.LIST

Engedélyezi a forrás- és tárgykódú sorok bekerülését a lista file-ba (alapértelmezés).

.XLIST

Tiltja a forrás- és tárgykódú sorok bekerülését a lista file-ba. Jól használható arra, hogy **INCLUDE** előtt tiltsuk a listázást, utána **.LIST** -el újra engedélyezzük, és ezzel az **INCLUDE** file-ok többszöri listázását elkerüljük.

.LFCOND

Minden feltételes blokk kerüljön listára.

.SFCOND

Csak a teljesülő feltételes blokkok kerüljenek listára (alapértelmezés).

.TFCOND

Vált a két előző listázási mód között.

.CREF

Készüljön kereszthivatkozási (cross-reference) tábla (alapértelmezés). Ez a tábla azt a célt szolgálja, hogy könnyen megtaláljuk az egyes változókra történő hivatkozásokat a programban.

.XCREF

Ne készüljön kereszthivatkozási tábla.

.LALL

Kerüljön listára a makró hívások kifejtése.

.SALL

Ne kerüljön listára a makró hívások kifejtése.

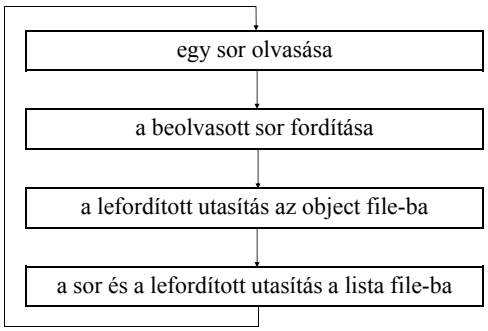
.XALL

A makró hívások kifejtéséből csak a kódot generáló rész kerüljön listára (alapértelmezés).

END kifejezés

A modul végét jelzi, **kifejezés** a program indítási címe. Ha a programunk több modulból áll, akkor természetesen csak egy modul végén adhatunk meg kezdő címet.

Assembler



előre hivatkozási probléma

Megoldási lehetőség:

Az assembler kétszer olvassa a program szövegét (két menet). Az első menet célja összegyűjteni, táblázatba foglalni a szimbólum definíciókat, így a második menet idején már minden (a programban definiált) szimbólum ismert, tehát a második menetben már nem jelentkezik az előre hivatkozási probléma.

Valahogy megpróbálni a fordítást egy menetben. Késleltetni a fordítást ott, ahol előre hivatkozás van, pl. táblázatba tenni a még le nem fordított részeket. A menet végén már minden szimbólum ismert, ekkor feldolgozni a táblázatot. Esetleg minden szimbólum definíciót követően azonnal feldolgozni a szimbólumra vonatkozó korábbi hivatkozásokat.

Mindkét esetben szükség van szimbólum tábla készítésére, de az utóbbi megoldásban a még le nem fordított utasítások miatt is szükség van táblázatra. További nehézséget jelent, hogy nem sorban készülnek el a tárgy kód (object code) utasításai, ezért ezeket pl. listába kell helyezni, majd rendezni a listát, és csak ezután történhet meg az object és a lista file elkészítése.

Manapság a legtöbb assembler két menetben működik.

Két menetes assembler, első menet

Legfontosabb feladata a szimbólum tábla felépítése.

A szimbólum tábla:

A szimbólum neve	értéke	egyéb információk
. . .	. . .	. . .

érték: – címke címe,
– változó címe,
– szimbolikus konstans értéke.

A szimbólum neve	értéke	egyéb információk
. . .	. . .	. . .

egyéb információk:
- típus,
- méret,
- szegmens neve, amelyben a szimbólum definiálva van,
- relokációs flag,
- . . .

Literál:

pl. az **IBM 370**-es gépcsaládon:

```
L       14,=F'5'   ; Load register 14 az 5-ös
                   ; Full Word konstanssal
```

Többek között a literálok gyakori használata vezetett a közvetlen operandus megadás kialakulásához és elterjedéséhez.

