Módszeres programozás: Programozási bevezető

Ebben füzetben a programozási alapismereteket tekintjük át.

Először hétköznapi példákon keresztül ismerkedünk az *algoritmus*okkal és az adatszerkezetekkel kapcsolatos fogalmakkal.

Ezután a *programkészítés* egyes lépéseit nézzük át nagyvonalúan, mintegy megelőlegezve a *Módszeres programozás* sorozat további köteteit.

Harmadiknak a *specifikáció*t állitjuk vizsgálataink középpontjába, mint a programkészités egyik legfontosabb, s alapvető lépését.

A következőkben végignézzük az algoritmusokban használt *alapelemek*et, melyeket az adatok, *adatszerkezetek* követnek.

Nyolc *algoritmusleíró eszköz*zel ismerkedhetünk meg a könyv hatodik fejezetében, az egyes eszközök értékelésével együtt.

Utolsóelőttiként a *dokumentáció* fajtáival és tulajdonságaival foglalkozunk.

Legvégül, de nem utolsósorban áttekintjük a fontos *programkészítési elvek*et, a megoldás elkészitésének stratégiájától az esztétikailag jól mutató "apróságokig".

Szlávi Péter — Zsakó László

Módszeres programozás:

Programozási bevezető

μlógia 18

ELTE IK Informatika Szakmódszertani Csoport

8., javított kiadás

Készült az NJSzT gondozásában
200 példányban
Felelős kiadó: Dr. Kozma László
Sorozatszerkesztő: Szlávi Péter – Zsakó László

Tartalomjegyzék

Előszó 5
I. A számítógép programozása – hétköznapi algoritmusok 7
1. Utcai telefonkészülék használati algoritmusa 7
2. Szörpautomata használati algoritmusa 8
3. Hogyan készül a szódavíz 11
4. Jegyárusító automata 15
5. Metróvezető algoritmusa 13
II. A programkészítés folyamata 21
III. Programozási fogalmak, specifikáció 23
IV. Algoritmikus szerkezetek 31
1. Az algoritmizálás alapjai 31
2. Algoritmikus elemek 34
2.1. Program 34
2.2. Értékadó utasítás 35
2.3. Beolvasó utasítás 35
2.4. Kiíró utasítás 35
2.5. Megjegyzés 36
2.6. Elágazások 36
2.7. Ciklusok 38
2.8. Eljárás, függvény, operátor 39
V. Adatok, adatszerkezetek 45
1. Az adatjellemzők összefoglalása 45
1.1. Azonosító 45
1.2. Hozzáférési jog 45
1.3. Kezdőérték 45
1.4. Hatáskör 45
1.5. Élettartam 46
1.6. Értéktípus 46
2. Az értéktípus 46
2.1. Az értéktípusról általánosságban 46
2.2. Az asszociált műveletek osztályozása 48
3. Egyszerű típusok 51
3.1. Egész típus 52
3.2. Valós típus 52
3.3. Logikai típus 53
3.4. Karaktertípus 53
3.5. Felsorolástípus 53

3.6. (Rész)Intervallumtípus ... 54
3.7. Valós résztípus ... 54
3.8. Tetszőleges típus résztípusa logikai formula alapján ... 55
4. Összetett (strukturált) típusok képzése ... 55
4.1. Rekordtípus-konstrukció ... 56
4.2. Alternatív rekordtípus-konstrukció ... 56
4.3. (Hatvány)Halmaztípus-konstrukció ... 57
4.4. Tömbtípus-konstrukció ... 58
4.5. Szövegtípus ... 58
5. Adatok leírása a programban ... 59
5.1. Típusdefiníció ... 59
5.2. Konstansok megadása ... 59
5.3. Változók deklarálása ... 59
VI. Algoritmusleíró eszközök ... 61
1. Folyamatábra ... 61
2. Struktogram ... 64
3. Jackson diagramok ... 65
4. Leírás fával ... 66
5. Leírás programozási nyelven ... 68
6. Leírás mondatokkal ... 68
7. Leírás mondatszerű elemekkel ... 69
8. Leírás absztrakt függvényekkel ... 69
VII. Dokumentálás ... 71
1. A dokumentáció fajtái ... 71
1.1. Fejlesztői dokumentáció ... 71
1.2. Felhasználói dokumentáció ... 72
1.3. Programismertető ... 72
1.4. Installálási kézikönyv, operátori kézikönyv ... 72
2. A dokumentáció tulajdonságai ... 73
2.1. Szerkezet ... 73
2.2. Forma ... 73
2.3. Stílus ... 73
VIII. Programkészítési elvek ... 75
1. Stratégiai elv ... 76
2. Taktikai elvek ... 77
3. Technológiai elvek ... 79
4. Technikai elvek ... 80
5. Esztétikai-ergonómiai elvek ... 81
Irodalomjegyzék ... 87

Előszó

Ez a füzet egy programozás módszertani sorozat első –bár időben nem elsőnek megjelent– kötete. Áttekintjük benne mindazokat a területeket, amelyek az ún. *amatőr programozási kultúra* részei. A szakirodalom ezt az ismeretkört a *programozás kicsiben (programming in small)* néven is illeti.

A teljes sorozat alapja, az 1986-ban a Műszaki Könyvkiadónál megjelent *Módszeres programozás* című könyvünk, illetve az arra is alapozott, 14 szerzőtársunkkal együtt írt, az OMIKK-nál megjelent *Számítástechnika középfokon*, illetve a Tanszékünk egyik füzetsorozatának két kiadványa: a *µlógia 8: A programkészítés technológiája* és a *µlógia Szilánkok 6: Adatok, adattípusok.*

Programozás kapcsán úgy tűnik, kétféle tevékenységről beszélhetünk. Egyrészt léteznek programkészítő szakemberek, akiknek ez a szakmájuk. Ők rendelkeznek mindazzal a módszertani ismerettel, programkészítést támogató eszközök ismeretével, amelyek akár tetszőlegesen nagyméretű, bonyolult rendszerek elkészítéséhez szükségesek.

Sok olyan célfeladat van azonban, amelyekhez nincs szükség egy szakember teljes szakmai fegyvertárára. (Gondoljunk arra, hogy az autónkat elvisszük-e szerelőhöz, ha kereket kell cserélni, lámpát kell cserélni, ... Kicsit hozzáértőbbek még néhány egyszerű javítást is maguk végeznek.) A programkészítés terén is sok olyan célprogram van, amelyet tanult emberek maguk is el tudnak (és kívánatos is, hogy el tudjanak) készíteni, szakember segítsége nélkül.

Ugyanígy természetesnek mondható az is, hogy az algoritmizálás, adatmodellezés ma már az általános kultúra részének tekinthető, így az iskolákban ilyen típusú ismeretekre (ezek tanítására) mindenképpen szükség van.

Van tehát kétféle, a programozással kapcsolatos „kultúrkör". Az egyik az átlag ember fejében meglévő ismereteken alapul, a másik a „profik piacközéppontú tudományán" nyugszik. Érdemes tehát mindenekelőtt összeszedni, összevetni: melyek is azok a szempontok, amelyek megkülönböztetik e kétféle programozási kultúrát? Az egyes szempontok nem különítik el élesen egymástól a kettőt, inkább utalások a különbözőségre.

Amatőr	Professzionális
Kis méretű feladatok megoldására viszonylag kis méretű programokat kell készíteni (50-1000 utasítás).	Nagy feladatok megoldására kell programokat, programrendszereket készíteni (néhány ezer, de gyakran több milliós nagyságrendű utasítás).
Ezeket a programokat általában egy vagy néhány ember írja, mindenki átlátja a teljes feladatot és a készülő megoldást.	Nagy programrendszereket programozói csoportok írnak, amelyek akár több száz főből is állhatnak. Mindenki csak a saját részét ismeri, s a többiekkel való kapcsolattartás módját.
Főfoglalkozásban nem programozó szakember írja, ismeretei a szakemberekénél lényegesen szűkebbek (e területen azonban lehet, hogy mélyebbek).	Szakemberek, sőt egyes célterületek specialistái készítik, a célterület és eszközei alapos ismeretével.

Az elkészült program viszonylag szűk körben használható, sokszor csak maga a készítő használja.	Az elkészült program gazdaságossági szempontok miatt minél szélesebb körben használandó.
A programot általában egyféle gépen használják.	A programot nagyszámú gépen gazdaságos használni.
Rövid életű programok, amelyek a változó igények miatt gyakran cserélődnek.	Hosszú életű programok, amelyek karbantartásáról, újabb változataik elterjesztéséről is gondoskodni kell.
Programkészítésnél a programozóval való párbeszédes kapcsolat fontos.	Programkészítéskor a modulokból való építkezés lehetősége az elsődleges.
Szerény hardver- és szoftverigények, kevés fejlesztési eszköz használata.	Sokféle hardver- és szoftverigény, sok fejlesztői szoftvereszköz kell hozzá.

I. A számítógép programozása – hétköznapi algoritmusok

Miből is áll egy feladat megoldása? Kiindulásul tegyük föl, hogy ismerjük a feladatot. „Csúnya" terminus technikus-szal mondva: elkészítettük a feladat *specifikáció*ját. Ezt a természetesnek ható dolgot azért kellett külön említeni, mert tapasztalat szerint éppen természetessége miatt sokszor ez a lépés, a specifikálás lépése marad végiggondolatlan, esetleges: sok-sok kérdést hagy nyitva, amelyek hiánya pillanatnyilag föl sem tűnik. A későbbi lépések során e hiányosságok halmozottan, ismételten föl-fölbukkannak, s emiatt kell több lépést többször is újra megtennünk.

A feladat biztos ismeretében már érdemben dönthetünk arról, hogy a feladat megoldásához milyen eszközöket, módszereket fogunk igénybe venni.

Ha ez megvan, meg kell terveznünk a feladat megoldását, számítógépes feladatmegoldás esetén meg kell alkotnunk a feladatot megoldó algoritmust. Ezt az algoritmust meg kell fogalmaznunk a számítógép számára is érthető nyelven, s valamilyen módon be kell juttatnunk a számítógépbe.

A számítógép ezek alapján már meg tudná adni a feladat megoldását, ha nem lennénk hajlamosak arra, hogy hibázzunk, elfelejtsünk valamit, azaz a számítógép nagyon gyakran nem azt a feladatot oldja meg az általunk megadott utasítássor kapcsán, mint amit mi vártunk tőle. Következő lépésben tehát meg kell győződnünk a megoldás helyességéről, s ha hibáztunk, akkor ki kell javítanunk.

Ahogyan a mindennapi életben, úgy a programokkal kapcsolatban is felmerülhetnek minőségi, gazdaságossági problémák, s nekünk ezekkel is foglalkozni kell.

A program egy termék, s mint minden termék, ő sem eladható értelmes használati utasítás nélkül, tehát még ilyet is írnunk kell.

A programkészítés során tehát több, különböző nehézségű és különböző kvalitásunkat próbára tevő tevékenységgel kell megbirkóznunk. Az egyik legnehezebbnek, legkreatívabbnak, az algoritmuskészítésnek az alapjaival ismerkedhetünk meg e fejezetben, sok hétköznapi példán keresztül.

1. Utcai telefonkészülék használati algoritmusa

Mit is nevezünk algoritmusnak, milyen jellemzői vannak, s hogyan kell ilyet csinálni? Vizsgáljunk meg egy egyszerű, mindenki által ismert, nap mint nap megoldott feladatot: hogyan is kell az utcán, egy telefonkészüléken telefonálni?[1]

```
Telefonhaszálat:
   Emelje le a kézibeszélőt!
   Várja meg a tárcshangot!
   Dobjon be egy 5 Ft-ost!
   Tárcsázzon!
Vége.
```

[1] Az itt közölt szöveges leírásnak megfelelő kis ábrákat, ún. ikonokat tartalmazó leírást találunk minden utcai telefonkészüléken. Mi itt csak „lefordítottuk szöveges magyarra".

Ez egy *algoritmus*, szemben a telefonkészülék formai leírásával (ami szintén a „telefonról szól"), amely semmiképpen sem az. Ez alapján az algoritmus jellemzői a következők:

1. végrehajtható,

*2. lépések*re bontva hajtható végre,

3. minden lépés vagy *elemi utasítás* vagy *további algoritmus*,

*4. pontos*nak kell lennie meghatározott *végrehajtási sorrend*del,

5. véges a leírás (bár a végrehajtása korántsem *mindig véges).*

Értelmezzük az egyes vonásait az algoritmusnak:

1. Lehet hozzá *végrehajtót* (egy valakit, egy gépet, egy processzort, egy LOGO-teknőcöt stb.) találni, neki, az „Ő nyelvén" szól.
2. *Világos* a végrehajtó számára, hogy mit tekintsen *lépés*nek (a neki szóló „mondat" egy szavának).
3. Elemi az, ami *értelme* felöl nincs kétsége a végrehajtónak, „magáért beszél", és persze képes is ennek „szellemében" cselekedni. Ha egy lépés önmagában nem egyértelmű (azaz számára új „fogalom"), akkor azt részletezni, értelmezni kell. Más szóval: meg kell adni a lépés (az új fogalom) részletes (további) részlépésekből összeálló algoritmusát. És ezt mindaddig folytatni, amíg a végrehajtásban már csak alapszavakkal kell megbirkóznia a végrehajtónak. Például a „Tárcsázzon!" felbontható 6-7-... számjegy egymásutáni tárcsázására, sőt egy számjegy tárcsázása különböző típusú telefonkészülékeken mást és mást jelenthet.
4. Nemcsak az egyes lépések önálló jelentése kell, hogy tiszta legyen, hanem az is, hogy melyiket mikor kell, lehet elvégezni; ez által lesz az *egész algoritmus* teljes, pontos. (Ahogy pl. a következő két nyelvtanilag helyes magyar mondat is azonos szavakból áll, de teljesen más értelemmel rendelkezik: „az úr ír", ill. „az ír úr", ugyanígy döntő az algoritmus lépéseinek sorrendje is.)
5. Ez azt jelenti, hogy előadódhatnak olyan helyzetek, amikor ez a *végesen* megfogalmazott parancs-sorozat precíz végrehajtása esetén soha sem ér véget. Gondoljunk csak arra, hogy ha a fenti algoritmust nem egy értelmes ember hajtaná végre, akkor a „Várja meg a tárcsahangot!" utasítás hatására rossz telefonkészülék esetén akár végtelen sokáig is várakozhatna.

2. Szörpautomata használati algoritmusa

Az előző példában egy olyan algoritmust láttunk, ahol ~~elemi tevékenységeket kellett végrehajtani, mindegyiket végre kellett hajtani, s adtunk hozzá egy egyértelmű végrehajtási sorrendet~~. Egy szörpautomata használatának menetén keresztül most azt vizsgáljuk, hogy elemi lépéseket hogyan, milyen sorrendben lehet végrehajtani.

```
Szörpautomata:
  Válaszd ki a megfelelő szörpöt!
  Dobj be egy 10 Ft-ost!
  Nyomd meg a kiválasztott szörphöz tartozó gombot!
  ISMÉTELD AMÍG nem telik meg a pohár: Nézd a poharat!
  Vedd ki a poharat!
Vége.
```

Az ismétlés tényét nyomatékosítjuk az `ISMÉTELD` kulcsszóval

Itt megjelent egy újdonság: **elemi tevékenység többszöri végrehajtása**. A végrehajtás számá egy *feltétel*től tettük függővé: addig kell várni, amíg megtelik a pohár. Az egymás után végrehajtandó elemi tevékenységek is különbözőek. Például a 'Vedd ki a poharat!' és az 'Idd meg a

szörpöt!' nevű tevékenységet *csak ilyen sorrendben* lehet elvégezni, fordítva nem, s egyszerre em. Ugyanakkor a 'Válaszd ki a megfelelő szörpöt!' és a 'Dobj be egy 10 Ft-ost!' tevékenységek **orrendje tetszőleges** lehet, sőt akár *egyszerre is* végezhetjük őket.

```
Szörpautomata:
  TETSZŐLEGES SORRENDBEN:
    Válaszd ki a megfelelő szörpöt!
    Dobj be egy 10 Ft-ost!
  Nyomd meg a kiválasztott szörphöz tartozó gombot!
  ISMÉTELD AMÍG nem telik meg a pohár: Nézd a poharat!
  Vedd ki a poharat!
Vége.
```

`TETSZŐLEGES SORRENDBEN` : a *tetszőlegesség* kifejezése

Tehát, ha **elemi tevékenységek mindegyikét végre kell hajtani**, akkor a végrehajtás sorrendjére három szabály is alkalmazható: **Adott sorrendben egymás után kell végrehajtani, tetszőleges sorrendben lehet egymás után végrehajtani, lehet párhuzamosan, egyszerre végrehajtani őket.**

A bevezetőben arról volt szó, hogy az algoritmus végrehajtása nem feltétlenül véges. Ez persze a legtöbb esetben (mint most is) hiba. Ezért, amikor elemi tevékenységek ismétlését írjuk le, akkor mindig meg kell gondolni, hogy az ismétlés befejeződik-e. Most például elképzelhető a végtelenség, ha az automata csak félig adja a poharat. Módosítsuk az ismétlés feltételét!

```
Szörpautomata:
  TETSZŐLEGES SORRENDBEN:
    Válaszd ki a megfelelő szörpöt!
    Dobj be egy 10 Ft-ost!
  Nyomd meg a kiválasztott szörphöz tartozó gombot!
  ISMÉTELD AMÍG folyik a szörp: Nézd a poharat!
  Vedd ki a poharat!
  [Idd meg a szörpöt!]
Vége.
```

Persze elképzelhető, hogy a pohár megtelik, s a szörp tovább folyik ..., érdemes ezen is elgondolkodni, de most mi más úton fogunk továbbhaladni.

Nézzünk egy másik problémát! Mit tehetünk, ha nincs 10 Ft-osunk és az automata elfogad 2 Ft-osokat is?

```
Szörpautomata:
  Válaszd ki a megfelelő szörpöt!
  HA van 5 db 2 Ft-osod AKKOR  Dobj be 5 db 2 Ft-ost!
                       KÜLÖNBEN Dobj be egy 10 Ft-ost!
  Nyomd meg a kiválasztott szörphöz tartozó gombot!
  ISMÉTELD AMÍG folyik a szörp: Nézd a poharat!
  Vedd ki a poharat!
Vége.
```

Újdonság itt az, hogy most megadtunk két tevékenységet ('Dobj be 5 db 2 Ft-ost!', valamint 'Dobj be egy 10 Ft-ost!'), de **közülük csak az egyiket kell elvégezni**, azaz *választani* kell közülük.

A megoldás nem tartalmazza azt a lehetőséget, amikor nincs sem 2, sem 10 Ft-osunk. Ugyanakkor, ha van 5 db 2 Ft-os és van 10 Ft-os is, akkor a fenti két lehetőségből bármelyiket választhatjuk.

```
Szörpautomata:
  Válaszd ki a megfelelő szörpöt!
  LEHETŐSÉGEK
    HA van 5 db 2 Ft-osod AKKOR Dobj be 5 db 2 Ft-ost!
    HA van 10 Ft-osod AKKOR Dobj be egy 10 Ft-ost!
  Nyomd meg a kiválasztott szörphöz tartozó gombot
  ISMÉTELD AMHG folyik a szörp: Nézd a poharat!
  Vedd ki a poharat!
Vége.
```

LEHETŐSÉGEK : a *tetszőlegesség* kifejezése

Tehát újdonságként megjelent a **több lehetőségből való választás**, amelyben ráadásul a **választás nem is egyértelmű**.

Nézzük meg, mit tehetünk, ha egyáltalán nem folyik szörp!

```
Szörpautomata:
  Válaszd ki a megfelelő szörpöt!
  LEHETŐSÉGEK:
    HA van 5 db 2 Ft-osod AKKOR Dobj be 5 db 2 Ft-ost!
    HA van 10 Ft-osod AKKOR Dobj be egy 10 Ft-ost!
  Nyomd meg a kiválasztott szörphöz tartozó gombot!
  HA nem folyik a pohárba a szörp AKKOR
    ISMÉTELD: üsd az automata oldalát[2] AMIG nem folyik a szörp
  ISMÉTELD AMHG folyik a szörp: Nézd a poharat!
  Vedd ki a poharat!
Vége.
```

Most egy olyan utasítást is be kellett vezetnünk, amely azt „szervezi meg", hogy valamit **vagy végre kell hajtani, vagy nem.** Fölbukkant egy újabb fajta ismétlés is, ebben **csak azután nézzük meg, hogy kell-e még ismételni, miután az ismétlődő utasítás(oka)t egyszer már végrehajtottuk.**

A 2 Ft-osokat persze egyenként kell bedobni, s nem egyszerre. Ezért a megoldás kicsit precízebben így fogalmazható meg:

```
Szörpautomata:
  Válaszd ki a megfelelő szörpöt!
  LEHETŐSÉGEK:
    HA van 5 db 2 Ft-osod AKKOR ISMÉTELD 5-ször: Dobj be egy 2 Ft-ost
    HA van 10 Ft-osod AKKOR Dobj be egy 10 Ft-ost!
  Nyomd meg a kiválasztott szörphöz tartozó gombot!
  HA nem folyik a pohárba a szörp AKKOR
    ISMÉTELD: üsd az automata oldalát AMHG nem folyik a szörp
  ISMÉTELD AMHG folyik a szörp: Nézd a poharat!
  Vedd ki a poharat!
Vége.
```

Harmadik fajta ismétlést láthatunk ebben a megoldásban: megmondtuk pontosan, hogy **hányszor kell az ismétlendőt megismételni.**

Nézzük meg, hogy az összetett algoritmusainkat milyen módon bontottuk elemibbekre!

1. Az összetett algoritmus *több* elemibb algoritmusból áll, és *mindegyik*et végre kell hajtani.
   a) A végrehajtási *sorrend adott*, s egyszerre *csak egy*et lehet végrehajtani.
   b) A végrehajtási *sorrend tetszőleges*, s egyszerre *csak egy*et lehet végrehajtani.
   c) A végrehajtási *sorrend tetszőleges*, akár egyszerre is végre lehet hajtani az *összes*et.
2. Az összetett algoritmus *egy vagy több* elemi algoritmusból áll, és ezek közül *csak legfeljebb az egyik*et kell végrehajtani.
   a) 1 elemi algoritmust vagy végrehajtunk, vagy nem.
   b) 2 elemi algoritmus közül egyértelműen választanunk kell az egyiket.
   c) Sok elemi algoritmus közül egyértelműen választanunk kell az egyiket.
   d) Sok elemi algoritmusok közül választanunk kell az egyiket, de a választás nem egyértelműen definiált.
3. Az összetett algoritmus *1* elemi algoritmus *ismétlés*éből áll.
   a) Tudjuk, hogy pontosan hányszor kell végrehajtani.
   b) Tudunk mondani egy feltételt, aminek teljesülése esetén még végre kell hajtani.
   c) Tudunk mondani egy feltételt, aminek teljesülése esetén még végre kell hajtani, de a feltétel vizsgálatához az ismétlendőt egyszer már végre kellett hajtani, azaz legalább egyszer „lefut" az ismétlendő.

Az 1/b és az 1/c megvalósítható az 1/a segítségével, a 2/d pedig a többi 2-es valamelyikével, így ezekkel külön nem foglalkozunk.

Az 1. megvalósítását a programozásban egymás utáni végrehajtásnak, **szekvenciá**nak nevezzük, a 2.-at **választás**nak vagy **elágazás**nak, a 3.-at pedig **ismétlés**nek vagy **ciklus**nak.

Az 1/b és a 2/d hozná be a programozásba az ún. **nemdeterminisztikusság**ot, az 1/c pedig a **párhuzamosság**ot.

3. Hogyan készül a szódavíz?

Minden számítógépes feladatmegoldásnak alfája az, hogy tisztázzuk: miből lehet kiindulni, mit kell kiszámolni, milyen eredményhez kell eljutni[3]. Vagyis a majdani program *bemenő* és *kimenő adat*airól kell pontos képet kialakítani. Ehhez először is megismerkednünk az adatok jellegzetességeivel. Az alábbi hétköznapi példa középpontjába egy „adatmetafora", a szódás szifon áll.

A feladat nagyon egyszerű: készítsünk szódavizet! Kezdjük azzal, hogy meggondoljuk, milyen kellékek szükségesek hozzá! Tehát szükséges:

- víz,
- patron és
- szifon.

A szódavíz-készítés nagyvonalú algoritmusa mindenki számára valahogy a következőképpen néz ki:

```
Szódavíz-készítés:
  Csavard szét a szifont!
  Öntsd bele a vizet!
  Csavard össze a szifont!
  Csavard rá a teli patront!
Vége (Igyál).
```

Annyi máris látszik, hogy kellékeink bizonyos szempontból alapvetően különböznek: a víz, a patron, amolyan „hozzáadandó" (*bemeneti* kellékek), amíg a szifon „adottság" (hisz ha ő nincs,

[2] Itt persze az ember ilyenkor „szokásos" reakcióját fogalmaztuk meg az algoritmusban. Lehet vérmérséklettől függően mást is tenni, pl.:

```
'ISMÉTELD: fohászkodj a szörpautomaták Szelleméhez AMÍG nem folyik a szörp'
```

[3] ... és persze *hogyan* ... (az ehhez szükséges ismeretek elemeivel barátkoztunk eddig).

az alapprobléma eleve föl sem merül). Sőt van olyan valami, ami az eljárás során keletkezik, ez a szódavíz (*kimeneti* kellék). A szódavízkészítés szempontjából a be- és a kimeneti valamik –összes függvén egymással– változnak, így ők *változó* „objektumok", amíg a szifon bizonyos alkatrésze *állandó*knak tekinthetők.

A kellékeink „sokfélesége" más nézőpontokból is vizsgálható, és jól mintázza az absztrakt adato jellemzőit is. Például a *szerkezet*ükre koncentrálva: amíg a víz, a patron egyszerűek (mondhatn *elemi*ek), addig a szifon bonyolult, szerkezettel rendelkező (úgymond: *összetett* valami). Újab nézőpontból: a kellékek *bizonyos vonása* szempontjából a vízmegléte, vagy a patron milyenség kérdésesnek tekinthető, a szifon részeinek meglétét eleve feltesszük (minthogy ennek megsé tésétől tartani annyi lenne, mint attól félni, hogy valaki kávéfőzővel akarna szódavizet csinálni).

Most már –hihetnénk– továbbléphetünk, s foglakozhatunk az algoritmus mélyebb régióinak fe vázolásával is. De mégsem. Ahhoz, hogy megfogalmazhassuk, e kellékekből milyen módon k szül a szódavíz: pontosítani kell a „tárgyiasult" részleteket, a kellékek szerkezetét, s csak e követően térhetünk rá az egyes „részeken" operáló utasítások ('Csavard szét...', 'Öntsd bele. stb.) megtervezésére.[4]

A kellékek részletezése (gyakran a részletezése helyett finomítása szót használnak):

- A patron lehet széndioxiddal (vagy mással) teli és lehet üres.
- A szifon testből és fejből áll.
- A test tartályt és benne levő csövet jelent.
- A tartályban –ideális esetben– víz és széndioxid keveredik össze szódavízzé.
- A fej számunkra érdekes részei: a szelep (amelyet tovább nem bontunk, hisz nem szerelni ak runk) és a patront tartó rész.
- A patronrész, a rajta elvégezhető és elvégezendő műveletek által, „szétesik" egy kosárra és benne levő (üres, vagy valamivel telt) patronra.

