```

Adjunk össze 10 számot

```
CODE
        LDC   10      ;ismétlések száma
        STA   db      ;ciklusváltozó
        LDC   0       ;kezdeti részösszeg
        STA   summa

ujra:   INP           ;következő szám be
        ADD   summa   ;részösszeghez hozzáadni
        STA   summa   ;tároljuk

        LDC   -1
        ADD   db      ;ciklusváltozó csökkentése
        STA   db

        JNZ   ujra    ;végeztünk?

        LDA   summa
        OUT
        STOP          ;program vége
ENDCODE
DATA
db:     DW    1       ;még kiírandó darabszám
summa:  DW    1       ;részösszeg
ENDDATA
```

Adjunk össze 10 memóriabeli számot

```
CODE
        LDC   0        ;kezdeti részösszeg
        STA   summa

ujra:   LDA   x        ;következő ??  szám be
        ADD   summa    ;részösszeghez hozzáadni
        STA   summa    ;tároljuk

        LDC   -1
        ADD   db       ;ciklusváltozó csökkentése
        STA   db

        JNZ   ujra     ;végeztünk?

        LDA   summa
        OUT
        STOP           ;program vége
ENDCODE
DATA
db:             10     ;még kiírandó darabszám
summa:  DW      1      ;részösszeg
x:              -3, 5, 2, 7, -9, -1, 2, 10, 0, 3
ENDDATA
```

Kétszeresen indirekt címzés

Utasítás:	Betöltés, (6)
Mnemonik:	`LDI` a
Hatása:	$[[a]] \mapsto$ `AC`
Jelzőbitek:	`ZERO`, `NEG`
Hiba:	Nem létező memóriarekesz

Utasítás:	Tárolás, (8)
Mnemonik:	`STI` a
Hatása:	`[AC]` $\mapsto [a]$
Jelzőbitek:	—
Hiba:	Nem létező memóriarekesz

Példa

```
CODE
          ...
        LDC    x
        STA    xcim
        LDI    xcim  ;5-öt tölt AC-ba
          ...
DATA
x:              5
xcim:   DW      1
          ...
```

Tényleg adjunk össze

```
CODE
        LDC   0        ;kezdeti részösszeg
        STA   summa
        LDC   x        ;első szám címe
        STA   kov      ;cím tárolása
ujra:   LDI   kov      ;következő szám be
        ADD   summa    ;részösszeghez hozzáadni
        STA   summa    ;tároljuk
        LDC   1
        ADD   kov      ;elemcím növelése
        STA   kov
        LDC   -1
        ADD   db       ;ciklusváltozó csökkentése
        STA   db
        JNZ   ujra     ;végeztünk?
        LDA   summa
        OUT
        STOP           ;program vége
ENDCODE
DATA
db:             10     ;még kiírandó darabszám
kov:    DW      1      ;következő elem címe
summa:  DW      1      ;részösszeg
x:              -3, 5, 2, 7, -9, -1, 2, 10, 0, 3
ENDDATA
```

Legnagyobb szám kiválasztása sorozatból

```
CODE
        INP             ;első szám beolvasása
        STA   legn      ;eddig ez a legnagyobb
ujra:   INP             ;köv.  szám beolvasása
        JZ    vege      ;0-ra vége
        GT    legn      ;
        JNR   marad     ;marad az eddigi
        STA   legn      ;új legnagyobb
marad:  JP    ujra      ;
vege:   LDA   legn      ;
        OUT             ;végeredmény
        STOP            ;program vége
ENDCODE
DATA
legn:   DW    1
ENDDATA
```

Számsorozat összeadása

```
CODE
        LDC   0
        STA   summa
ujra:   INP           ;köv.  szám beolvasása
        JZ    vege    ;0-ra vége
        ADD   summa   ;részösszeg növelése
        STA   summa   ;
        JP    ujra    ;
vege:   LDA   summa   ;
        OUT           ;végeredmény
        STOP          ;program vége
ENDCODE
DATA
summa:  DW    1
ENDDATA
```

Alprogramok

- Többször ismétlődő programrészleteket kényelmetlen mindig leírni

```
        ...
        LDC    1
        ADD    szamlalo
        STA    szamlalo
        ...
```

- Elég egyszer leírni és többször felhasználni

```
        ...
novel:  LDC    1
        ADD    szamlalo
        STA    szamlalo
        RET                ;alprogram vége
        ...
        CALL   novel       ;alprogram hívása
        ...
        CALL   novel       ;alprogram hívása

        ...
```

Utasítások alprogramokhoz

Utasítás:	Alprogram hívása, (21)
Mnemonik:	`CALL` a
Hatása:	$[\texttt{IP}] \mapsto \texttt{SP}$, $[\texttt{SP}] + 1 \mapsto \texttt{SP}$, $a \mapsto \texttt{IP}$
Jelzőbitek:	—
Hiba:	Verem betelik

Utasítás:	Visszatérés alprogramból, (22)
Mnemonik:	`RET`
Hatása:	$[\texttt{SP}] - 1 \mapsto \texttt{SP}$, $[[\texttt{SP}]] \mapsto \texttt{IP}$
Jelzőbitek:	—
Hiba:	Verem üres

Összeadás alprogramokkal

```
CODE
        LDC   0
        STA   summa
        LDC   x
        STA   kov
ujra:   LDI   kov
        ADD   summa
        STA   summa
        CALL  kovnov
        CALL  dbcsok
        JNZ   ujra
        LDA   summa
        OUT
        STOP
kovnov: LDC   1
        ADD   kov
        STA   kov
        RET
dbcsok: LDC   -1
        ADD   db
        STA   db
        RET
ENDCODE
```

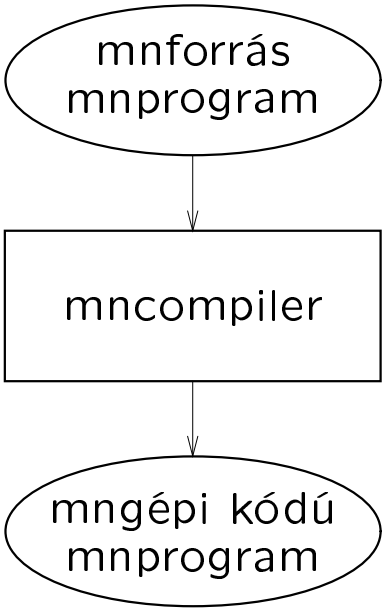
Összefoglaló

- Számítógép felépítése
  - Memória
  - CPU
  - Perifériák
- utasítások
  - Aritmetikai
  - Adatmozgató
  - Relációs
  - Vezérlésátadó
  - I/O
- Számítógép működése

Összefoglaló, folytatás

- Assembly nyelv
  - Programok felépítése
  - Mnemonikok
  - Címkék (adat- és utasítás)
- Programok
  - Egyszerű lineáris
  - Elágazásos
  - Korlátos ciklus
  - Tömbkezelés
  - Általános ciklus
  - Alprogramok

Magasszintű nyelvek



Mesterséges nyelvek

- Szintaxis: formai szabályok, mely jelsorozatok tartoznak a nyelvhez
- szemantika: jelentés, mit a jelentenek a nyelvhez tartozó jelsorozatok (szavak)

PASCAL nyelv elemei

- Változók (tárrekeszek)
- Deklarációk (`DATA ... ENDDATA`)
- Utasítások (`CODE ... ENDCODE`)
  - I/O
  - értékadás (aritmetikai, relációs, adatmozgató)
  - vezérlési szerkezetek (vezérlésátadó)

Első PASCAL program

```
PROGRAM osszeadas;
VAR
  x, y, z: INTEGER;
BEGIN
READLN( x);
READLN( y);
z := x+y;
WRITELN( z);
END.
```

Értékadó utasítás

- szintaxis: változó := kifejezés
- szemantika:
  - kifejezés kiértékelése, a változók értékét *kiolvassuk*
  - a kiszámított érték *beírása* a változóba

Mesterséges nyelvek szintaxisa

ábécé: véges halmaz, elemei a jelek vagy betűk

szó, mondat: az ábécé betűiből képzett véges jelsorozat

nyelv: szavak halmaza

példa

ábécé: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

szavak: $0, 7, 000, 123, 107, 96345325732, \ldots$

nyelv: páros számok

---

szintaxis: a nyelvet alkotó szóhalmaz megadása

szintaxisdiagram: szintaxis megadására szolgáló jelölésrendszer

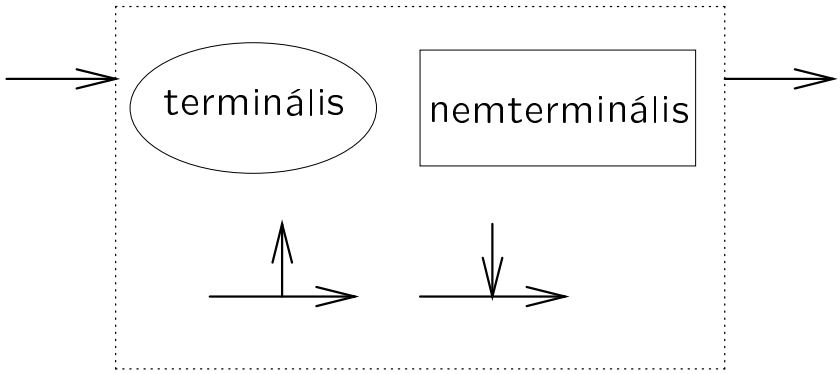
Szintaxisdiagram elemei

terminális: ábécé jeleiből képzett szó

nemterminális: egy (rész)nyelv neve

Szintaxisdiagram alakja

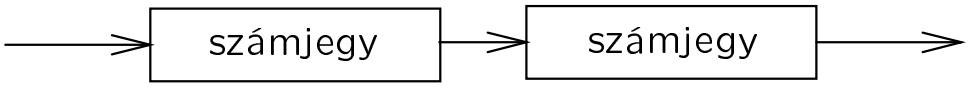
nemterminális



- Pontosan 1 benenő nyíl van
- Legalább 1 kimenő nyíl van
- Belül nyilakkal összekökött dobozok

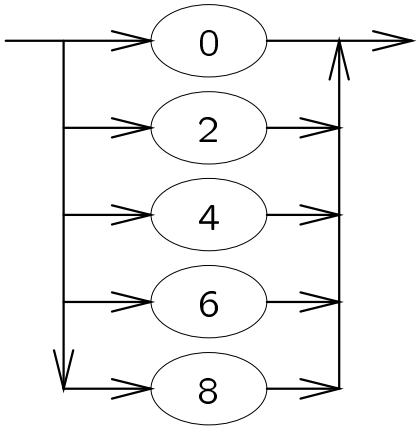
Szekvencia

kétjegyű szám

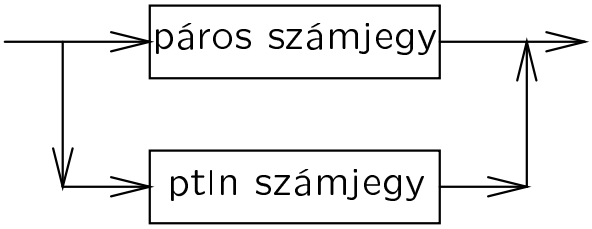


Alternatívák

páros számjegy

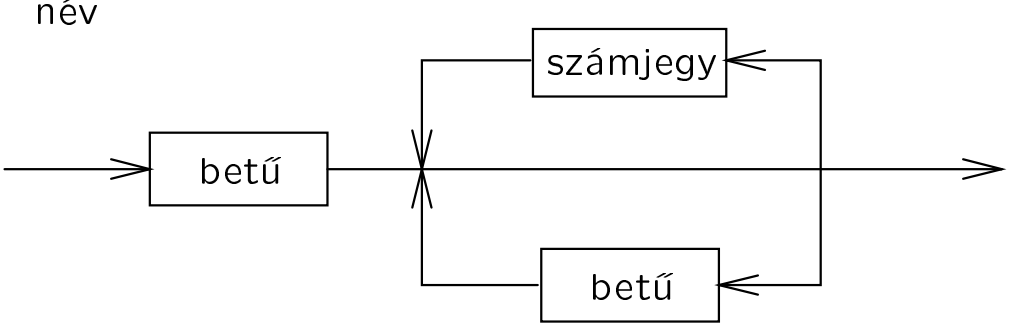


számjegy



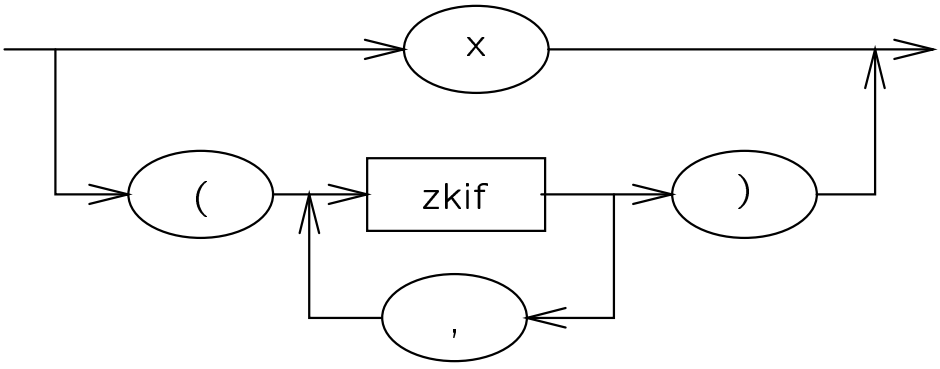
Hurkok

név



Önhivatkozás

zkif



Szintaxisdiagram értelmezése

- A bemenő nyilon be
- A nyilak irányát követve járjuk be a diagramot
- Az egyik kimenő nyilon távozunk
- Az érintett terminálisokat sorrendben lejegyezzük
- Az érintett nemterminálisok esetén az általuk definiált nyelv egy tetszőleges szavát jegyezzük le
- A szintaxisdiagram által definiált nyelv az összes fenti módon képezhető szavak halmaza

PASCAL szintaxis

- program fejléc
- deklarációs blokk
- program blokk

Deklarációs blokk

- Konstans deklaráció
- Típus deklaráció
- Változó deklaráció
- Alprogram deklaráció

Változó deklaráció

