0405 888/01

INFORMATIKA

KIRÁLY SÁNDOR

A PROGRAMOZÁS LOGIKÁJA I.

17

INFORMATIKA 17.

KIRÁLY SÁNDOR

PROGRAMOZÁS LOGIKÁJA I.

KÖZÉPISKOLAI TANKÖNYV

KÍSÉRLETI TANKÖNYV

SOROZATSZERKESZTŐ:

DOMBOVÁRI MÁTYÁS

LEKTOR:

DR. NÉMET ISTVÁN
ELTE KÍSÉRLETI GYAKORLÓ GIMNÁZIUM ÉS
SZAKKÖZÉPISKOLA

FELELŐS SZERKESZTŐ:

SZALAY ÁGNES

ISBN 963 7309 16 0

A KIADÁSÉRT FELELŐS: A GRADUATION BT ÜGYVEZETŐJE.
MŰSZAKI SZERKESZTŐ: BELEZNAINÉ S. ANNAMÁRIA
NYOMDA: TERCIA GMK., BUDAPEST

1995.

KÉSZÜLT A VILÁGBANKI PROGRAM KERETÉBEN,
A MŰVELŐDÉSI ÉS KÖZOKTATÁSI MINISZTÉRIUM ÉS
A KÖZISMERETI INFORMATIKA CSOPORT
TARTALMI GONDOZÁSÁBAN.

A TANKÖNYVVEL KAPCSOLATOS ÉSZREVÉTELEKET
SZÍVESEN FOGADJUK.

GRADUATION BT 2045 TÖRÖKBÁLINT, PF. 85.

TARTALOMJEGYZÉK

BEVEZETÉS ---------- *9*

I. A FELADATMEGOLDÁS LÉPÉSEI ÉS MÓDSZEREI ---------- *10*

1. A feladat meghatározása ---------- **10**

2. A feladatmegoldás lépései ---------- **11**

II. A PROGRAM ELEMEI, FELÉPÍTÉSE ---------- *15*

Karakter ---------- **15**

Többkarakteres szimbólumok ---------- **16**

Lexikai egységek ---------- **16**

Szimbolikus név - azonosító ---------- 16
Szimbolikus név - kulcsszavak, fenntartott szavak ---------- 16
Szimbolikus név - standard azonosítók ---------- 17
Címke ---------- 17
Megjegyzés ---------- 17
Változó ---------- 18
Nevesített konstans ---------- 21
Tipizált konstans ---------- 21
Konstans ---------- 22

Szintaktikai egységek - kifejezések ---------- **23**

Kifejezések kiértékelése ---------- 23

Utasítások ---------- **27**

Ugró utasítások ---------- 27
I/O utasítások ---------- 27
Feltételes utasítások ---------- 27
Üres utasítás ---------- 28
Ciklusok ---------- 28
1. Feltételes ciklus ---------- 29
2. Üres ciklus fogalma (terminológia) ---------- 29
3. Végtelen ciklus fogalma (terminológia, tárgyalása később) ---------- 30
4. Előírt lépésszámú ciklus: ---------- 30
5. Felsorolásos ciklus ---------- 32
6. Végtelen ciklus ---------- 32
7. Középfeltételes ciklus ---------- 33
8. Összetett ciklusok ---------- 33

III. PROGRAMEGYSÉGEK ---------- *34*

Az alprogram ---------- **34**

Blokk (a programegységek másik, speciális formája) ---------- **40**

IV. ADATTÍPUSOK, ADATSZERKEZETEK ---------- *42*

Az adatjellemzők összefoglalása ---------- **43**
Azonosító ---------- 43
Kezdőérték ---------- 44
Hatáskör ---------- 44
Élettartam ---------- 45

Az értéktípus ---------- **46**

Egyszerű típusok ---------- **46**
1. Egésztípus ---------- 46
2. Valós típus ---------- 47
3. Logikai típus ---------- 47
4. Karaktertípus ---------- 48
5. Mutató típus ---------- 48
6. Felsorolástípus ---------- 50

Összetett adattípusok, adatszerkezetek ---------- **51**
Tömb ---------- 51
Szöveg ---------- 54
Verem (stack) ---------- 55
Sor ---------- 56
Lista ---------- 57

V. ALGORITMUSLEÍRÓ ESZKÖZÖK ---------- *61*

1. Mondatszerű leírás ---------- **61**
Beolvasó és kiíró utasítás ---------- 61
Értékadó utasítás ---------- 62
Elágazások (feltételes utasítások) ---------- 62
Elágazás (többágú szelekció) ---------- 63
Ciklusszervező utasítások ---------- 63
Eljárások ---------- 64
Egyéb jelölések ---------- 65

2. Folyamatábra ---------- **66**

3. Struktogram ---------- **68**

VI. PROGRAMOZÁSI TÉTELEK ---------- *70*

Egy sorozathoz egy érték rendelése ---------- **70**

Az összegzés tétele ---------- 70
Az eldöntés tétele ---------- 71
A kiválasztás tétele ---------- 72
A keresés tételei ---------- 73
a) A lineáris keresés tétele ---------- 73
b) A logaritmikus keresés tétele ---------- 74
A megszámlálás tétele ---------- 74
A szélsőérték-kiválasztás tételei ---------- 75
a) A maximumkiválasztás tétele ---------- 75
b) A minimum kiválasztás tétele ---------- 76

Egy sorozathoz egy sorozat hozzárendelése ---------- 78
A kiválogatás tétele ---------- 78

Több sorozathoz egy sorozat hozzárendelése ---------- 80
A metszetképzés tétele ---------- 80
Az egyesítés (unióképzés) tétele ---------- 81
Az összefuttatás tétele ---------- 82

Rendezések ---------- 84
Buborékos rendezés ---------- 84
Rendezés minimum-kiválasztással ---------- 87
Rendezés közvetlen elemkiválasztással ---------- 87

Feladatok ---------- 90

VII. A PROGRAM HELYESSÉGE ---------- 94

Tesztelési módszerek ---------- 94
Statikus tesztelés ---------- 95
Dinamikus tesztelés ---------- 95
Feketedoboz módszerek ---------- 95
Fehérdoboz módszerek ---------- 96

Hibakeresés ---------- 96
Hibakeresési módszerek ---------- 96
Hibakeresési eszközök ---------- 97

Hatékonyság vizsgálat ---------- 98
A végrehajtási idő csökkentése ---------- 99
A ciklus végrehajtási számának csökkentése ---------- 99
Egy ciklus végrehajtási idejének csökkentése ---------- 101

A helyfoglalás csökkentése ---------- 102
Az adatok mennyiségének csökkentése ---------- 103
A programszöveg méretének csökkentése ---------- 104
A bonyolultság csökkentése ---------- 104

Lokális hatékonyság ---------- **105**

Program-transzformációk ---------- **105**

a) A ciklustól független utasítások kiemelése: ---------- 105
b) Ciklusok összevonása ---------- 106
c) Elágazások összevonása ---------- 107
d) Elágazások felesleges feltételeinek elhagyása ---------- 107
e) Összetett feltétel szétválasztása ---------- 107
f) Elágazás ágainak kiemelése ---------- 108

TÁRGYMUTATÓ ---------- ***109***

IRODALOMJEGYZÉK ---------- ***111***

BEVEZETÉS

Napjainkban a számítógép egyre inkább mindenki számára elérhető munkaeszköz, amely jelentős mértékben segíti, egyszerűsíti az emberek munkáját. A számítógép használatának megtanulása egyre inkább alapkövetelménnyé válik. Egyre többen gépelik, szerkesztik leveleiket, táblázataikat szövegszerkesztők, táblázatkezelők segítségével. Ezek az eszközök valójában számítógépes programok, amelyeket sokszor hosszú hónapokon keresztül, több tíz, száz programozó fejlesztett ki. Programok vezérlik a világ összes számítógépét, beleértve azokat is, amelyek nem egy íróasztalon, hanem például a repülőgépekben vagy autókban működnek.