E hosszadalmas szöveges leírást az alábbi tömörebb, s az „egymásba illeszkedést" is jól kifeje formalizmussal ismételjük meg. (Az a nem titkolt szándék van e mögött, hogy az adatok leírásá is valamilyen rövidebb, egyértelműbb nyelvezet felé induljunk el.)

```
Patron [∈{üres, CO2}]
Szifon = Test = Tartály = H2O
                          CO2
               Cső
         Fej   = Szelep
               Patronrész = Kosár
                            Patron [∈{üres, CO2}].
```

A kellékek finomítása után már magától értetődő az eljárásunk finomítása. Ez természetesen m mindig fog tartalmazni nem definiált részletű „makró tevékenységeket", melyeknek „csak" a c ja világos (a neve utal rá, és -esetleg- a szögletes zárójelek közé szúrt megjegyzések):

[4] Ezzel egy későbbi –a programozási stílusunkat nagyban befolyásoló– elvünk szellemében cselekszünk. Az elv állítja, hogy az adatok és a velük kapcsolatos tevékenységek tisztázásában együtt kell haladnunk, s kettejük kö az adatokra vonatkozó döntésünk a meghatározó.


```
Szódavíz-készítés:
  HA szifon üres AKKOR
    Csavard le a szifonról a fejet!                    [test:=szifon-fej]
    Vedd ki a csövet!                                  [tartály:=test-cső]
    Csavard le a fejről a patronrészt!                 [szelep:=fej-patronrész]
    Vedd ki a patronrészből a patront!                 [kosár:=patronrész-patron]
    Szifontestet öntsd tele vízzel![5]                 [Tartály:=víz]
    Tedd bele a csövet!                                [test:=tartály+cső]
    Csavard rá a testre a fejet!                       [szifon:=test+fej]
    Végy patront!                                      [=CO2]
    Cseréld ki a patronokat!                           [CO2 ⟷ üres]
    Csavard rá a szelepre a patronrészt!               [patronrész:=kosár+patron]
                                                       [Tartály:=Tartály+CO2]
vége.
```

Néhány probléma fölmerül a fenti algoritmussal kapcsolatosan, amelyek fölött nem szabad elsiklanunk. Ezek a meggondolni valók:

- bemenet problémája: csak azon „értékekre" várhatjuk el a jó működést, amelyeken „értelmezve" van az eljárásunk,
- másként -ügyesebben- is megfogalmazható az algoritmus (jobb idejekorán rájönni arra, hogy az eljárást nem végezhetjük el és akkor nem is próbálkozunk):

```
Szódavíz-készítés:
  Végy patront! [CO2] [6]
  HA szifon üres ÉS patron teli AKKOR
    Csavard le a szifonról a fejet!
    Vedd ki a csövet!
    Csavard le a fejről a patronrészt!
    Vedd ki a patronrészből a patront!
    Szifontestet öntsd tele vízzel!
    Tedd bele a csövet!
    Csavard rá a testre a fejet, hogy a szifon teljes legyen!
    Cseréld ki a patronokat!
    Csavard rá a szelepre a patronrészt, hogy a fej teljes legyen!
vége.
```

Megérett a helyzet arra, hogy megfogalmazzunk -az eddigi tapasztalatainkra építve- négy kulcsfontosságú programozási fogalmat: a *konstans*, a *változó*, az *értékadás* és a *típus* fogalmát. A szifon „állapotán" az eljárásunk folyamán változás történik. (Hisz ennek érdekében dolgozunk rajta.) Üresből szódavízzel telivé lesz. Eközben bizonyos részei (a tartályban a cső, a szelep a fejen) megmaradtak ugyanabban a minőségben, amiben kezdetben voltak. (Nagy bajt jelentene, ha nem így lenne!) És mindazt, amit leírtunk alkalmazhatjuk a világ összes szódás szifonjára, vagyis „nagyképűbben mondva": a szódás szifonok *kategóriá*jára.

Konstans: az az *adat*, amely a műveletvégzés során *nem változtat(hat)ja meg értékét*, mindvégig ugyanabban az „állapotban" marad.

Változó: az ilyen *adat*féleségnek lényegéhez tartozik a „változékonyság", más szóval: vonatkozhatnak rá olyan műveletek is, amelyek új értékkel látják el. Tudományosabban fogalmazva nem egyelemű az *állapothalmaza*; a változó mindig az állapothalmazbeli valamelyik állapotban van (éppen ez a változó értéke).

[5] Mi van, ha nincs víz?

[6] Ilyen és csak ilyen megfelelő!

Értékadás: az az utasítás, ami révén a pillanatnyi állapotból egy másikba (a meghatározottba) ke rül át a változó. (Nyilvánvaló, hogy konstans adatra nem vonatkozhat értékadás, a egy, kezdőértéket meghatározón kívül.)

Típus: olyan „megállapodás" (absztrakt kategória), amely adatok egy lehetséges körét jelöli ki a által, hogy rögzíti azok *állapothalmazát* és az elvégezhető *műveletek* arzenálját.

Vizsgáljuk meg e szempontból a szifont! Lehetséges állapotkomponensei:

- működőképesség szempontjából: jó vagy rossz (a „rossz"-ságot célszerű differenciálni[7]: össze szereletlen vagy hibás),
- használhatóság szempontjából: vízzel-, patronnal teltség.

Ilyen állapotdefiniálás mellett az alábbi, tipikus állapotokban található egy szifon:

- az elérendő állapot:
  (jó, (H_2O,CO_2)),
- egy „reményteljes" kezdőállapot:
  (jó, (,)),
- egy közbülső állapot (amikor szétszedés alatt áll):
  (összeszereletlen, (,üres)).

Érdemes e vizsgálatot egy egyszerű típusú adatra, mondjuk a patronra is elvégezni:

- töltő anyag szempontjából: CO_2 vagy más (pl. habpatron),
- teltség szempontjából: üres vagy teli.

(A két szempontot eggyé is tehetjük –mint, ahogy tettük is–, ha az állapothalmazt az üres, a CO_2 és a más értékhármassal definiáljuk.)

Nincs más hátra, mint a fent megfogalmazott résztevékenységeket továbbiakra (esetleg már ele miekre) bontani! (Figyeljük a szifonkomponens- és eljárás-finomításban a haladás párhuzamos ságát! Kissé formálisabban -egy-egy értékadással- igyekszünk megmutatni a tevékenységek álta elvégzett transzformációt.)

```
Csavard le a szifonról a fejet:
   test:=szifon-fej
Vége.[8]

Vedd ki a csövet:
   tartály:=test-cső
Vége.

Szifontestet öntsd tele vízzel:
   ISMÉTELD, AMÍG nincs tele
      Önts bele vizet!
Vége.
```

```
Csavard le a fejről a patronrészt:
   szelep:=fej-patronrész
Vége.

Vedd ki a patronrészből a patront:
   kosár:=patronrész-patron
Vége.
```

De lehet másként is, ha van egy „segédkellék": a decis pohár[9].

[7] Egy későbbi programvariánsra gondolunk ezzel.

[8] Vége, de minek? Csak a résztevékenységnek!

[9] Érdemes elgondolkodni: mi van, ha csak 3 decis „teli" pohárral dolgozhatnánk? („Túlcsordulás".) Mi a lényegi kü lönbség a pohárral, ill. a közvetlenül a csapból történő föltöltés között? („Diszkrét", ill. „folytonos" értékkészlet.)


```
ifontestet öntsd tele vízzel:
Csináld 10-szer
   Önts a pohárba vizet!
   Öntsd bele a pohár vizet!
ge.
```

egjegyzés: Célszerű egységesíteni az algoritmikus szókincset! 'Ismételd...', illetve 'Csináld...' ül válasszuk ki az egyiket, és a továbbiakban csak azt használjuk erre a célra! A következő, 4. ldában véglegesítjük a nyelvünk szókincsét.

```
dd bele a csövet:
 test:=tartály+cső
ge.

avard rá a testre a fejet, hogy a szifon teljes legyen:
 szifon:=test+fej
ge.

eréld ki a patronokat:
 Végy ki a patronos dobozból egy CO2 patront!
 Tedd a helyére az üres patront
 patronrész:=kosár+CO2 patron
ge.

avard rá a szelepre a patronrészt, hogy a fej teljes legyen:
 fej:=szelep+patronrész
ge.
```

legjegyzések:

Értelmesen lehetne rövidíteni az algoritmust, ha észrevesszük, hogy a két 'Csavard rá ...' eljárás „csak" a kellékekben tér el. Ha e konkrét kellékhármas (mit, mire, mivé) helyett csak jelölnénk a szerepüket, és e jelekkel fogalmaznánk meg a tevékenységet, akkor elegendő lenne csak egyszer lerögzíteni a csavarás hogyanját! E „szerepjátszásra" fölbérelt adatszimbólumokat *formális paraméter*eknek hívják. Ott, ahol viszont a konkrét adatokra alkalmazzuk, abban a leírásban *aktuális paraméter*ként fejezi ki, hogy vele történik mind az, ami az eljárásban zajlik.

```
Csavard rá az A-ra a B-t, hogy C teljes legyen:
   C:=A+B
Vége.
```

A :egyszer test, máskor szelep
B: egyszer fej, máskor patronrész
C: egyszer szifon, máskor fej

Szárnypróbálgatásként fogalmazzuk meg ugyanezt a feladatot 2 literes szódás szifon esetére!

4. Jegyárusító automata

Utolsó példánkkal egy vasútállomáson használható jegyárusító automata működését írjuk le. A korábbiakban már megállapítottuk, hogy a túlságosan nagy „szabadság" az algoritmusírásban felesleges bonyodalmakat okoz, célszerű lenne az algoritmusok leírására használatos nyelvünket egységesíteni, rögzíteni, s a leírás mögötti tartalomban megegyezni. Ezt majd a következő fejezetekben tesszük meg, de már ebben a példában is használjuk ezen -általunk javasolt- „szabványos" nyelvet.

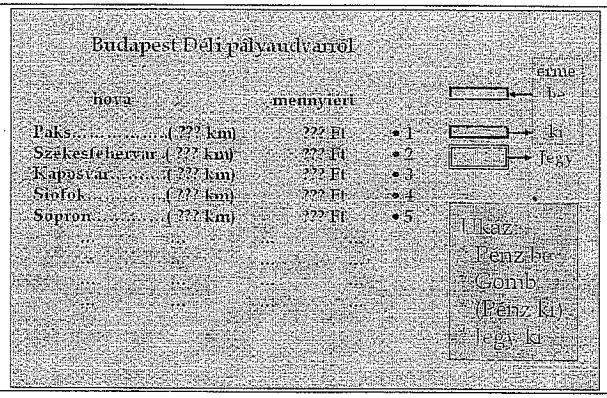
A szabványos algoritmikus elemeket **vastag** betűkkel jelöljük a továbbiakban. Ez a kiemelés

hasznunkra lesz, mivel első pillantásra is áttekinthetővé teszi a tevékenység(nek legalább a) sz
kezetét.

Elképzeljük a megoldást: az automatán –általános szokás szerint– látszanak azok az állomás
amelyekre jegyet képes adni. Olvasható az is, hogy mennyibe kerülnek az egyes jegyek. Példá
az alábbi képet társíthatjuk az automatához. (L. az 1. ábrát!)

Kezdjük ismét a kellékekkel! Szükség lesz:

- bedobott pénzre,
- visszajáró pénzre,
- jegyre,
- állomás(sorszám)ra.



1. ábra. Jegyárusító automata.

Az algoritmus legfelső szintjét nevezzük programnak. A megoldás tehát első közelítésben:

```
Program:
   [ visszajáró, bedobott, állomás, amelyek nem negatív egészek
     jegy : (távolság, ár), s ezek pozitív egészek ]
   Várd az állomást                              [állomás:=állomássorszám
   Számold meg a bedobott pénzt                  [bedobott:=bedobott pénz
   Számold ki a visszajárót                   [visszajáró:=visszajáró pénz
   Ha visszajáró<0 akkor                            [kevés a bedobott pénz
     Add vissza a bedobott pénzt                      [Ki : bedobott pénz
             különben
     Adj vissza                                            [Ki : visszajáró
     Add ki a jegyet                                [jegy:=(távolság, ár)
   Elágazás vége
Program vége.
```

Elágazásoknak algoritmusleíró nyelvünkben nemcsak az elejét, hanem a végét is jelezni fogju
kivéve, ha e nélkül is egyértelmű. A bekezdések mindig utalnak a struktúrára, azaz utasítás
együvé tartozó csoportját jelzik. Bekezdés vége a struktúra végét is jelöli.

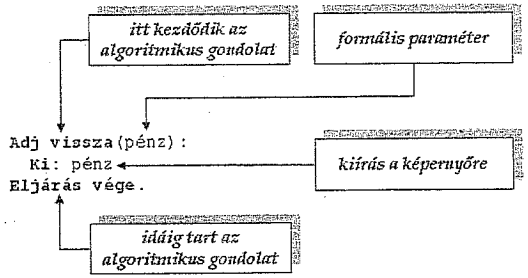
Megjegyzések:

1. A visszajáró pénzről juthat eszünkbe, hogy egységesíteni lehetne a pénz visszaadást, ha

'Adj vissza' tevékenység mindig a visszajáróban találja a neki megfelelő mennyiséget. Így az elágazás *akkor*-ága (azaz az **akkor** kulcs-szóval bevezetett struktúra) a következőképpen módosul:

```
visszajáró:=bedobott
Adj vissza       [Ki: visszajáró]
```

2. Ennél „igazibb" megoldás, ha –a már korábban körvonalazott– paraméterezett eljárással oldjuk meg a visszaadást; az alábbi módon:



Jegyezzük meg, hogy a *résztevékenység*eket algoritmizáló programrészeket *eljárások*nak nevezzük! Az eljárás mindig azzal a névvel kezdődik, amellyel korábban hivatkoztunk rá. Vagyis erre utalunk a programunk fent tisztázott szintjén két helyen is:

```
...
Ha visszajáró<0 akkor
  Add vissza a bedobott pénzt   ← Adj vissza(bedobott)
          Különben
  Adj vissza                    ← Adj vissza(visszajárót)
  Add ki a jegyet
Elágazás vége
...
```

Folytassuk a részletezést az egyes résztevékenységekkel! A 'Várd az állomást'-ban megjelenik a felhasználóval való kommunikáció másik utasítása (a **Ki:** párja), a beolvasás.

```
Várd az állomást:
  [ max : ennyi állomás van;
    táblázat(max,2) : ( (1.táv, 1.ár), (2.táv, 2. ár), ... ) ]
  Írj tájékoztatót                                    [ki : állomás-táblázat]
  Be: állomás                                                [1≤állomás≤max]
Eljárás vége.
```

Mint minden, ami fix, rögzített, ami eleve „kiírattatott" az automata dobozára, úgy az algoritmusbeli *táblázat* is „eleve megvan", s nem fog a használat által változni. Az ilyen adatféleséget korábban *konstans*nak neveztük. Itt persze az eddig megszokott „triviális" konstansoktól (5, 10, ..., 3.1415, ...) több szempontból is eltérő valamivel találkozunk. Első újdonság: általunk adott neve van (*táblázat*), második: „bonyolult" szerkezetű. Egy fontos fogalompárral érdemes megismerkednünk ezzel kapcsolatban: adataink lehetnek *struktúrálatlan*ok (mint a *max*, vagy a *bedobott* stb.), s lehet szerkezetük (amilyen a *táblázat*).

A pénz számlálására alapesetben elég egy beolvasó utasítást írni.

```
Számláld meg a bedobott pénzt:
  Be: bedobott
Eljárás vége.
```

Mivel az a megoldás nagyon elüt a valóságtól, ezért ezt a későbbiekben „részletesebbel", e reálisabbal helyettesítjük:[10]

```
Számláld meg a bedobott pénzt:
  [ pénz : pozitív egész értékű segédváltozó ]
  bedobott:=0
Ciklus amíg van még pénz[11]
  Be: pénz                                          [megengedett érm
  bedobott:=bedobott+pénz
  Ciklus vége
Eljárás vége.
```

Hogy mi a megengedett, attól először –legalábbis a kódolásnál– célszerű eltekinteni. Kiemelen az eljárásban a **Van még pénz** függvényszerű beépítése, amelynek a kérdésre adott válasz logi értékét kell előállítania. (Ennek a tevékenységnek –ti. összegzés– visszatérő voltáról esik m szó: *sorozatszámítás tétel* első előbukkanásának tekinthető.)

```
Számold ki a visszajárót:
  visszajáró:=visszajáró-táblázat(állomás,2)
Eljárás vége.
```

A pénz visszaadása első esetben szintén egyetlen kiírásból állhat.

```
Adj vissza:
  Ki: visszajáró
Eljárás vége.
```

Ezt az eljárást átalakíthatjuk olyanná, amely megmondja, hogy melyik pénzérméből mennyit k visszaadni.

```
Adj vissza:
  [ l : pozitív egész
    érmeszám : megengedett érmék száma
    érme(érmeszám) : a megengedett érmék növekvő értékű sorozata ]
  l:=érmeszám
  Ciklus amíg visszajáró>0
    Ciklus amíg visszajáró<érme(l)
      l:=l-1
    Ciklus vége
    Ki: érme(l)
    visszajáró:=visszajáró-érme(l)
  Ciklus vége
Eljárás vége.
```

A pénzvisszaadáshoz nem az egyetlen lehetséges elképzelést valósítottuk meg, hogy ti. a l kevesebb számú pénzben történjék. Ha a beolvasásnál ellenőriztük az érmék helyességét, akk már abban az eljárásban kellett az *érme* vektort deklarálnunk. (És íme egy újabb visszaköszö algoritmusfajta prototípusa: a *kiválasztás* tétel első előfordulása.)

[10] Vegyük észre, hogy csak azért tudjuk ilyen „lezserül" megtenni, mert a felhasználás helyén kellő eleganciá annyit mondtunk, hogy 'Számláld meg ...'. Ezzel a kézzel fogható haszonnal jár, ha nem akarunk elsőre nag markolni, és résztevékenységekben gondolkodni. Erre a jellegzetes hozzáállásra még vissza fogunk térni.

[11] Állapodjunk meg abban, hogy e ciklus mindaddig ismételteti a *ciklusmag*ot (a feltétel és a **Ciklus vége** közi utasításokat), amíg az **amíg** utáni *feltétel* igaz értékű; az ismétlés megszakad, amint hamissá válik a feltétel.


```
dd ki a jegyet:
  jegy:=táblázat(állomás,)  [a táblázat egy teljes sora: az állomásadik]
i: jegy
ljárás vége.
```

anulságként hívhatjuk föl a figyelmet arra, hogy „bonyolult" szerkezetű adatok egészére (*jegy*), agy annak még nem feltétlenül elemi részére (*táblázat* egy sorára) is lehet (illetve így lehet) ivatkozni.

```
an még pénz:
  [ van : szöveg segédváltozó ]
  Be: van   [van= „i" vagy „n"]
  Van még pénz:=(van="i")
üggvény vége.
```

A *függvény-eljárás*nak egyetlen érdekessége van: hogyan kell a függvény értékének megadásáról ondoskodni.

Az **Írj tájékoztatót'** eljárás tisztázását, mint algoritmikusan nem problematikust, a *kódolás*ra agyjuk (azaz csak a konkrét programozási nyelven megfogalmazandónak vesszük).

5. Metróvezető algoritmusa

A *párhuzamosság*gal a továbbiakban ritkán fogunk foglalkozni. Hogy ne maradjon ki teljesen ennek jellegzetes „problémavilága", az alábbiakban egy tanulságos példán keresztül vizsgáló-dunk. Nézzük meg, mit kell tennie egy metróvezetőnek a metróállomáson!

```
Metróvezető:
  Nyisd ki az ajtókat ÉS KÖZBEN mondd be az állomás nevét!
  ISMÉTELD: Nézd a peront AMÍG van még beszálló!
  Mondd, hogy az ajtók záródnak!
  Csukd be az ajtókat!
  Indulj ÉS KÖZBEN mondd be a következő állomás nevét!
Vége.
```

Ebben a példában tehát kimondottan párhuzamosan elvégzendő tevékenységeket adtunk meg, amelyeket az **ÉS KÖZBEN** szópár köt össze.

Párhuzamosan zajló „események" algoritmizálása közben (röviden: „párhuzamos környezetben") szükség lehet egyes folyamatok *várakoztatás*ára a részfolyamatok szinkronizálódása érdekében. Ezt láthatjuk a következő algoritmusvariációban.

```
Metróvezető:
  Nyisd ki az ajtókat ÉS KÖZBEN mondd be az állomás nevét!
  ISMÉTELD: Nézd a peront AMÍG van még beszálló és a várakozási idő≤1 perc!
  Mondd, hogy az ajtók záródnak!
  Csukd be az ajtókat!
  Indulj ÉS KÖZBEN mondd be a következő állomás nevét!
Vége.
```

Nemcsak elemi tevékenységek végezhetők párhuzamosan, hanem tetszőleges összetett (több utasítást tartalmazó) programstruktúrák is:

```
Metróvezető:
  Nyisd ki az ajtókat ÉS KÖZBEN mondd be az állomás nevét!
  ISMÉTELD: Nézd a peront ÉS KÖZBEN
            HA a várakozási idő>1/2 perc AKKOR Mondd, hogy igyekezzenek!
  AMÍG van még beszálló és a várakozási idő≤1 perc!
  Mondd, hogy az ajtók záródnak!
```

```
   Csukd be az ajtókat!
   Indulj ÉS KÖZBEN mondd be a következő állomás nevét!
Vége.
```


II. A programkészítés folyamata

A bevezető ismeretek után pontosíthatjuk a programkészítés folyamatát, annak egyes lépéseit. Talán most már senkit sem fog meglepni, hogy ez a szorosan vett programozási nyelven történő ún. *kódolás*on túl jónéhány más tevékenységet is tartalmazni fog.

Az első teendő a feladat pontos meghatározása, a **specifikáció**. Ez a feladat szöveges és formalizált, matematikai leírásán (a specifikáció ún. *szűkebb értelmezés*én) túl tartalmazza a megoldással szemben támasztott követelményeket, környezeti igényeket is (ami a specifikáció ún. *tágabb értelmezése*).

A specifikáció alapján meg lehet **tervezni** a programot, elkészülhet a megoldás **algoritmusa** és az algoritmus által használt **adatok leírása**. Az algoritmus és az adatszerkezet finomítása egymással párhuzamosan halad, egészen addig a szintig, amelyet a programozó ismeretei alapján már könnyen, hibamentesen képes kódolni. Gyakran előfordul, hogy a tervezés során derül fény a specifikáció hiányosságaira, így itt visszalépésekre számíthatunk.

Az algoritmusírás után következhet a **kódolás**. Ha a feladat kitűzője nem rögzítette, akkor ez előtt választhatunk a megoldáshoz programozási nyelvet. A kódolás eredménye a **programozási nyelven leírt program**.

A program első változatban általában sohasem hibátlan, a helyességéről csak akkor beszélhetünk, ha meggyőződtünk róla. A helyesség vizsgálatának egyik lehetséges módszere a **tesztelés**. Ennek során próbaadatokkal próbáljuk ki a programot, s az ezekre adott eredményből következtetünk a helyességre. (Ne legyenek illúzióink afelöl, hogy teszteléssel eldönthető egy program helyessége. Hisz hogy valójában helyes-e a program –sajnos– nem következik abból, hogy nem találtunk hibát.)

Ha a tesztelés során hibajelenséggel találkozunk, akkor következhet a **hibakeresés**, a hibajelenséget okozó utasítás megtalálása, majd pedig a **hibajavítás.** A hiba kijavítása több fázisba is visszanyúlhat. Elképzelhető, hogy kódolási hibát kell javítanunk, de az is lehet, hogy a hibát már a tervezésnél követtük el. Javítás után újra tesztelni kell, hiszen –legyünk őszinték magunkhoz!– nem kizárt, hogy hibásan javítunk, illetőleg –enyhe optimizmussal állítjuk:– a javítás újabb hibákat fed fel, ...

E folyamat végeredménye a **helyes program**. Ezzel azonban még korántsem fejeződik be a programkészítés. Most következnek a minőségi követelmények. Egyrészt a **hatékonyságot** kell vizsgálnunk (végrehajtási idő, helyfoglalás), másrészt a kényelmes **használhatóságot**. Itt újra visszaléphetünk a kódolási, illetve a tervezési fázisba is. Ezzel elérkeztünk a **jó program**hoz.

Egyetlen program –általánosabban fogalmazva: termék– sem **használható** megfelelő leírások, ún. **dokumentáció** nélkül, így még ez a teendő van hátra. Bár ezt a folyamat végén említjük, de korántsem jelenti azt, hogy időben is a végén kell elvégezni. A dokumentálást már a feladatmeghatározásnál el kell kezdeni, s folyamatosan, a befejezésig kell készíteni. (Az előbbi mondatban szereplő 'kell' szót senki se tekintse valami kellemetlen, kívülről ránk erőszakolt kényszernek, hisz nekünk, magunknak is támaszt jelenthet a dokumentáció „torzója" abban, hogy visszavisszanyúljunk korábbi döntéseinkhez, megállapodásokhoz, és nem kell rettegnünk emlékezetünk korlátai miatt.

„Hosszú életű", professzionális programoknál még ezután kezdődik egy igen nagy fontosságú

munka, a **karbantartás**. (Erről részletesebben azért nem szólunk, mert könyvecskénknek csak kezdők bevezetése volt célja, tehát ez most még nem lesz igazán érdekes számunkra.)

Programkészítési folyamat lépése	Mire keresi a választ	Mi a végterméke?	
Feladatmegfogalmazás	*Miből? Mit? Mi a kapcsolat?*	Specifikáció	A PROGRAM
Tervezés	*Hogyan ábrázoljam? Hogyan végezzem el?*	Adatleírás + algoritmus	
Kódolás	*A számítógép hogyan ábrázolja, és hogyan végezze el?*	Kód	
Tesztelés	*Helyes-e?*	Tesztesetek	A HELYES PROGRAM
Hibakeresés	*Hol a hiba?*	Hibahely-lista	
Hibajavítás	*Mi a hiba oka, hogyan küszöbölhető ki?*	Hiba-ok és javítási javaslat	
Hatékonysági tesztelés	*Hogyan függ a paraméterektől az időbeli működése és a helyfoglalása? Kielégítő-e?*	Hatékonysági tesztesetek	A JÓ PROGRAM
Rossz hatékonyságú pontok keresése	*Mely kódrész felelős a nem várt idő- / helyfelhasználásért?*	Problematikus helyek listája	
Hatékonyságnövelés	*Hogyan gyorsítható, hogyan tehető kisebb helyigényűvé?*	Probléma-ok és módosítási javaslat	
Dokumentálás	*Hogyan telepíthető, használható, hangolható?*	Dokumentáció	A PROGRAM MINT TERMÉK

1. ábra. A Programkészítés folyamata

III. Programozási fogalmak, specifikáció

programkészítés menetének első lépése a feladat meghatározása. Milyen is legyen ez, mit várnk el tőle? Nézzünk meg néhány –jónak tűnő– követelményt egyelőre címszavakban![1] A specikáció legyen:

helyes, egyértelmű, pontos, teljes;

rövid, tömör, ami legegyszerűbben úgy érhető el, hogy: ismert formalizmusokra építjük;

szemléletes, érthető (amit időnként nehezít a formalizáltság)!

Miből álljon a fenti feltételeknek megfelelő specifikáció, s milyen eszközökkel írhatjuk le? A érdés megválaszolásához tekintsünk három példát! (Ezeket fogjuk végig használni e fejezetben, a sorszámukkal hivatkozunk rájuk.)

. feladat (1. változat):

Adjuk meg egy másodfokú egyenlet megoldását!

. feladat (1. változat):

Adjuk meg N ember közül a legmagasabbat!

. feladat (1. változat):

Adjuk meg N ember közül a második legmagasabbat!

specifikáció első közelítésben lehetne a **feladatok szövege**. Ez azonban több problémát vethet el:

mi alapján adjuk meg a megoldást?

mit is kell pontosan megadni?

1. probléma:

A másodfokú egyenletet többféle alakban is megadhatjuk, például:
$ax^2+bx+c=0$ vagy $(x-a)(x-b)=0$.
Melyiket kell a feladat megoldásában használnunk?

2. probléma:

A legmagasabb ember megadása mit jelent? Adjuk meg a sorszámát, vagy a nevét, vagy a személyi számát, vagy a magasságát, esetleg ezek közül mindegyiket?

Megjegyzés: A 3. feladat problémái sokáig azonosak lesznek a 2. feladatéval, emiatt csak akkor foglalkozunk vele külön is, amikor újdonsággal találkozunk.

Tanulságként megállapíthatjuk, hogy a **specifikáció**nak tartalmaznia kell a **bemenő** és a **kimenő adatok leírását**. Pontosítsuk ezek alapján a fenti feladatokat.