```
VAR
  x, y, z: INTEGER;

VAR
  x, y: INTEGER;
  z: INTEGER;
```

Típusok

- Elemi típus
  - Beépített
  - Felsorolás
  - Intervallum
- Összetett típus
  - Tömb
  - Halmaz
  - Rekord
  - File
- Pointer

Típus által meghatározott jellemzők

- Értékkészlet (konstansok)
- Műveletek

INTEGER típus

- értékkészlet: $-\texttt{MAXINT} \cdots + \texttt{MAXINT}$
- konstansok alakja: $12, -35, \ldots$
- műveletek: $+, -, *, \texttt{div}, \texttt{mod}$

REAL típus

- értékkészlet: a valós számok egy véges részhalmaza
- konstansok alakja:
  $3\text{e}5 \equiv 3 \cdot 10^5, -1.7\text{e}11 \equiv -1.7 \cdot 10^{11}$
- műveletek: $+, -, *, /$

Utasítások

- Egyszerű
  - Értékadás
  - Input
  - Output
  - Alprogram hívása
- Összetett
  - Szelekciók
  - Ciklusok

Kifejezések

- Műveleti jelekkel összekapcsolt konstansok és változók
- A benne szereplő változók értékét *kiolvassuk*
- A kifejezés kiértékelés után egy értéket szolgáltat
- A kifejezés típusa az eredmény típusával egyezik meg

Értékadó utasítás

1. A változó és a kifejezés típusának meg kell egyeznie
2. Kiértékeli a kifejezést
3. A kifejezés értékét *beírja* a baloldalon megadott változóba

```
z := x+y
z := z+1
```

Input utasítás

A konzolon begépelt szám a megadott változóba kerül

Output utasítás

A kifejezés(ek) értékét kiírja a konzolra

```
PROGRAM haromszog_terulet;
VAR
  alap, magassag: REAL;
  terulet: REAL;
BEGIN
READLN( alap);
READLN( magassag);
terulet := ( alap * magassag) / 2;
WRITELN( terulet);
END.
```

BOOLEAN típus

- értékkészlet: `FALSE`, `TRUE`
- műveletek: `NOT`, `AND`, `OR`

NOT FALSE	TRUE
NOT TRUE	FALSE

AND	FALSE	TRUE
FALSE	FALSE	FALSE
TRUE	FALSE	TRUE

OR	FALSE	TRUE
FALSE	FALSE	TRUE
TRUE	TRUE	TRUE

Relációs műveletek

- Számok összehasonlításának eredménye `FALSE` vagy `TRUE`
- Relációk: =, <>, <, <=, >, >=
- `BOOLEAN` típusú kifejezést *feltétel*nek nevezzük

Példák feltételekre

- `5 < 3`
- `a >= 7`
- `x = y`
- `0 < x <= 10` {!!! nem jó !!!}
- `( 0 < x) AND ( x <= 10)`
- `( x < 10) OR ( 20 < x)`

Operátorok precedenciája

- $-5 * 3 + 7 \equiv$
  $((-5) * 3) + 7,$ nem $-(5 * (3 + 7))$
- NOT f_1 AND f_2 OR $f_3 \equiv$
  (NOT f_1) AND f_2) OR f_3

IF utasítás

1. Feltétel kiértékelése (TRUE:feltétel igaz, vagy teljesül)
2. Ha a feltétel igaz, akkor a THEN ágban található utasítást végrehajtja és az IF utasítás befejeződött.
3. Ha a feltétel hamis és van ELSE ág, akkor az ELSE ágban található utasítást végrehajtja és az IF utasítás befejeződött.
4. Ha a feltétel hamis és nincs ELSE ág, akkor az IF utasítás befejeződött.

```
PROGRAM abszolut;
VAR
  x: INTEGER;
BEGIN
READLN( x);      {szám beolvasása}
IF x < 0         {negatív}
THEN
     x := -1*x; {pozitívvá alakítjuk}
WRITELN( x);     {eredmény kiírása}
END.
```

```
PROGRAM nagyobb2;
VAR
  x, y: INTEGER;
BEGIN
READLN( x};
READLN( y);
IF x > y
THEN
     WRITELN( x)  {!!! nincs ; }
ELSE
     WRITELN( y);
END.
```

```
PROGRAM sorrend2;
VAR
  x, y, t: INTEGER;
BEGIN
IF x < y
THEN BEGIN    {összetett utasítás}
     t := x;
     x := y;
     y := t;
     END;
WRITELN( x);
WRITELN( y);
END.
```

```
PROGRAM mekkora;
VAR
  x: INTEGER;
BEGIN
READLN( x);
IF x <= 10
THEN
     IF x < 10
     THEN
          WRITELN( 'kisebb 10-nél')  {!;}
     ELSE
          WRITELN( 'egyenlő tízzel') {!;}
ELSE
     WRITELN( 'nagyobb');            {!;}
END.
```

```
PROGRAM furcsa;
VAR
  x: INTEGER;
BEGIN
READLN( x);
IF x <= 10
THEN
     IF x < 10
     THEN
          WRITELN( 'kisebb')
ELSE
     WRITELN( 'nagyobb')
END.
```

```
PROGRAM furcsajo;
VAR
  x: INTEGER;
BEGIN
READLN( x);
IF x <= 10
THEN BEGIN
     IF x < 10
     THEN
          WRITELN( 'kisebb')
     END
ELSE
     WRITELN( 'nagyobb')
END.
```

```
PROGRAM sorrend3;
VAR
  x, y, z: INTEGER;
BEGIN
READLN( x);
READLN( y);
READLN( z);
IF x < y
THEN
     IF z < x
     THEN WRITELN( z, '<=', x, '<=', y)
     ELSE
          IF z < y
          THEN WRITELN( x, '<=', z, '<=', y)
          ELSE WRITELN( x, '<=', y, '<=', z)
ELSE
     IF z < y
     THEN WRITELN( z, '<=', y, '<=', x)
     ELSE
          IF z < x
          THEN WRITELN( y, '<=', z, '<=', x)
          ELSE WRITELN( y, '<=', x, '<=', z)
END.
```

```
PROGRAM xsorrend3;
VAR
  x, y, z: INTEGER;
  a, b   : INTEGER;
BEGIN
READLN( x);
READLN( y);
READLN( z);
IF x < y
THEN BEGIN
     a := x;
     b := y;
     END
ELSE BEGIN
     a := y;
     b := x;
     END;
IF z < a
THEN
     WRITELN( z, '<=', a, '<=', b);
IF ( z >= a) AND ( z < b)
THEN
     WRITELN( a, '<=', z, '<=', b);
IF ( z >= b)
THEN
     WRITELN( a, '<=', b, '<=', z);
END.
```

Másodfokú egyenlet megoldása

```
PROGRAM masodfoku;
VAR
  a, b, c: REAL; {együtthatók}
  d:       REAL; {diszkrimináns}
  x1, x2:  REAL; {gyökök}
BEGIN
WRITELN( 'Másodfokú egyenlet megoldása.');
WRITELN( 'Kérem az együtthatókat:');
READLN( a);
READLN( b);
READLN( c);
IF a = 0.0
THEN
     WRITELN( 'Az egyenlet nem másodfokú.')
ELSE BEGIN
     d := b*b - 4*a*c;
     IF d < 0.0
     THEN
          WRITELN( 'Az egyenletnek nincs valós gyöke.')
     ELSE BEGIN
          WRITELN( 'Az egyenlet gyöke(i):');
          x1 := ( -b + sqrt( d))/( 2*a);
          WRITELN( 'x1=', x1);
          IF d > 0.0
          THEN BEGIN
               x2 := ( -b - sqrt( d))/( 2*a);
               WRITELN( 'x2=', x2);
               END
          END
     END
END.
```

FOR TO utasítás

- A kifejezések típusának meg kell egyezni a ciklusváltozó típusával.
- A ciklusváltozó nem lehet `REAL` típusú.

1. A kifejezések kiértékelődnek és a ciklusváltozó felveszi a kezdőértéket.
2. Ha a ciklusváltozó értéke nagyobb mint a végérték, akkor a `FOR` utasítás befejeződött.
3. A ciklusmag végrehajtódik.
4. Ha a ciklusváltozó egyenlő a végértékkel, akkor a `FOR` utasítás befejeződött.
5. A ciklusváltozó felveszi a következő értékét, majd vissza a 3. pontra.

Írjunk ki számokat

```
program szamok1;
var
  i: integer;
begin
writeln( '10 darab szám következik.');
for i:=1 to 10
do
   write( i:3);
writeln;
end.
```

```
program szamok2;
var
  i: integer;
begin
writeln( '10 darab szám következik.');
for i:=-1 downto -10
do
   write( i:4);
writeln;
end.
```

Összetett utasítás a ciklustörzsben

```
program summarum;
var
  summa, x: integer;
  i       : integer;
begin
writeln( '10 szám összegének kiírása.');
summa := 0;
for i:=1 to 10
do begin
   write( i, '.szám:');
   readln( x);
   summa := summa+x;
   end;
writeln( 'Az összeg:', summa);
end.
```

Egymásbaágyazott FOR utasítások

```
program szorzotabla;
var
  i, j: integer;
begin
writeln( 'Szorzótábla');
for i:=1 to 10
do begin
   for j:=1 to 10
   do
      write( i*j:4);
   writeln;
   end
end.
```

10-ből a legnagyobb

