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;
```



```
procedure MásodikMenet; {2. menet, vázlat}
const méret = 8; EndUtasítás = 99;
var
  HelySzámláló, osztály, hossz, kód: integer;
  VanInput: boolean;
  szimbólum, mnemo: array[1..méret] of char;
  sor: array[1..80] of char;
  operandusok[1..3] of integer;
    {op1, op2, címzési mód byte}
  object: [1..10] of byte;
begin
  Előkészítés2;
  {nincs TáblákInicializálása;}
  HelySzámláló := 0;
  VanInput = true;
```



```
while VanInput do begin      {sorok feldolgozása}
  SorOlvasás2(sor);
  {nincs Megőrzés(sor);}
  if NemKomment(sor) then begin  {nem kommentár}
    SzimbólumDef(sor, szimbólum);
    if szimbólum[1] <> ' ' then
                          {szimbólum definíció}
      SzimbólumEllenőrzés
        (sor, szimbólum, HelySzámláló);
    {nincs LiterálKeresés(sor, 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 HibásOpKód2;
      {Most készül a lista!}
    case osztály of
      0: hossz := fordít0(sor, operandusok);
      1: hossz := fordít1(sor, operandusok);
      ...
    end;
    Összeállítás
      (kód, osztály, operandusok, object);
    ObjectKiírás(object);
    Listázás(HelySzámláló, object, sor);
    HelySzámláló := HelySzámláló + hossz;
```

```
      if osztály = EndUtasítás then begin
        VanInput := false;
        {nincs LiterálTáblaRendezés;
        DuplikátumokKiszűrése;}
        Lezárások2;
      end;   {if osztály = }
    end;     {nem kommentár}
    else begin
       KommentárListázása;
    end;
  end;       {while VanInput }
end;         {2. menet}
```


Összeállítás = assemble

Az assembler számos hibát ismerhet fel:

- használt szimbólum nincs definiálva,
- egy szimbólum többször van definiálva,
- nem létező operációs kód,
- nincs elegendő operandus,
- túl sok operandus van,
- hibás kifejezés (pl. 9 egy oktális számban),
- illegális regiszter használat,
- típus hiba,
- nincs **END** utasítás,
- . . .

Számos olyan hibát azonban, melyet a magasabb szintű nyelvek fordítói könnyen felismernek – vagy egyáltalán elő se fordulhatnak – az assembler nem tud felderíteni:

- az eljárás hívás paramétereinek típusa nem megfelelő,
- a regiszter mentések és helyreállítások nem állnak „párban”,
- hibás vagy hiányzik a paraméter vagy a lokális változó terület ürítése a veremből,
- a hívás és a hívott eljárás helyén érvényes **ASSUME**-ok ellentmondásosak (nem feltétlenül hiba, de az lehet),
- . . .

Az object file nemcsak a lefordított utasításokat tartalmazza, hanem további – a szerkesztőnek szóló – információt is.

Makró generátor

Feladata a makró definíciók megjegyzése (pl. makró táblába helyezése) és a makró hívások kifejtése. Általában az assembler első menetében működik.

Makró definíciók felismerése

Amennyiben az assembler a forrás szöveg olvasása közben makró definíciót talál (ezt könnyű felismerni a műveleti kód részen lévő **MACRO** szó alapján), akkor a makró definíció teljes szövegét – az **ENDM** műveleti kódot tartalmazó sor végéig – elhelyezi a makró táblában. A felismerést és a tárolást a ***SorOlvasás*** vagy a ***PszeudoTáblában*** eljárás végezheti.

Makró hívások kifejtése

Az

```
OpKódTáblában(sor,mnemo, osztály, kód);
if osztály < 0 then       {nem létező utasítás}
    PszeudoTáblában(sor, mnemo, osztály, kód);
```

programrész után be kell szúrni az

```
if osztály < 0 then
    MakróTáblában(sor, mnemo, osztály, kód);
```

sorokat. A eljárás feladata a makró hívás felismerése és a makró helyettesítés is. A kifejtett makró egy pufferbe kerül, a puffer tartalma az **INCLUDE** utasításnál látottakhoz hasonlóan illeszthető a program szövegébe.

A makró kifejtés egy ciklusban:

```
EgySzóOlvasásaAMakróTörzsből;
 if FormálisParaméter then
   AMegfelelőAktuálisParaméterÁtmásolása;
 else
   ASzóÁtmásolása;
 ElválasztójelFeldolgozása;
```

Az ***ElválasztójelFeldolgozása*** legtöbbször az elválasztójel másolását jelenti, de a makró definícióban különleges szerepet játszó karakterek esetén ettől eltérő – magától értetődő – speciális feladatot kell végrehajtani.

A **`LOCAL`** utasítás feldolgozásához a makró generátor egy **`0`** kezdeti értékű változót használ. Makró híváskor a **`LOCAL`** utasításban szereplő szimbólumot, és az összes előfordulását a makró törzsben **`??xxxx`** alakú azonosítóval helyettesíti, ahol **`xxxx`** a változó aktuális értéke hexadecimális számrendszerben. A változó értékét minden a **`LOCAL`** utasításban szereplő szimbólum feldolgozása után **`1`**–gyel növeli.

Legegyszerűbb, ha a lokális szimbólumot formális paraméternek tekinti, és a generált **`??xxxx`** alakú azonosítót a megfelelő argumentumnak.

Szerkesztő

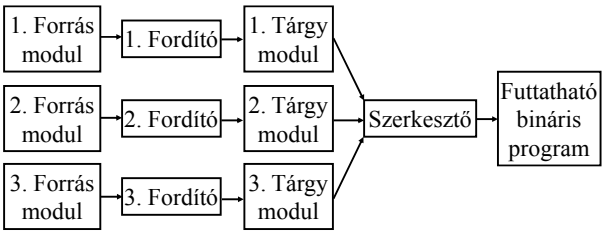
A következő feladatokat kell megoldania:

- az azonos nevű és osztályú szegmens szeletek egymáshoz illesztése a szegmens szeletek definíciójában megadott módon,
- a **`GROUP`** pszeudo utasítással egy csoportba sorolt szegmensek egymás után helyezése,
- a relokáció elvégzése,
- a külső hivatkozások (**`EXTRN`**) feloldása.

Az object file nemcsak a lefordított utasításokat tartalmazza, hanem további – a szerkesztőnek szóló – információt is.

Szerkesztő (~7.13. ábra)

Lehetővé teszi a program akár különböző nyelveken készített részleteinek összeillesztését.



Két menetben dolgozik:

- Az első menetben táblázatokat készít (térkép – map és globális szimbólum tábla).
- A második menetben a táblázatokban elhelyezett információk alapján elvégzi a szerkesztést.

A térkép (map) szegmens szeletenként a következő információt tartalmazza:

- modul név,
- szegmens név,
- osztály,
- illesztés típusa,
- kombinációs típus,
- hossz,
- kezdőcím,
- relokációs konstans.

Az első menet végén a térképet átrendezi, majd kitölti kezdőcímeket és a relokációs konstansokat.

A globális szimbólum tábla a **PUBLIC** változókból:

modul név	szegmens név	szimbólum	típus	cím
...	...	...	...	...

A **PUBLIC** utasítás nem tartalmazza a típust és a szegmens nevét, de az assembler ismeri, és el tudja helyezni a tárgy (object) modulban.
A cím a táblázat összeállításakor még relokálatlan címet jelent. Az első menet végén ebben a táblázatban is elvégezhető a relokáció.

Az assembler az **EXTRN** utasítás alapján a következő információt adja át:

		hivatkozás1		hivatkozás2		...
szimbólum	**típus**	**szegmens név**	**offszet cím**	**szegmens név**	**offszet cím**	...
...	...	...	...	...	...	...

Definiálatlan külső hivatkozások.

Moduláris programozás.

Object könyvtárak.

Dinamikus szerkesztés

Nagyméretű programokban bizonyos eljárások csak nagyon ritkán szükségesek. Ezeket nem kell statikusan a programhoz szerkeszteni.

Csatoló táblázat (Linkage Segment): minden esetleg szükséges eljáráshoz egy csatoló blokkot (struktúrát) tartalmaz.

```
CSAT_STR  STRUCT
CIM       DD     FAR PTR CSATOLO
NEV       DB     '         '; 8 szóköz
CSAT_STR  ENDS
```



```
CSAT_STR   STRUCT
CIM        DD     FAR PTR CSATOLO
NEV        DB     '         '; 8 szóköz
CSAT_STR   ENDS
```

Pl. a dinamikusan szerkesztendő **ALFA** eljáráshoz az:

```
ALFA       CSAT_STR     <, 'ALFA'>
```

csatoló blokk tartozhat. Az eljárás csatolása, hívása:

```
           MOV    BX, OFFSET ALFA
           CALL   [BX].CIM
```

Első híváskor a **CSATOLO** kapja meg a vezérlést. A csatolandó eljárás neve a **[BX].NEV** címen található. A név alapján betölti és a programhoz szerkeszti a megfelelő eljárást.


```
CSAT_STR   STRUCT
CIM        DD     FAR PTR CSATOLO
NEV        DB     '         '; 8 szóköz
CSAT_STR   ENDS
```

A szerkesztés végeztével **CIM**-et a most a programhoz szerkesztett eljárás kezdőcímére változtatja, és erre a címre adja a vezérlést:

```
           JMP    [BX].CIM
```

JMP, és nem **CALL**, hogy a veremben a **CSATOLO**-t hívó programhoz való visszatérés címe maradjon.

További hívások esetén a **CSATOLO** közbeiktatása nélkül azonnal végrehajtásra kerül az imént csatolt eljárás.

CSATOLO használhatja és módosíthatja a program szerkesztésekor készült térképet (map) és a globális szimbólumok táblázatát.

Szokásos megszorítás: a csatolandó eljárás nem tartalmazhat **EXTRN** utasítást, és egyetlen, a memóriába bárhová betölthető modulból áll.
Ekkor a szerkesztés magára a betöltésre, és ennek a tényét rögzítő adminisztrációra egyszerűsödik.

A dinamikusan szerkesztett eljárásokat könyvtárakba szokás foglalni (pl.: **.dll**), az eljárások általában többszöri belépést tesznek lehetővé (re-entrant).