1. feladat (2. változat):

Adjuk meg egy másodfokú egyenlet megoldását! Az egyenlet $ax^2+bx+c=0$ formában adott.

Bemenet: a,b,c – *együtthatók.*

Kimenet: x_1,x_2 – az *egyenlet megoldásai.*

[1] A továbbiakban a specifikáció szűkebb értelmezéséről lesz szó.

2. feladat (2. változat):

Adjuk meg N ember közül a legmagasabbat!

Bemenet: N – az emberek *száma*,

A – a *magasság*ukat tartalmazó *sorozat*.

Kimenet: MAX – a legmagasabb ember *sorszáma*.

3. feladat (2. változat):

Adjuk meg N ember közül a második legmagasabbat!

Bemenet: N – az emberek *száma*,

A – a magasságukat tartalmazó *sorozat*.

Kimenet: MAX2 – a második legmagasabb ember sorszáma,

MAG2 – a második legmagasabb ember magassága.

Egy olyan problémával folytatjuk, amely a fenti három példában a feladatszövegek alapján n nem mindig egyértelműen megoldható, mégis érdemes kitérni rá. Tudjuk-e, hogy a bemenő, ve a kimenő változók milyen értéket vehetnek fel?

B/1. probléma:

Megengedjük-e az a=0 esetet? Ha azt megengedjük, akkor lehet-e ebben az esetben És ilyenkor c milyen értékeket vehet fel?

B/2. probléma:

Az emberek magasságát milyen mértékegységben kell megadni? Az eredményül kapot szám milyen érték lehet: 1-től sorszámozunk vagy 0-tól?

Megállapíthatjuk tehát, hogy a specifikációban a bemeneti és a kimeneti változók **értékhal** is meg kell adnunk.

1. feladat (3. változat):

Adjuk meg egy másodfokú egyenlet megoldását! Az egyenlet $ax^2+bx+c=0$ formában a

Bemenet: a,b,c – együtthatók, tetszőleges *valós számok*.

Kimenet: x_1,x_2 – az egyenlet megoldásai, tetszőleges *valós számok*.

2. feladat (3. változat):

Adjuk meg N ember közül a legmagasabbat!

Bemenet: N – az emberek száma, *természetes szám*,

A – a magasságukat tartalmazó *sorozat*, *egész számok*, amelyek a magas centiméterben tartalmazzák (a sorozatot 1-től N-ig indexeljük).

Kimenet: MAX – a legmagasabb ember sorszáma, *1 és N közötti természetes szám*.

3. feladat (3. változat):

Adjuk meg N ember közül a második legmagasabbat!

Bemenet: N – az emberek száma, *természetes szám*,

A – a magasságukat tartalmazó *sorozat*, *egész számok*, amelyek a magasságo *centiméterben* tartalmazzák (a sorozatot 1-től N-ig indexeljük).

Kimenet: MAX2 – a második legmagasabb ember sorszáma, *1 és N közötti természetes szám*,

MAG2 – a második legmagasabb ember magassága, *valós szám*, amely a magasságot *méterben* adja meg.

st már a bemenő és a kimenő változók értékhalmazát pontosan meghatároztuk, csupán az a bléma, hogy a feladatban használt fogalmak, s az eredmények kiszámítási szabályát nem deáltuk.

. probléma:

Mit jelent az, hogy megoldás? Van-e tetszőleges a-ra, b-re és c-re megoldás? Ugyanúgy kell-e ezeket kiszámolni?

. probléma:

Mit jelent az, hogy második legnagyobb? A legnagyobbat kizárva, a maradék közül a legnagyobb, vagy pedig a legnagyobbnál alacsonyabbak közül a legnagyobb? (Ha a legmagasabbal egyforma magasságú ember létezik, akkor e két megfogalmazás nem ugyanazt jelenti!) Bár naiv kérdésnek tűnik, de azért feltesszük: hogyan kell a centimétert méterbe átszámítani? (Nem hangzik ennyire magától értetődőnek –legalábbis mi számunkra, nem „tőzsgyökeres" angolok számára–, ha pl. a kiinduló adatok hüvelykben lennének megadva és az eredményt lábban kellene előállítani.)

pecifikációnak tehát tartalmaznia kell a **feladatban használt fogalmak definícióját**, valamint **eredmény kiszámítási szabályát**. Itt lehetne megadni a bemenő adatokra vonatkozó összegéseket is. A bemenő, illetve a kimenő adatokra kirótt feltételeket nevezzük **előfeltétel**nek, tve **utófeltétel**nek. Az előfeltétel nagyon sokszor egy azonosan igaz állítás, azaz a bemenő tok értékhalmazát semmilyen „külön" feltétellel nem szorítjuk meg.

eladat (4. változat):

Adjuk meg egy másodfokú egyenlet megoldását! Az egyenlet $ax^2+bx+c=0$ formában adott.

Bemenet: a,b,c – együtthatók, tetszőleges *valós számok*.

Kimenet: sz – a megoldás „milyenségére utaló" *szöveg*

x_1,x_2 – az egyenlet megoldásai, tetszőleges *valós számok*,

Előfeltétel: *a=0 és b=0, akkor c=0.*

Utófeltétel: *ha a=0 és b=0 és c=0, akkor*

sz="nulladfokú az egyenlet, végtelen sok megoldása van"

ha a=0, akkor

x_1=-c/b, sz="elsőfokú az egyenlet, egy megoldása van"

*ha a≠0 és b^2<4*a*c, akkor*

sz="nincs valós megoldás"

*ha a≠0 és b^2=4*a*c, akkor*

*x_1=-b/(2*a), sz="egy kétszeres valós megoldás van"*

*ha a≠0 és b^2>4*a*c, akkor*

$x_1=\frac{-b+\sqrt{b^2-4*a*c}}{2*a}$, $x_2=\frac{-b-\sqrt{b^2-4*a*c}}{2*a}$

sz="két különböző valós megoldás van".

gjegyzés: itt egy igen ritka utófeltételt látunk, amelyben *explicite* fejeződik ki a kimenet és a menet közötti kapcsolat.

2. feladat (4. változat):

Adjuk meg N ember közül a legmagasabbat!

Bemenet: N – az emberek száma, *természetes szám*,

A – a magasságukat tartalmazó *sorozat*, *egész szám*ok, amelyek a magasságo *centiméterben* tartalmazzák (a sorozatot 1-től N-ig indexeljük).

Kimenet: MAX – a legmagasabb ember sorszáma, *1 és N közötti természetes szám.*

Előfeltétel: A_i*-k pozitívak.*

Utófeltétel: *MAX olyan 1 és N közötti szám, amelyre* A_{MAX} *nagyobb vagy egyenlő, m sorozat bármely eleme (az 1. és az N. között).*

3. feladat (4. változat):

Adjuk meg N ember közül a második legmagasabbat!

Bemenet: N – az emberek száma, *természetes szám*,

A – a magasságukat tartalmazó *sorozat*, *egész szám*ok, amelyek a magasságo centiméterben tartalmazzák (a sorozatot 1-től N-ig indexeljük).

Kimenet: MAX2 – a második legmagasabb ember sorszáma, *1 és N közötti természetes szám.*

MAG2 – a második legmagasabb ember magassága, *valós szám*, amely a magasságot *méterben* adja meg (*1 méter=100 centiméter*).

Előfeltétel: *N legalább 2 és van a legmagasabbtól különböző magasságú az A soroza és* A_i*-k pozitívak.*

Utófeltétel: *MAX2 olyan 1 és N közötti szám, amelyre* A_{MAX2} *kisebb, mint a sorozat le gyobb eleme, a többinél pedig nagyobb vagy egyenlő (a második legmaga ember a legmagasabbnál alacsonyabbak közül a legnagyobb), és MAG2 é egyenlő* A_{MAX2}*-vel átváltva cm-ről m-re.*

Újabb probléma merülhet fel bármelyik feladattal kapcsolatban: az eddigiek alapján a „vár lényegesen különböző –nyugodtan állíthatjuk: „banális"–, az elő- és utófeltételnek megf megoldást is tudunk készíteni.

D/1. probléma:

Az 'x_1:=0; a:=0; b:=1; c:=0; sz:="elsőfokú..."' utasítássorozat végrehajtása egy olyan potot eredményez, amely megfelel az utófeltételnek, de nyilvánvalóan nem helyes m dás *tetszőleges* a,b,c-re.

Itt persze arról a hallgatólagos feltételezésről (tehát még meg nem fogalmazott, ki nem mon van szó, hogy a *bemeneti változók értéke nem változik meg*. Ez sajnos, mint láttuk, nem felt nül igaz. A probléma megoldására kétféle utat követhetünk (a későbbiekben mindkettőt a mazni fogjuk):

- az utófeltételbe automatikusan beleértjük, hogy „*és a bemeneti változók értéke nem vál meg*", s külön kiemeljük, ha mégsem így van;
- az elő- és az utófeltételt a program paramétereire fogalmazzuk meg, amelyeket formá megkülönböztetünk a program változóitól, s emiatt nem a paraméterek fognak változni, ha a programbeli változók (ebben az esetben természetesen az elő- és az utófeltételben meg fogalmazni a paraméterek és a megfelelő programbeli változók értékének azonosságát).

második megoldásból az következik, hogy **meg kell különböztetnünk egymástól a feladat és program elő-, illetve utófeltételét!** Ez hosszadalmasabbá –bár precízebbé– teszi a feladat meggalmazását, emiatt ritkábban fogjuk alkalmazni.

lőfordulhat, hogy a feladat megfogalmazása alapján nem lehet egyértelműen meghatározni az edményt, ugyanis több, az utófeltételnek megfelelő megoldás is létezik.

/2. probléma:

Ha több, a legmagasabbal azonos magasságú ember van, akkor melyiket kell megadni eredményként?

z a jelenség a **feladat** ún. **nemdeterminisztikussága**. Az első fejezetben beszéltünk nemdeterinisztikus algoritmusokról is, de rögtön megállapítottuk, hogy ilyenekkel egyelőre nem fogulkozunk. Ehhez a nemdeterminisztikus feladathoz tehát determinisztikus programot kell írnunk, minek az utófeltétele már nem engedheti meg a nem egyértelműséget, a nemdeterminisztikusságot. E probléma miatt tehát *mindenképpen* **meg kell különböztetnünk egymástól a feladat és program elő-, illetve utófeltételét!**

feladat (5. változat):

Adjuk meg N ember közül a legmagasabbat!

Bemenet: N – az emberek száma, *természetes szám*,

A – a magasságukat tartalmazó *sorozat*, *egész szám*ok, amelyek a magasságot *centiméterben* tartalmazzák (a sorozatot 1-től N-ig indexeljük),

Kimenet: MAX – a legmagasabb ember sorszáma, *1 és N közötti természetes szám.*

Előfeltétel: A_i*-k pozitívak.*

Utófeltétel: *MAX olyan 1 és N közötti szám, amelyre* A_{MAX} *nagyobb vagy egyenlő, mint a sorozat bármely eleme (az 1. és az N. között).*

Program utófeltétel: *MAX olyan 1 és N közötti szám, amelyre* A_{MAX} *nagyobb vagy egyenlő, mint a sorozat bármely eleme (az 1. és az N. között), és előtte nincs vele egyenlő.*

Megállapíthatjuk ebből, hogy a **program utófeltétele lehet szigorúbb, mint a feladaté**, emellett **z előfeltétele pedig lehet gyengébb.**

Visszatekintve a specifikáció eddig „bejárt pályájára" egy szemléletes modellje körvonalazódik a eladatmegoldásnak. Nevezetesen: nyugodtan mondhatjuk azt, hogy a feladatot megoldó program gy olyan automatát határoz meg, amelynek pillanatnyi állapota a feladat paraméterei (a program áltozói) által „kifeszített" halmaz[2] egy eleme. Ezt a halmazt nevezzük a program **állapotteré**ek. Amikor megfogalmazzuk az előfeltételt, akkor tulajdonképpen kihasítjuk ebből az állapotérből azt a részt (azt az altért), amelyből indítva elvárhatjuk az automatánktól (amit a megoldó rogram vezérel), hogy a helyes eredményt előállítja egy végállapotában. A végállapotot jelöltük i az utófeltétellel.

Ezt a modellt elfogadva adódik még egy további megoldásra váró kérdés. Akkor ugyanis, amikor programot írjuk lépten-nyomon a részeredmények tárolására újabb és újabb változókat vezetünk e. Fölvetődik a kérdés: hogyan egyeztethető össze az imént elképzelt modellel? A válasz egy-

[2] Más szóval: e halmaz annyi dimenziós, ahány paraméterváltozója van a programnak; minden dimenzió egyik változó értékhalmaza. Tehát egy konkrét időpillanatban e „gép" állapota: a változóinak abban a pillanatban érvényes értékeinek együttese.

szerű: minden egyes újabb változó egy újabb dimenziót illeszt az eddig létrejött állapottér
Tehát a programozás folyamata –leegyszerűsítve a dolgot– nem áll másból, mint annak po
sításából, hogy hogyan is nézzen ki a megoldó automata állapottere (és persze: hogyan kel
egyik állapotból a másik állapotba jutnia). A feladatban szereplő paraméterek meghatár
„embrionális" állapotteret hívhatjuk paramétertérnek, ami csak altere a program valódi álla
terének. Ez is azt sugallja, hogy az feladat előfeltétele gyengébb (azaz az általa kijelölt álla
halmaz bővebb) lehet, mint a program előfeltétele. Például

F/1. probléma:

A másodfokú egyenlet kiszámításánál érdemes a diszkriminánst (b^2-4ac kifejezés) ért
csak egyszer kiszámítani, s az összes helyen, ahol ő szerepel, egy segédváltozót helye
síteni. Ez azt jelenti, hogy ezzel a segédváltozóval bővítjük az állapotteret, s figyelem
véve ezt az új „dimenziót" átalakítjuk az „állapot-átmeneti szabályokat" (a kiszámítás
sításait) úgy, hogy

1. legelőször is kapjon értelmes értéket ez a komponens is ('D:=b^2-4ac' értékadás bevez
tésével),
2. azokat a számításokat, ahol a diszkrimináns előfordult, módosítjuk azzal, hogy számí
helyett erre a komponensre (D-dimenzióra) történjék a hivatkozás.

Látható, hogy a szemléletes szöveges leírás a pontosság érdekében nagyon hosszúvá, akár
tekinthetetlenné is válhat. Szükségünk lenne egy eszközre, amellyel mindezt sokkal rövidebbe
leírhatnánk, persze az egyértelműség megtartása mellett. Ezzel a *Módszeres programozás* soro
következő tagjában (*Programozási tételek*) fogunk foglalkozni.

Foglaljuk most össze, hogy melyek a **specifikáció részei!** Ezek az eddigiek, valamint a prog
mra vonatkozó további megkötések lesznek.

1. A feladat specifikálása:

- a feladat szövege,
- a bemenő és a kimenő adatok elnevezése, értékhalmazának leírása,
- a feladat szövegében használt fogalmak definíciói (a fogalmak fölhasználásával),
- a bemenő adatokra felírt előfeltétel (a fogalmak fölhasználásával),
- a kimenő adatokra felírt utófeltétel.

2. A program specifikálása:

- a bemenő és a kimenő adatok elnevezése, értékhalmazának leírása,
- (a feladat elő-, illetve utófeltételétől esetleg különböző) program elő- és utófeltétel,
- a feladat megfogalmazásában használt fogalmak definíciói,

Absztrakt specifikáció

- a program környezetének leírása (számítógép, memória- és periféria-igény, programozási nyelv, s annak esetleges változata, szükséges file-ok stb.)
- a programmal szembeni egyéb követelmények (minőség, hatékonyság, hordozhatóság stb.).

Technikai specifikáció

A technikai specifikáció nélküli leírást a program *szűkebb* specifikációjának nevezik.

Egy utolsó gondolattal zárjuk a specifikációval kapcsolatos kérdéskört. A specifikáció precíz

ítése azzal jár, hogy komolyan belegondolunk a feladatba. Ez önmagában is jó, de ha ehhez
átehetjük azt is, hogy a specifikáció által már a megoldó algoritmus szerkezetéről is sokat
ndtunk, az külön öröm. Nézzük meg ezt, hogy is lehetséges ez!

ecifikáció, mint láttuk egy olyan leképezést határoz meg, amely a bemenő értékekhez hozzá-
leli a jó eredmény értékét. (Itt tehát azt az automatát, amelyhez a programot megírjuk, egy
pezést megvalósító gépnek tüntetjük föl. Ez a leképzés az ún. **programfüggvény**.) A függ-
yek konstruálásánál az alábbi eszközöket szokás igénybe venni:

ét függvény **kompozíció**ja:

$p(x):=f\circ g\,(x)$ ami ugyanaz, mint $f(g(x))$

lternatívával definiálás:

$p(x) := \begin{cases} f(x), ha\ T(x) \\ g(x), különben \end{cases}$ ami hosszasabban úgy fogalmazható meg, hogy $p(x)$ legyen $f(x)$-szel egyenlő, ha $T(x)$ igaz, különben pedig $g(x)$-szel.

ekurzióval definiálás:

$p(x) := \begin{cases} f(x), & ha\ T(x) \\ g\circ p\circ h(x), & különben \end{cases}$ Vagyis feltételtől függően vagy egyszerűen $f(x)$, vagy egy rekurzívan számolható $g\circ f\circ h(x)$.

tehát a specifikációban „hagyományos" függvénymanipulációkat alkalmazunk, akkor a követ-
ő kapcsolatot fedezhetjük föl ezen manipulációk és az algoritmikus szerkezetek között:

$p := f\circ g$

g és f kiszámításának **szekvenciája**:

```
p(x)≡
  y:=g(x); p:=f(y)
```

$p := \begin{cases} f \\ g \end{cases}$

T-től függő **elágazás**:

```
p(x)≡
  Ha T(x) akkor   p:=f(x)
          különben p:=g(x)
```

$p := \begin{cases} f \\ g\circ p\circ h \end{cases}$

ciklus vagy **rekurzió**:

```
p(x)≡
  Ha T(x) akkor   p:=f(x)
          különben y:=h(x); z:=p(y); p:=g(z)
```

ezzük be ezt a fejezetet egy jövőbe mutató megjegyzéssel. Kapcsolat létesíthető a specifikáció
egyes részei és az algoritmus egyes részei között. A kapcsolatról röviden:

Specifikáció	Algoritmus
menet, Kimenet	Típus-definíció, adat-deklaráció.
őfeltétel	Beolvasás ellenőrzéssel.
ófeltétel	A finomítások (eljárások, függvények...) törzse.
galomdefiníció	Finomítások definíciói.

IV. Algoritmikus szerkezetek

ejezet célja, hogy a programozással „frissen" barátkozók számára segítséget nyújtsunk az al-itmus megírásához az első lépésektől a komolyabb tervezés megkezdéséig. Emellett a jel-zetes algoritmikus gondolkodásba is betekintést is szeretnénk adni úgy a problémákról, mint k megoldási módszereiről.

Az algoritmizálás alapjai

gyalásmódunkra jellemző lesz az, hogy –körülbelül– azokból a kérdésekből indulunk ki –és an sorrendben-, amelyeket a kezdő programkészítő kimondatlanul is fel szokott tenni magának erencsésebb esetben meg is fogalmazza és neki is szegezi a rutinosabb programozóknak. eket a *naív* kérdéseket fogalmazzuk át precízebb, kicsit célratörőbb problémafelvetésekké, elyeket minden egyes későbbi feladat megoldásánál újra elő lehet venni és az adott konkrét blémára újra lehet gondolni. Így ez mintegy *vörös fonala* lehet a kezdeti feladatmegoldási eresésnek. (A matematikai problémamegoldásra ugyanezt a hozzáállást valósította meg évti-lekkel ezelőtt Pólya György, a nemzetközi hírű magyar matematikus; az Ő tiszteletére nevez-el a *precízebb* kérdéseket *pólyai kérdéseknek*.

> ***1. pólyai kérdés:*** Elegendők-e az ismereteink a feladatról?

g kell fogalmazni, hogy pontosan mit kíván a feladat, azaz milyen adat(ok)ból (input) mi(ke)t tput), s milyen összefüggés alapján kell „kiszámolni"? Ezt a pontos, precíz megfogalmazást ják **specifikáció**nak, amellyel az előző fejezetben már foglalkoztunk, s ezért most ezzel többet m foglakozunk.

> *2. pólyai kérdés:* Lehet-e tudni valamit a programról a konkrét feladattól függetlenül?

y gondoljuk, különösebb megfontolást nem igényel az, hogy lássuk: a legtöbb program nagy-ni szerkezetében három jól körülhatárolható résznek kell lennie (legalábbis e három funkciót ll megvalósítani). Ezek a funkcionális részegységek:

- **adatbeolvasás**,
- **feldolgozás**,
- **eredménymegjelenítés**.

em mindig érdemes, sőt nem mindig lehetséges e merev részfeladatokra bontás, de mi az yszerűség kedvéért csak ilyenekkel fogunk foglalkozni. (Ha nem lenne ilyen a feladat, akkor m lehetne pusztán a 'feldolgozás' részre koncentrálni, hanem azt össze kellene vonni vala-elyik másikkal. Ekkor a programunk bonyolódik, de a feladat nem megoldhatatlan.)

mi a beolvasást és a megjelenítést illeti, többnyire –algoritmikus szempontból– elég egyszerűen intézhetők. A három részfeladat közül tehát problematikusnak –általában– a középső tekinthető, y vizsgálódásunk középpontjába is ezt helyezzük. Folytassuk tehát a kérdezést:

3. pólyai kérdés: Elegendők-e ismereteink az adatokról?

Induljunk ki az input (=bemenő) és az output (=kimenő) adatok egy *formai* tulajdonságán „sokság"-nak a csoportosításából. Ilyen alapon a feladatok az alábbi félék lehetnek:

Input	Output
A. egyből	egyet
(pl. $x \mapsto e^x$)	
B. egyből	sok(félét)
(pl. $x \mapsto \sin(x), \cos(x), \mathrm{tg}(x)$)	
C. sok, azonos jellegűből	egyet
(pl. $x_i\ (i=1..n) \mapsto \sum x_i$)	
D. sok, azonos jellegűből	sok, azonos jellegűt
($x_i\ (i=1..n) \mapsto y_j\ (j=1..m)$: $y_j>0$)	

Megjegyzés: A sok, de különböző input adatból kiinduló feladat visszavezethető az A. vagy esetekre!

Hogy mit értsünk **egy** alatt, az nem mindig egyértelmű. A fogalom megvilágítására álljon itt hány példa (mert egyszerű, de precíz definíciót adni rá igen nehéz):

- egy nap időjárását jellemző paraméterek (maximális, minimális hőmérséklet, csapadékmennyiség),
- egy (x,y) paraméterpár, amely argumentuma egy f(x,y) kétváltozós függvénynek,
- egy mátrix dimenziói (indexhatárai),
- egy kísérlet mérési adatai.

Összefoglalva: több adat együttese is alkothat egységet (sőt egységnek tekintendő!), ha egy függvény vagy valami egységként elképzelendő dolog „állapotát" határozza meg.

E meggondolás arra jó, hogy a készülő program szerkezetéről legalább „heurisztikusan" meg junk valamit. Így a program:

A. egy „közönséges" transzformáció, amely például egy képlettel kiszámolható,

B. sok „közönséges" transzformáció, amelyek például egy-egy képlettel –elvileg akár párhuzamosan is, azaz egy időben– kiszámolhatók,

C. és D. esetek közös sajátossága, hogy valamit –éppen a „sok" egy elemének feldolgozását– sokszor elvégezni (esetleg akkumulálva az addigi számítás részeredményét).

4. pólyai kérdés: Nem lehet további ismereteket szerezni az adatokról? Nincs az adatainknak „finomabb" szerkezete?

Most az adatféleségek konkrétabb szerkezete alapján próbáljunk újabb információkat szerezendő programunkról! Még választási lehetőségünk is van. Ugyanis: e célból akár az input, az output adatokat szemügyre vehetjük. Az adatféleségekről:

A. „arctalan", azaz *szerkezetnélküli*:

skaláradat,

pl. egy vizsgált populáció egyedszáma, az adott helyen mért hőmérséklet;

k, azonos valamik *sokasága*:

vektor, **mátrix**, **halmaz**, **file**,

pl. korcsoportstruktúra, függvénytáblázat, virágszínek halmaza;

bb, esetleg *különféle* valaminek az *együttese*:

rekord,

pl. egy dátum (=év+hó+nap), egy gombahatározó egyes bejegyzései;

bb, esetleg *különféle* valaminek *többértelmű* („sokarcú") *együttese*:

alternatív (változó részt tartalmazó) **rekord**,

pl. egy személy adatai (nemtől függően: ha hölgy, akkor leánykori név; ha úr, akkor katonakönyvszám)

n ennyi ismeret után remélhetjük, hogy valamivel tisztábban tehető föl az „algoritmikus" és, a kérdés a hogyanra, és persze a válasz sem várat magára?

5. pólyai kérdés: Elegendőek-e ismereteink az algoritmusról?

véletlenül piszkálgattuk eddig az adatokat, pontosabban a *feladatot meghatározó adatok*at: öris, ezek azok a valamik, amik eleve ismertek; másodszor, ha a fenti osztályba sorolást akár nputra, akár az outputra elvégezzük egyértelműen meg tudjuk mondani a feldolgozás „lo-ját", vagyis az eredményt kiszámoló *program globális szerkezet*ét.

program egy elemi transzformáció, amely egy –esetleg bonyolultan kiszámolható **függvény** isát tartalmazó– **értékadás**, vagy **eljáráshívás** –output paraméter értékének visszaadása– formában realizálható[1], illetve –a tágabb környezetet is a program részének tekintve– egy **kiírás** y egy **beolvasás** révén változtatja meg a program állapotát.

program egy **ciklus**, amelynek magja az *ismétlődő elem kiszámolását* hivatott elvégezni (és ze megszervezi az ismétlődést magát).

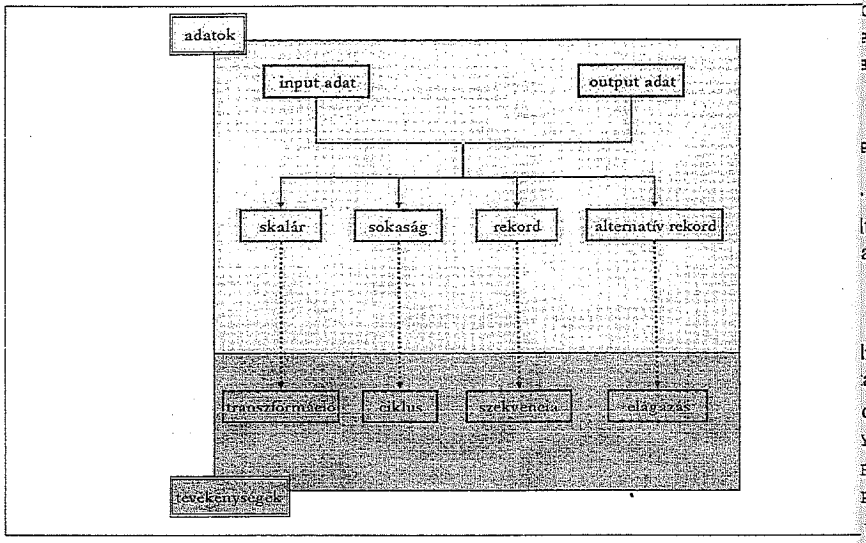
feldolgozás **szekvenciája** (egymásutánja) az egyes valamiket *feldolgozó tevékenységek*nek.

megoldás egy **elágazás** a „sokarcúság" szempontja szerint.

fontos kapcsolatot adatszerkezet és algoritmikus szerkezet között nevezik a szakirodalomban: *ktúraszerinti feldolgozás elv*ének.

1. ábra az eddigiek tömör összefoglalását adja.