```
program nagy10;
var
  x, max: integer;
  i     : integer;
begin
writeln( 'maximum kiválasztás.');
for i:=1 to 10
do begin
   write( i, '.szám:');
   readln( x);
   if ( i = 1) or ( x > max)
   then
        max := x
   end;
writeln( 'A legnagyobb:', max);
end.
```

CHAR típus

- értékkészlet: Az implementáció által meghatározott karakterek, tipikusan a kibővített ASCII jelkészlet.
  - 0–31: vezérlőkarakterek
  - 48–57: számjegyek
  - 65–91: nagybetűk
  - 128–255: nem szabványos karakterek
- konstansok alakja: `'A'`, `'5'`, `'?'`, ...
- műveletek:
  - `ord` : CHAR $\rightarrow$ INTEGER
    pl. `ord( 'A')` $\rightarrow$ `65`
  - `chr` : INTEGER $\rightarrow$ CHAR
    pl. `chr( 65)` $\rightarrow$ `'A'`
  - `pred` : CHAR $\rightarrow$ CHAR
    `pred( c)` $\equiv$ `chr( ord( c)-1)`
  - `succ` : CHAR $\rightarrow$ CHAR
    `succ( c)` $\equiv$ `chr( ord( c)+1)`

Karakterintervallum kiírása

```
program karint;
var
  elso, db, i: integer;
begin
writeln( 'karakterintervallum.');
write( 'első karakter kódja:');
readln( elso);
write( 'karakterek száma:');
readln( db);
for i:=elso to elso+db-1
do
   if ( i >= 32) and ( i <= 255)
   then
        write( chr( i));
writeln;
end.
```

```
program karint2;
var
  elso, utolso, c: char;
begin
writeln( 'karakterintervallum.');
write( 'első karakter:');
readln( elso);
write( 'utolsó karakter:');
readln( utolso);
for c:=elso to utolso
do
   write( c);
writeln;
end.
```

Karakterszámlálás

```
program aszamlalo;
var
  c: char;
  aszam, i: integer;
begin
writeln( ’‘A‘ betűk számlálása.’);
aszam := 0;
for i:=1 to 10
do begin
   write( i, ’.karakter:’);
   readln( c);
   if c = ’A’
   then
        aszam := aszam+1;
   end;
writeln( aszam, ’ db. ‘A‘ volt.’);
end.
```

Tömb változó

- egy név, több érték (skalár változó: egy név, egy érték).
- a komponensek egyforma típusúak.
- az indextípus határozza meg a tömb méretét, az értékkészlete minden eleméhez tartozik a tömbnek egy komponense.
- a komponensek indexeléssel érhetők el.
- tömbökön csak értékadás és egyenlőségvizsgálat van értelmezve.

```
program indexeles;
var
  c: char;
  t: array [ char] of integer;
begin
{...}
for c:= chr( 0) to chr( 255)
do
   t[ c] := 0;
{...}
end.
```

```
program karakterstatisztika;
var
  i: integer;
  c: char;
  t: array [ char] of integer;
begin
writeln( 'karakterstatisztika.');
for c:= chr( 0) to chr( 255)
do
   t[ c] := 0;
for i:=1 to 10
do begin
   write( i, '.karakter:');
   readln( c);
   t[ c] := t[ c]+1;
   end;
for c:=chr( 0) to chr( 255)
do
   if t[ c] > 0
   then
        writeln( '`', c, '` -> ', t[ c]);
end.
```

Intervallumtípus

- az alaptípus nem lehet `REAL`.
- értékkészlet: az alaptípus megadott részhalmaza.
- műveletek: az alaptípus műveletei.

```
program betustatisztika;
var
  i: 1..10;
  c: 'A'..'Z';
  t: array [ 'A'..'Z'] of 0..10;
begin
writeln( 'betűstatisztika.');
for c:= 'A' to 'Z'
do
   t[ c] := 0;
for i:=1 to 10
do begin
   write( i, '.karakter:');
   readln( c);
   t[ c] := t[ c]+1;
   end;
for c:='A' to 'Z'
do
   writeln( '`', c, '` -> ', t[ c]);
end.
```

Többdimenziós tömb

```
program matrixszorzas;
var
  a:    array [ 1..10] of array [ 1..10] of integer;
  b, c: array [ 1..10, 1..10] of integer;
  i, j, k: 1..10;
begin
{ ...
  a és b mátrixok feltöltése
}

for i:=1 to 10
do
   for j:=1 to 10
   do begin
      c[ i, j] := 0;
      for k:=1 to 10
      do
         c[ i][ j] := c[ i][ j] + a[ i, k]*b[ k, j];
      end;

{
  c mátrix már a és b szorzatát tartalmazza
  ...}
end.
```

Konstansok definiálása

- A konstansok olyan 'változók' amelyeknek az értéke nem változik.
- A definiáláskor kapnak értéket és a program futása során ez nem változhat meg (nem szerepelhetnek értékadó utasítás bal oldalán és input utasításban).
- Előnyök: Áttekinthetőbb program, módosításkor kevesebb hibalehetőség.

Típusok definiálása

- Nevet adunk egy általunk definiált típusnak.
- Az így definiált típusnév a beépített típusnevekkel megegyező módon használható.
- Előnyök: Áttekinthetőbb program, módosításkor kevesebb hibalehetőség.

```
program matrixszorzas2;
const
  meret= 10;
type
  index= 1..meret;
  matrix= array [ index, index] of integer;
var
  a, b, c: matrix;
  i, j, k: index;
begin
{ ...
  a és b mátrixok feltöltése
}

for i:=1 to meret
do
   for j:=1 to meret
   do begin
      c[ i, j] := 0;
      for k:=1 to meret
      do
         c[ i, j] := c[ i, j] + a[ i, k]*b[ k, j];
      end;

{
  c mátrix már a és b szorzatát tartalmazza
  ...}
end.
```

Egész számok sorbarendezése

```
program buborek;
const
  meret= 10;
type
  index= 1..meret;
  vektor= array [ index] of integer;
var
  t: vektor;
  i, j, m: index;
  x: integer;
begin
writeln( 'Buborékrendezés.');
{ vektor feltöltése }
for i:=1 to meret
do begin
   write( i, '.szám:');
   readln( t[ i]);
   end;

{ rendezés, ld. köv. oldal }

{ eredmény kiírása }
for i:=1 to meret
do
   writeln( t[ i]);
end.
```

Egész számok sorbarendezése, folyt.

```
for i:=meret downto 2
do begin
   { megkeressük a legnagyobbat }
   for j:=1 to i
   do
      if ( j=1) or ( t[ m] < t[ j])
      then
           m := j;
   { az m indexű a legnagyobb }
   { megjegyezzük }
   x := t[ m];
   { helyet csinálunk a tetején }
   for j:=m to i-1
   do
      t[ j] := t[ j+1];
   { legnagyobb a helyére }
   t[ i] := x;
   end;
```

STRING típus

- `STRING` $\approx$ `ARRAY [1..255] OF CHAR`
- `STRING[`N`]` $\approx$ `ARRAY [1..`N`] OF CHAR`
- konstansok: `'a'`, `'abcd'`, ...
- műveletek:
  - $\texttt{LENGTH} : \texttt{STRING} \to \texttt{INTEGER}$
  - $\texttt{CONCAT} : \texttt{STRING}^2 \to \texttt{STRING}$
  - $\texttt{POS} : \texttt{STRING}^2 \to \texttt{INTEGER}$
  - $\texttt{COPY} : \texttt{STRING} \times \texttt{INTEGER}^2 \to \texttt{STRING}$
  - $\texttt{DELETE} : \texttt{STRING} \times \texttt{INTEGER}^2$
  - $\texttt{INSERT} : \texttt{STRING}^2 \times \texttt{INTEGER}$

```
program string1;
var
  s: string;
  i: 1..255;
  ujsor: boolean;
begin
writeln( 'mondat szavai soronként.');
readln( s);
ujsor := false;
for i:=1 to length( s)
do
   if s[ i] = ' '
   then
        ujsor := true
   else begin
        if ujsor
        then
             writeln;
        write( s[ i]);
        ujsor := false;
        end;
writeln;
end.
```

```
program string2;
var
  s: string;
  i: 1..255;
  n: integer;
  szoban: boolean;
begin
writeln( 'mondat szavainak számlálása.');
readln( s);
n := 0;
szoban := false;
for i:=1 to length( s)
do
   if s[ i] = ' '
   then
        szoban := false
   else begin
        if not szoban
        then
             n := n+1;
        szoban := true;
        end;
writeln( 'a mondat ', n, ' szóból áll.');
end.
```

WHILE utasítás

1. Feltétel kiértékelése.
2. Ha az eredmény hamis, vége a `WHILE` utasításnak.
3. A ciklustörzs végrehajtása.
4. Vissza az 1. pontra.

```
program legn0;
var
  x, m: integer;
begin
writeln( 'legnagyobb 0-ra végződő sorozatból.');
write( 'számot kérek:');
readln( x);
m := x;
while x <> 0
do begin
   if x > m
   then
        m := x;
   write( 'számot kérek:');
   readln( x);
   end;
writeln( 'a legnagyobb: ', m);
end.
```

REPEAT-UNTIL utasítás

1. A ciklustörzs végrehajtása.
2. Feltétel kiértékelése.
3. Ha az eredmény igaz, a `REPEAT` utasítás végetért.
4. Vissza az 1. pontra.

```
program sum0;
var
  x, s: integer;
begin
writeln( '0-ra végződő sorozat összege.');
s := 0;

REPEAT
write( 'köv. szám: ');
readln( x);
s := s+x;
UNTIL x = 0;

writeln( 'Az összeg: ', s);
end.
```

Számkitalálós program

```
program kitalal;
uses
  crt;
const
  min = 1;
  max = 100;
type
  intervallum = min..max;
var
  also, felso, proba: intervallum;
  c: char;
begin
writeln;
writeln( 'kitalálósdi ', min, ' es ', max, ' között.');

also := min;
felso := max;

{ ld. köv. oldal}

writeln( 'tehát a megfejtés: ', also);
end.
```

kitalál program lényege

```
repeat
proba := ( also + felso) div 2;
write( 'talan ', proba, ' ? (n k e) ');
c := readkey;
write( c);
if c = 'k'
then
     if proba = also
     then
          write( ' <- ez mi?')
     else
          felso := proba-1
else
     if c = 'n'
     then
          also := proba+1
     else
          if c = 'e'
          then begin
               also := proba;
               felso := proba;
               end
          else
               write( ' <- ez mi?');
writeln;
until also = felso;
```

CASE utasítás

- A kifejezés és a címkék típusa meg kell egyezzen.
- Minden címke csak egyszer fordulhat elő.

1. Kifejezés kiértékelése.
2. Ha a kifejezés értéke megegyezik valamelyik címkével, akkor a címkéhez tartozó utasítást végrehajtja és a `CASE` utasítás befejeződött.
3. Ha a kifejezés értékének megfelelő címke nincs, de van `ELSE` "címke", akkor az ehhez tartozó utasítást végrehajtja és a `CASE` utasítás befejeződött.
4. Ha a kifejezés értékének megfelelő címke és `ELSE` sincs, akkor a `CASE` utasítás befejeződött.

kitalál alternatív megoldás

```
repeat
proba := ( also + felso) div 2;
write( 'talan ', proba, ' ? (n k e) ');
c := readkey;
write( c);
case c
of
  'k':
       if proba = also
       then
    write( ' <- ez mi?')
       else
    felso := proba-1;
  'n':
       also := proba+1;
  'e': begin
       also := proba;
       felso := proba;
       end;
  else
       write( ' <- ez mi?');
end;
writeln;
until also = felso;
```

Felsorolás típus

- A felsorolás típusnál az értékkészlethez tartozó elemeket a programozó sorolja fel.
- Az elemek tetszőleges, máshol nem használt nevek (azonosítók) lehetnek.
- A `READLN` utasítás csak `INTEGER`, `CHAR`, `REAL` és `STRING` típusú változókba tud beolvasni, ezért a felsorolás típus használata csak nagyobb programokban "belsejében" tipikus.