Ez a könyv a programkészítés *alapjaival* ismerteti meg az olvasót. Célja az, hogy témakörök elsajátítása után a tanuló - aki lehet akár 14, de 18 éves is - megismerje a programok elemeit, felépítését, szerkezetüket, a leggyakoribb típusokat, adatszerkezeteket, az egyszerűbb alapalgoritmusokat, valamint ismerje a hibakeresési eszközöket, tesztelési módokat. Segítségével a tanulók elsajátíthatják az algoritmus, illetve a programkészítés alapjait, ezek logikáját.

A könyv igyekszik nyelvfüggetlen lenni, de az algoritmusok leírása elsősorban a Turbo Pascal és a C++ nyelvek szabályait követi. Legeredményesebben akkor használható, ha a tanuló valamelyik programozási nyelv elemeit is elsajátítja, párhuzamosan a tankönyvi anyaggal. Így a könyv algoritmusai számítógépen futtatható programokká válnak.

Az első fejezet a feladatmegoldás lépeseivel, módszereivel ismerteti meg az olvasót. A könyv utolsó felében található feladatokat bizony már csak az első fejezetben leírtak alkalmazásával lehet megoldani.

A második és a harmadik fejezet a program elemeivel, a programok felépítésével foglalkozik. Ezen rész elsajátítása után tudhatjuk meg, hogyan is épül fel egy program, milyen kapcsolat van az egyes részek között. Ezt segíti a fejezet végén található néhány egyszerű feladat is

A negyedik fejezet témája az adattípusok és az adatszerkezetek. A rész elsajátítása után a tanulóknak ismerniük kell az egyszerű típusokat , és az összeetet adatszerkezethez kapcsolódó műveleteket.

Az ötödik fejezet néhány egyszerű algoritmus-leíró eszközzel ismereti meg a tanulót, majd - a hatodik fejezetben - konkrét algoritmusok következnek. Ezek olyan programozási tételek, amelyek szinte minden feladatban "visszaköszönnek", ismeretük létfontosságú a komolyabb feladatok megoldásához. A hatodik fejezet végén több tucat feladat található, melyek megoldása ajánlott az anyag alkalmazás szintű elsajátítása érdekében.

Az utolsó fejezet nemcsak a hatékony program készítésébe vezeti be az olvasót, hanem segít a már elkészült program hibáinak kijavításában, a programok tesztelésében.

I. A FELADATMEGOLDÁS LÉPÉSEI ÉS MÓDSZEREI

Amikor olyan feladatot szeretnénk megoldani számítógépen, amelyeket a rendelkezésre álló felhasználói programok segítségével (szövegszerkesztők, táblázatkezelők, stb.) nem lehet vagy túlságosan nehézkes megvalósítani, akkor bizony nekünk azt a programot elkészíteni, amely megoldja ezt a feladatot. Hogyan is történik egy ilyen program elkészítése?

Először pontosan ismerni kell a feladatot, majd ennek ismeretében el kell készíteni az algoritmust. Természetesen, ha számítógép segítségével akarjuk megoldani a feladatot, akkor az algoritmust kódolni kell, azaz valamilyen nyelven (valamilyen magas szintű programnyelv segítségével) meg kell írni a programot, majd ellenőrizni kell, hogy valóban jól működik-e. Ha már hiba nélkül elvégzi a kitűzött feladatot, akkor érdemes megvizsgálni, hogy elég hatékony-e ez a működés. Ha azt szeretnénk, hogy a programot más is használhassa, megérthesse (vagy a program készítője néhány hónap múlva), akkor bizony célszerű dokumentálni. A feladatmegoldás lépései tehát a következők:

- a feladat meghatározása
- az algoritmus elkészítése
- kódolás
- tesztelés, hibakeresés, javítás
- hatékonyság vizsgálata
- dokumentálás

1. A feladat meghatározása

A feladat megfogalmazását, meghatározását általában néhány leírt, kimondott mondattal elintézettnek szokták tekinteni. Pedig éppen a hiányos, pongyola feladatmegadás következménye lesz az, hogy az elkészült program nem azt a feladatot, nem úgy oldja meg, ahogy azt annak megfogalmazója - gondolatban - kívánta. A feladat megfogalmazása mindig legyen egyértelmű, pontos, teljes, rövid, tömör, szemléletes, érthető, tagolt formájú (feltéve, ha nem az a cél, hogy az algoritmus készítőjének nagy szabadságot adjunk). Mindezeket úgy valósítsa meg, hogy abból a program készítője egyértelműen és pontosan azt a feladatot és úgy oldja meg, ahogy azt a feladat kitűzője elvárta. A további viták elkerülése végett nem árt, ha a feladat kitűzése írásban rögzített! Példaként nézzük meg egy részfeladat kitűzését, megoldását.

Feladat:

Már ismerjük egy iskolában a tanulók év végi jegyeit az egyes tantárgyakból. Készítsük el azt a programot, amely az eredmények alapján elkészíti az iskolai rangsort. Írassuk ki a tanulók nevét, az átlagukat és egy csillag jelölje azokat, akik valamelyik tárgyból elégtelen osztályzatot kaptak.

Már ismert adatok:

- a tanulók száma az iskolában. Jelöljük ezt a számot N-nel
- a tantárgyak száma legyen T, amely 7 és 12 közé esik.
- a tantárgyak neve

- a tanulók neve
- a tanulók év végi jegyei az egyes tárgyakból

A részprogram bemenő adata

Látszólag pontosan sikerült megadni a problémát, ugye? Pedig ha elkezdjük megoldani a feladatot, hamarosan kiderül, hogy nagyon sok mindenről nem szól a feladat:

- az adatok a billentyűzetről kerülnek megadásra, vagy például már egy lemezen lévő állományban már megtalálhatóak?
- lehet-e T értékéke 7 vagy 12?
- hová kell kiríni a rangsort? Képernyőre vagy nyomtatóra? Esetleg mindkettőre?
- mi legyen a rangsor alapja? Valamelyik tantárgy, vagy a tantárgyak átlaga?
- növekvő vagy csökkenő sorrendben legyen a rangsor?
- hol legyen a csillag?

Még tovább lehetne folytatni a kérdések felsorolását, amelyek a pontatlan megfogalmazásból eredtek Ezen kérdések megfogalmazása azonban legyen az olvasó feladata!

Ha a kérdések megválaszolatlanok maradnak, akkor bizony a programozók többsége a legegyszerűbb megoldást fogja választani, nem pedig azt, amelyet a program használója, megrendelője szeretne. Nem szabad ezért ilyen sok szabadságot adni a programozó számára a feladat megoldása során.

A *frontális feladatmegoldás* "módszerét" ismeri meg először minden programozó (sajnos). Már a neve is mutatja: a feladatot egy egységként kezeli, azt egyszerre akarja megoldani minden előzetes felmérés, részekre bontás, átgondolás nélkül. Kis, tanuló feladatok megoldására még eredményesen használható.