[1] probléma „elodázását" megtestesítő eljárások és függvények, amelyek az általuk kijelölt részfeladatok megoldását hivatottak elvégezni, *argumentumuk*ban tartalmazzák –ún. *paraméter*ként– azokat az objektumokat, amelyekre onatkoznak; vagyis amelyekből kiindulva (*bemenő paraméterek*) számolják ki az eredményt jelentő objektumokat (*kimenő paraméterek*) értékét. Érdemes észrevenni, hogy az eljárások, függvények be- és kimenő paraméterei em mások, mint a megfelelő részfeladatok specifikációjának adat-megtestesülései.



1. ábra. Adatok, tevékenységek és kapcsolatuk.

Eszerint minden feladat akár kétféleképpen is megoldható: az egyik megoldás kiinduló pontj input szerkezete, a másodiké pedig az output. (Valójában általában kettőnél is több megoldás lálható.)

Elérkeztünk ahhoz a ponthoz, amikor az egyes algoritmikus elemeket (itt a következőkben), v mint adatszerkezeteket (a következő fejezetben) részletesen megvizsgáljunk, vagyis körvona zuk az algoritmikus „nyelvünket".

2. Algoritmikus elemek

2.1. Program

A program minden esetben utasítások sorozata lesz. Általában egy sorba egy utasítást írunk szükség esetén egy sorba több is írható (logikai összetartozásuk hangsúlyozásra), egymástól vel (vagy „:"-tal) elválasztva.

```
Program:[2]
   utasítássorok
Program vége.
```

Az egyes utasításokat a –sorok közötti, illetve soron belüli– leírásuk sorrendjében kell végre tani.

[2] Az algoritmusleíró nyelv alapszavait vastagon szedjük, így különböztetve meg őket az egyéb azonosítóktól.

da:

```
Program:
Be: OSZTANDÓ, OSZTÓ
 HÁNYADOS:=OSZTANDÓ Div OSZTÓ: MARADÉK:=OSZTANDÓ Mod OSZTÓ
 Ki: HÁNYADOS, MARADÉK
Program vége.
```

Értékadó utasítás

tozók legtöbbször *értékadó utasítás*sal kapnak értéket. Ezen utasítás az *értékadásjel* (:=) bal- alán a célként megjelölt változót, jobboldalán pedig a kiszámítandó kifejezést tartalmazza.[3]

```
azonosító := kifejezés
```

azonosító tetszőleges olyan objektum neve lehet, amelynek értéke megváltozhat, a *kifejezés* lig a matematikában és más tudományokban használt bármely *operátor*t, *függvény*t, *konstans*t almazhat.

da:

```
Y:=sin(X)
F:=e^k; G:=√(X-Y); H:=Y
P:=Q*R; S:=R^T
```

utóbbi példában mátrixok szerepelnek, a műveletek (*,T=transzponálás) értelemszerűen mát- műveletek.

Beolvasó utasítás

tszőleges adat beolvasására szolgál a felhasználó által kezelt perifériáról (billentyűzetről). vel a felhasználó nem része a programnak, ezért az általa beírt adatok és a program által várt *raméterek típusát*, *értékhalmazát* ellenőrizni kell. Az adatokat karakteresen kell megadni, s olvasáskor a megfelelő változó típusa szerinti automatikus *konverzió* történik.

```
Be: azonosítók      [feltételek]
```

szűkítő feltételek elmaradhatnak, ha a beolvasandó értékekre semmilyen előfeltevésünk nincs. együk észre ez az a feltétel, amit az előző –a specifikációról szóló– fejezetben *előfeltétel*ként öltünk meg.)

lda:

```
Be: OSZTANDÓ, OSZTÓ   [OSZTÓ≠0]
Be: X      [X(i)≥0: i=1..n]
```

. Kiíró utasítás

z előbbi utasítás ellentettje: a felhasználó által figyelt perifériára helyezi el az adatokat –a belső rázolástól függetlenül– karakteresen.

```
Ki: kifejezések      [formátum megkötés]
```

a kiírás *formátumá*ra van valamilyen speciális megkötésünk, akkor az itt szerepelhet, illetve a

léhány általánosításával programozási nyelvekben találkozhatunk. Pl. párhuzamos értékadás, többszörös értékadás stb.

legtöbb ilyet majd kódoláskor kell megfontolnunk.

2.5. Megjegyzés

Az algoritmusban elhelyezhetünk *magyarázó szövegek*et, a program állapotára vonatkozó sokat (pl. *ciklusinvariáns*t[4]), általában bármit, ami az olvashatóságot növeli. Ez a későbbi kánkat jelentősen megkönnyítheti. Formája:

```
[ tetszőleges szöveg ]
```

2.6. Elágazások

Az *elágazás*ok szerepe –mint azt a korábbiakban láttuk– feltételektől függő választás biz programrészek végrehajtása között.

A legegyszerűbb esetben dönthetünk egy utasítás(csoport) végrehajtás, illetve végrehajtásán hagyása mellett:

```
Ha logikai kifejezés akkor utasítássor
```

A *logikai kifejezés* a matematikában szokásos tetszőleges relációt, illetve logikai értékű k zést, valamint logikai műveleteket tartalmazhat. Például –a teljesség igénye nélkül– a kö zőket:

Relációk	Műveletek
$=, \neq, <, \leq, \geq, >$ $\in, \notin, \subset, \subseteq, \supseteq, \supset$	**és** ($\wedge$), **vagy** ($\vee$), **nem** ($\neg$)

Példa:

```
Ha I≤N és prím(I) akkor Ki: I
Ha I | N[5] és nem (páros(I) vagy I>N/2) akkor K:=I
```

Az utasítás második fajtájában két utasításcsoport végrehajtása között választunk a logik fejezés igazságértékétől függően. Két változata lehetséges:

```
Ha logikai kifejezés akkor  utasítássor1
                    különben utasítássor2
```

vagy

```
Ha logikai kifejezés akkor
     utasítássorok1
                    különben
     utasítássor2
Elágazás vége
```

Az első, rövidebb változatot akkor használjuk, ha ebből is egyértelmű, hogy az elágazás ág fejeződnek be. Ha az elágazás valamelyik ágán több sort is akarunk írni, akkor pedig a má változatra lesz szükségünk.

Mindkettő lényege: ha a logikai kifejezés igaz értékű, akkor az **akkor** alapszó utáni utasítá kell végrehajtani, ha pedig nem, akkor a **különben** alapszó utániakat.

[4] Azt az állítást nevezzük *ciklusinvariáns*nak, amely a ciklus minden egyes végrehajtásánál igaz marad. Egy ily lítás jellegzetesen írja le a ciklusbeli transzformációt. Ez által éppen annak helyességét lehet ezzel belátni.

[5] Értsd: 'I osztója N-nek'!

la:

```
a X<Y akkor  Ki: X,Y
    különben Ki: Y,X
a e∈H akkor  H:=H-{e}
 Ki: e," benne volt a halmazban."
    különben H:=H+{e}
 Ki: e," nem volt benne a halmazban."
lágazás vége
```

pzelhető, hogy az elágazás ágain újabb elágazás következik. Ennek gyakori esete, hogy az azás **különben**-ágán kell az újabbat elhelyezni. Ekkor tulajdonképpen az egyes ágak egyen-űak, s ez az elágazás leírásában is tükröződik.

```
Ha logikai kifejezés  akkor
                              utasítássorok1
különben ha logikai kifejezés akkor
                              utasítássorok2
                                különben
                              utasítássorokn
Elágazás végen
```

*ogikai kifejezések*et itt a felírás sorrendjében kell megvizsgálnunk, s az első igaz értékűnek felelő utasításcsoportot kell végrehajtani. Ha egyik sem teljesül, akkor az utolsó **különben**-n levő utasításokkal kell foglalkozni.

da:

```
Ha       I<0      akkor Ki: „negatív"
különben ha I=0   akkor Ki: „nulla"
különben            Ki: „pozitív"[6]
Elágazás vége
```

onyos esetekben a sokirányú elágazás feltételeinek kiértékelési sorrendjét nem akarjuk meg-ni (tetszőleges sorrendben, illetve párhuzamosan is elvégezhetők lehetnek). Ekkor egy újfajta gazást használunk:

```
Elágazás
   feltétel1   esetén utasítássorok1
   feltétel2   esetén utasítássorok2
     …
   feltételn   esetén utasítássoroksn
      egyéb  esetben utasítássoroksn+1
Elágazás végen
```

'egyéb' a **'nem** feltétel$_1$ **és nem** feltétel$_2$ **és … és nem** feltétel$_n$'-et rövidíti. az igaz feltételű elágazáságat kell végrehajtani. Ha több ilyen is van, akkor közülük tetszőleset, ha egy sincs, akkor az **egyéb esetben** alapszó utánit. Ez utóbbi ág el is maradhat, s ez ben ha egyik feltétel sem teljesül, akkor az elágazás egyik ágát sem hajtjuk végre.

ilágosabb, ha kiírjuk: **különben ha** I>0 **akkor Ki:** „pozitív"

Példa:

```
Elágazás
  I=0 esetén F:=0
  I=1 esetén F:=1
  I>1 esetén F:=Fibonacci_szám(I-1)+Fibonacci_szám(I-2)
Elágazás vége
```

Mivel nem fogunk sem párhuzamos, sem nemdeterminisztikus algoritmusokat írni, viszont lé
gesen áttekinthetőbb ez utóbbi sokirányú elágazás-szerkezet, ezért gyakran alkalmazzuk ez
első helyett, vele megegyező értelemben.

2.7. Ciklusok

Másik alapvető algoritmikus struktúránk volt a bevezető fejezetben a *ciklus*: utasítások me
métlése többször. Mint ott láttuk, többféle fajtája lehet, nézzük ezeket sorra!

```
Ciklus amíg logikai kifejezés
  utasításSorok
Ciklus vége
```

Az első lehetőség az ismétlést egy *feltétel*hez kötni: amíg a logikai kifejezés (az ún. *ciklusfelt*
igaz értékű, addig a ciklus **belsejében** szereplő utasításokat (az ún. *ciklusmag*ot) végre kell
tani. Ha a logikai kifejezés már **hamis** értékű, akkor a **Ciklus vége utáni** utasításnál kell folyt
a végrehajtást.

A felírás formájából is látszik, hogy itt a ciklusfeltételt a ciklusmag végrehajtása előtt kell
tékelni (ezt hívják *elöltesztelős ciklus*nak), s elképzelhető, hogy a ciklusmagot egyszer sem
juk végre.

Példa:

```
I:=2
Ciklus amíg I≤N és IN
  I:=I+1
Ciklus vége
```

A ciklusfeltételt nem csak a ciklusmag előtt, hanem mögötte is elhelyezhetjük. Ez lesz az ún
tultesztelős ciklus.

```
Ciklus
  utasításSorok
amíg logikai kifejezés
Ciklus vége
```

Ugyanúgy, mint az elöltesztelős ciklusnál, itt is akkor **fejezzük be** a végrehajtást, ha a logika
fejezés **hamis** értékű, s akkor kezdjük elölről a ciklusmagot, ha igaz.

Példa:

```
I:=0
Ciklus
  I:=I+1; Ki: NÉV(I)
amíg NÉV(I)≠"KISS ANNA"
Ciklus vége
```

Megjegyzés: Sok programozási nyelvben létezik e ciklusnak olyan változata, ahol nem a b
maradás, hanem a *kilépés feltételét* kell a hátultesztelős ciklusnál megfogalmazni, tehát
megfogalmazottnak épp az ellenkezőjét.

Ez a ciklus olyan, hogy a ciklusmagját egyszer mindenképpen végrehajtjuk, s csak utána viz

hogy kell-e még vagy sem.

elöltesztelős ciklusnak van egy speciális változata, amelyet külön is megvizsgálunk, először példával.

la: Össze kell adnunk egy számsorozat elemeit: az elsőt, a másodikat, ... utoljára az N.-et.

```
:=1: S:=0
iklus amíg I≤N
  S:=S+A(I): I:=I+1
iklus vége
```

→

```
S:=0
Ciklus I=1-től N-ig
  S:=S+A(I)
Ciklus vége
```

olyan elöltesztelős ciklusokat nevezzük *számlálós ciklus*nak, amelyekben egy változó adott rvallumon belüli összes értékére végre kell hajtanunk a ciklusmagot. Az elöltesztelősségből tkezik, hogy ha az intervallum üres, akkor a ciklusmagot egyszer sem hajtjuk végre.

jegyzés: Egyes programozási nyelvek, illetve azok egyes implementációi a számlálós ciklust ltesztelősként definiálják, emiatt használatával *üres intervallum* esetén óvatosan kell bán-k.

```
Ciklus cv=K-tól V-ig
  utasításSorok
Ciklus vége
```

lyen ciklusban a cv változó (az ún. *ciklusváltozó*) először felveszi a K (kezdő)értéket. Ha ez vég)értéknél kisebb, akkor végrehajtjuk a ciklusmagot, majd K értékét növeljük a [K..V] rvallum következő elemére, s újra vizsgáljuk a végrehajtás feltételét. Ha a ciklusváltozó túl-a végértéket, akkor a **Ciklus vége** utáni utasításnál kell folytatni a végrehajtást.

számlálós ciklus ciklusváltozója tetszőleges olyan változó lehet, amelynél van értelme *követ-* értékről beszélni, pl.:

```
iklus BETŰ='A'-tól 'Z'-ig
  Ki: BETŰ
iklus vége
```

es esetekben a számlálós ciklust nem kell az intervallum minden elemére végrehajtani, hanem minden valahányadikra, illetve nem növekvő, hanem csökkenő sorrendben kell haladni. kre az esetekre szolgál a számlálós ciklus második változata.

```
iklus cv=K-tól V-ig L-esével
  utasításSorok
iklus vége
```

yi a különbség az első változathoz képest, hogy a ciklusmag végrehajtása után a ciklusvál-ó nem a következő értéket veszi fel, hanem értéke az előző értékhez képest L-lel nő (azaz kken, ha L negatív volt).

da:

```
iklus X=-π-től π-ig 0.01-osával
  Ki: X,sin(X)
iklus vége
```

kor a *következést* a lépésköz egyértelműsíti.)

Eljárás, függvény, operátor

algoritmuskészítéssel foglalkozó első fejezetben már taglaltuk, hogy feladatainkat célszerű feladatokra bontani. Ezek a részfeladatok felelnek meg gondolkodásunk –absztrakciós– szint-ek. Célszerű, ha ennek is megfelel egy algoritmikus szerkezet.

Az olyan részfeladatokat megoldó részprogramokat, amelyek „valamit csinálnak", nevezzü
*járás*oknak. Az eljárások egyik fontos –de nem egyetlen– szerepe az absztrakció támoga
Formája:

```
Eljárás Eljárásnév:
  utasításssorok
Eljárás vége.
```

Az eljárás tehát –első közelítésben– formailag pont úgy néz ki, mint a (fő)program. Használa

```
Eljárásnév
```

egyszerűen a nevének leírásával történik. Az eljáráshívás tehát *önálló utasítás*ként szerepelhe

Az eljárás másik szerepét egy példán keresztül valósítjuk meg: a programunkban nagyon sok
lyen fordul elő az, hogy lapozni kell (várni kell a felhasználó beavatkozására, majd törölni k
képernyőt és folytatni a feldolgozást. Erre a célra szolgálhat a következő eljárás:

```
Lapozás:
   Ki: „Nyomj le egy billentyűt!"   [a kép legalsó sorába]
   Billentyűlenyomásra_várakozás
   Képtörlés
Eljárás vége.
```

Ezután ahány helyen kell lapozni, annyiszor kell leírni ennek az eljárásnak a hívását (s n
teljes eljárásszöveget). Ezzel megkaptuk az eljárások második szerepét: a kódrövidítést.

Sok olyan eset fordul azonban elő, amelyekben nem azonos kódrészt kellene rövidíteni, h
csak hasonlóakat.

Példa: az osztály fiú-, illetve lánytanulói átlaga az egyes tanulók átlageredménye alapján (F
L(M)).

```
Eljárás Átlagszámítás:
  FA:=0
  Ciklus I=1-től N-ig
    FA:=FA+F(I)
  Ciklus vége
  FA:=FA/N
  LA:=0
  Ciklus I=1-től M-ig
    LA:=LA+L(I)
  Ciklus vége
  LA:=LA/N
Eljárás vége.
```

→

```
Eljárás Átlagszámítás:
  Átlag(N,F,FA)
  Átlag(M,L,LA)
Eljárás vége.

Eljárás Átlag(DB,V,ÁT):
  ÁT:=0
  Ciklus I=1-től DB-ig
    ÁT:=ÁT+V(I)
  Ciklus vége
  ÁT:=ÁT/DB
Eljárás vége.
```

Nyilvánvaló, hogy itt egy kódrészlet olyan megismétléséről van szó, amelyben három válto
kell cserélni, s minden más marad változatlan. Vegyük észre, hogy az algoritmus megf
mazása során számos változót használtunk föl különösebb előzetes jellemzés nélkül (FA, I,
F, LA, L). Némelyek a feladatból következtethető jelentéssel bírnak, másokra a használat
jából lehet következtetni. Bár érezhető, hogy ez így nem kellően precíz hozzáállás, mégis eb
fejezetben az adatok pontosításától eltekintünk; a következő fejezetet viszont csak ennek
teljük.

Az eljárások tehát *paraméterezhető*ek lesznek, s így általánosabban adhatók meg. Definiál
meg kell adni a paraméterek eljáráson belül használt nevét (ún. *formális paraméter*), h
különböző helyeken különböző paramétcrekkel használhatják. Az eljárás hívásánál pedig fe
sorolni azon konkrét változókat, konstansokat, kifejezéseket (az ún. *aktuális paramétert*

lyekkel az eljárást el kell végezni.

enti példából azonban az is látszik, hogy a paraméterek legalább kétfélék lehetnek: amiből az
rás „számol" valamit (*bemenő paramétere*k), illetve, amit eredményül ad (*kimenő paramété-*
). Harmadik fajta lehetne a be– és kimenő paraméter, amelynek az értékét az eljárás felhasz-
a, majd meg is változtatja.

másik csoportosításban a paraméter értéke az eljárásban vagy megváltozik (ún. *változó*
améter), vagy pedig nem (ún. *konstans paraméter*). Az eljárások leírásában, illetve hívásában
ez utóbbit követjük:

```
Eljárás Eljárásnév(formális paraméterek):
  utasításssorok
Eljárás vége.
```

```
Eljárás Eljárásnév(aktuális paraméterek)
```

ben a leírásban a *formális paramétere*k a legegyszerűbb esetben változók felsorolását jelentik.
ki akarjuk emelni, hogy mely paraméterek változhatnak meg, akkor írjuk ki eléjük a **Változó**,
tve a **Konstans** alapszavakat, s ha még a típusukat is meg akarjuk adni, akkor azt írjuk
géjük! A formális és az aktuális paramétereket felsorolásuk sorrendjében feleltetjük meg egy-
snak.

da:

```
Eljárás Lapozás(Konstans CIM:Szöveg):
  ... eljárástörzs ...
Eljárás vége.
Eljárás Átlag(Konstans DB,V, Változó ÁT):[7]
  ... eljárástörzs ...
Eljárás vége.
```

egy formális paramétert konstansként deklarálunk, akkor az eljárás hívásakor aktuális para-
terként konstans vagy kifejezés is szerepelhet, ha azonban változóként, akkor hívni is csak
ltozóval lehet.

aphelyzetben az eljárások csak azokat a változókat használhatják, amelyeket paraméterként
gadtunk számukra, a bővebb definíciójuk alapján azonban használhatnak ezen kívül tetsző-
es *globális változót*, amelyet a feladat részekre bontásának egy, az őt tartalmazó szintje de-
iált.

ézzünk erre példaként egy konkrét mátrix (A(N,M)) adott sorának sorösszegét kiszámoló el-
rást!

```
Eljárás Sorösszeg(I,ÖSSZ):
  ÖSSZ:=0
  Ciklus J=1-től M-ig
    ÖSSZ:=ÖSSZ+A(I,J)
  Ciklus vége
Eljárás vége.
```

z önállóan definiált részfeladatokat azonban néha kényelmesebb lenne másképpen használni. E
ás módszer ugyanaz, mint a matematika függvényeinek használati módja: a függvényértékek

[7] A fenti felírásmód is korrekt, de kevesebb információt tartalmaz; illetve még ennél is több lehetne az eljárásfejben, ha az előző eljáráshoz hasonlóan a paraméterek típusát is leírnánk.

kifejezésekben használhatók. Az eljárás melletti másik absztrakciós eszköz lesz a *függvény*.[8] I
finiálása hasonló az eljáráséhoz, használata formailag ugyanaz, csak nem önálló utasításként,
nem kifejezésben szerepelhet.

```
Függvény Függvénynév(formális paraméterek): függvényérték típusa
  utasítássorok
  Függvénynév := kifejezés
Függvény vége.
```

Megjegyzés: Elképzelhető lenne egy másik szintaxis is:

```
típus Függvény Függvénynév(formális paraméterek):
  ...
Függvény vége.
```

Így jár el pl. az ELAN nyelv is.

A *függvény* hívása ekkor az alábbi módón történhet (de természetesen egy olyan típusú kifeje
részeként, amilyen típusú értéket szolgáltat a függvény):

```
... Függvénynév(aktuális paraméterek) ...
```

Megjegyzés: A rövidebb, kevesebb információt tartalmazó felírásban a függvényérték típusa,
a definiálás legelején szereplő **Függvény** alapszó is elhagyható (bár nem javasolható).

A függvény formális paraméterei csak konstansok lehetnek, őket megváltoztatni nem tudja.

Példa: a fiúk vagy a lányok átlaga a jobb?

```
Eljárás Átlagszámítás:
  Ha Átlag(N,F)>Átlag(M,L) akkor Ki: „Nagyobb a fiúk átlaga"
                           különben Ki: „Nagyobb a lányok átlaga"
Eljárás vége.

Függvény Átlag(DB,V): Valós
  ÁT:=0
  Ciklus I=1-től DB-ig
    ÁT:=ÁT+V(I)
  Ciklus vége
  Átlag:=ÁT/DB
Függvény vége.
```

Vannak speciális függvények, amelyeket a többiektől eltérő módon használunk, jelüket nem
argumentumuk elé, hanem *argumentumaik közé* írjuk. Ezek az ún. *operátor*ok, vagy magyar
*művelet*ek. Ilyenek a szokásos aritmetikai, logikai műveletek, relációk. Ha egy adattípushoz ilyen
jellegű műveletet kell készíteni, akkor operátordeklarációt írunk.

```
Művelet Operátornév(formális paraméterek): értékének típusa[9]
  utasítássorok
  Operátornév := érték
Művelet vége.
```

```
... aktuális paraméter1 Operátornév aktuális paraméter2 ...
```

Az operátornak kettő vagy egy argumentuma lehet (utóbbi esetben az aktuális paraméter$_1$ el

[8] Függvényeljárásnak is hívják az eljáráshoz hasonló szerepe miatt.

[9] Szokás a **Művelet** helyett **Operátor** kulcsszót használni.

la: térbeli helyvektorok összeadása (a **Helyvektor** típust már definiáltnak föltételezzük)

```
űvelet +(Konstans A,B: Helyvektor): Helyvektor
 Ciklus I=1-től 3-ig
   C(I):=A(I)+B(I)
 Ciklus vége
 + := C
űvelet vége.
```

ldában jól látszik, hogy időnként eléggé suta módon lehet csak hivatkozni az operátor szim-
mára (ami itt a „+"). Ilyen és egyéb okok miatt (gondoljunk az ún. szabad szintaxisú operá-
kra) bevezetjük az operátorok időleges átnevezését lehetővé tevő **Másként** kulcsszóval be-
tett utasítást a művelet definiálásában:

```
Művelet Operátornév(formális paraméterek): értékének típusa
  Másként operátorszimbólummal összekapcsolt kifejezése a paramétereknek
  utasítássorok
  Operátornév := érték
Művelet vége.
```

dául:

```
űvelet VektorÖsszeg(Konstans A,B: helyvektor): Helyvektor
 Másként A + B
 Ciklus I=1-től 3-ig
   C(I):=A(I)+B(I)
 Ciklus vége
 VektorÖsszeg := C
űvelet vége.
```

veletek felhasználása az elvárható módon történik.

dául, ha az `A`, `B` és `C` `Helyvektor` típusú adatok, akkor az `A` és `B` összegvektorát kapjuk `C`-ben:

```
C := A + B
```

árások, függvények paramétere az eddigiekben csak konstans vagy változó lehetett, de min-
képpen adat. A következőkben ezt általánosítjuk.

például egy függvénytabellázó eljárást írunk, akkor abban nem csak a táblázatban szereplő
kintervallum, illetve lépésköz lehet paraméter, hanem maga a függvény is. Az első általánosí-
tehát az *eljárás-*, illetve a *függvényparaméter*.

da:

```
Eljárás Függvénytabellázás(Függvény f(Valós): Valós,
                           Konstans A,B,C: Valós):
 Ciklus X=A-tól B-ig C-esével
   Ki: X,f(X)
 Ciklus vége
Eljárás vége.
```

bekeretezett paraméterész a paraméterfüggvényt leíró rész. Ebből derül ki, hogy az eljárásban
`f` formális névvel hivatkozunk rá, és neki egyetlen **Valós** paramétere lehet, s eredménye is
lós típusú.

újabb általánosítást egy példapáros vezeti be. Két feladatot nézünk mindegyikben, s a közös
lemzőjük alapján találjuk meg az eljárások paraméterfogalmának kiteljesítését.

Példa:

egy osztály 5-ös osztályzatú tanulói

```
Eljárás: Kiválogatás(N,OSZT,JEGY):
  Ciklus I=1-től N-ig
    Ha OSZT(I)=JEGY akkor Ki: I
  Ciklus vége
Eljárás vége.
```

egy névsor Pál keresztnevű tagjai

```
Eljárás Kiválogatás(N,SOR,NÉV):
  Ciklus I=1-től N-ig
    Ha SOR(I)=NÉV akkor Ki: I
  Ciklus vége
Eljárás vége.
```

Ránézésre a két eljárás lényegében azonos, jó lenne csak egyszer megírni. A baj az, hogy a vektor egész számokat tartalmaz, s a JEGY is egész értékű, míg a SOR szövegeket tartalmaz NÉV is szöveg.

A megoldás az *eljárás paraméterezése típussal*: legyen paraméter az elemek típusa is.

```
Eljárás Kiválogatás(Típus Elemtípus,
                    Konstans N: Egész,
                             X: Tömb(1..N,Elemtípus),
                             E: Elemtípus):
  Ciklus I=1-től N-ig
    Ha X(I)=E akkor Ki: I
  Ciklus vége
Eljárás vége.
```

Ezt az eljárást a fenti két feladatban a következőképpen hívhatjuk:

```
Kiválogatás(Egész,N,OSZT,JEGY)
Kiválogatás(Szöveg,N,SOR,NÉV)
```

Összeköthetjük ezt a bővítést az előzővel: tetszőleges tulajdonságú elemeket válogassunk ki tetszőleges típusú elemeket tartalmazó sorozatból!

```
Eljárás Kiválogatás(Típus Elemtípus,
                    Konstans N: Egész,
                             X: Tömb(1..N,Elemtípus),
                    Függvény t(Elemtípus): Logikai):
  Ciklus I=1-től N-ig
    Ha t(X(I)) akkor Ki: I
  Ciklus vége
Eljárás vége.
```

Ebben az esetben az első példabeli hívás a következő lehet:

```
Kiválogatás(Egész,N,OSZT,Ötös)
```

ahol az Ötös függvény így néz ki:

```
Függvény Ötös(Konstans J: Egész): Logikai
  Ötös:=(J=5)
Függvény vége.
```


V. Adatok, adatszerkezetek

z adatjellemzők összefoglalása

zonosító

jelsorozat, amellyel hivatkozhatunk a tartalmára, amely által módosíthatjuk tartalmát.

ul:

i, j, n, a – *változó*k nevei;

π, –128, 3.14,

"Eredmény=", IGAZ – *konstans*okat azonosító jelsorozat.

ük észre a változó nevének janusarcúságát! Az **x:=x+1** *értékadás*ban a baloldali **x** *címhivat-*st reprezentál, amíg a jobboldalon ugyanő *értékhivatkozás*t jelent[1].