```
type
  honap = (jan, feb, mar, apr, maj, jun,
           jul, aug, sze, okt, nov, dec);
  nyar= jun..aug;
var
  mainap= ( hetfo, kedd, szerda, csutortok,
            pentek, szombat, vasarnap);
  ehavi, elozohavi: honap;
begin
{...}
ehavi := nov;
elozohavi := pred( ehavi);
for mainap:=hetfo to pentek
do
   {...}
```

Sorszámozott típusok

- A sorszámozott típusok értékkészletének elemeit sorba lehet állítani (legkisebbtől a legnagyobbig).
- Az elemeknek a sorbaállítás alapján egy sorszámot adunk.
- *Csak* sorszámozott típus lehet:
  - `FOR` utasításban ciklusváltozó típusa.
  - Tömb indextípusa.
  - Intervallumtípus alaptípusa.
  - `CASE` utasításban cimke típusa.

Sorszámozott típusok

- A PASCAL sorszámozott típusai:
  - `INTEGER`, minden szám sorszáma saját maga.
  - `BOOLEAN`, $\texttt{ord(false)} \rightarrow 0, \texttt{ord(true)} \rightarrow 1$
  - `CHAR`, a sorszámok az ASCII kódok.
  - Intervallum típus, a sorszámok megegyeznek az alaptípus sorszámaival.
  - Felsorolás típus, a sorszámok 0-tól *elemszám*-1–ig terjednek.
- Műveletek sorszámozott típusokon:
  - $\texttt{ORD} : \texttt{T} \rightarrow \texttt{INTEGER}$
  - $\texttt{PRED} : \texttt{T} \rightarrow \texttt{T}$
  - $\texttt{SUCC} : \texttt{T} \rightarrow \texttt{T}$

Eljárások

- Az eljárások a programozó által definiált utasítások.
- Hívásuk a programban az eljárás nevének leírásával történik.
- Az assembly alprogramok PASCAL-beli megfelelője.
- Eljárásfej, eljárástörzs, paraméterek.

```
procedure proc(formális paraméterek);
eljárástörzs
 ...
begin
 ...
proc(aktuális paraméterek);
 ...
end.
```

```
program legn0;
var
  x, m: integer;

procedure diszsor;
begin
writeln( '----------------------');
end;
procedure beolvas;
begin
write( 'számot kérek: ');
readln( x);
end.
procedure frissit;
begin
if x > m
then
     m := x;
end;

begin { program kezdete}
diszsor;
writeln( '0-ra végződő számsor legnagyobb eleme.');
beolvas;
m := x;
while x <> 0
do begin
   frissit;
   beolvas;
   end;
writeln( 'a legnagyobb: ', m);
diszsor;
end.
```

Paraméteres eljárások

- Az eljárások paraméres formában is definiálhatók.
- A híváskor megadott paraméter értéke befolyásolhatja az eljárás működését.
- A formális paraméterek kezdőértékkel rendelkező változók.

```
...
procedure diszsor( hossz: integer);
var
  i: integer;
begin
for i:=1 to hossz
do
   write( '-');
writeln;
end;
...
begin
...
diszsor( 15);
...
diszsor( 32);
...
end.
```

Paraméterátadás

- A formális és az aktuális paraméterek számának meg kell egyezni, a felírási sorrend alapján kerülnek párosításra.
- Az összetartozó formális és aktuális paraméterek típusának meg kell egyeznie.

Érték szerinti paraméterátadás

- Az aktuális paraméter kifejezés lehet.
- A kifejezés kiértékelésre kerül, a kiszámított érték lesz a formális paraméterként megadott változó kezdőértéke.
- A formális paraméter egy önálló változó.

Cím szerinti paraméterátadás

- Az aktuális paraméter változó lehet.
- Az eljárás futása alatt a formális paraméter ugyanarra a tárolórekeszre hivatkozik, mint az aktuális paraméter.

Cím szerinti paraméterátadás

```
program cimszpld;
var
  x: integer;

procedure novel( var z: integer);
begin
z := z+1;
end;

begin
x := 1;
novel( x);
writeln( x);
end.
```

Azonosítók hatásköre

- A program egymást tartalmazó hatásköri blokkokra osztható, a program egésze és minden eljárás is egy-egy blokk.
- Egy azonosító hatásköre a programszöveg azon része, ahol az azonosítóra hivatkozhatunk.
- Hatásköri szabályok:
  - Egy azonosító hatásköre a deklaráció helyétől az őt tartalmazö legszűkebb blokk végéig tart.
  - Egy külső blokkban deklarált azonosító hatásköre “lyukas” lehet, ha belső blokkban is használjuk ugyanazt a nevet.

```
program tabla;
const
  meret= 10;
var
  i: integer;

procedure sor( x: integer);
var
  i: integer;
begin
for i:=1 to meret
do
   write( i*x:4);
end;

begin
...
for i:=1 to meret
do begin
   sor( i);
   writeln;
   end;
...
end.
```

Terminológia

Lokális azonosító : Egy blokk és egy azonosító viszonyát fejezi ki: Az azonosító az adott blokkban van deklarálva.

Globális azonosító :

- Egy blokk és egy azonosító viszonyában: Az azonosító az adott blokkon kívül van deklarálva.
- Megnevezett blokk híján: az azonosító a legkülső blokkban van deklarálva.

Változók élettartama

- Egy változó élettartama alatt a program futásának azt az időszakát értjük, amikor számára tárolóhely van foglalva, ez több szakaszból is állhat.
- Egy változó számára akkor foglalódik le tárolóhely, amikor a vezérlés belép a változót tartalmazó blokkba (elindul az eljárás).
- A változó számára lefoglalt tárolóhely felszabadul, amikor kilépünk a blokkból (végetér az eljárás).
- Következmény: egy eljárás lokális változói két hívás között nem tartják meg az értéküket.

```
program miez;
var
  x: integer;

procedure proc;
var
  y: integer;
begin
if x = 1
then y := 1
else y := y+1;
writeln( y);
end;

begin
x := 1;
proc;
x := 2;
proc;
end.
```

```
program p;
var
  x: integer;

procedure z;
var
  x: integer;
begin
x := 2;
end;

begin
x := 1;
z;
writeln( x);
end;
```

```
program p;
var
  x: integer;

procedure z( x: integer);
begin
x := 2;
end;

begin
x := 1;
z( x);
writeln( x);
end;
```

```
program p;
var
  x: integer;

procedure z( var y: integer);
begin
y := 2;
end;

begin
x := 1;
z( x);
writeln( x);
end;
```

```
program p;
var
  x: integer;

procedure z( var a: integer);
var
  b: integer;
begin
if a = 0
then
     b := a
else
     a : b+1;
end;

begin
x := 0;
z( x);
x := 5;
z( x);
writeln( x);
end;
```

```
program p;
var
  x: integer;

procedure z( var a: integer);
const
  b: integer= 17;
begin
if a = 0
then
     b := a
else
     a : b+1;
end;

begin
x := 0;
z( x);
x := 5;
z( x);
writeln( x);
end;
```

Függvények

- A függvény olyan eljárás, aminek visszatérési értéke van, vagyis befejeződése után egy értéket szolgáltat.
- Míg az eljárások hívása eljáráshívás utasítással történik, a függvények hívása csak kifejezésben fordulhat elő.
- A visszatérési értéket a függvény "nevének kell értékül adni".

```
program vektorsum;
const
  meret= 10;
type
  index= 1..meret;
  vektor= array [index] of real;
var
  avektor: vektor;

procedure novel( var r: real; n: real);
begin
r := r + n;
end;

function summazo( vektor v): real;
var
  i: index;
  sum: real;
begin
sum := 0;
for i:=1 to meret
do
   novel( sum, v[ i]);
summazo := sum;
end;

begin
{ avektor feltöltése }
writeln( 'a számok összege:', summazo( avektor));
...
end.
```

`record` típus

- A tömbtípushoz hasonlóan egy `record` típusú változó több részváltozót tartalmazhat (mezők).
- Akkor használjuk, ha több, különböző típusú adatok akarunk egy egységként tárolni.
- Értékadás és komponens kiválasztás művelet értelmezett rajtuk.

komponensek ...	`array`	`record`
... típusa	megegyezik	lehet különböző
... száma	rögzített	rögzített
... elérése	indexeléssel	minősített hivatkozással

Példa record-ra

```
type
  honap   = ( jan, feb, mar, apr,
              maj, jun, jul, aug,
              sze, okt, nov, dec);

  datum   = RECORD
              ev:     1900..3000;
              ho:     1..12;
              nap:    1..31;
              END;

  szemely = RECORD
              nev:     string[ 30];
              szulido: datum;
              hazas:   boolean;
              END;
var
  d: datum;
  xy: szemely;
```

```
d.ev := 1999;
xy.hazas := true;
xy.szulido := d;
```

Program vázlat

```
program adatbazis;
const
  max= 100;
type
  ...
var
  meret: 0..max;
  adatok: array[ 1..max] of szemely;

function hozzaad( x: szemely): boolean;
begin
if meret = max
then
     hozzaad := false
else begin
     inc( meret);
     adatok[ meret] := x;
     hozzaad := true;
     end;
end;
```

```
procedure listaz(
  lnev: boolean;lev: boolean; lhaz: boolean);
var
  i: 1..max;
begin
writeln( 'személyi adatok:');
for i:=1 to meret
do begin
   writeln( 'rekord sorszám:', i:3);
   if lnev
   then
        writeln( 'név:', adatok[ i].nev);
   if lev
   then
        writeln( 'születési év:',
                 adatok[ i].szulido.ev);
   if lhaz
   then
        if adatok[ i].hazas
        then
             writeln( 'házas')
        else
             writeln( 'nem házas');
   end;
end;
```

with utasítás

- Ha egy rekord több mezőjét használjuk egyszerre, a `with` utasítással a program rövidebbé tehető.

```
procedure beolvas( var x: szemely);
var
  c: char;
begin
writeln( 'Kérem a személy adatait!');
write( 'név:');
readln( x.nev);
write( 'születési év:');
readln( x.szulido.ev);
write( 'hónap:');
readln( x.szulido.ho);
write( 'nap:');
readln( x.szulido.nap);
write( 'házas (i/n):');
readln( c);
x.hazas := ( c = 'i' or c = 'I');
end;
```

```
procedure beolvas( var x: szemely);
var
  c: char;
begin
writeln( 'Kérem a személy adatait!');
with x
do begin
   write( 'név:');
   readln( nev);
   write( 'születési év:');
   readln( szulido.ev);
   write( 'hónap:');
   readln( szulido.ho);
   write( 'nap:');
   readln( szulido.nap);
   write( 'házas (i/n):');
   readln( c);
   hazas := ( c = 'i') or (c = 'I');
   end;
end;
```

```
with x
do begin
   write( 'név:');
   readln( nev);
   with szulido
   do begin
      write( 'születési év:');
      readln( ev);
      write( 'hónap:');
      readln( ho);
      write( 'nap:');
      readln( nap);
      end;
   write( 'házas (i/n):');
   readln( c);
   hazas := ( c = 'i') or ( c = 'I');
   end;
```

```
with r1                  with r1, r2
do                       do ...
   with r2
    do ...
```

Pointer típus

- A pointer (mutató) olyan változó, amely egy másik változó címét tartalmazza.
- Értékkészlet: érvényes címek + `NIL`
- Pointerekkel kapcsolatos műveletek:
  - változó címének megállapítása
  - indirekció: a pointer által meghatározott változó "megkeresése"
  - értékadás: azonos típusú pointerek értékei áttölthetők egymásba
  - összehasonlítás: egyenlő =, nem egyenlő <>

```
var
  x, y: integer;
  p, q: ^integer;
begin
p := addr( x);  {x címe p-be}
q := @y;        {y címe q-ba}
p^ := 5;        {x := 5}
```