Legjellemzőbb hibái: a kész programnak nincs szerkezete, csupa trükk az egész, módosítani, javítani, fejleszteni szinte lehetetlen, a tesztelése véget nem érő, dokumentációja nincs, vagy hiányos, használata bizonytalan, csak idő kérdése, hogy működését senki se értse (a készítője sem!).

2. A feladatmegoldás lépései

A helyes megoldó a feladat felülről lefelé történő lebontását fogja alkalmazni legtöbbször (*top-down* módszer). Hangsúlyozni kell, hogy a feladat megoldásának ebben a szakaszában nem programot, hanem algoritmust készítünk. Nem foglalkozunk azzal, hogy később a program milyen nyelven milyen gépre készül el. A feladat ilyen szintű megoldása, annak rögzítése hatalmas előnyökkel jár. A majdani aktuális nyelv és gép sajátosságait figyelmen kívül hagyhatjuk, csak a feladat logikájára, a megoldás ebből adódó szerkezetére, az egyes részek kapcsolódására, a szükséges adatszerkezetekre, adatokra, mindezek rögzítésére koncentrálhatunk. A feladat algoritmusos megoldása után választhatjuk ki a megoldásra legmegfelelőbb nyelvet, sőt, esetleg gépet is.

Vizsgáljuk meg e felülről lefelé haladó módszer részleteit. Alapvető stratégiai elvünk meglehetősen régi. Oszd meg és uralkodj! A programozók ezt az elvet a *lépésenkénti finomítás elvének* nevezik. Mondanivalója: oszd részekre a feladatot, mert így az egyes részeket egymástól függetlenül - természetesen azokat egymáshoz szigorúan illesztve -

oldhatod meg, és uralkodhatsz az egész feladat felett. Első közelítésben szinte minden feladat a következő három részre osztható:

- bemenő adatok bevitele,
- azokon a szükséges műveletek elvégzése,
- az eredmények megjelenítése, kiírása, rögzítése.

Ez a felosztás azonban még nem elegendő. Az egyes részeket tovább kell finomítani. Az első - és minden további - szint részfeladatai közötti kapcsolatot, harmóniát is biztosítani, rögzíteni kell. Az egyes részfeladatok közötti kapcsolatot azok be- és kimeneti adatainak illesztésével biztosíthatjuk. Feladatunkat tehát az első szinten néhány jól körülhatárolható részfeladatra osztjuk és ezek egymás közötti kapcsolatát is meghatározzuk. A következő szinteken pedig addig folytatjuk az előző szint részfeladatainak további részekre bontását és azok ismételt egymáshoz illesztését, amíg olyan elemi részfeladatokig nem jutunk, melyek kódolása már nem okoz problémát, mert arra nagyon egyszerű szabály adható meg.

	1. szintű program
2. szint	2.1 részprogram, 2.2 részprogram ...
3. szint	2.1.1 részprogram, 2.1.2 részprogram ...

Nézzük a lépésenkénti finomítás elvének taktikai elveit:

a) A *párhuzamos finomítás elve*, mely azt jelenti, hogy a finomítást az adott szint minden részfeladatára végezzük el, ne rohanjunk előre egy-egy könnyebb ágon. Az egyes szintek finomításához szorosan hozzátartozik az adatok finomítása és egyes részfeladatok adatkapcsolatainak meghatározása, rögzítése is!

b) A *döntések elhalasztásának elve*, mely szerint egyszerre csak kevés dologról, de azokról következetesen döntsünk. A részfeladatokat kevés (3-5) további részre bontsuk csak fel, és az újabb részek lehetőleg egyenlő súlyúak legyenek. Halasszuk későbbre azokat a döntéseket, melyek meghozatalához még nincs elegendő ismeretünk.

c) *A vissza az ősökhöz elvre* akkor van szükségünk, ha a fenti elvek betartása és minden óvatosságunk ellenére zsákutcába jutottunk. Ekkor nem elegendő az adott szint újbóli végiggondolása, mindenképpen lépjünk vissza az előző szintre.

d) A *nyílt rendszerű felépítés elve* szerint lehetőleg ne az éppen adott feladatra, hanem a minél általánosabban magába foglaló feladatkörre vonatkozó és azt megoldó algoritmust készítsük el.

e) A *döntések kimondásának* (el nem mulasztásának, nyilvántartásának) *elve*.

A ki nem mondott, de hallgatólagosan meghozott döntések rengeteg bajt okozhatnak! Szóban forgó feltételezések, döntések legtöbbször az adatokkal kapcsolatosak. Ott feledkezünk meg leginkább arról, hogy egy adat nem lehet negatív vagy 0, mert az algoritmus készítésekor ez még nyilvánvaló, de ha nem mondjuk ki, nem rögzítjük, akkor a kódolásnál el is feledkezünk ennek a fontos ténynek az ellenőrzéséről.

f) Az *adatok elszigetelésének elve* azt mondja ki, hogy az egyes programrészek csak finomításkor meghatározott módon kapcsolódhatnak egymáshoz. Ezért az egyes programrészekhez tartozó adatokat ki kell jelölni és szigorúan el kell különíteni más

programrészektől. Az adatokat legegyszerűbb a programban betöltött szerepük alapján osztályozni, elkülöníteni. Ennek alapján beszélünk közös (*globális*) adatokról, melyekhez elvben a program összes részegysége hozzáférhet, módosíthat. Az eljárás hívójával kapcsolatot teremteni ezen kívül ún. bemenet (input), valamint kimeneti (output) adatokkal lehet. A csak egy programrészhez tartozó helyi (*lokális*) adatokat más programegység nem használhatja.

A megoldás lépésenkénti finomításával előbb-utóbb eljutunk egy olyan szintre. amelyen az adott nyelven rendelkezésünkre álló utasításokból már felépíthetjük, létrehozhatjuk az éppen szükséges eljárásokat. Persze akár így is kezdhettük volna: a meglévő eljárásokból létrehozunk olyan magasabb szintű eljárásokat, amelyeket szükségesnek gondolunk, ezekből még magasabb szintű eljárásokat állítunk össze, stb. Ezt alulról felfelé haladó (*bottom-up*) módszernek nevezzük. A kétféle módszer a gyakorlatban kiegészíti egymást, de a felülről lefelé haladóé az elsőbbség. A számítógépes program megoldásánál ugyanis eleinte felülről lefelé haladunk, csak később, egyes részletek kidolgozásakor építkezünk alulról felfelé.

A *kritikus részek kiemelésének* módszerét ugyancsak nagy gyakorlattal rendelkező programozóknak lehet ajánlani. Tulajdonképpen itt is felülről lefelé haladunk, de nem dolgozunk ki részletesen (tovább bontva) minden részfeladatot, nem írunk le részletes algoritmust, mert akkora gyakorlattal rendelkezünk már, hogy egy-egy nem túl bonyolult részfeladat megfogalmazása, illesztése után képesek vagyunk annak hibátlan kódolására. Ilyen feladat lehet például a fejezet elején megfogalmazott rangsoros példa. Részletes algoritmust csak a feladat legbonyolultabb, kritikus részeiről készítünk.