Hozzáférési jog

okat *módosítani*, illetve értéküket *lekérdezni*, használni lehet; eszerint egy adat hozzáférés pontjából négyféle lehet:

	módosítható	lekérdezhető	
független	nem	nem	*független*
input	nem	igen	*konstans*
output	igen	nem	*virgin*
I/O	igen	igen	*változó*
a külvilágban			a memóriában

Kezdőérték

üületéskor hozzárendelt érték. Konstansoknál nyilvánvaló; változóknál deklarációban kap-e, ató-e, vagy futáskor szerez értéket magának. Vegyük észre, ha a *'nem definiált'* érték egy iális érték, akkor mód van ellenőrzésre is (*virgin*)!

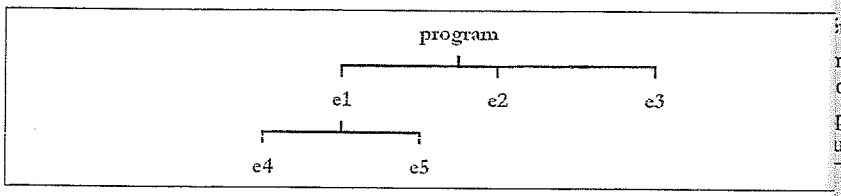
Hatáskör

rogramszöveg azon tartománya, amelyben az adathoz hozzáférés megengedett (nem füg-en).

lául az egyes eljárások használják egymás változóit, konstansait.

1. ábrán vázolt programhierarchia mindegyik eljárása (*e1*, ..., *e5*) használ egy i azonosítójú ozót. Kérdés, milyen kapcsolatban állhatnak ezek az azonos nevű változók egymással, ha az rások kapcsolatát az alábbi séma írja le?

ért fontos az eljárásoknál a paraméterek ilyen szempontú megkülönböztetése: annak explicit jelzése, hogy *input* gy *output* (is!) paraméter, ami a paraméterátadás-átvétel szempontjából *érték-* vagy *címhivatkozás*t jelent.



1. ábra. Példa a hatáskörre.

Elképzelhető válaszok az i változók viszonyáról:

1. e1,...,e5-beni ugyanaz; *globális*[2]
2. e1, e4, e5-ben i ugyanaz
   de e2, ill. e3-belitől független; } *lokális*[3]
3. e1,...,e5-ben mind különböző. *lokális*[4]

1.5. Élettartam

A *futási időnek az az intervalluma, amelyben az adat azonosítója végig ugyanazt az objel jelöli.*

Például:

- a globális változók születéstől a program teljes futásidejében élnek;
- a lokális változók csak addig élnek, amíg az az eljárás (programrész) aktív, am deklarálták; illetve ezek is a kóddal együtt születnek meg fordításkor, hozzátapadnak, d „aktivizálódnak" amíg a kód megfelelő része nem „mozgatja".

A dinamikus adatok életét (futási időben) nem deklaratívan működő utasítások határozzá létrehozzák, megszüntetik őket (így nem kötődnek a hatáskörhöz...).

1.6. Értéktípus

Az adatoknak az a tulajdonsága, hogy *értékei mely* **halmazból** származnak és **tevékenység** (függvények, operátorok, utasítások) *mely „készlete*, amely *létrehozza, felépíti, lerombolja szekre bontja*", alkalmazható rá.

2. Az értéktípus

2.1. Az értéktípusról általánosságban

Összetettség (strukturáltság) szempontjából beszélhetünk *strukturálatlan* (vagy *skalár*) típ ha (az adott szinten) szerkezetet nem tulajdonítunk neki; vagy *strukturált* (más szóval: *össz* típusról, ha (elemibb) összetevőkre bontjuk.

Például, ha az (A(),N) vektort

- *skalár*nak tekintjük, akkor értéke az a byte- (bit-) sorozat (=tárterület), amely az A() „

[2] ti. az egész programra nézve

[3] az e1-re nézve; de az alatt levő szintek szempontjából *globális*

[4] minden eljárásra nézve; mondhatnánk: *saját*

értékét „egységes egészként" tartalmazza, részeire nem alkalmazandó semmilyen művelet;

nint egy N-dimenziós *vektor*t használjuk, akkor értéke az őt alkotó, önállóan is elérhető vek- orelemek N-esének aktuális értéke.

pvető *adatstrukturálási módok*, amelyekkel egyszerűbbekből (bázistípusból) összetettebb ukturált) *adatszerkezetek* készíthetők:

strukturálási mód	→	adatszerkezet
esztszorzat	→	rekord
egkülönböztetett) egyesítés	→	változó résszel rendelkező rekord (alternatív rekord)
lmazképzés	→	halmaz
rált megadás	→	sorozatok: tömb, sor, file ...

usok *megadása* **értékhalmaz**uk és a **típushoz asszociált műveletek** kettősének típusnévhez delését jelenti. Az **értékhalmaz** azon **konstansok halmaza**, amelyből veheti az ebbe a típusba ozó adat értékeit. Ezen halmazok némelyike –a típus gyakori előfordulása miatt– eleve defi- lt szokott lenni (**elemi**ek), és mintegy a továbbiak –összetettebbek– bázistípusaként használha- öl. A **típushoz asszociált műveletek** az adott értékhalmazon értelmezett (és a típushoz szoro- kapcsolódó) operációkat (függvényeket, műveleteket) jelentik.

következőkben példákat is adunk, amelyeket az adatleíró nyelv szintaxisának megfelelően ık le. Előzetesen –és most nem részletezve– a következő jelöléseket fogjuk alkalmazni.

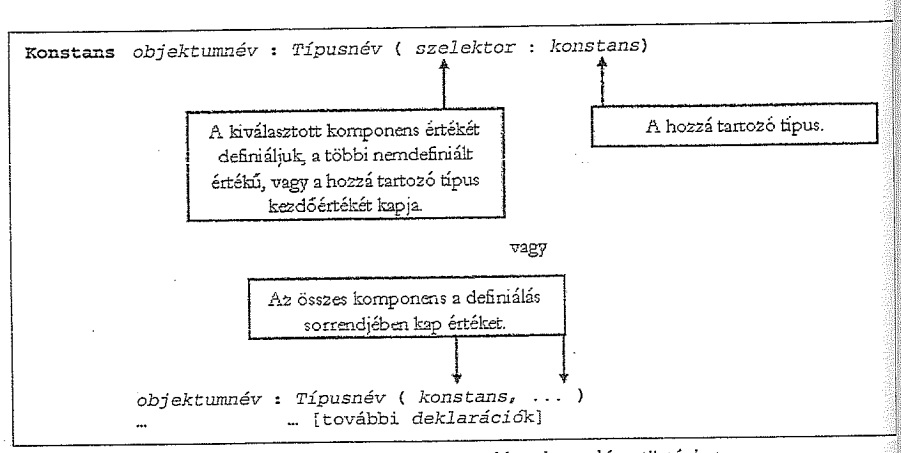
vvel látjuk el a konstrukciónak megfelelő típust, hogy későbbiek során egyszerűen –csupán évél– lehessen hivatkozni rá:

```
Típus     Típusnév = Típuskonstrukció  [≡típusmegadás]
          ...              ...     [további típusdefiníciók]
```

egyes objektumokhoz[5] hozzárendeljük a típusukat, amely rögzíti, hogy milyen halmazból ve- ik az értékeiket, és milyen operációk vonatkozhatnak rájuk:

```
Változó   objektumnév : Típusnév
          ...              ... [további deklarációk]
```

bjektum szót fogunk használni, ha az adatot akár *konstans*, akár *változó* tartalmazhatja.



Megjegyzés: a *változók kezdőértékadás*a a konstansokhoz hasonlóan történhet meg.

2.2. Az asszociált műveletek osztályozása

Az alábbiakban a típusokhoz hozzárendelendő, hozzáértendő, asszociálható műveleteket csoportosítjuk: *értékadás*, *típusátviteli függvény* (*konstrukciós*, *szelekciós függvények*, *azonosság* és más *reláció*, *Számosság*, *Min- és Max-*, *Rend-függvény*), *transzformációs* és egyéb függvények.

A. Értékadás = azonos típusúak közötti adatmozgatás, másolatkészítés. Az értékadás műveleti jele a ':='; szemantikája: *céladat* értéke ettől a pillanattól kezdve ugyanaz, mint *tárgyadaté* (a következő módosításig). Későbbiekben látni fogjuk, hogy nem egyetlen lehetőség az értékadás megvalósításánál a fenti *értékmásolás*, hanem sok esetben –különösen nagy „terjedelmű", dinamikusan épített struktúrák esetében, vagy címszerinti paraméterátadásnál– *értékmegosztás*sal rendelődik az új érték a struktúrához. Ez azt jelenti, hogy ugyanazon memóriatartomány –az érték– több objektumhoz is hozzátartozik. Így nem várt jelenségek is előfordulhatnak: egyes objektumok értékei váratlanul megváltoznak.

B. Típusátviteli függvények = valamely típus értékeit egy másik típus értékeire képezik le.

A típusátviteli függvényeknek nevezetes fajtái vannak, ezek: konstrukciós, szelekciós, illetve speciális (de nélkülözhetetlen) egyéb függvények. Tudnivalók ezekről:

° *Konstrukciós függvények* = *strukturált érték létrehozása* (egyesítése) komponensek értékeiből. A **függvény neve** mindig az **adott típus nevé**vel egyezik meg, paraméterei a típusdefinícióban szereplő (al)típusú adatok (definíciós sorrendben) lehetnek. Ha *felsorolás*- (3.5.), vagy *intervallumtípus*ra (3.6. rész) alkalmazzuk, akkor a függvény jelentése értelemszerűen megváltozik. Nevezetesen: *a típus értékkészletének „argumentumadik" konstansát* adja értékül. *Elemi típusok*ra alkalmazva (ennek csak konstans deklarálásánál van értelme) magát *az argumentum*ot jelenti (tehát mint egy identikus függvény működik).

Például:

```
Típus    Hely = Tömb(1..3:Valós)[6]
Változó  pont : Hely
         ...
```

Ekkor a *Hely*(2.0,4.0,6.0) függvény értéke a (2.0,4.0,6.0) valós értékű vektor maga. Így az értékadás utasítást felhasználva a következő tömörítésre van módunk: a

```
pont(1):=2.0
pont(2):=2.0   } helyett pont:=Hely(2.0,4.0,6.0)
pont(3):=6.0
```

Van úgy, hogy a konstrukciós függvény elrejtve tevékenykedik: amikor a **'Be:'** vagy a **'Ki:'** utasítás hajtódik végre, akkor automatikusan konverzió történik a *Szöveg* és az argumentumbeli objektum típusa között. Vagyis a **'Ki:'** esetén egy *Szöveg*konstrukció zajlik, amíg a **'Be:'** estében egy más típusú érték születik.

° *Szelekciós függvények* = strukturált adat komponenseihez való hozzáférés eszköze. Ezen függvény jelölését illetően –s a „néven túli" formalizmusra gondolunk (pl. infix, prefix stb.)– igen nagy változatosság jellemző a nyelvekre. Erről győződhetünk meg majd a IV. fejezetben.

Például:

```
Típus    Hely = Tömb(1..3:Valós)
Változó  pont : Hely
         x : Valós
         ...
x:=pont(2)
         ...
```

értékadásban a `pont(2)` kifejezésben a '(.)' szelekciós függvényt használtuk a `Hely`vektor 2. komponensének (2. vektorkoordinátájának) kiválasztására; vagy a

```
pont:=Hely(pont(1)^2,pont(2)^2,pont(3)^2)
```

értékadásban a `pont` vektor 1., 2. és 3. koordinátáinak négyzetéből „összeálló" vektor szerepel a jobb oldalán, amelyet (egy `Hely`típusú tömbértékké egyesít a `Hely` konstrukciós függvény) ugyancsak a `pont` vektor kap értékül.

Vannak explicit névvel ellátott szelekciós függvények. Erre példa az alábbi, a *vermek*re vonatkozó részlet. Vegyük észre, hogy annak ellenére soroltuk a szelekciós függvények közé, hogy formálisan nem is függvényeljárásként írtuk. Vagyis nem a leírás módja a fontos, hanem az elvégzett leképezés milyensége!

Például:

```
Típus    Veremtip = Verem(Egész)
Változó  v : Veremtip
         e : Egész
         ...
Veremből(v,e) [v→e]
         ...
```

Itt a `Veremből(.)` tevékenység a tulajdonképpeni szelekciós „függvény" maga.

° *Azonosság* = két, azonos típusú adat értékegyezőségét vizsgáló logikai értékű függvény. (A

[6] A `Tömb(1..3:Valós)` jelentése: 1..3 indexű tömbje valósaknak (l. IV./4.-nél).

legfontosabb konverziós függvényként is tekinthetjük.) Jele az = (egyenlőségjel).

Például:

```
Típus    Hely = Tömb(1..3:Valós)
Változó  pont : Hely
   ...
pont:=Hely(4.5, 5.2, 8.9)
...
```

értékadás után a `pont=Hely(4.5, 5.6, 7.8)` reláció értéke hamis.

° *Számosságfüggvény* = megadja (ha megadható), hogy mennyi az adott típus „számossága", azaz az értékhalmazát alkotó konstansok száma. Fontos tudni, hogy ez „nem-véges" típusokra implementációfüggő értékű! (L. még a *Rend*-függvényt, továbbá a későbbi példákat!)

° *Min/Max-függvény* = az értékhalmaz legkisebb, illetve legnagyobb eleme (feltéve, hogy rendezett típusról van szó). A függvény nevére az alábbi konvenció érvényes: *Min*-nel vagy *Max*-szal kezdődik, és a típus nevével folytatódik (mintha az adott típus első, illetve utolsó konstansa lenne). Annak érdekében, hogy ne korlátozzuk a szabad azonosítóválasztást a függvénynév két részét egy olyan karakterrel választjuk el, amely egyéb célú fölhasználása igen ritka, ez a jel az *aposztrof* (') . Természetesen az azonosítókban való alkalmazásánál ezt figyelembe kell venni! (Nem-véges típusnál implementációfüggő, vagy nem definiált!)

Például:

```
Típus    Index = 0 .. 99
Változó  n,m,i : Egész
   ...
n:=Max'Index; m:=Min'Index; i:=Számosság'Index
```

egyenértékű az `n:=99; m:=0; i:=100` értékadásokkal (mivel az *Index*típus minimális értéke konstansa a 0, a maximális pedig a 99).

Megjegyzések:

1. A *Számosság*, a *Min* és *Max* függvényeket nyugodtan tekinthetjük az adott típushoz tartozó a típust jellemző *konstans*oknak, s nem függvénynek, hisz ténylegesen nincs is argumentumuk, amihez rendelnék az értéket; továbbá a típus definiálásának pillanatában „rögzül" is az értékük.
2. Elő fog fordulni, hogy a számtípusú *Max*-ot, illetve *Min*-t –az „egészséges lustaság jegyében"– a rövidebb és megszokottabb +∞, illetve -∞ szimbólumokkal jelöljük.

° *Rend*-függvény[7]= (rendezett típus esetén[8]) az adat értékhalmazbeli sorszámát szolgáltatja (Nem-véges típusnál implementációfüggő, vagy nem definiált!) Érdemes észrevenni a előbbi függvények kapcsolatát: `Számosság'Típus-1=Rend(Max'Típus)`

[7] Hívhatjuk *Sorszám*-függvénynek is.

[8] Az persze elképzelhető, hogy a rendezés nem eleve adott, hanem „külön" hozzá kell definiálni egy megfelelő *kai operátor*t.

Például:

```
Típus    Index = -3 .. 3
Változó  n,m: Egész
         ...
n:=Rend(Index⁹(-3)); m:=Rend(Index(3))
         ...
```

egyenértékű az `n:=0; m:=6` értékadásokkal; ugyanis az intervallumtípus a belső ábrázolásban mindig **0-tól folyamatosan rendeli az intervallumbeli konstansokhoz a (belső) indexeket**, egészen a `Rend(Max'Típus)-1`-ig.

° *Egyéb relációk* = (csak rendezett típusokon) a szokásos <, ≤, >, ≥, ≠ relációk.

C. Transzformációs függvények = a típuson (esetleg direktszorzatán) értelmezett, a típusra képező függvények. Ezek –értelemszerűen– típustól függően mások és mások lehetnek.

Például:

egészeken : `+` , `-` , `*` , `Div` (egészosztás), `Mod` (maradékképzés)

valósokon : `+` , `-` , `*` , `/` , `^`

számvektorokon (nyilván azonos dimenziósokon): `+` , `-`, `*` (vektoriális szorzás)

D. Egyéb függvények = pl. ide sorolhatjuk azokat a transzformációs függvényeket, amelyek kivezetnek az értelmezési tartomány típusából, vagy vegyes típusúak (ti. az értelmezési tartományuk), de specialitásuknál fogva nem kerültek egyik korábbi függvényosztályba sem.

Például:

értelmezési tartomány típusa	→	értékkészlet típusa
egész × vektor	→	vektor
vektor × vektor	→	vektor
vektor × mátrix	→	vektor

Néhány –javarészt a *sorozattípusú*akra vonatkozó– függvényt is ide sorolhatunk, amelyek feladata pl. egy komponens elérésének csupán „előkészítése".

Például:

a *Következő*re(l) vagy az *Előző*re(l) függvények egy l *List*atípusú objektumra vonatkozólag.

Utolsó megjegyzésként a típusokról általánosságban:

Amikor egy típus kerül szóba a programírás során, akkor az összes vele kapcsolatos kellékére gondolunk, úgy a *konstans*aira, mint az *asszociált művelet*einek teljes garnitúrájára (*Min*-, *Max*-, *Rend*-, *Sorszám*- és transzformációs stb. függvények, operátorok ...). Ez különösen akkor válik nem magától értetődő, sőt fontos megjegyzéssé, amikor a típust mint *paramétert* szeretnénk kezelni.

3. Egyszerű típusok

Ennek a fejezetnek az a célja, hogy definiálja az „eredendően" szerkezetnélküli típusokat. Megadjuk ezek *értékhalmaz*át, a hozzátapadó *művelet*ek, *reláció*k körét. A definíció mikéntjét a „szokás" és a „kívánalom" határozza meg.

[9] Kénytelenek vagyunk közbeiktatni az *Index* típuskonstrukciós függvényt, hiszen nem egyértelmű a -3 „hovatartozása" (lehet *Egész* és lehet *Index* típusú is).

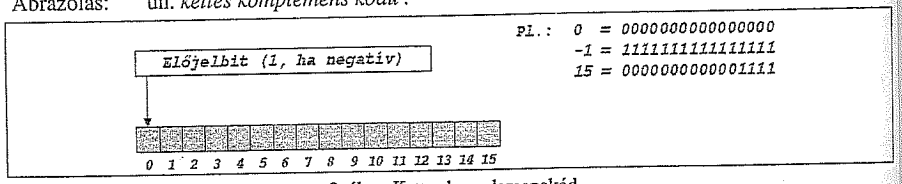
3.1. Egész típus

Értékhalmaz: -32768..32767 [10]
(*Min'Egész..Max'Egész*)

Műveletek: + , - , * , **Div** (egészosztás), ^ (pozitív egészkitevős hatványozás), **Mod** , - (unáris mínusz) .

Relációk: = , < , ≤ , ≥ , > , ≠ .

Ábrázolás: ún. *kettes komplemens kódú* .



2. ábra. Kettes komplemenskód.

Például:

```
Változó  i : Egész
Konstans N : Egész(35)
```

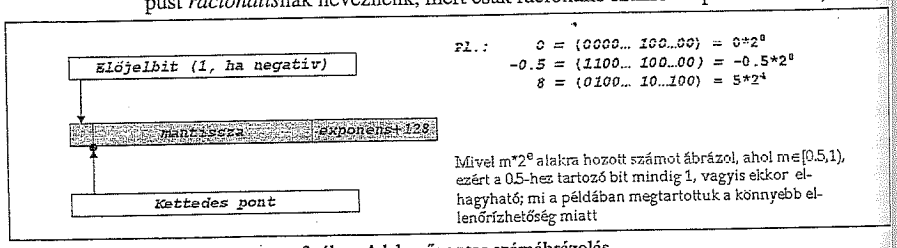
3.2. Valós típus

Értékhalmaz: ????..????
(*Min'Valós..Max'Valós* nem definiáltak, vagy implementációfüggő)

Műveletek: + , - , * , / , ^ , - (unáris mínusz) .

Relációk: = , < , ≤ , ≥ , > , ≠ .

Ábrázolás: ún. *lebegőpontos ábrázolás* (mint az alábbiakból kitűnik pontosabb lenne, ha e típust *racionális*nak neveznénk, mert csak racionális számot képes ábrázolni) .



3. ábra. A lebegőpontos számábrázolás.

Például:

```
Változó  r : Valós
Konstans pi : Valós(3.141592)
```

[10] Természetesen itt és később is az ilyen természetű adatokat egy konkrét ábrázolással összefüggésben kell érteni! Annak érdekében, hogy ilyen konkrét értékekkel nem kelljen megterhelni a programozó agyát, vezettük be az előző fejezetben a *Min*, *Max* típusfüggvényeket.

3.3. Logikai típus

Értékhalmaz: *H*amis..*I*gaz
(*Min'Logikai..Max'Logikai*: *Hamis*, illetve *Igaz*)

Műveletek: **nem** , **és** , **vagy** (a szokásos logikai műveletek) .

Relációk: = , < , ≤ , ≥ , > , ≠ (a belső ábrázolásuk alapján) .

Ábrázolás: 0 = *Hamis*, -1 = *Igaz* (néha 1 = *Igaz*) .

Például:

```
Változó  siker : Logikai
Konstans nem : Logikai(Hamis)
```

3.4. Karaktertípus

Értékhalmaz: 0..255 -*kód*ú jelek
(*Min'Karakter..Max'Karakter*: a 0, illetve a 255 kódú karakter)

Műveletek: nincs .

Relációk: = , < , ≤ , ≥ , > , ≠ (a belső ábrázolásuk alapján) .

Az ASCII karakterkészlet néhány kódjáról:

0-31: ún. kontroll karakterek;

32-64: szóköz, szokásos írásjelek, ill. számjegyek (48-57 = '0'-'9');

65-91: nagybetűk;

92- : kisbetűk, grafikus jelek (erősen implementációfüggően).

Például:

```
Változó betű : Karakter
Konstans szóköz : Karakter(" ")
```

vagy

```
Konstans szóköz : Karakter(32)
```

3.5. Felsorolástípus

Mindazon típusokat, amelyek értékkészletét konstansainak egyszerű fölsorolásával adhatjuk meg *diszkrét típus*nak hívjuk. Speciálisan tehát ilyen az egész, a logikai, a karakter, de lehet bármilyen –a program írója által kreált– *absztrakt konstans*okat tartalmazó ún. *absztrakt felsorolástípus* is.

Értékhalmaz: (konstans$_1$, konstans$_2$, ... , konstans$_N$)
(*Min'Típus..Max'Típus*: konstans$_1$..konstans$_N$)

A konstansok maguk a típus értékkészletét meghatározó rendezett absztrakt értékek.

Műveletek: a rendezettségre építenek az alábbi függvények; nem értelmezett helyeken nemdefiniált az értékük:

Következő(*típusbeli kifejezés*),

Előző(*típusbeli kifejezés*),

Rend(*típusbeli kifejezés*).

A deklarációban a kezdőérték megadásához a korábbi szokás szerint használható a *típusnév* nevű konstrukciós függvény. Emellett azonban a végrehajtáskor sokszor jó haszonnal jár az alábbi értelmezésű párja:

Típusnév(0..**Rend**(**Max**'*Típusnév*)).

Ez utóbbi függvény jelentése: a paraméterként „megadottadik" típusbeli konstans. Tehát mintha ez a *Rend*-függvény inverzeként viselkednék!

Relációk: $=, <, \leq, \geq, >, \neq$ (a *felsorolás sorrendje* egyben a *rendezés* is)

Például:

```
Típus     Hét = ( hétfő, kedd, szerda, csütörtök, péntek, szombat, vasárnap )
Változó   tegnap, ma, holnap : Hét
Konstans  ünnepnap : Hét(vasárnap)
          i  : Egész
...
tegnap:=Előző(ma); i:=Rend(ma)
Ha ma=Max'Hét akkor  holnap:=Min'Hét
             különben holnap:=Következő(ma)
...
```

3.6. (Rész)Intervallumtípus (diszkrétből származtatott típus)

Értékhalmaz: $konstans_1$..$konstans_2$

(*Min'Típus*..*Max'Típus*: $konstans_1$..$konstans_2$)

A származtatás által meghatározott bázistípus adott részhalmaza, helyesebben részintervalluma.

Műveletek: ugyanazok, amik a bázistípuson értelmezettek (a felsorolástípusoknál említett *típuskonstrukciós* függvényt itt nem definiáljuk; l. a példa utáni megjegyzést!).

Relációk: ugyanazok, amik a bázistípuson értelmezettek.

Például:

```
Típus     Hét= (hétfő, kedd, szerda, csütörtök, péntek, szombat, vasárnap)
          Munkanap = hétfő..péntek
          Hétvége  = szombat..vasárnap
Konstans  utolsónap : Munkanap(péntek)
Változó   mnapindex : Munkanap
```

Megjegyzés: a típuskonstrukciós függvény nem egyértelműségéről:

1) *Hétvége*(0)=szombat, *Hétvége*(1)=vasárnap lenne a kívánatos a *Hétvége*típus definíciója alapján, de mivel a
2) *Rend*(szombat)=5, *Rend*(vasárnap)=6 –hisz *Hét*-beliek–, logikus a *Hétvége*(5)=szombat, *Hétvége*(6)=vasárnap lenne –az inverz tulajdonság miatt–; s ez ellentmondást eredményez.

3.7. Valós résztípus

Értékhalmaz: valós $kons_1$ .. valós $kons_2$valós $kons_3$**-lépéssel**

(*Min'Típus*..*Max'Típus*: $kons_1$.. $kons_2$)

Olyan valósakat tartalmaz, amelyek előállíthatók a $kons_1$+i*$kons_3$ formulával, ahol i=0..$kons_2$-$kons_1$ / $kons_3$.

Műveletek: Korlátozott valós műveletek (csak a formulával előállíthatók jöhetnek ki – automatikus kerekítéssel, ha szükséges), vagy ha nem ilyen, akkor mint „egyszerű" valós értékkel lehet csak továbbszámolni.

Relációk: valós relációk.

Például:

```
Konstans LépésKöz : Egész(0.01)
Típus  TrigÉrt  = -π..π LépésKöz-lépéssel
Konstans Középső : TrigÉrt((((Max'TrigÉrt-Min'TrigÉrt) / LépésKöz) / 2)
                           *LépésKöz+Min'TrigÉrt)
Változó  x : TrigÉrt
```

3.8. Tetszőleges típus résztípusa logikai formula alapján

Egy érdekes általánosítása az intervallumtípusnak: nem első és utolsó elem által meghatározott része egy értékhalmaznak, hanem valamilyen (predikátummal, azaz logikai formulával definiált) tulajdonságnak eleget tevő elemeinek részhalmaza.

Értékhalmaz: *BázisTípus*(x : *predikátum*(x))

Olyan $x \in \{BázisTípus\}$, amelyre a *predikátum*(x) teljesül.

Műveletek: Korlátozott *BázisTípusú* műveletek, ha ez kivezet az értékhalmazból, akkor már csak mint *BázisTípusú* értékkel lehet manipulálni. Minden speciális, csak rá vonatkozó műveletet külön eljárás, illetve függvény formájában kell definiálni. Elsőként magát a predikátumot. Mivel nem feltétlenül alkotnak összefüggő halmazt, ezért a *rákövetkezés* sincs eleve definiálva.

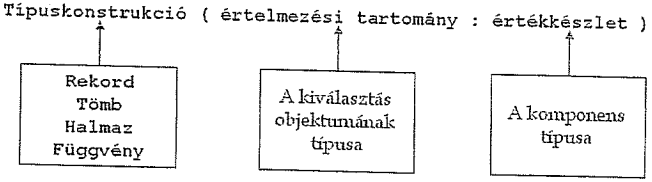
Relációk: *BázisTípusú* relációk.