```
q^ := p^;          {y := x}
p := q;            {p is y-ra mutat}
p^ := 3;           {y := 3}
end.
```

Dinamikus változók

	Statikus változó	Dinamikus változó
hatáskör	blokkhoz kötött	nincs nevük, nem értelmezhető
élettartam	automatikus létrehozás és megszüntetés	explicit létrehozás és megszüntetés

- A dinamikus változók egy elkülönített memóriaterületen tárolódnak (heap, vagy halom)
- Bármilyen típusú változó lehet statikus és dinamikus is.
- A pointerek általában dinamikus változók címét tartalmazzák.

Dinamikus változók kezelése

- A heap-ben lefoglalt, illetve rendelkezésre álló helyet nyilván kell tartani, ezt a feladatot végzi a 'heap manager'.
- Létrehozás: `new( p)`
  - Létrehoz egy `p` típusának megfelelő dinamikus változót.
  - A `p` pointert beállítja úgy, hogy az a létrehozott változóra mutasson.
  - Ha nincs elég memória, akkor `NIL`-re állítja a pointert.
- Megszüntetés: `dispose( p)`
  - Megszünteti a dinamikus változót, felszabadítja az általa lefoglalt helyet.
  - A `p` pointer értékét *nem* módosítja.

Dinamikus adatszerkezetek

- Ha a feldolgozandó input adatok mennyisége nem ismert, akkor legtöbbször dinamikus adatszerkezeteket kell használni.
- A dinamikus adatszerkezetek olyan dinamikus rekordokból állnak, amelyek tartalmaznak egy vagy több pointer mezőt.

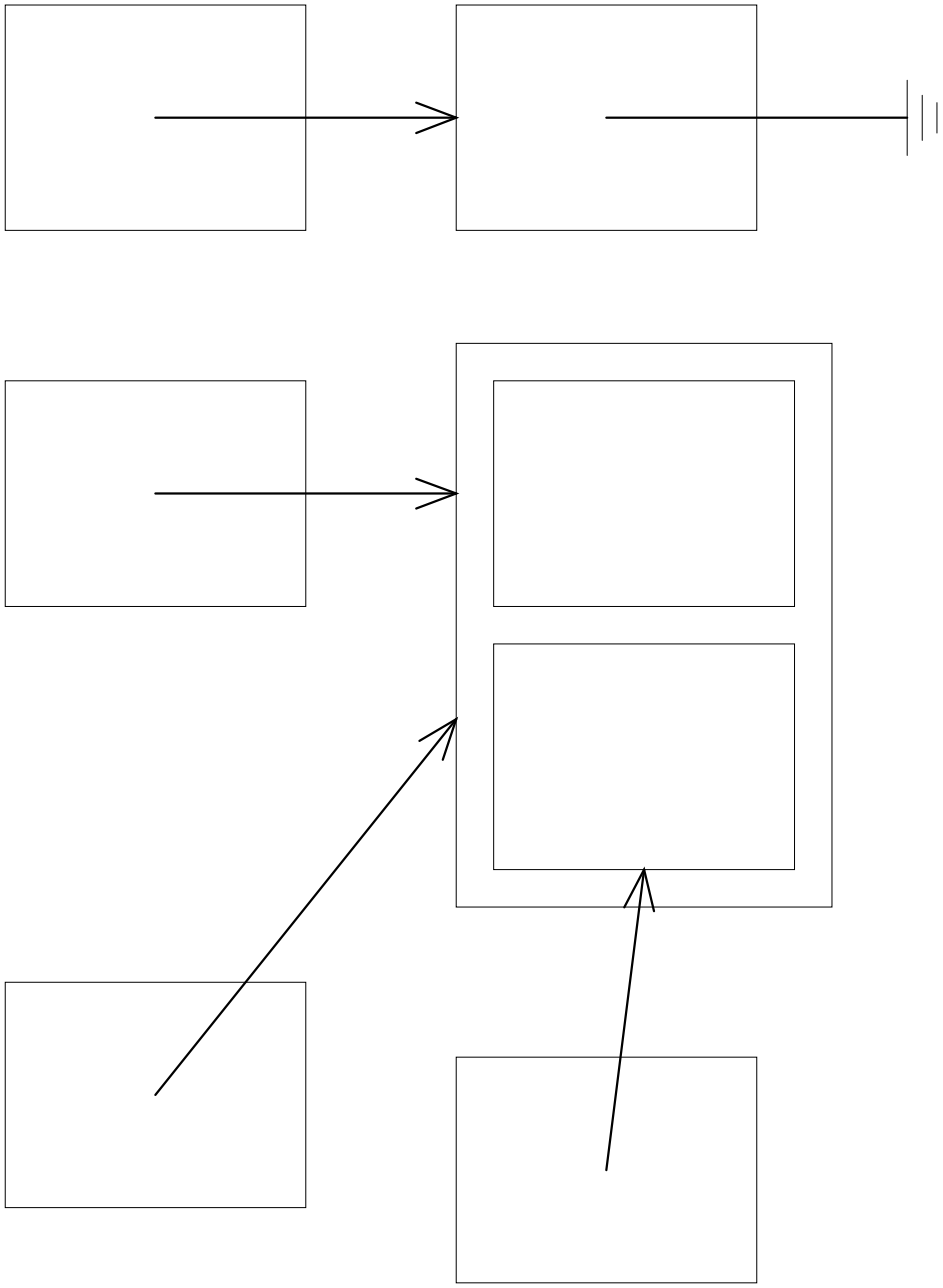
```
type
  ListaElemPtr = ^ListaElem;
  ListaElem    = record
                   adat: integer;
                   kov:  ListaElemPtr;
                   end;
var
  ListaFej: ListaElemPtr;
```

Új elem hozzáadása listához

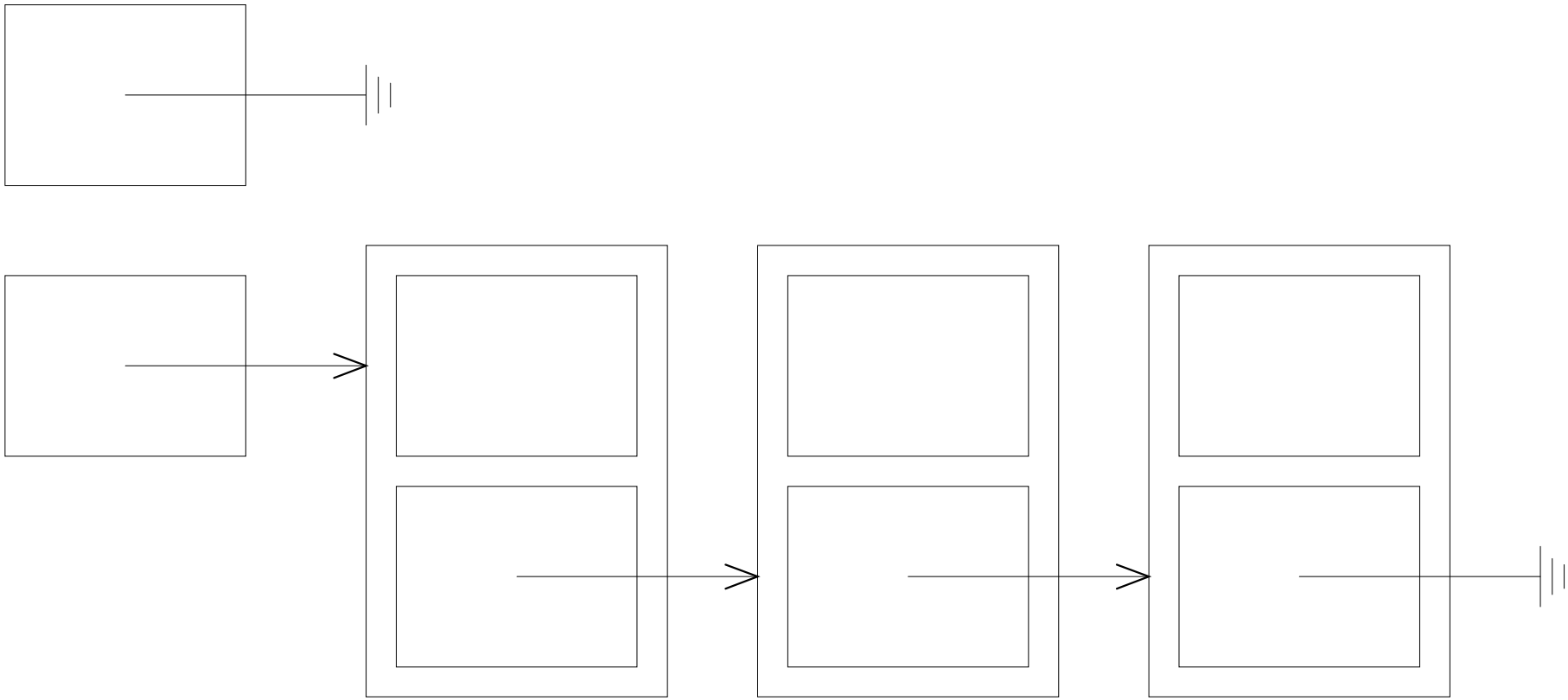
```
procedure ujelem( ujadat: integer);
var
  ujelemptr: ListaElemPtr;
begin
new( ujelemptr);
ujelemptr^.adat := ujadat;
ujelemptr^.kov := ListaFej;
ListaFej := ujelemptr;
end;
```

