Makró definíció belsejében meghívható az éppen definiálás alatt lévő makró is (a makró hívás ezáltal rekurzívvá válik).

```
PUSHALL  macro    reg1,reg2,reg3,reg4,reg5
IFNB     <reg1>             ;; ha a paraméter nem üres
         push     reg1      ;; az első regiszter mentése
         PUSHALL  reg2,reg3,reg4,reg5   ;; rekurzió
ENDIF
         ENDM
```

Most pl. a

```
         PUSHALL  ax, bx, cx
```

makró hívás hatása:

```
         push     ax
         push     bx
         push     cx
```



```
PUSHALL  macro    reg1,reg2,reg3,reg4,reg5
IFNB     <reg1>             ;; ha a paraméter nem üres
         push     reg1      ;; az első regiszter mentése
         PUSHALL  reg2,reg3,reg4,reg5   ;; rekurzió
ENDIF
         ENDM

         PUSHALL  ax, bx, cx
```

makró hívás hatása:

```
         push     ax
         PUSHALL  bx, cx
```

az újabb hívás hatása:

```
         push     bx
         PUSHALL  cx
```

az újabb hívás hatása:

```
         push     cx
         PUSHALL
```

ennek hatására nem generálódik semmi.


```
FL_CALLELJ     =   0
CALLELJ    macro ;; Eljárást beépítő és felhívó makró
         LOCAL  FIRST    ;; nem lenne fontos
IF       FL_CALLELJ      ;; a 2. hívástól igaz
         call   Elj      ;; elég felhívni az eljárást
         EXITM           ;; makró helyettesítés vége
ENDIF
FL_CALLELJ     =   1     ;; csak az első híváskor
         JMP    FIRST    ;; jut érvényre
Elj      proc            ;; eljárás deklaráció
         ...
         ret
Elj      endp
FIRST:   call   Elj      ;; az eljárás felhívása
         endm
```



```
FL_CALLELJ     =   1     ;; csak az első híváskor
         JMP    FIRST    ;; jut érvényre
Elj      proc            ;; eljárás deklaráció
         ...
         ret
Elj      endp
FIRST:   call   Elj      ;; az eljárás felhívása
```

Az első **CALLELJ** hívás hatására az

```
FL_CALLELJ     =   1
         JMP       ??0000
Elj      proc
         ...
         ret
Elj      endp
??0000:  call   Elj
```



```
         call   Elj      ;; elég felhívni az eljárást
         EXITM           ;; makró helyettesítés vége
```

A további **CALLELJ** hívások esetén csak egyetlen utasítás, a

```
         call   Elj
```

utasítás generálódik.

A megoldás előnye, hogy az eljárás akkor és csak akkor része a programnak, ha a program tartalmazza az eljárás felhívását is, és mégsem kell törődjünk azzal, hogy hozzá kell-e szerkesztenünk a programhoz vagy se.

Megváltoztathatunk egy makró definíciót azáltal, hogy újra definiáljuk.

Makró definíción belül előfordulhat másik makró definíció.

E két lehetőség kombinációjából adódik, hogy a makró definíción belül megadhatunk ugyanarra a makró névre egy másik definíciót, ezáltal készíthető olyan makró, amely „átdefiniálja” önmagát.

Az önmagát átdefiniáló makrók esetében a belső és külső definíciót lezáró **ENDM** utasítások között egyetlen utasítás sem szerepelhet – még kommentár sem!

Önmagát „átdefiniáló” makró (az előző feladat másik megoldása):

```
CALLELJ2    macro           ; külső makró definíció
            jmp     FIRST
Elj2        proc            ; eljárás deklaráció
            ...
            ret
Elj2        endp
FIRST:      call    Elj2    ; eljárás hívás
CALLELJ2    MACRO           ; belső makró definíció
            call    Elj2    ; eljárás hívás
            ENDM        ; belső makró definíció vége
            endm        ; külső makró definíció vége
```


CALLELJ2 első hívásakor a kifejtés eredménye:

```
            jmp     FIRST
Elj2        proc            ; eljárás deklaráció
            ...
            ret
Elj2        endp
FIRST:      call    Elj2    ; eljárás hívás
CALLELJ2    MACRO           ; belső makró definíció
            call    Elj2    ; eljárás hívás
            ENDM        ; belső makró definíció vége
```


A kifejtés **CALLELJ2** újabb definícióját tartalmazza, ez felülírja az eredeti definíciót, és a továbbiak során ez a definíció érvényes. Ez alapján a későbbi **CALLELJ2** hívások esetén

```
call    Elj2
```

a kifejtés eredménye.