Például:

```
Típus  TermészetesSzám = Egész(i : i>0)
       PrímSzám = TermészetesSzám(n : Prím?(n))
Függvény Prím?(Konstans x:TermészetesSzám): Logikai
  ...
  Prím?:=...
Függvény vége.
```

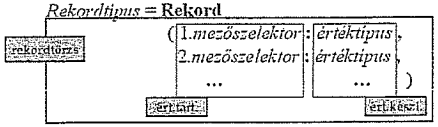
4. Összetett (strukturált) típusok képzése

Ebben a fejezetben valójában nem *típus*okról lesz szó (egyetlen kivétellel: a szövegtípus), hanem olyan **típuskonstrukciós eszköz**ökről, amelyekkel elemibbekből új, összetett struktúrák típusait lehet létrehozni. A struktúrák összetett volta miatt egy új keletű „problémával" találkozunk: a komponensekre való hivatkozás szükségességével, akár a felépítés, akár a rájuk vonatkozó hivatkozás során. Minden *összetett típust* tehát –mintegy függvényként tekintve– *két „típusgarnitúra" határoz meg* (az „összeépítésük" miként jén túl): az *értelmezési tartomány típusa* (vagyis ahogy a komponensek kiválaszthatók) és az *értékkészlet típusa* (vagyis a struktúrát alkotó egyes komponensek típusai). Az hogy egyes típusok miként szerveződnek egy összetett típussá azt írjuk le elsőként. Általános szabály lesz a típusmegadásnál, hogy a két meghatározó típust az alábbi szintaktikával „egyesítjük" a létrehozandó típusban:



4.1. Rekord-típuskonstrukció

Strukturálás:



Értékhalmaz: a mezők *értéktípus*ai által meghatározott alaphalmazok direktszorzata.

Műveletek: *szelekciós* függvény ('*.mezőszelektor*' nevű)[11],
konstrukciós függvény (*Rekordtípus* nevű),
elképzelhetők: *transzformációs* függvények, amelyek a teljes rekordstruktúrát érintik.

Relációk: = (mezőnkénti egyezés) , ≠ .

Például:

```
Típus     Komplex = Rekord
                    (re : Valós,
                     im : Valós)
          Dátum= Rekord
                    (év  : 0..2000,
                     hó  : 1..12,
                     nap : 1..31)
Változó   a, b : Komplex
Konstans  egység1 : Komplex(1,0)
          egység2 : Komplex(0,1)
          Jézus: Dátum(év:0, hó:12, nap:25)
          PéterPál: Dátum(hó:6, nap:29)
```

4.2. Alternatív rekord-típuskonstrukció

Strukturálás: *Rekordtípus* = **Rekord**

mezőszelektor$_1$: *Értéktípus$_1$* ,
mezőszelektor$_2$: *Értéktípus$_2$* } *állandó rész*
... ...
Alternatívák
felt.kif$_1$ **esetén** rekordleírás$_1$
felt.kif$_2$ **esetén** rekordleírás$_2$ } *változó rész*
... ...
Alternatívák vége)

(Itt a *felt.kif$_j$* feltételes kifejezést –leggyakrabban relációt–, a *rekordleírás$_j$* a rekord törzsét jelöli, változó rész tetszőleges mezőmegadás helyén szerepelhet, egy rekordon belül akár több is.)

Értékhalmaz: az egyes alternatív összeállítású mezők értéktípusainak direktszorzat-uniója.

Műveletek: *szelekciós* függvény ('*.mezőszelektor*' nevű),
konstrukciós függvény (*Rekordtípus* nevű),

[11] Természetesen nem ez az egyetlen szintaktikai lehetőség a *szelekció* leírására, hanem az egyes mezőket kiválaszthatnánk „tömbszerűen" (a mezőnévvel „indexelve"), vagy függvényszerűen (melyeknek neve a mezőnév, argumentuma az adott rekordstruktúra).

elképzelhetők *transzformációs* függvények is, amelyek a teljes rekordstruktúrát érintik.

Relációk: = (mezőnkénti), ≠ .

Például:

```
Konstans  ffi : Egész (1)
          nő : Egész (2)
Típus     Dátum = ...
          Kontroll = Függvény(ffi..nő,Dátum,0..999 : 0..9)
          Szemszám = Rekord
                     ( nem : ff..nő
                       szülidő : Dátum
                       sorszám : 0..999
                       ellenőr : Kontroll(nem, szülidő,sorszám)
                     )
          Személy = Rekord
                     ( szsz : Szemszám
                       név : Szöveg
                       Alternatívák
                         nem=nő  esetén lnév Szöveg
                         nem=ffi esetén katsz : Egész
                       Alternatívák vége)
Konstans  Jézus : Személy((1,                    a nem mező = ffi;
                           0.12.25,              az szsz mező tartalma;
                           123),                 az ellenőr mező automatikusan
                                                 töltődik föl;
                           'Jézus Krisztus',     a név és
                           (123456789)           a katsz mező tartalma;
                          )
```

4.3. (Hatvány)Halmaz-típuskonstrukció

Strukturálás: *Halmaztípus* = **Halmaz**(*Elemtípus*)[12]

Értékhalmaz: az alaphalmaz (amely az *Elemtípus* által van meghatározva) iteráltja („mely elemek vannak benne a halmazban").

Műveletek: * (metszet), + (egyesítés), - (különbség), *Üres* (üres halmaz létrehozása: eljárás), vagy *Üres'Halmaztípus* előre definiált konstans, *Üres*? (logikai értékű függvény).

Relációk: = , < , ≤ , ≥ , > , ≠ (parciális rendezés: a tartalmazás alapján).

Például:

```
Típus     Nap= 0 .. 23
          Foglalt  = Halmaz(Nap)
Változó   ma,holnap: Foglalt
Konstans  munkanap : Foglalt (7..12,14..20)
…
Üres(ma)  [ma „szabad"]  vagy  ma:=Üres'Foglalt
Ha Üres?(holnap) akkor ma:=munkanap
…
```

[12] Mivel a „halmazság" tulajdonság nem teszi szükségessé az elemeinek közvetlen kiválaszthatóságát (emiatt *halmaz*, s nem *tömb*!), ezért nincs is komponenseire vonatkozó *szelekciós* függvény, s így az „értelmezési tartomány" megadásától a definícióban el is tekintünk.

4.4. Tömb-típuskonstrukció

Struktúrálás: *Tömbtípus* = **Tömb**(*Indextípus* : *Értéktípus*)

Értékhalmaz: az alaphalmaz „*index*terjedelemszeres" hatványa (*index*terjedelemnyi *Értéktípus*ú elemek „egyesített" halmaza).

Műveletek: szelekciós függvény, amellyel az egyes elemek kiválaszthatók: '(.)', ahol a '.' egy *Indextípusú* kifejezést jelöl.

Relációk: = (tömbelemenkénti egyezés), ≠ .

Például:

```
Konstans  n: Egész(99)
Típus     Elem= Valós
          Vektor = Tömb(1..n : Elem)
Változó   v: Vektor
          i: 1..n
Be: v
Ciklus i=2-től n-1-ig
  Ha T(v(i)) akkor v(i):=(v(i-1)+v(i+1))/2
Ciklus vége
```

4.5. Szövegtípus

Strukturálás: annak ellenére, hogy egy nyilvánvalóan összetett típusról van szó, nincs értelme *strukturálás*ról beszélni, mivel ez teljesen kötött, alig ad valami szabadságot a programozónak; csak annyit, hogy jogában áll a maximális *karakterszámot* előre meghatározni:

Szövegtípus = **Szöveg**(*MaxHossz*) .

Értékhalmaz: *MaxHossz*nyi karaktersorozatok halmaza (karaktertípus *MaxHossz*-szoros direktszorzata).

Műveletek: *Hossz*, +, *Balrész*, *Jobbrész*, *Jele* (adottadik jele), *Üres* ("" = szövegkonstans)[13].

Relációk: = , < , ≤ , ≥ , > , ≠ (alfabetikus rendezés).

Például:

```
Típus     Név  = Szöveg(30)
Konstans  UtolsóUtáni: Név("zzzzzzzzzzzzzzzzzzzzzzzzzzzzzz")
Változó   névsor  : Tömb(1..40: Név)
          i,j,db  : 1..40
...
Ciklus i=1-től db-1-ig
  első:=névsor(i) : elsőind:=i
  Ciklus j=i+1-től db-ig
    Ha első>névsor(j) akkor első:=névsor(j) : elsőind:=j
  Ciklus vége
  Csere(i,elsőind)
Ciklus vége
...
```

[13] Implementációnként eltérő műveletkészletet definiálnak.

5. Adatok leírása a programban

Tekintsük át végezetül az adatok leírását egy az egyben!

5.1. Típusdefiníció *(típusmegadás)*

Cél egy „egész" adatosztály (*adatuniverzum*) megtervezése: értékhalmazostul (*adatállapotok halmaza*: *mozgástér*), operációstul (*adatállapot-változást okozó leképezések*: *mozgástörvény*ek). A definíció csak az osztályt hozza létre, üresen (az univerzum még „kihalt").

Olyan esetben, amelyben előre definiált „kész" *típus*okra, *típuskonstrukció*kra építjük az új típust, megelégedve az „eleve meglévő lehetőségekkel", igen egyszerűen elintézhető a dolog:

Például:

```
Típus  Alszín = ( piros, zöld, kék )
       Dátum  = ?
       Kontroll = Függvény(1..2,dátum, 0..999 : 0..9)
       Színek = Halmaz(Alszín)
       Index  = 1..maxN
       TömbN  = Tömb(Index:Színek)
       Szemsz = Rekord
          ( nem : 1..2
            szülidő : Dátum
            sorszám : 0..999
            ellenőr : Kontroll(nem,szülidő,sorszám)
          )
```

5.2. Konstansok megadása

A cél a már meglévő adatosztályba konstans adatokat elhelyezni (az univerzumot „állócsillagokkal benépesíteni").

Például:

```
Konstans  fekete : Színek () [üres halmaz]
          fehér  : Színek (piros, zöld, kék)
          maxN: Egész  (1000)
```

5.3. Változók deklarálása

Az előzőhöz hasonlatosan itt is újabb és újabb elemek kerülnek az adatok absztrakt keretébe, de ezek kifejezetten „mozgékonyságra", változékonyságra teremtődnek (ezek lesznek az univerzumunk „bolygói", melyek mozgását szabályozzák a definiált mozgástörvények).

Megjegyzés: A kezdőérték (kezdőállapot) a konstansoknál megszokott módon adható, de nem kötelező. Ha nem rendelünk kezdőértéket hozzájuk, akkor vagy az ún. nemdefiniált értékkel töltődnek föl, vagy a típushoz definiált iniciális értékkel.

A típus-, a konstans-, illetve a változómegadások felváltva követhetik egymást. Egy töretlen megadási sorozat előtt elegendő csak *egyszer* közölni, hogy milyen megadási sorozat jön. Ennek akkor van értelme, ha a programot viszonylagosan független részekből (*pszeudomodul*okból) építjük föl, s ilyenkor célszerű az önálló részeket elkülönítetten felépíteni. Ez szokott előfordulni, ha egy típus már „majdnem" kész, csak apróbb kiegészítésekre szorul; ebben az esetben nem készítünk teljesen új moduldefiníciót.

Például:

```
Konstans  ...
          ...
Típus     ...
          ...
Változó   ...
          ...
... eljárások, függvények ...
```

1. „pszeudomodul"

```
Konstans  ...
          ...
Típus     ...
          ...
Változó   ...
          ...
... eljárások, függvények ...
```

2. „pszeudomodul"

VI. Algoritmusleíró eszközök

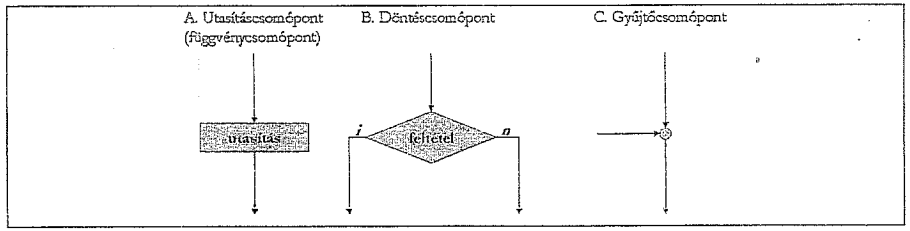
Az algoritmusleíró eszközök használatának célja a feladatok megoldásának leírása programozási nyelvtől független nyelven. A programozási nyelvek ugyanis szigorú szintaxisúak, a tervezés szempontjából lényegtelen sallangokat tartalmaznak. A programozási nyelven történő tervezés esetén nehézzé válhat a program átírása más nyelvre, más gépre.

Mindegyik algoritmusleíró eszközzel leírjuk egy feladat megoldását: N tanuló évvégi átlagának ismeretében adjuk meg a jeles átlagú tanulók számát!

1. Folyamatábra

Az egyik legkorábban kialakult algoritmusleíró eszköz a programot *gráf*ként írja le. A programgráf egy irányított gráf, amely csomópontokból és őket összekötő élekből áll, egyetlen induló és befejező éle van, az induló élből bármely csomópont elérhető, s bármely csomópontból el lehet jutni a befejező élre.

A folyamatábra alapesetben háromféle csomópontot tartalmaz:



1. ábra. Folyamatábra-szimbólumok.

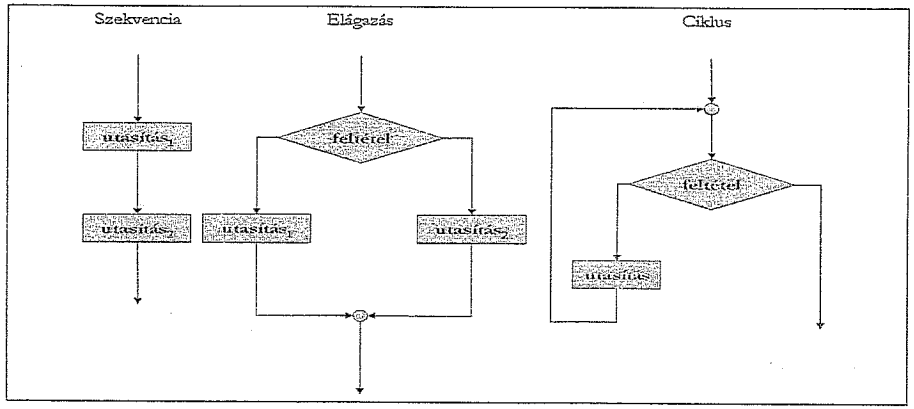
Az utasításcsomóponton áthaladva végre kell hajtani a bele írt utasítást. A döntéscsomópontban levő feltétel igaz értéke esetén az **i** betűvel jelölt élen, hamis értéke esetén pedig az **n** betűvel jelölt élen kell továbbhaladni. A gyűjtőcsomóponton való áthaladás nem változtatja meg a program állapotát.

A program e háromfajta csomópontból felépített gráf, bemenő éle a „semmiből" jön, kimenő éle a „semmibe" megy. A szokásos algoritmikus struktúrák ezekből a közvetkezőképpen építhetők fel (csak a struktúrát jelöljük, a tartalmat nem). L. a 2. ábrát.

Ez az algoritmusleíró eszköz jól használható az algoritmusok végrehajtásának követésére, hiszen a végrehajtás a programgráf csomópontjainak bejárása az élek mentén.

Ezen előnnyel szemben azonban több súlyos hátrányos jellemzővel rendelkezik. Terjedelmes, szövegszerkesztővel nehezen készíthető, javítása rendkívül nehézkes. Nagy programok leírása könnyen áttekinthetetlen ábrákhoz vezethet (nem fér ki egy lapra, a gráf élei többszörösen keresztezik egymást stb.). Alapproblémája, hogy strukturális alapelemei nem azonosak a szokásos algoritmikus szerkezetekkel, sőt segítségükkel más struktúrák is létrehozhatók. Ezen más struktúrák veszélyességével, illetve használatuknak szükségtelenségével ebben a füzetben nem foglalkozunk, érdeklődő Olvasóinknak ajánljuk Varga László: *Programok analízise és szintézise* című

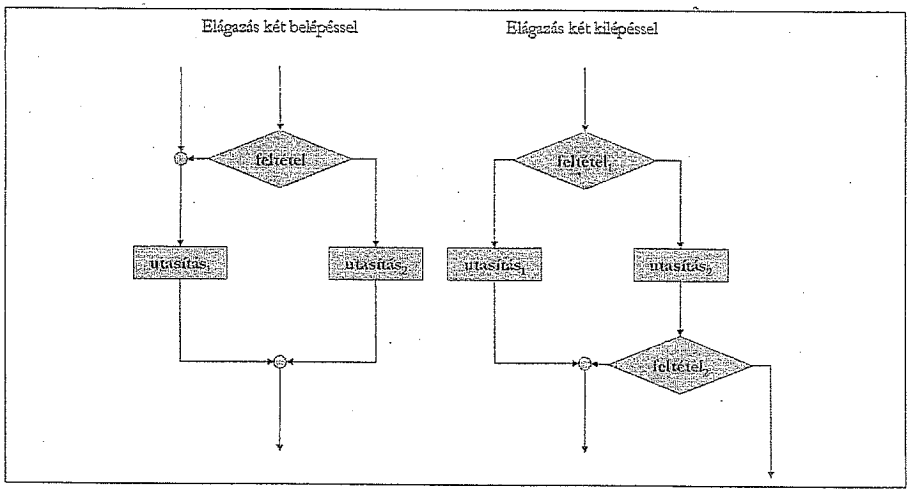
könyvét.



2. ábra. Elemi strukturált programok folyamatábra-szimbólumokkal.

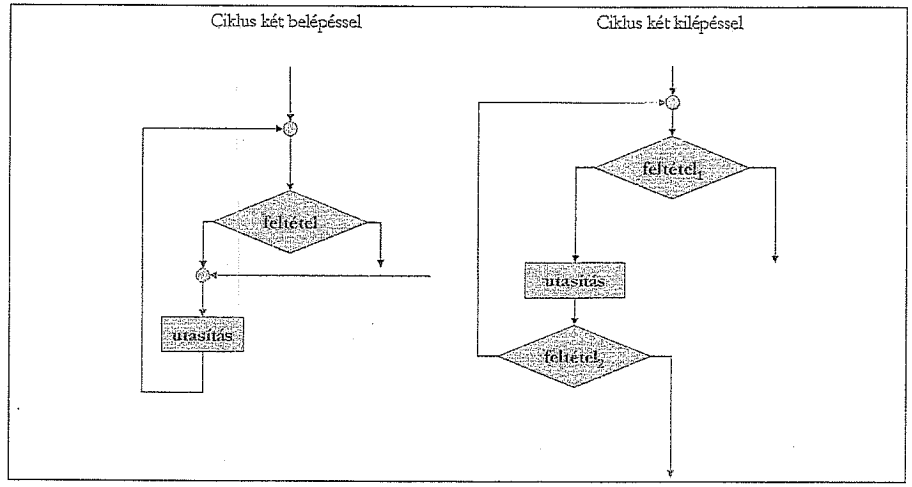
Definíció: **Strukturált programnak** nevezzük azt a programot, amely csak a fenti hárôm algoritmikus szerkezetet (szekvencia, elágazás, ciklus) tartalmazza.[1]

Vannak olyan programok, amelyek nem csak ezeket az ún. strukturált alapszerkezeteket tartalmazzák, hanem másokat. E mások a következőfélék lehetnek:



3. ábra. Nem strukturált alapszerkezetek – elágazások.

[1] Mi csak ilyenekkel foglalkozunk.

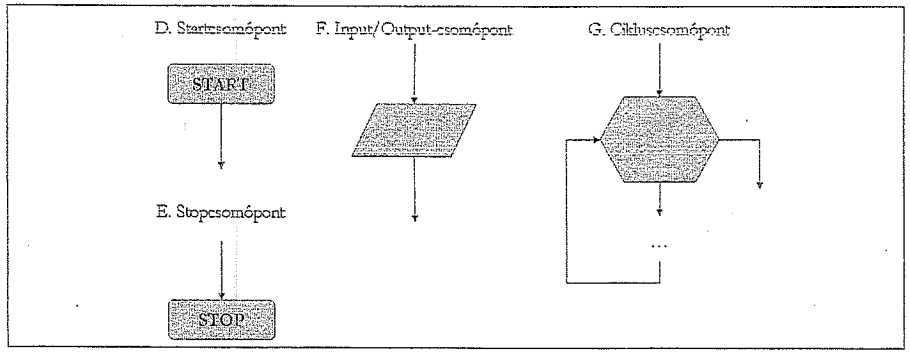


4. ábra. Nem strukturált alapszerkezetek – ciklusok.

Mind a négy nem strukturált alapszerkezetről látható, hogy bonyolultabbak a strukturált alapszerkezeteknél, alkalmazásuk nehezebb, bonyolultabb programot eredményez.[2]

Szigorú szabályok önkéntes betartásával persze ezt az eszközt is lehet használni strukturált programok írására, de a sok más lehetőség túlságosan csábító lehet, s az ettől való eltérésért sokszor nagyon nagy árat kell fizetni.

E minimális csomóponkészletet a folyamatábráknál néhány újabb elemmel szokták bővíteni - ezek a leírást nem teszik lényegileg bővebbé, csupán kényelmességi, olvashatósági szempontok miatt definiálták őket. Ezek közül néhány, a korábban elkezdett betűjelölést folytatva:



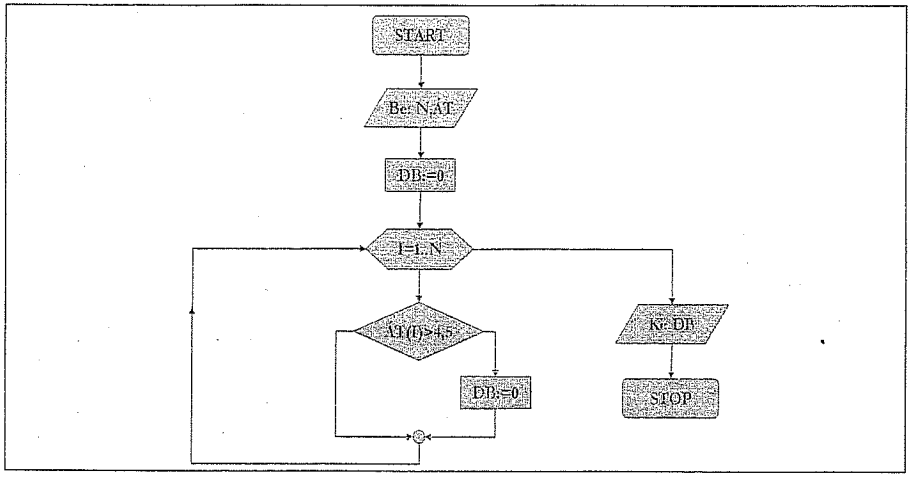
5. ábra. További folyamatábra-szimbólumok.

Start és Stop csomópont a programban egyetlen lehet, az eddigi „semmiből" jövő, illetve „sem-

[2] Az előbb javasolt könyvben megtalálhatjuk annak bizonyítását, hogy nincs is rájuk szükség.

mibe" menő élhez helyezzük el őket. Az input-output csomópontba beolvasást, illetve kiírást tehetünk, jelölve, hogy éppen melyikről van is szó. A ciklus csomópontot számlálós ciklusoknál alkalmazzuk, a csomópontban szerepelnie kell a ciklusváltozónak, a ciklusváltozó kezdeti-, valamint végértékének.

A kitűzött példa ezen eszközökkel készült megoldását láthatjuk a következő ábrán.



6. ábra. Példaprogram folyamatábrával.

2. Struktogram

Ez az eszköz az előző hibáit próbálja kiküszöbölni azzal, hogy a programgráfot élek nélkül ábrázolja. Így egyetlen egy alapelem marad, a téglalap:

7. ábra. A struktogram alapeleme.

Ezzel az alapelemmel építhetjük fel a szokásos strukturált alapszerkezeteket (és csak azokat).

8. ábra. A struktogram összetett alapszerkezetei.

Szekvenciánál a téglalapok egymás alatti sorrendje dönti el a végrehajtás sorrendjét. Az elágazás-feltétel igaz értéke esetén az **i** betűvel jelölt baloldali téglalap utasítását kell végrehajtani, hamis értéke esetén pedig az **n** betűvel jelölt jobboldali téglalapét. Ha az elágazás valamelyik ága üres, akkor a neki megfelelő téglalap is üres marad. A ciklus elöltesztelős, azaz a benne levő utasítást mindaddig végre kell hajtani, amíg a feltétel igaz.

Az utasítások helyén lehet egyetlen elemi utasítás, lehet a három algoritmikus szerkezet valamelyike és lehet egy eljáráshívás. Ezt a leíró eszközt még többféle elemmel szokták bővíteni: az eljárásdefinícióval, a sokirányú elágazással, illetve a hátultesztelős ciklussal:

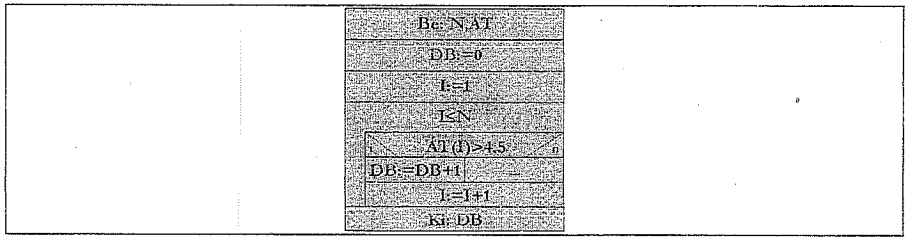
9. ábra. A struktogram további összetett alapszerkezetei.

Sokirányú elágazásnál azt az ágat kell végrehajtani, amelynek igaz értékű a feltétele (közülük minden esetben pontosan egy teljesülhet).

Ez az eszköz egyértelműen csak strukturált programok írására alkalmas, így kiküszöböli az előző hiányosságainak egy részét. Megmarad azonban a „rajzosság" miatt a terjedelmesség, nehéz javíthatóság. Veszítünk viszont a gráf éleinek elhagyása miatt a követhetőségből.

A lokális adatokat az eljárások téglalapjai mellett, az eljárásnév után sorolhatjuk fel.

Nézzük meg ezzel az eszközzel leírva az előző példát!



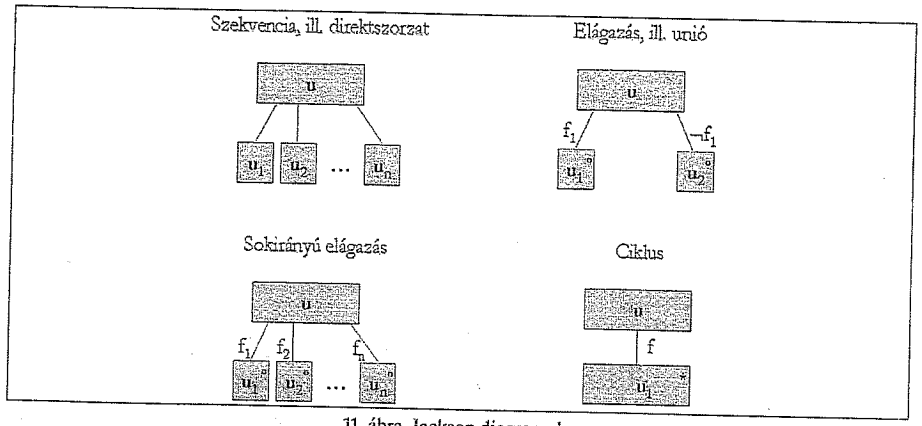
10. ábra. Példaprogram struktogrammal.

3. Jackson diagramok

A harmadik rajzos eszköz az adat- és az algoritmikus szerkezetek leírására egységes ábrakészletet definiál. Ahogyan az algoritmikus alapszerkezetek a szekvencia, az elágazás és a ciklus, úgy a nekik megfelelő adatszerkezetek a direktszorzat, az unió, illetve a sokaság. Ezeknek a Jackson diagramban háromféle struktúra felel meg. (L. a 11. ábrán.)