```
procedure ujelem( ujadat: integer);
var
  regifej: ListaElemPtr;
begin
regifej := ListaFej;
new( ListaFej);
ListaFej^.adat := ujadat;
ListaFej^.kov  := regifej;
end;
```

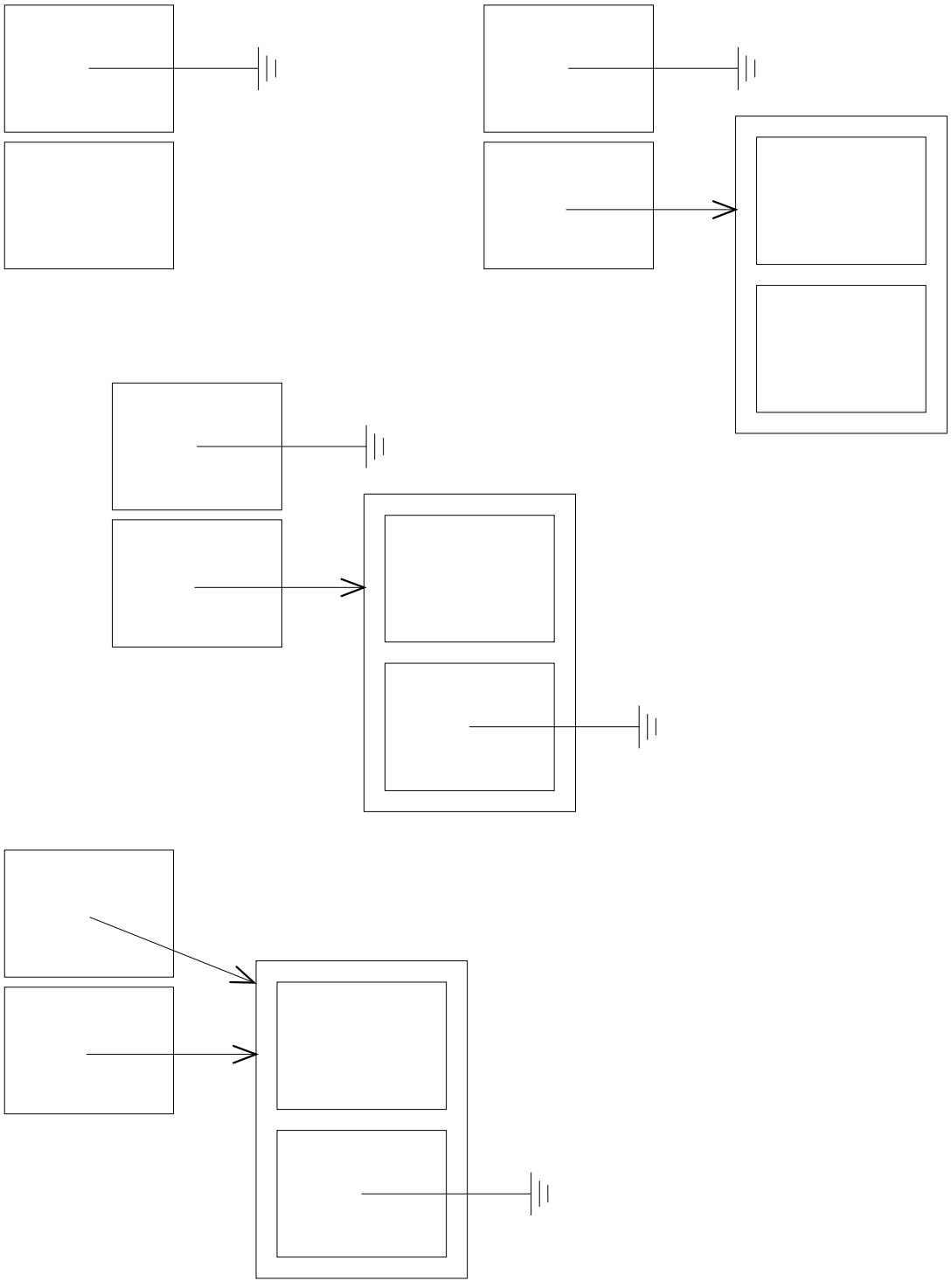
Pointerek ábrázolása



Láncolt lista



Beszúrás lista elejére



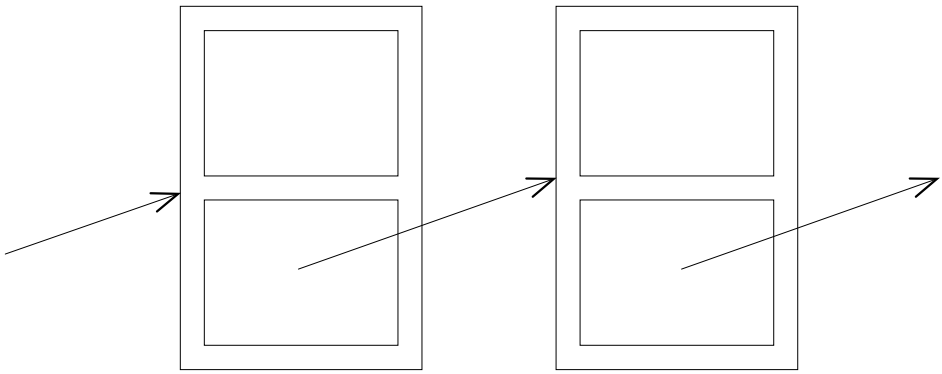
Lista elemeinek megszámlálása

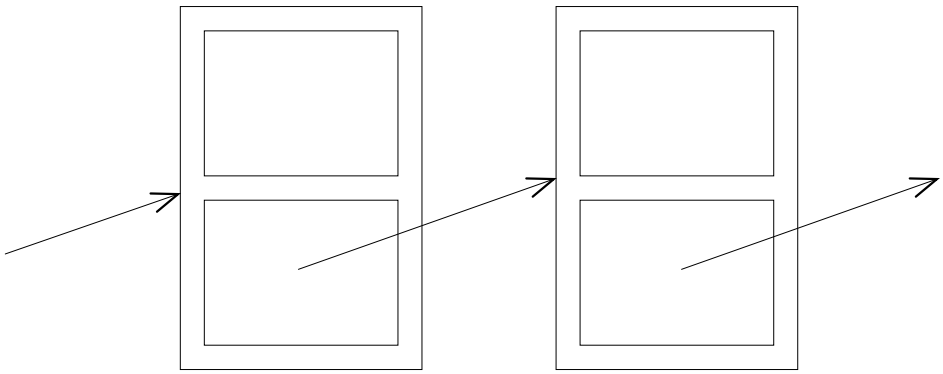
```
function hossz: integer;
var
  elemptr: ListaElemPtr;
  h: integer;
begin
elemptr := ListaFej;
h := 0;
while elemptr <> NIL
do begin
   inc( h);
   elemptr := elemptr^.kov;
   end;
hossz := h;
end;
```

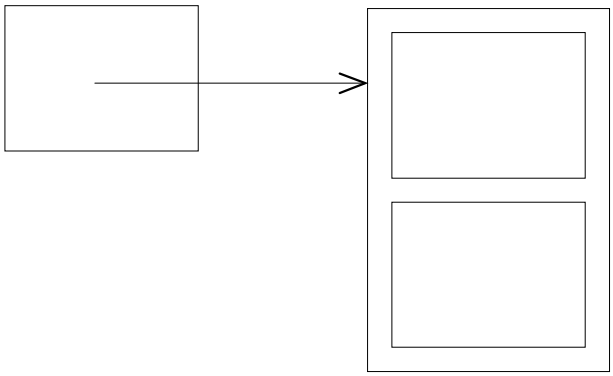
Beszúrás lista tetszőleges helyére

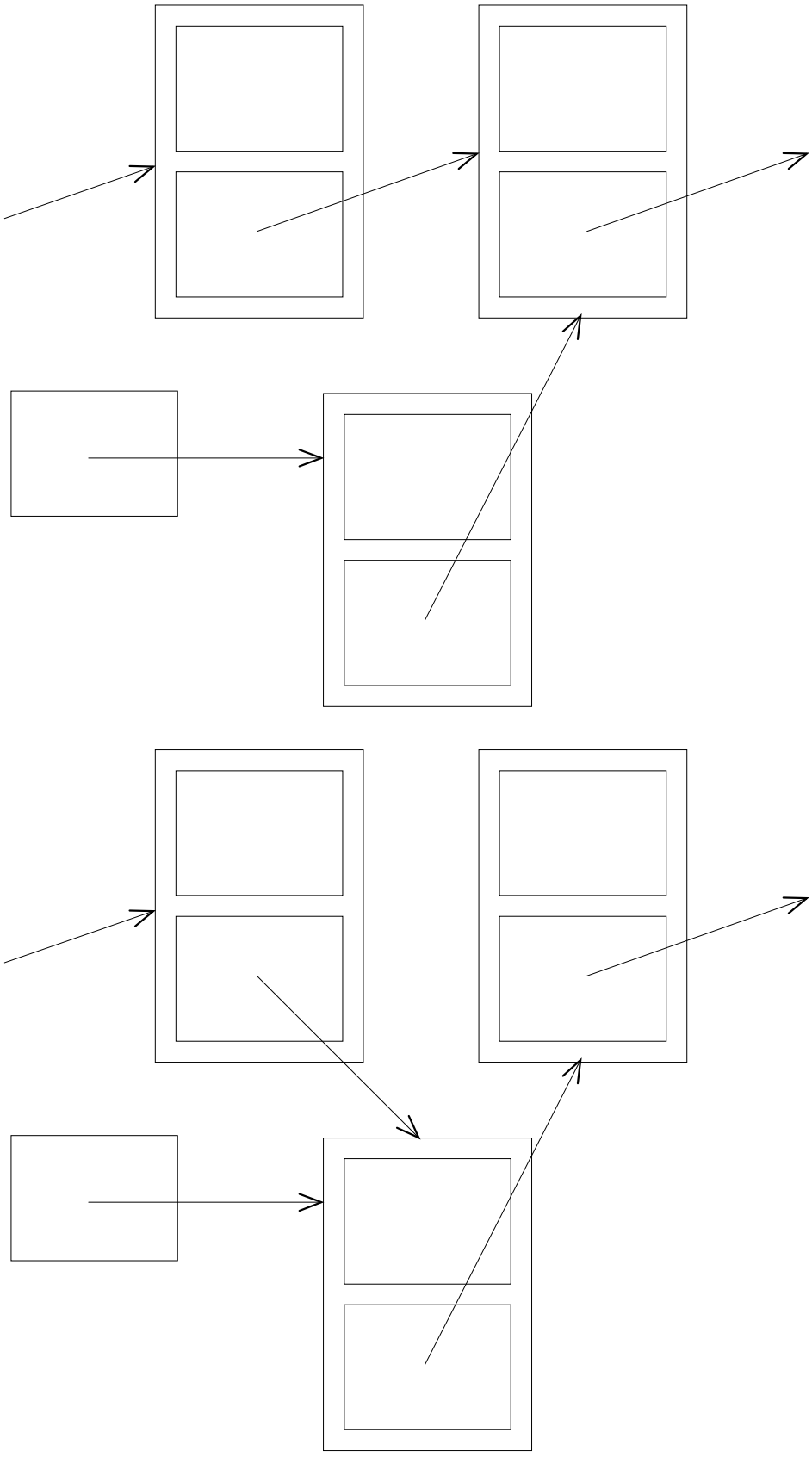
- Két paraméter kell
  - Hova szúrunk be: Pointerre mutató pointer.
  - Mit szúrunk be.

```
type
  ListaElemPPtr: ^ListaElemPtr;
...
procedure beszuras(
    pp: ListaElemPPtr; x: integer);
var
  ujelemptr: ListaElemPtr;
begin
new( ujelemptr);
ujelemptr^.adat := x;
ujelemptr^.kov := pp^;
pp^ := ujelemptr;
end;
```









Törlés listából

```
procedure torles( pp: ListaElemPPtr);
var
  elemptr: ListaElemPtr;
begin
{ pp és pp^ nem lehet NIL}
elemptr := pp^;
pp^ := elemptr^.kov;
dispose( elemptr);
end;
```

Lista megszüntetése

```
procedure megszuntetes;
begin
while ListaFej <> NIL
do
   torles( @ListaFej);
end;
```

Biztonságos indirekció

- Az indirekció csak nem-NIL pointerre alkalmazható.
- Alkalmazása esetén biztosnak kell lennünk abban, hogy az adott pointer nem lehet NIL.
- Módszer: Minden indirekcióhoz meg kell tudni mutatni azt a helyet a programban, ahol a hozzá tartozó pointert ellenőriztük, hogy nem NIL-e.

Keresés listában

```
function kereses( x: integer): ListaElemPPtr;
var
  elempptr: ListaElemPPtr;
begin
elempptr := @ListaFej;
{ !!! strict kiértékelés !!!}
while ( elempptr^ <> NIL) and ( elempptr^^.adat < x)
do
   elempptr := addr( elempptr^^.kov);
kereses := elempptr;
end;
```

BOOLEAN műveletek kiértékelése

- `FALSE and x` értéke mindig `FALSE`
- `TRUE or x` értéke mindig `TRUE`
- A `BOOLEAN` műveleteknek kétféle kiértékelési módja van:

  full : Mindenképpen kiértékeli mindkét paramétert.

  strict : Ha az első paraméter egyértelműen meghatározza az eredményt, akkor a a második paramétert nem értékeli ki.

- Kiértékelési mód beállítása
  - Fejlesztőkörnyezetben `Options/Compiler/Boolean Evaluation` menüpont
  - Forrásszövegben `{$B+}`, illetve `{$B-}`

0-ra végződő sorozat rendezése

```
program listasrendezo;
type
  ListaElemPtr = ^ListaElem;
  ListaElemPPtr = ^ListaElemPtr;
  ListaElem = record ...
var
  ListaFej: ListaElemPtr;
  x: integer;

procedure beszuras ...
procedure torles ...
procedure megszuntetes ...
function kereses ...

begin
ListaFej := NIL;
readln( x);
while ( x <> 0)
do begin
   beszuras( kereses( x), x);
   readln( x);
   end;
{eredmény kiírás, HF}
megszuntetes;
end.
```

Stack (verem) adattípus

- A verem olyan tároló, ami megőrzi a beletett dolgok sorrendjét. A tetejére tehetünk új elemet, illetve a tetejéről levehetjük a legfelsőt.
- Értékkészlet: értékek véges sorozatai
- Műveletek:
  - `push`: Új elem berakása
  - `pop`: Elem elvétele
  - `empty`: Üresség ellenőrzése
- Nem beéptett típus a PASCAL nyelvben, megvalósításához pl. láncolt lista használható.