Megjegyezzük, hogy most is szerencsésebb lett volna a **FIRST** címkét lokálissá tenni. Igaz, hogy csak egyszer generálódik, de így a **CALLELJ2** makró használójának tudnia kell, hogy a **FIRST** címke már „foglalt”!

Ha egy **M_név** makró definíciójára nincs szükség a továbbiak során, akkor a

```
PURGE   M_név
```

pszeudo utasítással kitörölhetjük.

Blokk ismétlés

Nemcsak a blokk definíciójának kezdetét jelölik ki, hanem a kifejtést (hívást) is előírják. A program más részéről nem is hívhatók.

Blokk ismétlés **kifejezés**-szer:

```
REPT    kifejezés
...     ; ez a rész ismétlődik
ENDM
```


A korábban ismertetett kopogást így is megoldhattuk volna:

A korábbi megoldás:

```
KOPOG   macro   n
        LOCAL   ujra
        mov     cx,n
ujra:   KOPP
        loop    ujra
        endm
```

```
        REPT    N
        KOPP
        ENDM
```

Ha pl. **N=3**, akkor ennek a hatására a

```
        KOPP
        KOPP
        KOPP
```

makró hívások generálódnak.

Megjegyzés: Most **N** nem lehet változó – fordítási időben ismert kell legyen az értéke!

Blokk ismétlés argumentum lista szerint:

```
IRP     par, <arg1[,arg2...]>
... ;   ez a rész többször bemásolásra
... ;   kerül úgy, hogy par rendre
... ;   fölveszi az arg1,arg2... értéket
ENDM

IRP     x, <1,2,3>
db      x
ENDM

db      1
db      2
db      3
```


Blokk ismétlés string alapján:

```
IRPC    par,string
...     ; ez a rész kerül többször bemásolásra úgy,
...     ; hogy par rendre fölveszi
...     ; a string karaktereit
ENDM
```

Ezt a **string**-et nem kell idézőjelek közé tenni (az újabb ismétlés jelentene). Ha a **string**-en belül pl. szóköz vagy , is előfordul, akkor **<>** jelek közé kell tenni.

Az előző feladatot így is megoldhattuk volna:

```
IRPC    x,123
db      x
ENDM
```


Másik példa:

```
IRPC    x,ABCDEFGHIJKLMNOPQRSTUVWXYZ
db      '&x'        ;; nagy betűk
db      '&x'+20h    ;; kis betűk
ENDM
```

Hatása:

```
db      'A'
db      'A'+20h             a kódja
. . .
db      'Z'
db      'Z'+20h             z kódja
```

Fontos az **&** jel, nélküle **'x'**-ben **x** nem paraméter, hanem string lenne!

Makró definíció tartalmazhat blokk ismétlést, és blokk ismétlés is tartalmazhat makró definíciót vagy makró hívást.

Pl.: A bit léptető és forgató utasítás kiterjesztésnek egy újabb megoldása:

```
; makrót definiáló blokkismétlés
      IRP   OP,<RCR,RCL,ROR,ROL,SAR,SAL>
OP&S  MACRO OPERANDUS,N
      mov   cl, N
      OP    OPERANDUS,cl
      ENDM
      ENDM
```

Ennek a megoldásnak előnye, hogy nem kell külön meghívnunk a külső makrót az egyes utasításokkal, mert ezt elvégzi helyettünk az **IRP** blokk ismétlés.


```
; makrót definiáló blokkismétlés
        IRP     OP,<RCR,RCL,ROR,ROL,SAR,SAL>
OP&S    MACRO   OPERANDUS,N
        mov     cl, N
        OP      OPERANDUS,cl
        ENDM
        ENDM
```

hatása:

```
RCRS    MACRO   OPERANDUS,N
        mov     cl, N
        RCR     OPERANDUS,cl
        ENDM
RCLS    MACRO   OPERANDUS,N
        ...
```


Szegmens, szegmens csoport

sz_név **SEGMENT** *aling_type combine_type 'osztály'*

. . . szegmens

sz_név **ENDS**

sz_név a szegmens (szelet) neve.

A fordító az azonos nevű szegmens szeleteket úgy tekinti, mintha folyamatosan, egyetlen szegmens szeletbe írtuk volna.

Az azonos nevű szegmens szeletek paraméterei egy modulon belül nem változhatnak.

A szerkesztő egy memória szegmensbe szerkeszti az azonos nevű szegmenseket.

aling_type (illesztés típusa): a szerkesztőnek szóló információ. Azt mondja meg, hogy a szegmens szelet milyen címen kezdődjön:

BYTE 1-gyel,

WORD 2-vel,

DWORD 4-gyel,

PARA 16-tal,

PAGE 256-tal osztható címen.