Nézzük meg, hogyan is végezzük el a feladat részfeladatokra bontását?
A kérdés megválaszolásához a mindennapi életből kell kiindulnunk. Egy bonyolult tevékenységet elemibbekből háromféleképpen építhetünk fel. Lehet több elemi tevékenységet egymás után végezni (számítástechnikai szakkifejezés: *szekvencia, felsorolás*), lehet egy feltételtől függően egyik vagy másik tevékenységet végezni (*elágazás, választás*). A legbonyolultabb esetben egy elemi tevékenységet végzünk el sokszor (*ciklus, ismétlés*).

3. Dokumentálás

Kétféle dokumentáció létezik: felhasználói és fejlesztői. A felhasználói dokumentáció önálló, a kész programmal együtt átadandó szöveg, mely megad a program használatához minden szükséges információt.
Tartalmazza:
- a feladat megfogalmazását,
- a program nyelvét,
- az esetleges más nyelvű rendszerek funkcióit
- a futtatáshoz szükséges géptípust, konfigurációt
- a program betöltését, installálását, indítását (részletesen).
- a program használatát (menüket, háttérműveleteket, funkcióbillentyűket).

- hibajelzéseket, a hibák javítási módját,
- egy tipikus futtatás teljes leírását (képernyőkkel, beavatkozásokkal),
- a program fejlesztési lehetőségeit, annak feltételeit.

A fejlesztői dokumentáció a fejlesztők számára szól és a program fejlesztésével *párhuzamosan* készül. Részei:

- a feladat megfogalmazása, pontosítása,
- az algoritmus összes szintjének részletes leírása
- a gépi és nyelvi igények,
- változótábla (név, típus, jelleg, egység, kód),
- az egyes szintek, részfeladatok, rutinok,
- az egyes eljárások hierarchiáját megadó táblázat,
- a program fejlesztési lehetőségei, annak feltételei,
- a program teljes listája, valamint egy háttértáron őrzött példánya (lehetőleg nem egy háttértáron).

A fejezet elején ismertetett feladat esetében 3 nagy részproblémát kell megoldani

- az adatok beolvasása
- rangsor készítés
- kiíratás.

Az első probléma esetében meg kell oldani az adatok tárolási módját és a beolvasás módját. Ismerjük-e a tanulók számát, vagy egy bizonyos jel megadásáig folytassuk a bevitelt?

A rangsor készítésekor két feladatot kell megoldani: egyik a jegyek átlagának kiszámítása tanulónként, majd az adatok rendezése a kiszámított átlag szerint csökkenő sorrendben.

A harmadik probléma esetében tudjuk, hogy ki kell íratni az összes adatot, vizsgálni kell, hogy kapott-e valamelyik tanuló elégtelent. Nyomtatóra történő kiírás esetén gondoskodni kell lapszámlálóról, fejlécről és lapdobásról

II. A PROGRAM ELEMEI, FELÉPÍTÉSE

Az elkészített algoritmus illetve egy program tulajdonképpen egy szöveg. Innentől kezdve lentről felfelé építsük fel a szöveget, nézzük meg, hogy milyen elemekből lehet felépíteni egy programszöveget.[1]

Az elemek a következőek:

1. Karakter.
2. Többkarakteres szimbólumok
3. Lexikai egységek (lexikális egységek, amelyek karakterekből épülnek fel). Ide tartoznak:
   - szimbolikus név (azonosító)
   - kulcsszavak (fenntartott szavak)
   - standard azonosítók
   - címke
   - megjegyzés
   - változó
   - nevesített konstans
   - tipizált konstans
   - konstans

4. Szintaktikai egységek (szintaktikus egységek)
Karakterekből és/vagy lexikai egységekből épülnek fel. Ide tartozik:
- kifejezés

5. Utasítások:
Karakterekből és/vagy lexikai egységekből és/vagy szintaktikai egységekből épülnek fel.

6. Program: utasításokból épül fel

Az imperatív nyelveknek[2], azon belül az eljárás-orientált nyelveknek a felépítése: karakter - program. Ezeken a fogalmakon fogunk végighaladni.

Karakter

A szöveg legkisebb alkotóeleme. Minden nyelvnek meg van a saját karakterkészlete. A karakterkészlet operációs rendszer illetve szoftver függő. A karakterkészlet egy adott

---

[1] A felosztás csak az ún. imperatív nyelvekre igaz, de ezek manapság az igazán elterjedt nyelvek, pl. C, Pascal, COBOL, PL/I.

[2] Az imperatív ("parancsoló") nyelvek esetében a programozónak kell meghatároznia, hogy a problémát milyen módon, hogyan oldja meg az utasítások felhasználásával. Nyelvek alatt természetesen programozási nyelveket kell érteni.

logika szerinti leképezése a szabványos kódrendszer (pl. ASCII, ANSI vagy az EBCDIC stb.).

Mit is tartalmaz egy nyelv karakterkészlete:

1. Minden nyelv karakterkészletében ott van az angol ábéce 26 betűje. Betű alatt még néhány speciális jel is értendő: Pl."_" (aláhúzásjel) vagy a "#". Sok nyelvben ezek a jelek betűknek számítanak, de ez implementáció[3] függvénye.

Néhány nyelv csak a nagybetűket ismeri el betűnek és a kisbetűket nem tekinti betűnek, de pl. a PASCAL, C a kis és nagybetűket egyaránt betűnek tekinti.

2. Minden nyelv karakterkészletében benne van a 10 db decimális számjegy (0-9).

3. Speciális karakterek

Meglehetősen nagy eltérés van a nyelvek között - ide tartoznak:,.:;!() stb. Amennyiben a magyar ékezetes betűk a karakterkészletben benne vannak, akkor speciális karakterek - a speciális karakterekbe nagyon sok minden beletartozik. A speciális karakterek között kitüntetett szerepe van egy karakternek, a szóköznek, mely létezik minden nyelvben. Ha azt akarom, hogy látható legyen a szóköz karakter, akkor a látható szóközöket lefektetett, száraival felfelé álló szögletes zárójellel jelölhetem.

Többkarakteres szimbólumok

Bizonyos nyelvek értelmeznek olyan egységeket, melyek több karakterből állnak, de egy egységek, ezért egy egységként kezelendők.

Pl. <>, <=, >=, többkarakteres műveleti jelek a PASCAL-ban (máshol nem biztos, hogy müveleti jelek)

Lexikai egységek

Szimbolikus név - azonosító

Olyan karaktersorozat, amely betűvel kezdődik és betűvel vagy számjeggyel folytatódik. Vagy az implementáció, vagy a hivatkozási nyelv megszab egy felső határt az azonosító hosszának - nyilvánvaló, hogy tetszőleges hosszúságú azonosító nem létezik, legalább implementációs szinten van felső határ. Az azonosító szolgál arra, hogy a programozó a saját programozói objektumait elnevezze. Programozói objektumnak a neve mindig azonosító. *Nagyon sok mindennek lesz neve, és az mind azonosító lesz!*

Szimbolikus név - kulcsszavak, fenntartott szavak

Olyan speciális karaktersorozatok, amelyeknek a nyelv tulajdonít értelmet és ez az értelem nem változtatható meg (csak abban az értelemben használható). Pl. a PASCAL-ban a GOTO szónak van egy meghatározott jelentése illetve szerepe, ezért másra nem használhatom, csak arra, amire a nyelv szánja (menj oda, ugorj oda).

[3] Egy adott programozási nyelv konkrét gépi megvalósítása

Vannak azonban olyan nyelvek, amelyekben fenntartott szavak nem léteznek, mivel nem minden nyelv védi a saját szavait. Pl.: a PL/I-ben nem létezik, de a PASCAL-ban igen - minden PASCAL verziónál ott vannak a kulcsszavak.