```
type
  Stack= ListaElemPtr;
```

Stack műveletek

```
procedure push( s: Stack; x: integer);
begin
beszuras( @s, x);
end;

function pop( s: Stack): integer;
begin
{ ne hívjuk üres stackkel !!!}
pop := s^.adat;
torles( @s);
end;

function empty( s: Stack): boolean;
begin
empty := s = NIL;
end;
```

RPN kifejezések

- Az RPN kifejezésekkel zárójelek nélkül lehet felírni a számításokat.
- A műveleti jeleket nem az argumentumok közé, hanem utánuk tesszük.
- `2 + 4` $\rightarrow$ `2 4 +`
- `(2 + 4) * 3` $\rightarrow$ `2 4 + 3 *`
- `8 / (3 - 1)` $\rightarrow$ `8 3 1 - + /`
- Nem minden számokból és műveleti jelekből álló sorozat ad értelmes kifejezést.
- `3 4 + 2 * + 6`

A kifejezések kiértékelése

- A kiértékeléshez szükség van egy veremre, amiben számokat tárolunk.
- A kifejezés elemeit (számok és jelek) balról jobbra haladva feldolgozzuk.
- Ha számmal találkozunk, azt bedobjuk a verembe.
- Ha műveleti jelhez érünk, akkor
  - kiveszünk a veremből megfelelő számú számot,
  - elvégezzük rajtuk a műveletet,
  - majd az eredményt visszatesszük a verembe.
- Ha végeztünk, akkor a verem tetején lévő szám adja a végeredményt.

Egyszerűsítő feltételek

- A kifejezésben egyjegyű számokat, valamint a +,*,/ bináris és a - unáris műveleti jelet fogjuk használni.
- Minden egyéb jel hibás input-ot jelent.

Részfeladatokra bontás

```
kifejezés beolvasása
while nincs hiba és nem értünk a végére
do
   if kifejezés i.-edik komponense szám
   then
        bedobjuk a verembe
   else
        elvégezzük a műveletet
eredmény kiírása
```

```
program rpn;
type
  ListaElemPtr = ...
  ListaElem = record ...
  Stack = ListaElemPtr;

procedure beszuras ...
procedure torles ...
procedure push ...
function pop ...
function empty ...

var
  kif: string;
  s: Stack;
  i: integer;
  hiba: ( nincs, karakter, verem);

procedure add; ...
procedure neg; ...
procedure mul; ...
procedure div; ...
```

```
begin
write( 'kérem a kifejezést:');
readln( kif);
i := 1;
hiba := nincs;
while ( hiba = nincs) and ( i <= length( kif))
do begin
   if ( kif[ i] >= '0') and ( kif[ i] <= '9')
   then
        push( s, kif[ i]-ord( '0'))
   else
        case kif[ i]
        of
           '+': add;
           '-': neg;
           '*': mul;
           '/': div;
           else
                hiba := karakter;
        end;
   if hiba = nincs
   then
        inc( i);
   end;
if empty( s)
then
     hiba := verem;
case hiba
of
  nincs:    writeln( 'az eredmény:', pop( s));
  karakter: writeln( 'hibás karakter:', kif[ i]);
  verem:    writeln( 'a verem idő előtt kiürült');
end;
end.
```

Műveletek megvalósítása

```
procedure add;
var
  x: integer;
begin
if empty( s)
then
     hiba := verem
else begin
     x := pop( s);
     if empty( s)
     then
          hiba := verem
     else
          push( s, x+pop( s))
     end;
end;

procedure mul;
...
```

```
procedure sub;
var
  x, y: integer;
begin
if empty( s)
then
     hiba := verem
else begin
     x := pop( s);
     if empty( s)
     then
          hiba := verem
     else begin
          y := pop( s);  { !!! }
          push( s, y/x);
          end;
     end;
end;

procedure neg;
begin
if empty( s)
then
     hiba := verem
else
     push( s, -pop( s));
end;
```

Rekurzió

- Egy eljárás (függvény is) hívhatja saját magát közvetlenül, vagy közvetetten; az ilyet *rekurzív* (önhivatkozó) eljárásnak nevezzük.
- A végtelen hívási láncot elkerülendő a rekurzív eljárás mindig tartalmaz olyan szelekciós utasítást, amelynek nem mindegyik ágában hívja meg magát.
- A rekurzív hívást tehát minden esetben feltétel(ek) kiértékelése előzi meg, bizonyos eset(ek)ben sor kerül a rekurzív hívásra, más eset(ek)ben nem: *bázis eset*.
- Eljáráshívás esetén az eljárásba való belépéskor az érték szerinti paraméterek, valamint a lokális változók számára hely foglalódik. Emiatt rekurzió esetén minden híváshoz új változókészlet tartozik.

Egyszerű rekurzív függvények

```
function factorial( x: longint): longint;
begin
if x = 0
then
     factorial := 1
else
     factorial := factorial( x-1)*x;
end;
```

```
function hossz( lista: listaelemptr): integer;
begin
if lista == NIL
then
     hossz := 0
else
     hossz := 1 + hossz( lista^.kov);
end;
```

Rekurzív adatszerkezetek

- Egy dinamikus adatszerkezet rekurzív, ha "önhivatkozó" rekordokból épül fel.
- A legegyszerűbb rekurzív adatszerkezet a láncolt lista.

Bináris fa

```
type
  BinfaPtr = ^BinfaCsucs;
  BinfaCsucs = record
                 adat: integer;
                 bal : BinfaPtr;
                 jobb: BinfaPtr;
                 end;
```

- Elnevezések: Gyökér, levél, belső csúcs, részfa

Fabejárás

- A bejárás a fa összes csúcsában lévő adat valamilyen sorrendben történő feldolgozását jelenti.
- Bejárási stratégiák:
  - Preorder: gyökér, bal részfa, jobb részfa
  - Inorder: bal részfa, gyökér, jobb részfa
  - Postorder: bal részfa, jobb részfa, gyökér

```
procedure inorder( faptr: BinfaPtr);
begin
if faptr <> NIL
then begin
     inorder( faptr^.bal);
     {valami faptr^.adat-tal}
     inorder( faptr^.jobb);
     end;
end;
```

Rendezés bináris fával

- *Rendezett bináris fá*nak nevezünk egy fát, ha minden csúcsára igaz, hogy a benne tárolt elem nem kisebb a bal részfában tárolt elemeknél és nem nagyobb a jobb részfában tárolt elemeknél.
- Rendezett bináris fa inorder bejárása a tárolt elemeket nagyság szerint növekvő sorrendben dolgozza fel.
- Rendezési elv:
  - Üres fából indulunk ki (ez rendezett).
  - A rendezendő elemeket a fa rendezettségének fenntartása mellett egyesével elhelyezzük a fában.
  - Inorder bejárással megkapjuk a rendezett sorozatot.

Elem beszúrása bináris fába

```
procedure faba( var binfa: BinfaPtr; x: integer);
begin
if binfa = NIL
then begin
     new( binfa);
     binfa^.adat := x;
     binfa^.bal := NIL;
     binfa^.jobb := NIL;
     end
else
     if x <= binfa^.adat
     then
          faba( binfa^.bal, x)
     else
          faba( binfa^.jobb, x);
end;
```

```
program binfarendezes;
type
  BinfaPtr = ^BinfaCsucs;
  BinfaCsucs = ...

procedure inorder( binfa: BinfaPtr);
begin
...
   {valami:}
   writeln( binfa^.adat:4);
...
end;

procedure faba ...

var
  gyoker: BinfaPtr;
  x: integer;

begin
gyoker := NIL;
writeln( 'kérem a számokat:');
readln( x);
while x <> 0
do begin
   faba( gyoker, x);
   readln( x);
   end;
writeln( 'rendezett sorozat:');
inorder( gyoker);
end.
```

`File` típus

- A `file` összetett típus, hasonlít a tömb típushoz.
- Komponensei egyforma típusúak (nem lehet `file` típus).
- Mérete nem előre rögzített, a program futása közben változhat.
- A komponensek sorszáma egész szám, az első komponens a 0 sorszámú, 1-esével növekszik.
- A `file`-okat a feldolgozás megkezdése előtt meg kell *nyitni* olvasásra, vagy írásra.
- A komponenseket *szekvenciális elérés*sel (egymás után egyesével) lehet feldolgozni.

Külső adatállományok

- `File` típusú változón keresztül háttértárolón elhelyzett adatokat, adatállományokat dolgozhatunk fel.
- A feldolgozás megkezdése előtt az adatállományt hozzá kell rendelnünk a `file`-hoz.
- A `file` változón végzett műveletek a hozzá rendelt külső adatállományon hajtódnak végre.
- Ugyanaz a `file` változó különböző időpontokban más-más külső adatállományhoz tartozhat.

`file` deklarációk

```
type
  intfile  = file of integer;
  szemfile = file of szemely;
var
  intf:      intfile;
  szemelyek: szemfile;
  realf:     file of real;
```

`File` feldolgozás menete

- `File` változó és külső adatállomány összerendelése.
- `File` megnyitása olvasásra vagy írásra.
- Feldolgozás: komponensek kiolvasása vagy beírása szekvenciálisan. A megnyitott `file`-hoz tartozik egy *file pointer*, ami jelzi, hogy hányadik komponensnél tartunk.
- `File` lezárása (pufferelés!)

`File` típushoz tartozó eljárások

- Összerendelés:

  `procedure assign(var f:file; n:string);`

- Megnyitás
  - Olvasásra: `procedure reset( var f:file);`
    - A hozzárendelt `file`-nak léteznie kell.
    - A `file` pointert 0-ra állítja.
  - Írásra: `procedure rewrite( var f:file);`
    - Ha a `file` nem létezik, akkor létrehoz egy üreset.
    - Ha a `file` létezik, akkor törli a tartalmát.
    - A `file` pointert 0-ra állítja.
- Komponensek számának lekérdezése:

  `function filesize( var f:file):longint;`

- `File` végének ellenőrzése:

  `function eof( var f:file):boolean;`

`File` típushoz tartozó eljárások

- Feldolgozás
  - Következő komponens beolvasása

    `procedure read( var f: file; var x: ?);`

    * A megadott változóba beolvassa a `file` pointer által meghatározott komponenst.
    * A `file` pointert eggyel növeli.
  - Új komponens kiírása

    `procedure write( var f: file, x: ?);`

    * A `file` pointer által megadott pozícióra kiírja a megadott értéket.
    * A `file` pointert eggyel növeli.
- Lezárás: `procedure close( var f: file);`
  - A pufferben lévő adatokat kiírja az állományba.
  - A `file` változót hozzárendelhetjük másik állományhoz.

Véletlen számok

- Lehetőség (és szükség) van arra, hogy "véletlen" számokat állítsunk elő.