Akkor van jelentősége, ha a szegmens szelet egy másik modulban lévő ugyanilyen nevű szegmens szelet folytatása.

combine_type (kombinációs típus): a szerkesztőnek szóló üzenet. Lehet:

PUBLIC: (alapértelmezés) az azonos nevű szegmens szeletek egymás folytatásaként szerkesztendők.

COMMON: az azonos nevű szegmens szeletek azonos címre szerkesztendők. Az így keletkező terület hossza megegyezik a leghosszabb ilyen szegmens szelet hosszával. A **COMMON** hatása csak különböző modulokban megírt szegmens szeletekre érvényesül.

MEMORY: a szerkesztő ezt a szegmenst az összes többi szegmens fölé fogja szerkeszteni, mindig a program legmagasabb címre kerülő része (a Microsoft **LINK** programja ezt nem támogatja).

STACK: a stack részeként szerkesztendő a szegmens szelet, egyebekben megegyezik a **PUBLIC**-kal. Amennyiben van **STACK** kombinációs típusú szegmens a programban, akkor **SS** és **SP** úgy inicializálódik, hogy **SS** az utolsó **STACK** kombinációs típusú szegmensre mutat, **SP** értéke pedig ennek a szegmensnek a hossza.

AT kif: a **kif** sorszámú paragrafusra kerül a szegmens szelet. Alkalmas lehet pl. a port-okhoz kapcsolódó memória címek szimbolikus definiálására.

A szegmens osztály legtöbbször **CODE**, **DATA**, **CONSTANT**, **STACK**, **MEMORY**.

Beágyazott (nested) szegmensek

```
message   MACRO     text
          LOCAL     symbol
ADAT      SEGMENT   PARA PUBLIC 'DATA'
symbol    DB        &text
          DB        13,10,"$"
ADAT      ENDS
          mov       ah, 09h
          mov       dx, OFFSET symbol
          int       21h
          ENDM
          . . .
KOD       SEGMENT   PARA PUBLIC 'COCE'
          . . .
          message   "Please insert disk"
```


Az **ASSUME** utasítás az assembler-t informálja arról, hogy a címzésekhez a szegmens regisztereket milyen tartalommal használhatja, más szóval, hogy melyik szegmens regiszter melyik szegmensnek a szegmens címét tartalmazza (melyik szegmensre mutat):

ASSUME sz_reg1:sz_név1[, sz_reg2:sz_név2 ...]

ASSUME sz_reg1:sz_név1[, sz_reg2:sz_név2 ...]

Az **ASSUME** utasításban felsorolt szegmenseket „aktív"-nak nevezzük.

Az **ASSUME** utasítás nem gondoskodik a szegmens regiszterek megfelelő tartalommal történő feltöltéséről! Ez a programozó feladata!

Az **ASSUME** utasítás hatása egy-egy szegmens regiszterre vonatkozóan mindaddig érvényes, amíg egy másik **ASSUME** utasítással mást nem mondunk az illető regiszterről.

A **GROUP** utasítással csoportosíthatjuk a szegmenseinket:

```
G_nev   GROUP S_név1[, S_név2...]
```

Az egy csoportba sorolt szegmenseket a szerkesztő a memória egy szegmensébe helyezi. Ha ilyenkor az **ASSUME** utasításban a csoport nevét adjuk meg, és ennek megfelelően állítjuk be a bázis regisztert, akkor a csoport minden szegmensének minden elemére tudunk hivatkozni.
Ilyenkor egy változó **OFFSET**-je és effektív címe (**EA**) nem feltétlenül egyezik meg.


```
GRP     GROUP       ADAT1,ADAT2
ADAT1   SEGMENT     para public 'data'
A       dw          1111h
...
ADAT1   ENDS

ADAT2   SEGMENT     para public 'data'
W       dw          2222h
...
ADAT2   ENDS

code  segment     para public 'code'
      ASSUME      CS:code, DS:GRP
      ASSUME      SS:stack, ES:nothing
...
```



```
GRP   GROUP  ADAT1,ADAT2
ADAT1 SEGMENT ...
A     dw     1111h
...
ADAT1 ENDS

ADAT2 SEGMENT ...
W     dw     2222h
...
ADAT2 ENDS
```

```
code  segment ...
      ASSUME CS:code, DS:GRP
      ASSUME SS:stack, ...
      ...
```

```
      MOV SI,OFFSET W ; SI ⇐ W offset-je: 0
; az ADAT2 szegmens elejétől mért távolság
      MOV     AX,[SI]  ; AX ⇐ 1111h,
; de!!!
      LEA     SI, W    ; SI ⇐ effektív címe:
; a GRP szegmens csoport elejétől mért távolság
      MOV     AX,[SI]  ; AX ⇐ 2222h.
```