Egy lehetséges operációs kód tábla részlete:

mnemonic	op1	op2	kód	hossz	osztály
AAA	-	-	37	1	6
ADD	reg8	reg8	02	2	10
ADD	reg16	reg16	03	2	11
. . .	. . .	. . .	. . .	. . .	. . .
AND	reg8	reg8	22	2	10
AND	reg16	reg16	23	2	11
. . .	. . .	. . .	. . .	. . .	. . .


```
procedure ElsőMenet;  {1. menet, vázlat}
const méret = 8; EndUtasítás = 99;
var
  HelySzámláló, osztály, hossz, kód:
    integer;
  VanInput: boolean;
  szimbólum, literál, mnemo:
     array[1..méret] of char;
  sor: array[1..80] of char;
begin
  Előkészítés;
  TáblákIinicializálása;
  HelySzámláló := 0;
  VanInput = true;
```



```
while VanInput do begin     {sorok feldolgozása}
  SorOlvasás(sor);
  Megőrzés(sor);
  if NemKomment(sor) then begin  {nem kommentár}
    SzimbólumDef(sor, szimbólum);
    if szimbólum[1] <> ' ' then
                              {szimbólum definíció}
      ÚjSzimbólum(sor,szimbólum, HelySzámláló);
    LiterálKeresés(sor, literál);
    if literál[1] <> ' ' then
      ÚjLiterál(literál);
    hossz := 0;
    OpKódKeresés(sor, mnemo);
    OpKódTáblában(sor, mnemo, osztály, kód);
```

```
      if osztály < 0 then   {nem létező utasítás}
        PszeudoTáblában(sor,mnemo, osztály, kód);
      if osztály < 0 then HibásOpKód;
      hossz := típus(osztály);  {utasítás hossza}
      HelySzámláló := HelySzámláló + hossz;
      if osztály = EndUtasítás then begin
        VanInput := false;
        LiterálTáblaRendezés;
        DuplikátumokKiszűrése;
        Lezárások;
      end;   {if osztály = }
    end;     {nem kommentár}
  end;       { while VanInput }
end;         {1. menet}
```


OpKódKeresés eljárás triviális, mindössze az a feladata, hogy a **sor**-ban az első szóköz után a látható karaktereket a következő szóközig terjedően **mnemo**-ba másolja.

OpKódTáblában eljárás meglehetősen bonyolult, az operandusok elemzésével kell megállapítania, hogy az utasítás melyik **osztály**-ba tartozik. Látszólag feleslegesen határozza meg a **kód**-ot, de a többi feladata mellett ez már nagyon egyszerű, és így ez a függvény a második menetben változtatás nélkül alkalmazható.

Az **OpKódTáblában** eljárás nem alkalmas pl. az **ORG** pszeudo utasítás feldolgozására! Nem ismeri a **HelySzámláló**-t.

A **SorOlvasás(sor); Megőrzés(sor);** arra utal, hogy a második menetben olvashatjuk az első menet eredményét. Pl. az első menet folyamán szokás elvégezni az **INCLUDE** utasításokhoz, a makró definíciókhoz és makró hívásokhoz tartozó feladatokat.

Az **Előkészítés** valami ilyesmi lehet:

```
 Push(NIL);      {sehova mutató pointer a verembe}
  InputFileNyitás;
  p = ProgramSzövegKezdete;
...
```

A továbbiak során **p** mutatja a következő feldolgozandó karaktert.

A **SorOlvasás** eljárás:

```
begin
  while p↑ = EOF do begin
    Pop(p);
    if p = NIL then ENDHiba; {nincs END utasítás}
  end;
  EgySorOlvasás(sor);
end;
```


Ha **sor** történetesen **INCLUDE** utasítás, akkor az **EgySorOlvasás** eljárás ezt a következőképpen dolgozhatja fel:

```
Push(p);
IncludeFileNyitás;
p = IncludeSzövegKezdete;
```