```
function random( x:  word): word;
```

- $0 \leq$ `random(x)` $< x$
- Az előállított számsorozatot egyértelműen meghatározza az első tagja, amit a **RandSeed** globális változó tárol.
- Ugyanarra az értékre állítva a **RandSeed**-et, ugyanazt a véletlen sorozatot kapjuk.
- Ha minden programfutáskor másik sorozatot akarunk, akkor a program elején meg kell hívni a **Randomize** eljárást (ez "véletlen" értékre állítja a **RandSeed** változót).

Adott tartományba eső véletlenszám

```
function rnd( min, max: integer): integer;
begin
rnd := min + random( max-min+1);
end;
```

Véletlen sorozat kiírása

```
program veletlenszamok;
const
  db = 10;
  min = -20;
  max = 100;

var
  i: integer;

function rnd ...

begin
randomize;
for i:=1 to db
do
   writeln( i:3, ':', rnd( min, max):4);
end.
```

File létrehozása

```
program filetrehozas;
const
  meret = 500;
  min   = -100;
  max   =  100;

var
  i:    integer;
  intf: file of integer;

function rnd ...

begin
randomize;
assign( intf, 'adatok.int');
rewrite( intf);
for i:=1 to meret
do
   write( intf, rnd( min, max));
writeln( 'a file mérete:', filesize( intf));
close( intf);
end.
```

File beolvasása

```
program fileolvasas;
var
  intf: file of integer;
  i, x: integer;

begin
assign( intf, 'adatok.int');
reset( intf);
writeln( 'file mérete:', filesize( intf));
i := 0;
while not eof( intf)
do begin
   read( intf, x);
   writeln( i:4, ':', x:5);
   inc( i);
   end;
close( intf);
end.
```

Nagy mennyiségű adat rendezése

- Ha a rendezendő adatok nem férnek el a memóriában, akkor két lépcsőben történik a rendezés:
  1. Az adatokat kisebb csoportokra bontjuk, a csoportokat egyenként rendezzük. Így rendezett részsorozatokat kapunk.
  2. A rendezett részsorozatokat *összefésüléssel* kettesével egyesítjük egyetlen rendezett sorozattá. Így N részsorozat esetén $N-1$ összefésülés után előáll a rendezett sorozat.
- Természetesen az eredeti és a részsorozatok tárolására is file-okat használunk.

Összefésüléses példa

- Rendezendő sorozat:
  $4, 6, 2, 1, 8, 9, 4, 2, 8, 0, 1, 3, 5, 6, 8, 3, 1, 5, 1$
- Részsorozatok:
  - $4, 6, 2, 1, 8, 9, 4, 2$
  - $8, 0, 1, 3, 5, 6$
  - $8, 3, 1, 5, 1$
- Rendezett részsorozatok:
  - $1, 2, 2, 4, 4, 6, 8, 9$
  - $0, 1, 3, 5, 6, 8$
  - $1, 1, 3, 5, 8$
- Első összefésülés után:
  - $0, 1, 1, 2, 2, 3, 4, 4, 5, 6, 6, 8, 8, 9$
  - $1, 1, 3, 5, 8$
- Második összefésülés után:
  $0, 1, 1, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 8, 8, 8, 9$

Összefésülés menete

- A rendezett részsorozatok elemeit csak szekvenciális eléréssel dolgozhatjuk fel, mert file-ban vannak.
- Az összefésülés során a két részsorozatot tároló file-t olvassuk és létrehozunk (írunk) egy új file-t az összefésült sorozat számára.
- Az összefésülés során a részsorozatokból egyenként átmásoljuk az elemeket az eredmény sorozatba. Az átmásolás addig folytatódik, amíg mindkét részsorozat el nem fogy.
- A program minden lépésben arról dönt, hogy melyik részsorozat első elemét másolja át az eredmény sorozatba: nyilván a kisebbet választja.

Összefésülés lépésenként

1.sorozat: 1, 2, 6, 8, 9
2.sorozat: 0, 1, 1, 3, 3, 4, 5, 5
eredmény:

1.sorozat: 1, 2, 6, 8, 9
2.sorozat: 1, 1, 3, 3, 4, 5, 5
eredmény: 0

1.sorozat: 2, 6, 8, 9
2.sorozat: 1, 1, 3, 3, 4, 5, 5
eredmény: 0, 1

1.sorozat: 2, 6, 8, 9
2.sorozat: 1, 3, 3, 4, 5, 5
eredmény: 0, 1, 1

1.sorozat: 2, 6, 8, 9
2.sorozat: 3, 3, 4, 5, 5
eredmény: 0, 1, 1, 1

1.sorozat: 6, 8, 9
2.sorozat: 3, 3, 4, 5, 5
eredmény: 0, 1, 1, 1, 2

1.sorozat: 6, 8, 9
2.sorozat: 3, 4, 5, 5
eredmény: 0, 1, 1, 1, 2, 3

1.sorozat: 6, 8, 9
2.sorozat: 4, 5, 5
eredmény: 0, 1, 1, 1, 2, 3, 3

Összefésülés lépésenként,folyt.

1.sorozat: 6, 8, 9
2.sorozat: 5, 5
eredmény: 0, 1, 1, 1, 2, 3, 3, 4

1.sorozat: 6, 8, 9
2.sorozat: 5
eredmény: 0, 1, 1, 1, 2, 3, 3, 4, 5

1.sorozat: 6, 8, 9
2.sorozat:
eredmény: 0, 1, 1, 1, 2, 3, 3, 4, 5, 5

1.sorozat: 8, 9
2.sorozat:
eredmény: 0, 1, 1, 1, 2, 3, 3, 4, 5, 5, 6

1.sorozat: 9
2.sorozat:
eredmény: 0, 1, 1, 1, 2, 3, 3, 4, 5, 5, 6, 8

1.sorozat:
2.sorozat:
eredmény: 0, 1, 1, 1, 2, 3, 3, 4, 5, 5, 6, 8, 9

Program

```
type
  intfile = file of integer;

  status = record
           f:     intfile;
           value: integer;
           flag:  boolean;
           end;

function mergeEof( var s: status): boolean;
begin
with s
do
   mergeEof := eof( f) and not flag;
end;
```

```
procedure nextinp( var s: status);
begin
with s
do if not flag
   then begin
        read( f, value);
        flag := true;
        end;
end;

procedure nextout( var s : status;
                   var fo: intfile);
begin
with s
do begin
   write( fo, value);
   flag := false;
   end;
end;
```

```
procedure merge( finp1, finp2, fout: string);
type
  status = ...

function mergeEof ...
procedure nextinp ...
procedure nextout ...

var
  fo: intfile;
  s : array[ 1..2] of status;
  i : 1..2;

procedure mopen;;
procedure phase1;
procedure phase2;
procedure mclose;

begin
mopen;
phase1;
phase2;
mclose;
end;
```

Megnyitás és inicializálás

```
procedure mopen;
begin
assign( s[ 1].f, finp1);
assign( s[ 2].f, finp2);

for i:=1 to 2
do with s[ i]
   do begin
      reset( f);
      flag := false;
      end;

assign( fo, fout);
rewrite( fo);
end;
```

Első fázis

```
procedure phase1;
begin
while not ( mergeEof( s[ 1]) or mergeEof( s[ 2]))
do begin
   nextinp( s[ 1]);
   nextinp( s[ 2]);

   if ( s[ 1].value <= s[ 2].value
   then i := 1
   else i := 2;

   nextout( s[ i], fo);
   end;
end;
```

Befejezés

```
procedure phase2;
begin
for i:=1 to 2
do while not mergeEof( s[ i])
   do begin
      nextinp( s[ i]);
      nextout( s[ i], fo);
      end;
end;

procedure mclose;
begin
close( fo);
close( s[ 1].f);
close( s[ 2].f);
end;
```

Moduláris programozás

- Ha a programban változtatunk, akkor az egészet újra kell fordítani. Ez sok időt vesz igénybe, ha a program hosszú.
- Ha többen dolgoznak egy nagyobb feladat részfeladatain, jó volna, ha az egyes részeket külön le lehetne fordítani (futtatni nem biztos).

- Egy PASCAL program több modulra bontható, ezeket külön file-okban tároljuk.
- Egy modul változtatása esetén általában nem kell az egész programot (összes modult) újrafordítani.

Modulok szerkezete

- Minden programban van egy főmodul és valahány almodul.
- Eddig csak olyan programot írtunk, ami egyetlen főmodulból állt.
- Az almodulok két részből állnak:
  - Interface rész — kapcsolódási felület, csak deklarációk.
  - Implementációs rész — a funkciókat megvalósító program.
- A modulok hierarchikus kapcsolatban állnak egymással, ennek a hierarchiának a csúcsán a főmodul helyezkedik el.
- Minden modul függ az alatta lévőktől, így a főmodul minden más modultól függ.

Modulok függése

- Egy modul kétféleképpen függhet egy másiktól:
  - Interface-en keresztül
  - Implementáción keresztül
- Interface függés esetén egy modul megváltozottnak tekinthető, amennyiben bármelyik olyan modul megváltozik, amitől ő függ.
- Ha valamelyik modulban változtatunk, minden olyan modult újra kell fordítani, amelyik ennek hatására megváltozott.

Újrafordítás

- Ha a modulokat külön le lehet fordítani, akkor ezeknek a tárgykódja nyilván külön file-ba kerül.
- Kérdés: hogyan lesz egy végrehajtható állomány a tárgykódokból?
- Válasz: a program fordítása két fázisra van bontva:
  1. Tárgykód létrehozása modulonként (fordítási fázis).
  2. Tárgykódok összeállítása egyetlen végrehajtható állománnyá (szerkesztési fázis)
- Fentiek miatt egy változtatás hatására általában nem kell az egész programot (összes modult) újrafordítani.
- A szerkesztést persze mindig meg kell csinálni.

Modulok szerkezete

- Az interface rész konstansok, típusok, változók, eljárások és függvények deklarációit tartalmazza.
- Az interface részben deklarált objektumok hatásköre kiterjed azokra a modulokra, amelyek függnek ettől a modultól:
  - Interface függés esetén a függő modul egészére kiterjed a hatáskör.
  - Implementáció függés esetén csak a függő modul implementációs részére terjed ki a hatáskör.

Modul szintaxis

```
unit xxx;      {filenév: xxx.pas}
interface
uses           {kihagyható}
  yyy, zzz;

const ...
type  ...
var   ...
procedure ...
function  ...

implementation
uses           {kihagyható}
  aaa, bbb, ccc;

const ...
type  ...
var   ...
procedure ...
function  ...

begin
{utasítások}  {kihagyható}
end.
```

Modul szemantika

- A modul implementációs részében ki kell fejteni azokat az eljárásokat és függvényeket, amelyeket az interface rész deklarál.
- Az interface részben deklaráltakon kívül egyéb objektumok is szerepelhetnek az implementációs részben.
- Az modulok inicializációs részei a főprogram első utasításának végrehajtása előtt futnak le.
- A hatáskör kiterjesztés miatt egy függő modul használhat minden olyan objektumot, ami a neki közvetlenül alárendelt modulok interface részében található:
  - Felhasználhatja a konstansokat és típusokokat.
  - Kiolvashatja és megváltoztathatja a változók értékét.
  - Meghívhatja az eljárásokat és függvényeket.

Stack megvalósítása modullal

```
unit intstack;
interface
type
  stackElemTipus = integer;
var
  stackError: boolean;

procedure clear;
procedure push( x: stackElemTipus);
function pop: stackElemTipus;
function empty: boolean;

implementation

const
  kapacitas = 100;
type
  stack = array[ 1..kapacitas]
            of stackElemTipus;

var
  s : stack;
  sp: 0..kapacitas;
```

Stack megvalósítása modullal

```
procedure clear;
begin
sp := 0;
stackError := false;
end;

procedure push( x: stackElemTipus);
begin
stackError := sp = kapacitas;
if not stackError
then begin
     inc( sp);
     s[ sp] := x;
     end;
end;
```

Stack megvalósítása modullal

```
function pop: stackElemTipus;
begin
stackError := sp = 0;
if not stackError
then begin
     pop := s[ sp];
     dec( sp);
     end;
end;

function empty: boolean;
begin
stackError := false;
empty := sp = 0;
end;

begin {inicializáció}
clear;
end.
```

Stack modul használata

```
program stackdemo;
uses
  intstack;

var
  i: integer;

begin
if empty
then
     writeln( 'A stack üres')
else
     eriteln( 'A stack nem üres');

for i:=1 to 102
do begin
   push( random( 1000));
   if stackError
   then
        writeln( 'Valami nem ok');
   end;

end.
```