Szimbolikus név - standard azonosítók

Olyan speciális karaktersorozatok, amelyeknek a nyelv tulajdonít valamilyen értelmezést, de ez a programozó által megváltoztatható. Valamilyen nyelv által értelmezett karaktersorozat használható, ha az azonosító megfelel programazonosítónak is - programozó megváltoztathatja a jelentését. Pl. PASCAL-ban a függvénynevek standard azonosítók. Általában minden nyelvben vannak ilyen azonosítók.

Címke

Az utasítások megkülönböztetésére szolgál (ez az irányítószám) - a program egy adott utasításánál lehessen hivatkozni valamelyik másik utasításra (egyértelműen). A címkének az alakját, tehát hogy milyen formában, hogyan lehet megadni, megint csak a nyelv definiálja és nyelvenként különböző. Pl.: standard PASCAL-ban a címkék négyjegyű, előjel nélküli egész számok lehetnek - a Turbo Pascal-ban(TP) ezen túlmenően még tetszőleges azonosító is lehet a címke. Alakja: **címke**: utasítás (utasítások előtt állhat a címke, tőle kettősponttal elválasztva).

Megjegyzés

Ez az elem a program olvasójának szól (tájékoztató jellegű, magyarázó szövegeket helyezhetünk el a program szövegében). A fordítóprogram nem fog majd a megjegyzéssel foglalkozni, mint a program szövegével általában, hanem kihagyja. A nyelvek definiálják, hogyan helyezhetünk el a program szövegében megjegyzéseket: Pl. C-ben /* */. vagy TP esetén { }. *A PASCAL azt mondja, megjegyzés elhelyezhető bárhova, ahol szóköz szerepel.* Ebben a könyvben a megjegyzések **-gal fognak kezdődni

A program szövegének felépítése

Hogyan kell felépíteni a program szövegét? Programozói nyelveknél lényeges kérdés. Kétféle programozói nyelv-filozófia van ebből a szempontból (kétféle megközelítési mód):

- *kötött formátumú nyelvek*

A korábbi nyelvek kötött formátumú nyelvek voltak, ezek lyukkártyákhoz kötődtek (80 pozíciós kártyákban kellett gondolkodni). A programot úgy kellett megírni, hogy a kártya a bizonyos oszlopain adott elemeket kellett elhelyezni (tehát kötött volt, hogy az első 8 oszlopba mit írhatok stb.). Ennek bizonyos átöröklődései ma is léteznek, mert mikrogépes környezetben van még kötött formátumú nyelv (nem mindegy, hogy a sorokba mit írok).

- *szabad formátumú nyelvek*

Az utasításokat tetszőleges modon helyezhetem el, egy sorba sok utasítást stb. írhatok - nincs kötöttség, nincs megmondva, hogy bizonyos pozíciókon mik jelenhetnek meg. Pl. PASCAL, C.

A két elv között van egy lényeges különbség. A kötött formátumú nyelvnél meg volt szabva, hogy, hova lehet írni az utasítást (megvolt az utasítás helye, vagy megvolt a címke helye). Ezzel szemben a szabad formátumú nyelvekben nincs ilyen hely. Éppen ezért valamilyen módon meg kell különböztetni, el kell határolni egymástól az utasításokat tehát meg kell mondani, hogy az utasítás innentől idáig tart. Itt két lehetőség van

1) Az ALGOL, PL/I, C azt mondja, hogy minden utasítást egy pontosvesszővel kell lezárni. Tehát egy utasításhoz szigorúan hozzátartozik a pontosvessző, mert ez jelzi a végét, és értelemszerűen 2 pontosvessző között van az utasítás.

2) Ezzel szemben a standard PASCAL azt mondja, hogy a pontosvessző utasítások elhatárolására szolgál akkor, ha mással már nem határoltunk el utasítást. Tehát a standard PASCAL-ban a pontosvessző nem utasítást záró jel, hanem utasítások elválasztására szolgál (utasításszeparátor). A standard PASCAL-ban utasítást elhatárolhatnak alapszavak, kulcsszavak. Két utasítás között vagy valamilyen kitüntetett kulcsszó szerepel, vagy pontosvessző; itt a 'vagy'-on van a hangsúly, mert a kettő együtt nem szerepelhet. A TP(Turbo Pascal) ezt kiküszöböli, ugyanis azt mondja, hogy mindkettő lehet együtt. Pl. a standard PASCAL-ban az END egy kulcsszó (alapszó); ahol END van, ott nem szabad pontosvesszőt szerepeltetni. (END előtt a pontosvesszőt a standard PASCAL visszautasítja.) A TP elfogadja a pontosvesszőt, azt mondja, hogy fölöslegesen kiírtam egy utasításszeparátort.

Változó

Eljárás-orientált nyelvek legfontosabb objektuma. Minden változónak egyedi neve van ezeken a nyelveken a programozás igazából a változók értékeinek megváltoztatása. A változó olyan objektum, melynek 4 komponense van

1. Név

- tetszőleges, a programozó által választott *azonosító* - minden változónak egyedi neve van

2. Cím

- minden változónak van egy címkomponense, amely egy tárbeli címet jelent
- az értékkomponens jelenik meg ezen a bizonyos tárcímen.

3. Érték

4. Az *attribútumok* azt írják le, hogy milyen értékeket vehet fel az adott változó, azon a bizonyos tárterületen milyen értékek jelenhetnek meg. Tehát az értékeket jellemzik az attribútumok, a változó által felvehető értékekről adnak információt.

A változó a szövegben mindig a nevével jelenik meg. Mindjárt tisztázzuk, hogy a név leírása melyik komponenst jelenti a szövegben. Innentől kezdve akkor a névhez kell valamilyen értelemben hozzárendelni az attribútumokat, hozzárendelni a címet és az értéket (a név képviseli a változót és a névhez rendelem hozzá a többi komponenst).

Az attribútumok névhez rendelése

a) A programozó speciális utasításban, ún. deklarációs utasításban rendeli hozzá a névhez az attribútumot - amikor a fordítóprogram találkozik ezzel a deklarációs utasítással, a fordítás pillanatában vagy a fordítás közben a névhez az attribútumok egyértelműen hozzárendelődnek és ezek nem változtathatók meg.

b) A programozó nem mondja meg az attribútumokat (ami kvázi egyenértékű azzal hogy nem szerepeltetheti deklarációs utasításban a nevet). Ebben az esetben a fordítóprogram rendel hozzá attribútumokat a név, vagy a név kezdőbetűje alapján.

c) Egy névhez az attribútumok futás közben rendelődnek hozzá attól függően, hogy milyen értéket kapott a változó. Futás közben az attribútumok megváltoztathatók (tehát más-más attribútum értékeket vehet fel futás közben a változó).

A nyelvek az adott lehetőségek közül valamelyiket (esetleg többet is egyszerre) választják - ez általában hivatkozási nyelv szintű döntés. Pl.: a PASCAL azt mondja, minden változót deklarálni kell (szigorúság). Más nyelvek az első és második változatot használják. PL/I, FORTRAN, bizonyos BASIC-ek - a névnél típusjelző karakterek, vagy egyebek vannak és a név alapján dől el az attribútum. A harmadik lehetőség a klasszikus nyelvekben nem nagyon van, de az egyéb nyelveknél van ilyen lehetőség (a 3. kategóriájú nyelveknél van olyan lehetőség, hogy futás közben dőlnek el az attribútumok és azok megváltoztathatók, például DBASE).