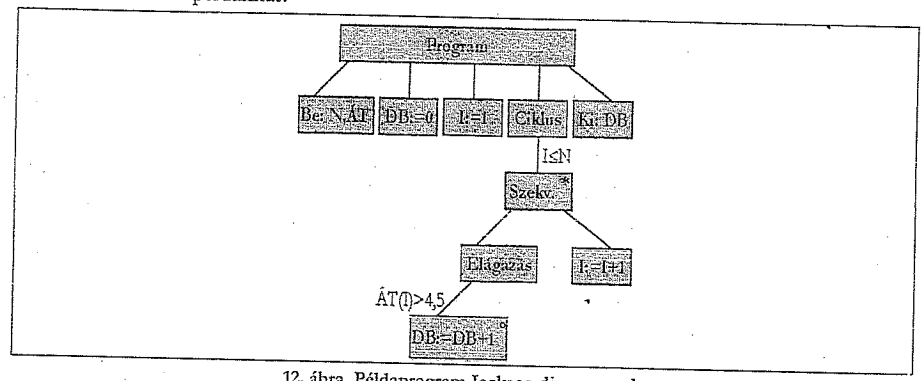
Szekvencia esetén az utasításokat balról jobbra haladva kell végrehajtani, a kétféle elágazásnál az igaz feltételű, jobb felső sarkában kis karikával (o) jelölt téglalap tartalmát, ciklusnál pedig mindaddig, amíg a feltétel teljesül, a jobb felső sarkában csillaggal (*) jelöltét.

Az új eszköz egyértelmű előnye az algoritmus- és adatleírás egységessége, áttekinthetősége azonban az eddigieknél is rosszabb. A szekvenciát itt azonnal, mint sok utasítás szekvenciáját definiáltuk, a többi alapelem a szokásossal megegyező.



11. ábra. Jackson diagramok.

Nézzük ezzel is a példánkat!



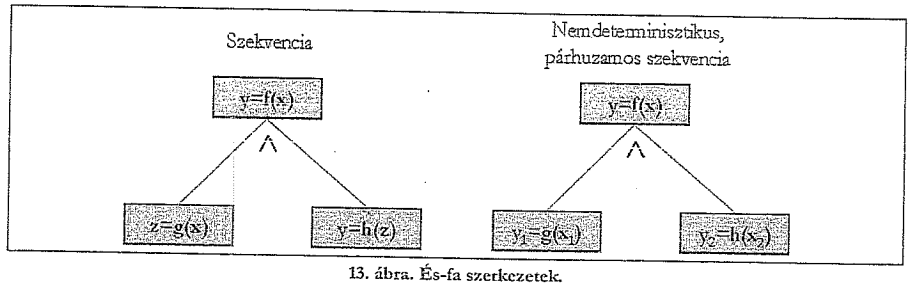
12. ábra. Példaprogram Jackson diagrammal.

A példában szereplő Program, Ciklus, Szekv., Elágazás elnevezésű csomópontok helyett konkrét feladatokban valami „értelmes" elnevezést célszerű alkalmazni.

4. Leírás fával

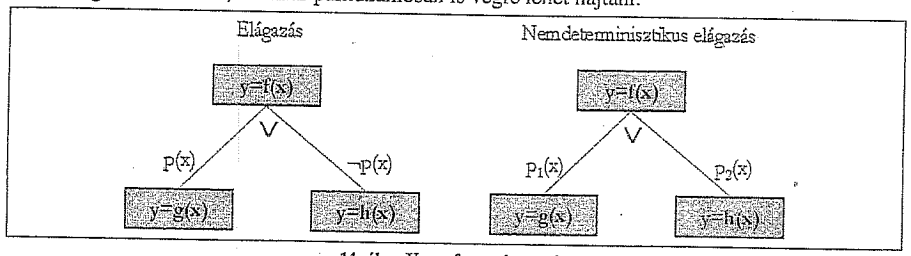
Ez a leíróeszköz nagyon hasonló az előzőhöz, a programgráfot **és-vagy-fával** írja le. A fa elágazásai **és-típusúak**, ha mindegyik ágat végre kell hajtani; **vagy-típusúak**, ha az ágak egyikét kell végrehajtani. A fa ciklust nem tartalmaz, a ciklusokat rekurzívan írhatjuk le. Az eszköz jellemzője, hogy alkalmas nemdeterminisztikus, illetve párhuzamos algoritmusok leírására is.

A fa csomópontjaiban ennél a módszernél mindig a szükséges input→output leképezés függvényformája szerepel, a fában lefelé haladva egyre inkább konkrétabb függvényekkel. Nézzük végig a szokásos algoritmikus szerkezeteket ezzel az eszközzel.



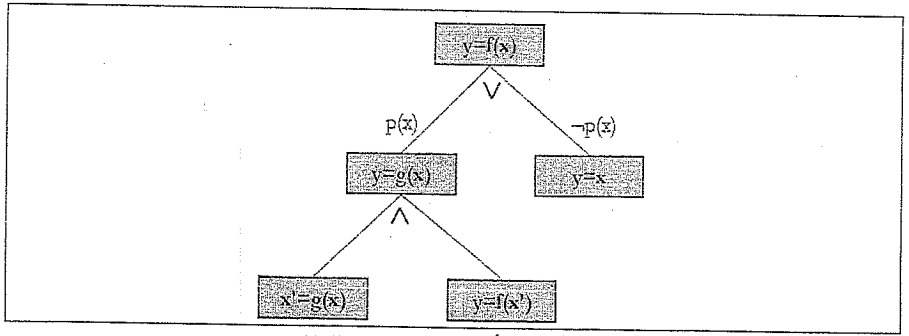
13. ábra. És-fa szerkezetek.

Mindkét fa lehet többágú is. A felírási formájuk teljesen egyforma, a determinisztikus sorrendet csak a leképezések input-output hozzárendelése rögzíti a baloldali fában. A jobboldali ágait tetszőleges sorrendben, sőt akár párhuzamosan is végre lehet hajtani.



14. ábra. Vagy-fa szerkezetek.

A jobboldali fa lehet többágú is. A determinisztikus esetben a feltételek kötelezően kizárólag, s közülük az igaz feltételű ág hajtandó végre, a jobboldali, nemdeterminisztikus elágazásban egyszerre több feltétel is lehet igaz értékű, s ilyenkor a nekik megfelelő ágak közül bármelyik végrehajtható.



15. ábra. Példaprogram És-Vagy-fával.

Itt az x' változó bevezetése jelenti a ciklusmag előbbi végrehajtását, s utána az y=f(x) leképezés újbóli végrehajtását.

Beolvasás, illetve kiírás ennél az eszköznél nem létezik, helyette az input→output leképezést kell

megadnunk. Elemi tevékenységként pedig a függvénydefiníció helyetti értékadás jel (:=) szerepel.

Ez az eszköz nyilvánvalóan szélesebb alkalmazási lehetőségű az eddigieknél, áttekinthetőnek azonban semmiképpen nem mondható.

5. Leírás programozási nyelven

A gráfot alkalmazó, rajzos algoritmusleíró eszközök után áttérünk a szöveges eszközökre. Az egyik legelső szöveges algoritmusleíró nyelv egy egyszerű, de mégis általános célú programozási nyelv volt: az ALGOL60. Gyakorlati célú programozási nyelvként viszonylag szűkebb körben terjedt el, algoritmusleíró nyelvként azonban nagyon sok, a 60-as, 70-es években megjelent könyvben használták. Ugyanezt a célt szolgálja napjainkban az egyszerűsített Pascal, az ELAN stb.

A bevezetőben említett negatív hatások mellett a programozási nyelv használatának még egy rossz tulajdonsága van: a szigorú szabályain könnyíteni kell, ha jó algoritmusleíró nyelvként szeretnénk használni, de ezt a könnyítést nehéz meghatározni, ráadásul kódolásnál csak zavarokat okozhat.

6. Leírás mondatokkal

Sorszámozott utasítású programozási nyelvekhez (mint a BASIC, az assembly nyelvek, esetleg a FORTRAN) illeszkedik ez az eszköz. Utasításai sorszámozott mondatok.

A mondatok leírhatnak értékadást, beolvasást, kiírást, adott sorszámú mondattal folyatatást, valamint feltételes szerkezetet.

```
Értékadás: 5. Legyen I értéke I+1!
Beolvasás: 6. Olvasd be a K-t!
Kiírás:    7. Írd ki I*I-t!
Folytatás: 8. Folytasd a 15. soron!
Feltétel:  9. Ha I<N akkor ... ← itt az előző négy sorbeli bárme- lyike
                                 lehet.
```

Ennél az eszköznél a követést a sorszámok biztosítják. Az így leírt programnak azonban semmi szerkezete nem fog látszani, áttekinthetetlen, olvashatatlan lesz.

Szinte semmi előnye és rengeteg hátránya van. Csak a történelmi szerepe miatt írtuk róla e néhány sort.

Elrettentés kedvéért nézzük azonban meg ezzel az eszközzel is a példánkat!

```
1. Olvasd be N-t és ÁT-ot!
2. Legyen DB értéke 0!
3. Legyen I értéke 1!
4. Ha I>N, akkor Folytasd a 10. sornál!
5. Ha ÁT(1)>4,5, akkor Folytasd a 7. sornál!
6. Folytasd a 8. sornál!
7. Legyen DN értéke DB+1!
8. Legyen I értéke I+1!
```



```
9. Folytasd a 4. sornál!
10. Írd ki DB értékét!
11. Vége.
```

7. Leírás mondatszerű elemekkel

Ez az az eszköz, amely e füzetben, s a sorozat összes többi tagjában is szerepel. Az előző részekben az adatok és az algoritmikus elemek megismerésekor ezt használtuk, emiatt úgy gondoljuk, hogy újbóli részletes ismertetésére itt már nincs szükség.

Elemei nem teljes mondatok, hanem mondatszerű elemek (befejezetlen, hiányos mondatok).

Alapvető előnye, hogy struktúrái megfelelnek a szokásos Neumann-elvű programozási nyelvi struktúráknak, ezért kódolása rendkívül egyszerű. Egyik legalkalmasabb például az ELAN nyelv, ahol a kódolása gyakorlatilag angolra fordítást jelent.

Csak a strukturált programozás alapszerkezeteit tartalmazza, így ettől eltérni nem lehetséges. Megfelelő programkészítési elvek (l. e füzet utolsó részét – Programkészítési elvek) használata esetén egy optimális algoritmusleíró eszközt kapunk.

Bevezető példánk ezzel az eszközzel:

```
Program:
  Be: N, ÁT  [N természetes szám, I≤ÁT(I)≤N ∀I=1..N]
  DB:=0
  Ciklus I=1-től N-ig
    Ha ÁT(I)>4,5 akkor DB:=DB+1
  Ciklus vége
  Ki: DB
Program vége.
```

8. Leírás absztrakt függvényekkel

A szöveges algoritmusleíró eszközök után áttérünk a matematikai eszközökkel való algoritmusleírásra. Egy ilyen eszközzel fogunk megismerkedni, amely formulákkal írja le a programot. A legegyszerűbb formulák a szokásos algoritmikus szerkezetekhez tartoznak:

```
Szekvencia:  P=SEQ(Q,R)
Elágazás:    P=IF(f;Q,R)
Ciklus:      P=DO(f;Q), illetve P=UNTIL(Q;f)
```

A kisbetűs jelölés logikai formulát, a nagybetűs pedig utasítást jelent. Az utasításokat a formulán belül vessző, a feltételeket az utasításoktól pontosvessző választja el. Szekvencián belül kettőnél több utasítás is lehet.

Kicsit bonyolultabbak a *Programozási tételek* című füzetben szereplő típusalgoritmusok formulái. Nézzünk meg ezek közül néhányat (a típusparamétereket dőlt betűkkel jelöljük):

Az Elemtípus típusú elemeket tartalmazó, N elemű X vektorban van-e t tulajdonságú elem:

```
P=Eldöntés(Elemtípus;t;N,X,VAN)
```

Az Elemtípus típusú elemeket tartalmazó, N elemű X vektor maximális értékű eleme sorszámának a meghatározása:

`P=Maximumkiválasztás(Elemtípus;N,X,MAX)`

Az Elemtípus típusú elemeket tartalmazó, N elemű X vektor t tulajdonságú elemei sorszámainak kigyűjtése a DB elemű Y vektorba:

`P=Kiválogatás(Elemtípus;t;N,X,DB,Y)`

Nyilvánvaló, hogy ez az eszköz sem eredményez áttekinthető programot, de ennek nem is az a célja. Elsősorban arra a célra szolgál, hogy automatikusan, programmal elemezzünk algoritmusokat, s matematikai eszközökkel állapítsuk meg jellemzőiket.

Nézzük meg ezzel az eszközzel szokásos példánkat!

```
Program=SEQ( Be: N, Be: ÁT, DB:=0, I:=1,
             DO( I≤N; SEQ( IF( ÁT(I)>4.5; DB:=DB+1, ),
                           I:=I+1)),
             Ki: DB )
```


VII. Dokumentálás

A program egy **termék**, s egy terméknek mindig rendelkezni kell különböző leírásokkal.

Mire is lehet szükség egy program kapcsán? Először is egy leendő*felhasználó*nak el kell döntenie, hogy milyen programot, akar használni. Ha a programot megvette, akkor el szeretné helyezni a számítógépén, majd használni szeretné, s a felhasználásban segítséget vár.

Nemcsak a felhasználónak van szüksége dokumentációra, hanem a *fejlesztő*nek, karbantartónak is (nem véletlenül adnak például háztartási készülékeinkhez műszaki leírást is).

Nyilvánvaló, hogy ez a két- vagy többfajta dokumentáció másoknak szól, így nem egy egységes dokumentációról fogunk beszélni, hanem többfajta dokumentumról.

1. A dokumentáció fajtái

1.1. Fejlesztői dokumentáció

Minden egyes dokumentumnál fel kell tennünk azt a kérdést, hogy *„Kinek szól?"*. Az e kérdésre adott válasz egyértelműen meghatározza, hogy az ilyen dokumentációban minek kell szerepelnie.

A fejlesztői dokumentációt használja az, akinek a programban hibát kell keresnie, a hibát ki kell javítania, a programot hatékonyabbra kell írnia, át kell vinnie más gépre, át kell írnia más nyelvre, valamint tovább kell fejlesztenie.

Ezen emberek munkájának megkönnyítése érdekében a fejlesztői dokumentációban szerepeljen:

- *Specifikáció*k, követelményanalízis (korlátozások, pl. megkívánt hatékonysági jellemzők, alkalmazandó adatszerkezetek). A feladat és a megoldástól elvárt követelmények meghatározása. Ezt még a feladat kitűzője adta, vagy vele történt megbeszélés során pontosodott a megoldás első lépéseként.
- *Futási környezet leírása*: számítógép, operációs rendszer, memóriaméret, (speciális) perifériaigény, grafikus kártya (felbontóképesség, színek száma ...), ...
- *Fejlesztői környezet leírása*: a választott *programnyelv*(ek), és verziószáma(i); eljárás-könyvtárak, unit-ok (azaz a szükséges „programdarabok" file-jai).
- Az *algoritmus*ok és az *adat*ok (típusok, osztályok, program-konstansok) leírása, ezek kapcsolata. Döntések, más alternatívák, érvek, magyarázatok.
- *Kód*, implementációs szabványok (ún. *kódolási szabályok*; egyéni konvenciók), döntések.
- *Tesztese*tek, azaz milyen (jellegzetes) bemeneti adatokra, milyen eredménnyel „válaszol" a program.
- *Hatékonyság*i mérések (hatékonysági tesztesetek), megfontolások, javaslatok az esetleges hatékonyabbra írásra.
- *Fejlesztés*i lehetőségek.
- A *készítő* adatai.

A programkészítés közben keletkezhetnek olyan dokumentumok, amelyek az elkészítést támogatják, majd lényegük később beépülhet a fejlesztői dokumentációba. Ezek többek között a kö-

vetkezők:

Koncepcióterv

Feladata a probléma lehetséges megoldásainak bemutatása és értékelése. Felhasználásával a feladat kitűzője választhat megoldási lehetőségek között.

Rendszerterv

Tartalmaznia kell a szükséges hardver-szoftver környezet leírását, amely meghatározza a rendszer használhatósági körét. Itt konkrét döntéseket kell megfogalmazni, következményeikkel, indoklásukkal együtt. Ez az a dokumentum, amelynek alapján a tényleges programozási munka elkezdődhet.

1.2. Felhasználói dokumentáció

Ezt a dokumentumot használja a felhasználó, az üzembehelyező, a betanító.

Nekik szükségük van a következőkre:

- A *feladat*. Egy rövid összefoglaló leírás is kell az áttekintés miatt és egy részletes a pontos használathoz.
- *Futási környezet leírása*: számítógép, operációs rendszer, memóriaméret, perifériaigény, grafikus kártya, ... [1]
- A *használat* leírása. Hogyan kell a programot betölteni/elindítani, milyen kérdéseket tesz fel, mik a lehetséges válaszok, mik a program egyes lépései, lehetőségei. (Nagyvonalú funkcionális leírás.)
- *Bemenő adatok, eredmények*, szolgáltatások részletes leírása, mit, mikor, milyen sorrendben kell megadni. (Részletes funkcionális leírás.)
- *Mintaalkalmazás* – példafutás. A felhasználó –főleg a betanító– ez alapján tudja előre –gép nélkül– „elképzelni" a programot.
- *Hibaüzenetek* és a hibák lehetséges okai. Mi a teendő valamely hibaüzenet láttán.

Látható ebből, hogy a felhasználói és a fejlesztői dokumentáció több közös jellemzőt tartalmaz.

1.3. Programismertető

A programismertető célja a vásárló, programkereső ember meggyőzése arról, hogy e program felel meg leginkább igényeinek. Ez a hangzatos, reklám jellegű stílus mellett a következőket igényli:

- A *feladat rövid*, szöveges leírása, áttekintési céllal.
- A *program tulajdonságai*nak rövid leírása.
- Minimális *hardver és szoftver* (operációs rendszer és esetlegesen megkívánt egyéb, a programmal együtt nem „szállított" szoftver kellékek, pl. driverek, dll-ek stb.) környezet.

1.4. Installálási kézikönyv, operátori kézikönyv

Nagyobb programok esetén külön *installálási* (üzembehelyezési) kézikönyvet mellékelnek, más-

[1] Megegyezik a fejlesztői dokumentáció ugyanilyen című részével.

kor ez a felhasználói dokumentáció része. Ebben szerepel mindaz az információ, aminek segítségével egy-több generálólemezről a program elhelyezhető gépünkön úgy, hogy az aktuális környezetben optimálisan működjön. (Ügyelni kell arra, hogy az installáció minél kevesebb számítástechnikai ismerettel végrehajtható legyen! Fel lehet használni olyan célprogramokat, amelyeket kifejezetten erre terveztek, de legalább egy batch-programmal automatizálni kell. Ilyen automatizmus esetén is dokumentálni kell a installációs folyamat lépéseit.)

Az *operátori kézikönyv* olyan rendszereknél különül el a felhasználói kézikönyvtől, ahol más a program felhasználója és más a kezelője.

2. A dokumentáció tulajdonságai

2.1. Szerkezet

A dokumentáció elsődleges célja segítségnyújtás a program leendő felhasználóinak, továbbfejlesztőinek. Ezért olyannak kell lennie, hogy minden számukra szükséges tudnivalóhoz *könnyen hozzájuthassanak*. Ehhez elsődleges szempont természetesen, hogy a dokumentáció mindezeket tartalmazza, de a használatát egyéb követelmények betartásával jelentősen megkönnyíthetjük. Ezek a következők:

- A dokumentáció *ne legyen túl hosszú*, hiszen egy program használatához senki sem akar egy „regényt" elolvasni.
- A dokumentáció *ne legyen túl rövid*, mert akkor tömörsége miatt érthetetlen lesz, s így használhatatlan.
- A dokumentáció *legyen világosan tagolt*, s a tagolás segítse elő az egyes tudnivalók gyors keresését.
- A dokumentáció *legyen tömör*: az Olvasója ne vesszen el a részletekben.
- A dokumentáció *legyen olvasható*: a túlságos (és kizárólagos) formalizálás az érthetőség rovására megy.
- A dokumentáció *legyen pontos*: Olvasója minden kérdésére tartalmazza a választ.

2.2. Forma

A dokumentáció használatát néhány formai jellemző nagyban megkönnyítheti. Ezek egyike a *tartalomjegyzék*. Másik, dokumentációkban ennél ritkábban használt, de néha kifejezetten nagy segítséget nyújtó eszköz: az *index*.

Az nyilvánvaló, hogy világos szerkezetű kell legyen: kitalálható mondanivalójú fejezetekre bontva. További stílusjegyek megegyeznek bármely szakmai kiadványéval.

2.3. Stílus

Végezetül essen néhány szó az egyes dokumentációk stílusáról! A programismertető egyértelműen a reklámcélra szolgál. Itt dicsérni kell a programot, kiemelve jó tulajdonságait.

A felhasználói dokumentáció elsősorban részletes szöveges leírás, amely időnként lehet „szájbarágó" is. (Célszerű figyelembe venni a várható felhasználói kört a leírás részletességének, a

szájbarágosság szintjének megtervezésénél.[2])

A fejlesztői dokumentációban minden más szempontnál fontosabb a pontosság, emiatt ebben kerülhet elő a matematikai leírás, a formális specifikáció.

Az installálási, illetve az operátori kézikönyv elsősorban utasítások, teendők pontos felsorolása, utalva a lehetséges válaszok következményeire.

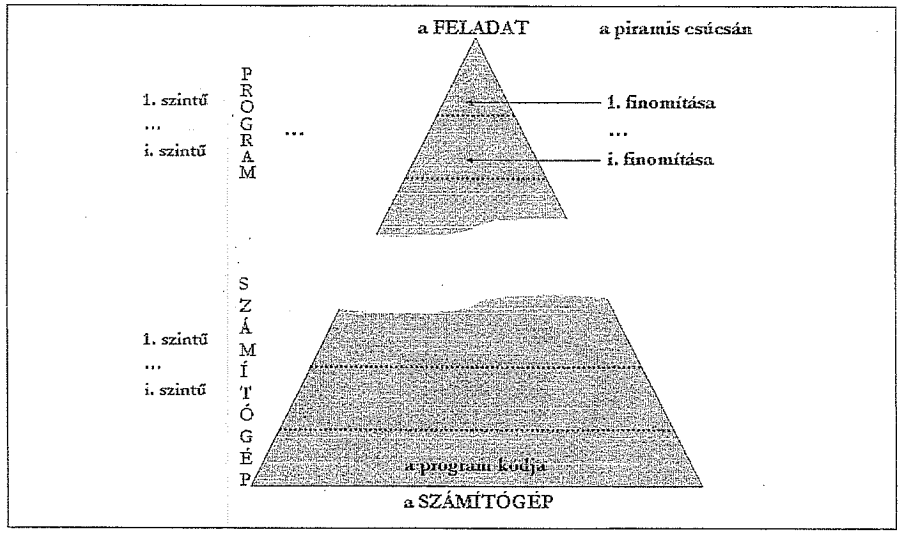
[2] Pl. teljességgel fölösleges egy Windows környezetben futó program esetében az ablakkezelésről általánosságban szólni.

VIII. Programkészítési elvek

1. Stratégiai elv

Egyik legfontosabb, sokféleképpen alkalmazható elvünk az ókori latin kultúrából ránk maradt *oszd meg és uralkodj* elve alapján fogalmazható meg: *oszd részekre*, majd a részek független megoldásával az egész feladatot könnyebben oldhatod meg. Így programod könnyen kézben tarthatod, vagyis *uralkodhatsz* felette.

Ezt a **stratégiai elv**et tartjuk szem előtt akkor, amikor gondolkodásmódunkat kívánjuk helyes mederbe terelni. *Lépésenkénti finomítás*nak nevezik ezt az elvet a feladatmegoldás filozófiájában. A feladat megoldását először átfogóan végezzük el, nem törődve a részletekkel, amelyekről érdemben amúgy sem tudnánk helyesen dönteni az adott pillanatban sok, még pontosan előre nem látható probléma miatt. Tehát a feladatot néhány (=nem túl sok!) részfeladatra bontjuk. Úgy is mondhatnánk: a feladatot megoldjuk a legfelső szinten. Ha volna olyan gép, amelyen léteznének azok az utasítások, amiket mi részfeladatokként megadunk, akkor máris futtatható lenne a program.[1] Ezt az eljárást fogjuk követni az egyes részfeladatok megoldásakor is (a *részek feletti uralkodás* érdekében), mindaddig, amíg olyan utasítások szintjéig nem érünk, amelyeket gépünk (kódolás után) már végre tud hajtani. (Piramis elv)

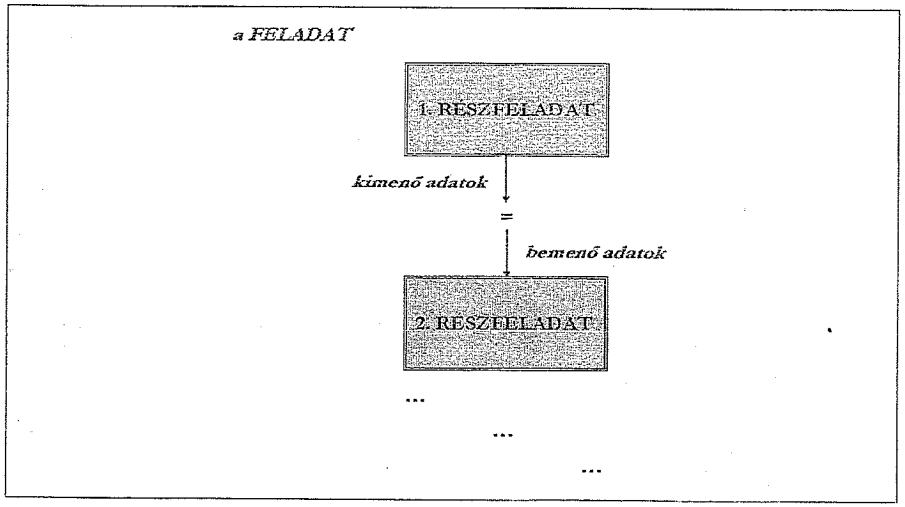


1. ábra. A 'Piramis elv'.

[1] Dijkstra-féle gyöngysor modell.

A részfeladatokra bontás a következőket foglalja magában: pontosan ki kell jelölni, hogy az adott részművelet milyen adatokat kezel, milyeneket állít elő, és ezeket miként kell egymáshoz rendelni; vagyis ha *ez* az adat jött a részművelethez, akkor *azt* az adatot kell kapjuk eredményként. A *hogyan*ról most nem elmélkedünk.

Nyilvánvaló, hogy két azonos szinten definiált részfeladat között biztosítani kell a harmóniát úgy, hogy a végrehajtásban előbb következő az utána következő adatait szolgáltassa.



2. ábra. Részfeladatokra bontás: a piramis egy szintje („=" helyett esetleg: „C").

A részfeladatokra bontáskor persze felmerülhetnek illesztési problémák.

Az elmondottakból már körvonalazódik a *lépésenként* elnevezés értelme. Óvatosan, megfontoltan haladunk az ismeretlenbe; vigyázunk, hogy minden lépésnél megtartsuk uralmunkat az éppen megoldandó részfeladat felett. A *finomítás* pedig e lépésenkénti megközelítés miként jére utal. A lépés során kirajzolódott elemibb, egymástól már jól elhatárolható, s ezért egymástól függetlenül kezelhető[2] tevékenységek még elemibbekre bontását jelenti.

Ez a módszer tehát a **program felülről lefelé való kifejtése** (top-down programozás), amely a problémaanalizáláson, dekomponáláson, részekre osztáson alapul.

Elképzelhető egy másik módszer is, amelyben a piramist alulról kezdve építjük fel. Itt azonban nagy gyakorlat szükséges ahhoz, hogy a csúcsra érve tényleg a kitűzött feladatot oldjuk meg.

Ez utóbbi módszer a **program alulról felfelé való felépítése** (bottom-up programozás), amely szintetizáláson alapul.

[2] Ezek a tevékenységek függetlenül kezelhetők, hiszen az egymáshoz való viszonyukat az illeszkedéseik megadásakor már pontosan figyelembe vettük.