A címkomponens névhez rendelése

A címkomponenssel kapcsolatban tárkiosztásról szoktak beszélni.

a) *Statikus tárkiosztás*: ez az az eset, amikor a fordítóprogram (szerkesztő) rendel a változóhoz címkomponenst és ez a címkomponens a futás során állandó. Lényeg tehát, hogy fordítás alatt eldől egy változó címkomponense és ez futás közben nem változik - egy adott változó mindig az adott tárterületen van, ahogy elindultunk.

b) *Dinamikus tárkezelés*: a blokkszerkezetű nyelvek esetén van (lényeges, hogy a nyelv blokkszerkezetű legyen) annak jelentősége, hogy a változóhoz címkomponens a futás közben rendelődik. Amikor egy olyan blokkra kerül a vezérlés, amelynek az adott változó lokális változója (korábban a változónak nem volt címkomponense), akkor lesz címkomponens, és a címkomponens addig van, amíg az adott blokk be nem fejeződik. A futásidő bizonyos részeiben van címkomponens a változóhoz hozzárendelve és ez a futás idő különböző pontjaiban más-más cím lehet. Az a lényeg, hogy futás közben a futás idő különböző részeiben különböző memóriahelyeken helyezkedhet el a változó - változik a változó címkomponense futás közben.

c) *Programozó által vezérelt tárkiosztás*: ez az az eset, amikor futás közben a programozó mondja meg, hogy mikor rendelődjön hozzá az adott változóhoz tárterület. Van külön utasítás, amelyikkel megmondhatom, hogy ehhez a változóhoz most kérek tárterületet hozzárendelni.

Programozó által vezérelt tárkiosztás is 3 féle lehet:

a) Megmondom az abszolút tárcímet (tehát megmondom, hogy a tárban ide tegye).

b) Más, címmel rendelkező objektumokhoz képest helyezem el (tehát nem abszolút mondom meg, hogy a 28-as bájt, hanem azt mondom meg, hogy az ilyen másik változóhoz képest hova rakja. Közvetett hivatkozás a címre egy olyan objektum segítségével, aminek már van címe (tehát még mindig én mondom meg).

c) A rendszernek szólok, hogy helyezze el valahova, de nekem fogalmam sincs róla, hogy hová fogja helyezni, csak azt kérem a rendszertől, hogy most ennek a változónak legyen címkomponense. Pl.: a C-ben megvan ez a lehetőség, a tárkezelése egészen speciális TP (Turbo Pascal)-ban mindhárom vezérelt tárkiosztás létezik.

Értékkomponens névhez rendelése

1. Bizonyos nyelvekben megengedik azt, hogy fordítás közben egy változóhoz értéket rendeljek hozzá (szerkesztés). Lényege az, amikor a program elindul, a változó már értékkel rendelkezzen. Ez megint csak deklarációs utasításban történhet. Ezt az egészet úgy hívják, hogy kezdőértékadás.

2. A másik lehetőség, hogy futás közben adok értéket a változónak. Futás közben háromféle módon tudok értéket adni:
- értékadó utasítással
- beolvasó utasítással
- paraméterátadással (lásd később)

Pl. PASCAL-ban: változó:=kifejezés

Általában a nyelvekben az értékadó utasítások szerkezete ilyen: változó neve, valamilyen értékadásjel és egy kifejezés. Az értékadás jel vagy a ":=" vagy csak a "=" . A PASCAL az egyenlőségjelet fenntarja egyenlőségvizsgálatra, ez meg (:=) a legyen egyenlőnek a jelölése. Néhány nyelv cifrázza, mert megengedi azt, hogy több változót felsoroljunk az értékadó utasítás bal oldalán. Pl.: C++ - ez az ún. többszörös értékadás, de van olyan BASIC verzió is, amely ezt megengedi

A program szövegében leírt változónév mit jelent? Pl., értékadó utasításban ez mit jelent? x:=x+1.

Az általános szabály a következő: ha egy változó nevét kifejezésben szerepeltetem, akkor az az értékkomponensét jelenti. Ebből következik, ha kifejezésben hivatkozok egy változóra, akkor annak már értékkel kell rendelkezni. Értékadó utasítás bal oldalán szereplő változónév viszont a címkomponenst jelenti, képviseli. Tehát x eddigi értékéhez vegyünk hozzá 1-et és helyezzük el azon a címkomponensen, ahol x van, amire az x mutat, vagy amihez az x-et hozzárendeltem. Ez alól kivétel a C, ugyanis a C-ben explicit(közvetlen) módon megmondhatjuk, hogy egy változónak a címkomponensét vagy értékkomponensét akarom használni valahol. Az értékkomponensre visszatérve, az implementációk nagy része azt csinálja, hogy amikor címkomponenst rendel a változóhoz, akkor az értékkomponenst valamilyen speciális értékre állítja - implicit(közvetett)

kezdőértékadást végeznek el az implementációk (nem minden nyelvben, de az implementációk jó része ilyen). Pl.: a számértékű változókat lenullázzák. A tárterület lefoglalása egyben a tárterületnek valamilyen értékkel történő feltöltését is jelenti - nagyon sok implementáció így csinálja.

A standard PASCAL-ban nincs kezdőértékadási lehetőség, a TP-ben van (tipizált konstans).

Konverzió: értékadó utasítás bal oldalán áll egy változó, jobb oldalán egy kifejezés. Mi van akkor, ha a változónak az attribútumai és a kifejezésnek az attribútumai nem egyeznek meg? Lehet-e ilyen, ha igen, akkor mi van? A válasz: konverzió szükséges. A konverzió problémája értékadásnál merül fel, tehát amikor a bal oldalon álló változó attribútumai mások, mint annak az értéknek az attribútumai, amit "kiszámolok" a jobb oldalon. Ilyenkor van szükség például a PASCAL-ban a típuskonverziós függvényekre.

Nevesített konstans

Ez egy olyan programozói objektum, aminek van egy neve, vannak attribútumai, van ugyan címkomponense, de ezt a címkomponenst nem illik, vagy nem lehet használni és van értékkomponense, amely állandó.

A nevesített konstansnál a következő kérdéseket kell megválaszolni:
1. Van-e a nyelv által definiált nevesített konstans (standard nevesített konstans)?
2. Ha van a programozó által deklarálható nevesített konstans, akkor ezt az értéket (konstans értékét) hogyan tudja megadni a programozó?
Konstanssal vagy kifejezéssel
Fordítási időben vagy futási időben dől el a konstansnak az erteke.
A nyelvek mást-mást mondanak.
3. Milyen típusú értékek lehetnek?

A PASCAL-ban van nevesített konstans, sőt standard(állandó) nevesített konstansok is léteznek, pl. FALSE, TRUE, NIL.
A programozó is definiálhat nevesített konstanst, a következő deklarációs utasítással.
Konstans a konstans neve= érték

Innentől kezdve a szövegben, a nevesített konstans nevének a leírása mindig az értéket képviseli. A deklaráció leírása után ennek a konstansnak az értéke nem változtatható.

Tipizált konstans

Majdnem úgy néz ki, mint a nevesített konstans. Konstansnál kell deklarálni, azzal a lehetőséggel, hogy az egyenlőségjel helyett : és = jel szerepel és ezzel egy változó

deklaráltam. Ezt hívja a TP tipizált konstansnak, de ennek felülírhatom az értékét, tehát változó.

Arra a kérdésre válaszolva, hogy van-e olyan lehetőség, hogy a programozó definiálja a nevesített konstanst, a válasz: van. A PASCAL azt mondja tehát, hogy csak konstanssal adhatom meg és a fordítás során rögzítődik az érték. Tehát futás során az érték már nem módosulhat.

Konstans

A konstansnak nincs neve. Egy normál programozó számára nem létezik címkomponense. Az értéke állandó. Olyan objektumai a programozásnak, amelyek önmagukat definiálják és nincs deklaráció, nincs egyéb. Tehát egy-egy konstans, egy-egy karaktersorozat, amely saját magát definiálja és meghatározza az attribútumait.

Pl.: Felírtam azt, hogy 11. Ránézésre ez egy egész szám, az értéke 11. Nyilvánvalóan az értéket nem változtathatom meg, mert ha azt mondom 12. az egy másik konstans.

A TP-ben a következő konstansok léteznek:

Numerikus konstansok:

1. Egész konstans:
- olyan decimális egész szám, amelynek lehet előjele (ha nincs előjele, pozitív egésznek számít).
2. Előjel nélküli egész: időnként a PASCAL-ban ez egy külön fogalom - olyan egész, melynek nem lehet előjele.
3. Hexadecimális egész:
- lehet előjele - a szám előtt közvetlenül egy $ áll és utána hexadecimális számjegyek (Pl. $2FA)
- speciális esetben használhatjuk, de nem általánosan.
4. Tizedes törtnek nevezett konstans: alakja egy egész szám, egy pont és a pont után legalább egy decimális számjegy és azon felül tetszőlegesen sok decimális számjegy. A lényeg, hogy a tizedes törtben van tizedespont. (Pl. 12.2)
5. Exponenciális konstans: egy egészet vagy egy tizedes törtet követ egy E betű és utána áll egy egész - ez egy olyan konstans, mely a szám normál alakját reprezentálja. Az E betű a lényeg. (Pl. 12.1222E6 jelentése $12.1222 \cdot 10^6$)

Logikai konstans

A PASCAL-ban nincs, csak logikai nevesített konstans (TRUE, FALSE). Ezek nevek és nem konstansok - ez azért érdekes, mert pl. ezek standard azonosítók a PASCAL-ban.
Más nyelvekben létezik ilyen, hogy logikai konstans.

Karakteres konstans (szöveges konstans, karakterlánc, sztringkonstans)

A standard és a TP-nak is konstanstípusa. Alakja a következő: aposztrófok között tetszőleges karaktersorozat, pl. 'almafa', 'DE JO VOLT'

A PASCAL-ban legalább egy karakternek kell lenni az aposztrófok között, feltéve, ha nem üres sztringet akarok használni. Nyilvánvalóan mindig van felső határ, meg van adva, mennyi lehet (ha normálisan használom, akkor minden belefér). Ez az egyetlen objektum, egyetlen lexikális elem, amikor a szóköz ugyanolyan értékű karakter, mint bármi más karakter. Itt nem elhatárolójel, hanem karakteres konstans - mindenütt másutt határolójel.

Szintaktikai egységek - kifejezések

A kifejezés olyan objektum, amelynek attribútuma és értéke van. Itt a típus megegyezik az attribútummal, más attribútum nincs. Mondhatom azt, hogy típusa és értéke van. A kifejezés arra szolgál, hogy a program bizonyos pontjain bizonyos értékeket állítsunk elő, amelyeket majd tovább akarunk feldolgozni.

Egy kifejezés formálisan operandusokból, műveleti jelekből és kerek zárójelekből épül fel. Operandus konstans, nevesített konstans, változó és függvényhívás lehet. Önmagában már egy operandus is kifejezést ad. A műveleti jeleket minden nyelv definiálja. Ez nyelvenként elég különböző lehet (vannak standard műveleti jelek). Legalapvetőbb kérdés a kifejezések kiértékelése. A kifejezés kiértékelése nem jelent mást, mint: mi lesz a kifejezés értéke és mi lesz a típusai.

Kifejezések kiértékelése

A kifejezések kiértékeléséhez kapcsolódóan a programozási nyelvek
- elsőként a precedencia szabályt alkalmazzák
- másodikként a balról-jobbra vagy jobbról-balra szabályt.

Minden nyelv felállít műveleti jelei között egy erősorrendet (precedenciát, prioritási sorrendet), az erősebb műveleti jel valamilyen értelemben hamarabb kerül kiértékelésre. Ha van egy kifejezésem, és ha a kifejezésben zárójelet alkalmazok, akkor azért alkalmazom, hogy a műveletek nyelv által definiált (standard) precedenciáját (sorrendiségét) megváltoztassam. Egy kifejezésen belül tetszőlegesen sok (redundáns) zárójelet alkalmazhatok. Pl. a PASCAL műveleti jeleinek precedencia szabálya:

( )	zárójelek
not @ ^ + -	egyoperandusú műveleti jelek
* / div mod shl shr and	
+ - or xor	
< > = <= >= <> in	

(A DIV egész osztás, azaz 8 DIV 3 = 2, a MOD maradékképzés, tehát 9 MOD 4 = [illegible])

Az egy sorba írt műveleti jelek azonos precedenciájúak. A PASCAL egyértelműen a balról-jobbra szabályt vallja, a C nyelv viszont a jobbról-balra szabályt.

Ha van egy kifejezésünk, akkor annak kiértékelése a következő műveleti sorrendben megy végbe: elindulunk balról és megvizsgáljuk az első két műveleti jelet. Ha a bal oldali

erősebb, akkor elvégzem. Ha azonos, akkor is elvégzem a bal oldalit. Ha pedig a jobb oldali erősebb, továbblépek a következő két műveleti jel prioritásának vizsgálatára. Ilyen értelemben a kijelölt műveleti sorban a legelső műveletnek az elvégzése nagyjából egyértelmű. A balról-jobbra szabály úgy jelenik meg, hogy azonos prioritású műveletek közül a bal oldalit fogjuk elvégezni. Ha elvégeztünk egy műveletet, csökken az operandusok száma. Innentől kezdve nem egyértelmű a kifejezés kiértékelése. Kétféleképpen mehetünk tovább:

- visszamegyünk a kifejezés elejére és megint keressük az első legerősebbet és azt fogjuk elvégezni
- ha elvégeztük az első legerősebbet, akkor továbbmegyünk és keressük a következő legerősebbet.

Ez a döntés a folytatásról erősen implementáció-függő. TP (Turbo Pascal) esetén a TP optimalizáló dönti el ezen szabályokon belül a folytatást. Bizonyos nyelvek ezekben az esetekben azt is mondhatják, hogy azonos prioritású műveletek jobbról-balra végzendők el.

Hogyan jutunk el a kifejezés típusához? Erre kétféle választ adnak a nyelvek:

1. Erősen típusos nyelvek - *típus-egyenértékűség*:

Azokat a nyelveket - és lesz még egy-két jellemző -, amelyek a kifejezésekben csak azonos típusú operandusokat engednek meg, erősen típusos nyelveknek hívjuk. Itt nincs probléma, mert a művelet elvégzése után a típus adva van. Erősen típusos nyelveknél felvetődik a típus-egyenértéküség problémája: mikor mondhatjuk azt, hogy két típus megegyezik? A típus-egyenértéküség kifejezéseken túlmenően a következő helyeken jelenik meg problémaként: értékadás, indexek illetve indexkifejezések, paraméterátadás.