2. Taktikai elvek

Milyen taktikai elveket –vagy kissé szerényebben szólva, jó tanácsokat– adhatunk a *lépésenkénti finomítás* elvének megvalósításához?

A. A párhuzamos finomítás elve

A szint összes részfeladatára kell elvégezni a finomítást. Nem szabad előre sietni valamelyik könnyebbnek vélt ágon, mert előfordulhat, hogy munkánk kárba vész (egy esetleges visszalépés miatt, s kár lenne ilyen lélektani terhet nyakunkba venni).

Egy-egy szinthez szervesen hozzátartoznak a részfeladatok adatai is. Így világos, hogy az adatok finomítása során sem szaladhatunk előbbre, mint amit a szint eljárásai megkívánnak.

A szint eljárásai –mint a piramis elvből nyilvánvaló– a feladat teljes megoldását adják. Ha lenne olyan gép, amely ezeket az utasításokat végre tudná hajtani, akkor készen is lennénk. Ha nincs ilyen gép, akkor –legalább– egy szinttel lejjebb kell lépnünk.

B. A döntések elhalasztásának elve

Egyszerre csak kevés dologról, de következetesen kell rendelkezni. A problémát kevés, de jól körülhatárolt részproblémára kell bontani. Ám óvakodni kell a másik véglettől is, hiszen a túl kevés részre bontás általában nem vezet optimális döntésekhez, mert nem azonos nehézségű, súlyú részproblémákat eredményez.

A részfeladatok pontos körvonalazásához eleinte még nincs elegendő ismeret birtokunkban, s ha döntésünket később felül kell bíráljuk, még visszalépések is adódhatnak. Az a jó döntés, amely a későbbiek során a *legkevésbé köti meg kezünket.*

Célszerű az adatok és eljárások finomításánál minél későbbre halasztani azokat a döntéseket, amelyek kihasználják a gép, illetve a programozási nyelv konkrét sajátosságait. Annál jobb a programunk, minél többet tartalmaz a feladat lényegi felépítéséből és minél kevesebbet a gép, illetve a nyelv kötöttségeiből.

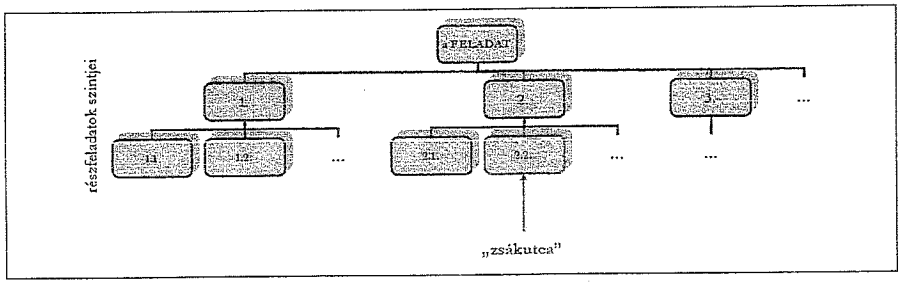
Ugyanezen elv alapján célszerű a bonyolult döntéseket későbbre hagyni, az algoritmus finomítása során ugyanis nagy valószínűséggel ezek egyszerűbbé válnak.

C. Vissza az ősökhöz elv

Erre akkor van szükségünk, amikor körültekintő megfontolásaink ellenére zsákutcába kerülünk. Ekkor vissza kell lépni az előző szinthez (az őshöz), és újra végig kell gondolni a részfeladatokra bontást, a keserű tapasztalatok figyelembevételével.

Vegyük példának a 3. ábra által leírt szituációt! Bármennyire is csábító lenne a zsákutcából a 2. részfeladat újra átgondolásával, feladatának újra meghatározásával kijutni, elkerülhetetlen a 2.-at tartalmazó minden részfeladatát (az 1.-t és a 3.-at ... is) újra megfontolni, hogy a beépített elem hézagmentesen illeszkedjen.

A visszalépés nem hiba, csupán egy előre nem látható hatású döntés következménye. A visszalépések e programkészítési folyamatban természetesek.



3. ábra. 'Vissza az ősökhöz' elv alkalmazása.

D. A nyílt rendszer felépítés elve

Nemcsak a **feladat**ra, hanem a feladatot is tartalmazó **feladatkör**re alkalmazható programot érdemes definiálni. Ezzel elkerülhetjük a későbbi kényszerű feladat-, vagy programidomítási gondokat, azaz nem egy „nyárra" hozunk létre programot. (Szokás ezt még a *feladat általánosítás* elveként is emlegetni.)

Itt tehát nagyobb szabadságot kell adnunk az előfeltételben, több lehetséges eredményt az utófeltételben, csak az a kérdés, hogy meddig érdemes elmenni?

Az általánosításnak nyilvánvalóan vannak hatékonyságrontó vonatkozásai, valamint a programozási munkát növelők is. Az előbbi kérdésre a válasz nyilvánvalóan az, hogy addig, amíg a befektetett munka nem növekszik meg túlságosan, s a kapott eredmény még megfelelő hatékonyságú lesz.

E. A döntések kinyilvánításának elve

A ki nem mondott, de hallgatólagosan meghozott döntések rugalmatlanná és „álnokká" teszik a programot. Szorosan összefügg azzal, hogy a fejlesztői dokumentációt a program tervezésével, írásával párhuzamosan kell készíteni. (Ebben úgy írjuk le a programot, hogy működését más is megértse, szükség esetén módosítsa.)

F. Az adatok elszigetelésének elve

Ez az elv azt mondja ki, hogy a programot biztos mederben csak úgy lehet tartani, ha az egyes programegységek a megtervezett illesztéseken, kapcsolatokon kívül egymással nem beszélhetnek. Ezért tehát a programegységekhez tartozó adatokat ki kell jelölni, és el kell szigetelni más programegységektől.

Az adatok kijelölését legcélszerűbb a programegységben betöltött szerepük alapján csoportosítani: **közös** (pontosabban fogalmazva: **globális**), ezen belül **bemeneti (input)**, és **kimeneti (output)** adatok.

A programegységhez és csak hozzá tartozó, saját (pontosabban fogalmazva: **lokális**) adatok, melyektől –mint később látni fogjuk– a jó helykihasználás miatt hasznos elválasztani az ún. **munka**adatokat (igazából ezeket szokták a programozók **saját**nak nevezni). Ez utóbbiak a programegységnek csak egyszeri működéséhez, időlegesen felhasznált segédadatok.

Így a következő ábrához hasonló táblázatot (az úgy nevezett: 'kereszthivatkotás- vagy kereszt-referencia-tábla') építhetünk föl a program létrehozása közben. A táblázat tartalmazza az adat

nevét, azt a szintszámot vagy eljárásnevet, amelyben létrehoztuk, és a hatáskörét (az egész szinthez vagy csak a szint egy eljárásához tartozik).

Adat	Szint	Hatáskör
sor_számláló	lapozás	munka
lap_számláló	lapozás	be-kimenet
...	...	...

Megjegyzés: E táblázatot praktikusan még további oszlopokkal bővíthetjük, úgy fogja igazán jól betölteni szerepét. Vegyük észre, hogy nem nehéz olyan programot készíteni, amivel ez a tábla automatikusan generálható (az érintett program forrás kódjából).

G. A párhuzamos ágak függetlenségének elve

Szintenként az egyes részfeladatok között semmilyen vezérlési, illetve adatforgalom nem lehetséges, mindezt az eggyel magasabb szint feladata megoldani. Tehát egy szinten –sőt különböző szinteken, de független kifejtési ágakon– levő eljárások egymást nem hívhatják, egymás változóit nem is láthatják.

H. Szintenkénti teljes kifejtés elve

Minden szint a probléma teljes megoldását adja egy olyan absztrakt gépen, amelynek utasításai a szinten nem definiált, primitív eljárások, valamint elemi tevékenységek. Emiatt a szint leírásának már tartalmaznia kell a majdani alatta levő szint eljárásainak specifikációját is.

A résztevékenységek interface-iknek illeszkedniük kell egymáshoz: $Kimenet_i$+$Előfeltétel_i \subseteq Bemenet_{i+1}$+$Utófeltétel_{i+1}$!

3. Technológiai elvek

A programírás már ecsetelt didaktikai elveitől a **technológiai elvek** vezetnek el bennünket a kóddal kapcsolatos **technikai elvek** birodalmához. Ezek az elvek az *algoritmus* (és a kód) írására, annak *szabályaira* vonatkoznak.

A. Algoritmusleírási szabályok

Kevés, de egyértelmű szabályt kell kialakítani az algoritmusok leírására. Ez legyen számunkra kellően kényelmes gondolataink szárnyalásához, de a kényelem nem mehet a precizitás, az egyértelműség rovására! Ebben a legfontosabb algoritmikus szerkezeteknek helyet kell kapniuk.

Melyek is ezek a nevezetes szerkezetek?

Az adatokat *beolvasó* és *kiíró utasítások* az „ablak" szerepét játsszák a külvilág felől, illetve a program felhasználója felé.

A program változóinak értékkel való ellátását az *értékadó utasítások* végzik.

A feltételektől függő végrehajtást teszik lehetővé az ún. *feltételes utasítások, elágazások.*

A számítógépre szánt feladatok mindegyike feltételezi bizonyos részfeladatok ismételt elvégzését. A számítógép erősségét: a gyorsaságot éppen ezek a „mechanikus" ismétlések használják ki a legjobban! Ezek megvalósítása a *ciklus utasítások* segítségével történik.

A program adott szintjén elemi utasításként felhasznált, meghatározott, de nem finomított rész-

programok (*eljárások, függvények, operátorok)* beépítését (az ún. eljáráshívást) is meg kell oldanunk nyelvünkben.

Természetesen a felhasznált és még hiányzó *eljárások finomítása* (másként szólva: az eljárás kifejtése) sem hiányozhat.

Az algoritmusleírás mellett e nyelvek rendelkeznie kell az *adatok* (*konstansok, változók*) és *típusok* leírására szolgáló eszközökkel is.

B. Értelmes sorokra tördelés – világos tagolás

Kérdés, hogy mit írjunk egy sorba, mit több sorba. Alapelképzelésünk lehetne például az, hogy minden utasítást külön sorba kell tenni. Ezt a következőképpen módosítjuk: kerüljenek egy sorba azok az utasítások, amelyek szervesen összekapcsolhatók, s egy sorba írásukkal a program még áttekinthető marad.

C. Bekezdéses leírás

Ahogy az író is főbb gondolatait, a történés főbb eseményeit fejezetekbe tömöríti, ahogy az egyes epizódok külön-külön bekezdéseket alkotnak a fejezeten belül, valahogy úgy kell algoritmikus gondolatainkat, az algoritmus főbb eseményeit, epizódjait is jól láthatóan elkülöníteni a programban. A program teljes levezetése (finomítása) után a program szerkezetének vissza kell tükröznie a szintekre tagozódást: egy szint elemi utasításai a bekezdések azonos szintjeit alkossák!

Egyes nyelvi szövegszerkesztők automatikusan a bekezdéses leírásnak megfelelően tördelik programunkat.

D. Összetett struktúrák zárójelezése

Az algoritmusokban szereplő *elágazások, ciklusok, eljárások*, valamint az *összetett adatstruktúrák* úgy ismerhetők fel könnyen, ha nem csak az elejüket jelzi valamilyen nyelvi elem, hanem a végüket is egyértelműen rögzítjük.

E. A „beszédes" azonosítók elve

A *konstansoknak, változóknak, típusoknak, eljárásoknak, függvényeknek, operátoroknak* olyan nevet érdemes adni, ami utal arra, hogy mire használjuk. Ez kizárja az azonosítók keveredését: hiszen a név sugallja funkciót, az algoritmusban betöltött szerepet. Nagy segítséget nyújt a kódoláskor is, pl. lehetővé teszi, hogy minimális számú változót rendeljünk az adatokhoz, hiszen a munkaváltozókhoz azonos neveket is rendelhetünk.

Nem minden esetben a hosszú azonosítók a *beszédesek*, például ha egy fizikai képlettel (E= m*g*h) dolgozunk, akkor éppen ezek az egybetűs jelölések a beszédesek, ha pedig mátrixösszeadásra definiálunk egy operátort, akkor azt célszerű a + jellel jelölni.

4. Technikai elvek

A továbbiakban technikai jellegű ismérveket sorolunk fel, amelyek a *program kód*jával kapcsolatosak. Inkább úgy mondhatjuk, hogy az előzőek a program megírásához szükségesek, ez utóbbiak pedig a program használhatóságához elengedhetetlenek. Ilyen értelemben beszélhetünk a „csak" **helyes** programról, amely a feladat logikája szempontjából tökéletes, és a **jó** programról, amely ezen túl elő is segíti saját felhasználását.

A. Barátságosság, udvariasság

Az udvarias program bemutatkozással kezdi ténykedését (tájékoztató), s ezzel tudatja a felhasználójával képességeit, szolgáltatásait, használatának mikéntjét. Az udvariasság másik fontos megnyilvánulása, hogy a program futása során megjelenő kérdések bárki számára –azaz a nem számítástechnikus szakemberek számára is– érthetők, és a válaszok a lehető legegyszerűbben megadhatók legyenek. Így pl. nem rabolja feleslegesen a felhasználó türelmét azzal, hogy a már megadott adatokból kiszámítható, származtatható adatokat kér.

B. Biztonságosság

A „bolond-biztos" program, amit a kísérletezni vágyó, vagy éppen balszerencsés felhasználó sem képes ellenőrizetlen vágányokra terelni azáltal, hogy nem a megfelelő módon, vagy nem a megfelelő pillanatban válaszol a feltett kérdésére. Ennek érdekében a program kritikus pontjait, azaz ahol a felhasználó közvetlenül avatkozik be a program további menetébe, nagy odafigyeléssel kell megírni. Az esetleges hibalehetőségekre fel kell készíteni a programot úgy, hogy a felhasználónak lehetősége legyen a helyesbítésre is. (Itt használjuk ki a specifikáció előfeltétel részében leírtakat.)

Nem támaszkodhatunk a számítógép, illetve az értelmező vagy fordítóprogram eleve meglevő hibajelzéseire. Ezek ugyanis arra valók , hogy segítségükkel felderíthessük és kijavíthassuk az esetleges programhibákat, tehát a program írója, nem pedig a használója számára készültek. A felhasználónak végeredményben semmi köze sincs ahhoz, hogy a program nem közvetlenül, hanem valamilyen programozási nyelv közvetítésével, bábáskodásával mozgatja a számítógépet.

A 'bolond-biztos' tulajdonság egyébként nemcsak e technikai elv, hanem a **piramis elv** közvetlen folyamánya is, mert ahogy megadtuk az egyes eljárások közötti elvárásainkat (ti. egy eljárás végső feltételei nem lehetnek bővebbek, mint az őt követő eljárás bemeneti feltételei), pontosan ugyanúgy kell rögzíteni a „külvilág" és a beolvasó eljárás kapcsolatára vonatkozó kívánalmakat is. A külvilág azonban a programnak nem része, s így nem is programozható, ezért a bemeneti eljárások kezdő feltételeinek körét megfelelően ki kell szélesíteni. Ez azt jelenti, hogy a bemenő adatok értéktartományát, összeférhetőségét ellenőrizni kell. Ha ezek az eredeti feltételeket nem teljesítik, akkor jelezni kell, majd új adatbevitelt kell kezdeményezni.

C. Jól olvasható program

A program módosításakor, továbbfejlesztésekor óriási előnyt jelent, ha nem kell a programunk minden mellékes vonását újra feltérképezni a megértéshez, hanem a lényeges tulajdonságai a program megfelelő helyén könnyen kiolvasható formában megtalálhatók, s így a sebész magabiztos mozdulatával nyúlhatunk bele a program legérzékenyebb részeibe is.

Már két idevágó elvet is említettünk, a **bekezdéses leírás** és az **összetett utasítások zárójelezése** elveket. Ezt egészíthetjük ki a kódoláskor különösen nagy jelentőségűvé váló '**jó magyarázatok** (kommentek)' **elvé**vel. A programozási nyelvre való áttéréskor ugyanis –a programozási nyelv kötöttségei miatt– sok, az algoritmust nagyban jellemző tulajdonság elveszne, ha ezeket az információkat nem őriznénk meg egy-egy jól megfogalmazott megjegyzés formájában.

D. A (jól) dokumentált program

Sokszor nincs lehetőség –a program méretére rótt korlátozások miatt– arra, hogy az előző elvet maradéktalanul megvalósíthassuk, ekkor le kell írni a program fontos vonásait: az algoritmusát (felépítését), a változóit és ezek szerepét, értelmezését, értéktartományát, hatáskörét stb., a kódo-

lásnál követett szabályokat (a leíró és a programozási nyelv utasításainak, illetve változóinak megfeleltetését). Ezeket a dokumentációban is rögzíteni kell, amelyben ezen kívül még foglalkozni kell a használat mikéntjével, és az esetleges, előrelátható fejlesztési lehetőségekkel is.

5. Esztétikai-ergonómiai elvek

A következőkben a már ismertetett udvariasságra és bolondbiztosságra vonatkozó elv finomításairól, a program **emberközelségéről** lesz szó. Ezek a használatot befolyásoló tényezőkre hívják föl a figyelmet. Legfőbb mondanivalójuk, hogy nagy gondot kell fordítani a program által megjelenített információk külalakjára. Ide nemcsak az eredmény jellegű kiírandók tartoznak, hanem pl. a tájékoztató, a felhasználóval való párbeszéd módja is. (L. még a µlógia 21. Kódolási technikák fejezetét.)

A. Lapkezelési technika

A kiírandó szövegek, adatok logikai egységekre bontva, jól különüljenek el, egyszerre csak annyi és olyan ütemezésben, amennyit s ahogy a felhasználó be tud fogadni. Ennek megvalósítására szokták alkalmazni a lapkezelés technikáját.

Mit is jelent a lapkezelés? Egyszerre egy képernyőlapnyi információt jelenítünk meg, s a felhasználónak lehetősége van lapozásra, például egy adott billentyű lenyomásával jelzi a gépnek: „Elolvastam! Lapozhatsz!". (Többek között ennek megvalósítására használható a '**Várj amíg szükséges**' utasítás.) Nem szerencsés ez esetben az adott ideig történő várakozás – gondoljunk a különböző olvasási sebességű felhasználókra!

Nyomtató esetén e várakozásra nincs szükség, viszont újdonságként felmerülhet a lapszámozás, illetve a fejléc vagy lábléc írása.

Képernyőkezelés esetén is lehetőséget kell teremtenünk arra, hogy az aktuális képernyőtartalmat kinyomtathassuk.

Mit kell még figyelembe venni egy lap összeállításánál? Ügyelni kell a **képernyőlap arányos kitöltésére**, és jó, ha az egy lapon belül szereplő, logikailag szorosan össze nem tartozó információk egymástól elkülönülnek. Az elkülönítés megoldható üres sorok beiktatásával, az egyes részek szakaszokkal való elkülönítésével, illetve bekeretezésével.

A mondanivalónk legfontosabb elemeit –a gép adta lehetőségek figyelembevételével– kiemeljük (inverz betűkkel, vagy bekeretezve, vagy más színű háttérrel, illetve betűkkel stb.)

B. Menütechnika

A lapkezeléssel szorosan összefüggő módszer, amely a felhasználóval való párbeszéd elegáns megszervezésére alkalmas. Általában bonyolult szolgáltatásokkal rendelkező programoknál használatos, amelyből a felhasználó –akár egy menüből– kiválaszthatja a számára szükséges lehetőséget.

Minden egyes válaszával (válaszcsoporttal) a kérdések egy nagy hányadát kizárja, ezeket a számítógépnek fel sem kell tennie, megkímélve a felhasználót a fölösleges válaszadásoktól. (Hierarchikus menürendszer.)

A **menü** egy lap (vagy ablaka), amelyen megjelennek a választási lehetőségek, amelyek közül sorszámmal (vagy kezdőbetűvel), illetve rámutatással (kurzormozgató billentyűk vagy egér segítségével) választhatunk.

A program főmenüjében célszerűen szerepel egy *Munka befejezése* menüpont, a többi menüpont végrehajtása után pedig újra e főmenü jelenik meg. Az egyes almenük hasonló elven épülhetnek fel, de ezekben a befejezés helyett a *Vissza az előző menühöz* pont választható.

C. Ikontechnika

A szöveges menüknél esetenként gyorsabban felismerhetők az egyes választási lehetőségek, ha azokat kicsi jellemző ábrával, ún. ikonnal jelenítjük meg. ezek közül rámutatással (kurzormozgató billentyűk vagy egér segítségével) választhatunk.

Ez a technika azonban könnyen veszélyessé válhat: a túl sok és túl kicsi ikon a képet áttekinthetetlenné teheti.

D. Értelmezési tartomány kijelzése

A kérdéseknél nagyon sokszor épp az okoz bizonytalanságot, hogy a felhasználónak fogalma sincs arról, hogy az adatot milyen mértékegységben kell megadni. Ezért a kérdés szövege mellett célszerű közölni az **adat mértékegységét,** sőt –ha nem magától értetődő, akkor– még az **értéktartományt** is.

Így elkerülhető, hogy pl. a program egy szöget radiánban vár, a gyanútlan felhasználó pedig a legnagyobb természetességgel fokban adja meg az értéket. Az ilyesmiből származó hibát nyilván nem kell ecsetelnünk.

E. A fontos adatok kiemelése

Nem csak az információk könnyebb megértése szempontjából van jelentősége –amint ezt már a **laptechnikánál** taglaltuk–, hanem hasznos a program állapotának, meghatározó paramétereinek azonnali visszajelzésekor is.

Például, amikor a számítógép egy hosszadalmas számítást végez, vagy bármilyen időigényes tevékenységbe fog –a hangsúly a hosszadalmasságon, időigényességen van!–, akkor ne maradjon el időnként egy-egy kiírás, ami értesíti a felhasználót, hogy mely tevékenységgel bíbelődik éppen a program, és hogy még kis türelmet kér. Látványos lehet ilyen esetekben közölni azt –esetleg grafikus formában is–, hogy a feldolgozás hány százalékánál tart éppen a program.

F. Tördelés

A legelemibb elvárás a képernyőn megjelenő szövegekkel szemben, hogy a sorok/szavak tördelése a helyesírás szabályainak megfeleljen. Ne sajnálja a programozó a fáradságot mondanivalójának gördülékeny megfogalmazására, szép elhelyezésére, hiszen csak ily módon kaphat mindenki számára kellemes programot!

G. Következetesség

Következetes beolvasási és kiírási szokások is fontosak. Tartsunk mértékletességet a beolvasási módszerek változatosságában. Nem díjazzák a felhasználók kiterjedt programozási ismereteinket, ha a választ hol <ENTER>-rel lezárva, hol anélkül várja a program.

Hasonló probléma az IGEN-NEM választ igénylő kérdések sokféle feldolgozási lehetősége, válasszunk egyfajtát, s ahhoz ragaszkodjunk.

Ha lehetőségünk van rá, akkor a **lapkezelési technikához** kapcsolódva az azonos jellegű kérdések, illetve eredményadatok a lapok azonos helyein jelenjenek meg.

H. A hibajelzés követelményei

A hibák kézben tartásának szükségességéről már volt szó, de a hibák jelzésének mikéntje is jellemzi a programot. Igyekezni kell a hibajelzés legmegfelelőbb módjának kiválasztására. Ehhez a következő szempontokat érdemes megfontolni:

- **A hibajelzés ideje**. Hibát akkor kell jelezni, amikor bekövetkezett, nem pedig valamely következményekor! Tipikusan „bosszantó" lehet több száz adat beolvasása után olyan üzenetet látni, hogy kezdjük elölről a begépelést, mert a legelső rossz volt.
- **A hibajelzés „látvány-környezete"**. Ha a kezelői hiba javítása után folytatható a végrehajtás, akkor a képernyőlátványt vissza kell-e, illetve vissza lehet-e állítani?
- **A hibajelzés időtartama**. Mindig a felhasználó dönthessen a továbbhaladásról! Kerüljük az adott időtartamig megjelenő hibajelzéseket, a felhasználó ugyanis lehet lassúbb vagy gyorsabb, esetleg éppen nem a képernyőre figyel, s így elmulaszthatja a hibajelzést.
- **A hibajelzés mozgósító ereje**. Biztosan észlelhető legyen, érthető legyen, azaz ne legyen túl rövid –csak a program írója ismeri a „megfejtést", esetleg külön búvárkodást igényel–, túl hosszú –az ember az ilyenkor természetes türelmetlensége miatt csak hevenyészve képes végigszaladni a leírt „regényen"–, várható felhasználói számára érthető szakkifejezéseket tartalmazzon.

Felesleges azonban abban az esetben külön hibajelzés szöveget kiírni, amikor a kérdés szövegéből egyértelmű, hogy a felhasználó mit rontott el. Ekkor elég például egy hangjelzés, majd a kérdés újrafeltevése.

I. Naplózás

A program futása során több olyan *esemény* következhet be, amelyeket jó feljegyezni a későbbi esetleges feldolgozás érdekében. A felhasználó –ha ott is ül a számítógép mellett– nem biztos hogy megteszi ezeket. Ennek megoldására szolgál az ilyen események automatikus file-ba írása, a naplózás. E file többnyire egy egyszerű szerkezetű szöveges file, amit a használó könnyen (egy „igénytelen" szövegszerkesztővel is) képes megjeleníteni, nyomtatni.

J. Makrók, funkcióbillentyűk

Érdemes lehet egyes funkciókhoz, funkciócsoportokhoz egy-egy billentyűt hozzárendelni, s annak bármikori lenyomása a megfelelő funkciók végrehajtását jelenti.

Például szimulációs programokban gyakran találkozunk olyan funkcióbillentyűkkel, amelyek a szimuláció leállítására, újraparaméterezésére, megjelenítésmódjának változtatására, részleges összesítések elkészítésére stb. vonatkoznak.

K. Segítség

Egy tipikus funkcióbillentyű a segítség (HELP) billentyű. Ennek lenyomása a futás bármely pillanatában a program aktuális állapotáról szükséges tudnivalók kiírását eredményezze.

Ennek egy hasznos formája a menüben mozgás alatti segítség, amely az aktuális menüpont részletes leírását adja a felhasználó kívánságára.

L. Ablaktechnika

A homogén képernyő helyett célszerű olyan lapokat, ún. ablakokat használni, amelyek a képernyő elkülönített részein jelennek meg.

Egy ablak mindig egy keret, és egy a belsejében levő tartalom. Az ablak kiírásakor a képernyőn alatta lévő részt eltakarja, s levételekor újra megjelenik az eltakart rész.

Ablakokat használhatunk a segítségszöveg megjelenítésére, hibajelzésre, menük kezelésére, a program állapotának kijelzésére stb.

Irodalomjegyzék

1. O.J.Dahl–E.W.Dijkstra–C.A.R.Hoare: „*Strukturált programozás*", Műszaki Könyvkiadó, 1978
2. Varga L.: „*Programok analízise és szintézise*", Akadémiai Kiadó, 1981
3. Fóthi Á.: „*Bevezetés a programozáshoz*", Tankönyvkiadó, egyetemi jegyzet, 1983
4. Szlávi P.–Zsakó L.: „A programozás ABC-je = az ABC' programozása", ELTE TTK Számítástechnikai Tanszék, ABC-s füzetek, 1984
5. Szlávi P.–Zsakó L.: „*Módszeres programozás*", Műszaki Könyvkiadó, 1986
6. Szlávi P.– Zsakó L. et al.: „*Számítástechnika középfokon*"., OMIKK, 1987
7. Hanák P.:"Programozás ELANnal", Műszaki Könyvkiadó, 1988
8. Varga L.: „*A programozási módszertan elmélete. I-II*", Tankönyvkiadó, 1989-90
9. Aszalós J.–Erki I.: „*Bevezetés a strukturált programozásba*", SZÁMOK, 1980
10. Hvorecky J.–Kelemen J.: „*Ötlettől az algoritmusig*", Tankönyvkiadó, 1987
11. C.H.A.Koster: „*Programozás felülnézetben*", Műszaki Könyvkiadó, 1988