2. Vegyes típusú kifejezések megengedettek - *típuskényszerítés*:

A műveleti jel két oldalán különböző típusú operandusok lehetnek. Megmondja, hogy milyen típuskeveredéseket enged meg valamilyen műveleti jelnél. A hivatkozási nyelv definiálja, hogy ezt hogyan kell értelmezni, ill. milyen szituációban mi lesz az eredménynek a típusa, és mi lesz a végeredmény: ezt a jelenséget típuskényszerítésnek nevezzük (nyelvek másik csoportja).

Hogy még cifrább legyen a dolog, van a nyelveknek egy közbeeső csoportja, pl. a PASCAL, amely igazában erősen típusos nyelv. Ez azt jelenti, hogy megkövetelik, hogy az első pontban említett szituációkban megegyezzenek a típusok de ettől el lehet térni bizonyos esetekben

3. *Típus-kompatibilitás*:

Enyhített típus-egyenértekűség - meg kell egyezni ilyen szituációkban a típusoknak. de mégiscsak van valamilyen engedmény és ekkor beszélünk típus-kompatibilitásról (valamilyen értelemben mégiscsak lehet keverni a típusokat).

Pl PASCAL-ban: a:=b. megy, ez értékadás-kompatibilitás

b:=a nem biztos, hogy megy

Kifejezés esetén a zárójelezés minden esetben kiemelési jelentő: a legelső zárójelben található részkifejezést kell legelőször kiértékelni. A PASCAL lelkivilága olyan, ha egy kifejezés 4 operandusnál többet tartalmaz, zárójelezzünk

Attól függően, hogy egy kifejezésnek az értéke milyen típusú, szokás
- logika
- aritmetikai
- szöveges
- stb.

kifejezésről beszélni

Logikai kifejezések (speciális kifejezések)

pl. (x<>0) and (y/x<1)
Ha az első része igaz, akkor továbbmegy és a kifejezés második részétől függ a logikai érték, ha nem igaz, akkor a második értéktől már nem függ a logikai érték, de ekkor a második rész nem kiértékelhető, mert nullával való osztás van benne. Ez illusztráció ahhoz, hogy bizonyos nyelvek a logikai kifejezések kiértékelésénél alkalmazhatják a *rövidzár* műveletet (csak logikai kifejezésekkel kapcsolatban lehet).
Egy logikai kifejezés esetén a következőket mondhatják a nyelvek:
- mindentől függetlenül a teljes logikai kifejezést ki kell értékelni (végig kell értékelni)
- a logikai kifejezést csak addig kell kiértékelni, amíg egyértelműen el nem dől az értéke (ha egyértelműen eldőlt a logikai érték, nem megyünk tovább).

Továbbá:
- az egyik lehetőség, hogy a nyelv eldönti, a fenti két lehetőség közül melyiket választja és én nem tudom befolyásolni
- a másik lehetőség (amit a TP mond), hogy én dönthetek a program futása előtt (programfutáshoz kapcsolódóan), hogy melyiket akarom (a standard PASCAL-ban nincs erre mód). A TP-ben egy *direktíva* segítségével tudom beállítani az üzemmódot {$B}
- harmadik lehetőség, hogy külön logikai műveleti jelek vannak - vannak olyan nyelvek, amelyekben más logikai műveleti jel van az egyikre és más a másikra - az egyiknél végig megtörténik a kiértékelés, a másiknál nem. E utóbbi a rövidzár művelet, amikor nem értékeli ki végig (ilyen van az ADA-ban vagy a C-ben).

Nézzük most meg példákon keresztül miről is volt szó eddig!

1) Nevesített konstans megadása:
Konstans Hossza = 15
Kulcsszó Azonosító 1 karakter numerikus konstans (két karakterből áll)
Itt deklaráltunk egy nevesített konstanst, vagyis innentől kezdve a 15 neve a programban a Hossza lesz. A Hossza pedig nem más mint egy azonosító.

2) Egy tipizált konstans deklarálása:
Konstans Mondat: **Szöveg** = "Programozok"
Kulcsszó Azonosító Kulcsszó 1 karakter szöveges konstans

"Programozok" szövegnek adtunk egy nevet: Mondat. Itt most a konstans típusát (a típusokról a következő fejezetekben lesz szó) is megadtuk, ezért nevezzük tipizált, azaz típussal rendelkező konstansnak

3) Változó deklarálása:

Változó számol: **Egész**

Kulcsszó Azonosító Kulcsszó

A fenti példában egy számol *nevű* változót deklaráltunk. [1] Ennek típusa **Egész**, ami mögött majd ott az attribútum(a nyelv adja meg, hogy mi jellemzi az **Egész** típust). A deklarálás pillanatában már lefoglalásra kerül a változó számára egy cím a memóriában Ezen a címen fog majd elhelyezkedni a változó mindenkori értéke. A deklarálás után a TP-ben és a C++-ben nem történik kezdeti értékadás, vagyis nem tudjuk, hogy most milyen értéke van a változónak. Ezért a számol:=számol+1 utasítás eredményeképpen nem tudjunk megmondani a számol változó értékét. Nőtt eggyel, de mihez képest? Ezért lehet szükség a kezdeti értékadásra. számol:=0. Ezzel megtörténik a név és az értékkomponens összerendelése A többi komponens névhez rendelése a deklaráció során történt

4) Mennyi 9 DIV 2 MOD 3 * -2 a TP-ben?

Mivel a DIV és a MOD azonos precedenciájú, ezért először a DIV hajtódik végre, vagyis 9 DIV 2 = 4.

4 MOD 3 * -2 amit ki kell értékelni. A balról-jobbra szabály miatt 4 MOD 3 =1

1 * -2 Itt most a - előjel műveletet jelent, ezért ez hajtódik végre először, majd a szorzás.

Az eredmény tehát -2 lesz!

5) Deklaráljunk most egy olyan változót, amely egész számokat vehet fel értékül

Változó AFA: **Egész**

Tegyük fel, hogy a programunkban az AFA:= 36 / 2 MOD 3 értékadó utasítás szerepel. Az értékadás jobb oldalán egy kifejezés, amelyet ki kell értékelni:

36 / 2 sajnos nem 18, hanem 18.0, vagyis egy valós szám. A MOD viszont csak egész számokra van értelmezve, így egy erősen típusos nyelvnél a 18.0 MOD 3 nem végezhető el, a fordító hibát jelez.

Ha az AFA:= 36 MOD 2 / 3 utasítást nézzük, akkor kicsit más a helyzet, hiszen 36 MOD 2 = 0, de 0 / 3 sajnos nem 0, hanem 0.0. Az AFA változó azonban csak egész értéket vehet fel, a 0.0 viszont egy valós szám Így erősen típusos nyelveknél ismét nem történik meg az értékadás.

[1] A könyvben többször fogunk használni ékezetes karaktereket az azonosítókban Vigyázzunk azonban arra hogy a nyelvek többsége nem engedi ezek használatát!

Utasítások

Osztályozásuk:

- értékadó utasítások
- ugró utasítások[vezérlő utasítások (a program vezérlési szerkezetét adják meg)]
- feltételes utasítások
- ciklusszervező utasítások
- I/O utasítások
- egyéb utasítások.

Ugró utasítások

- alakja: GOTO címke (a PASCAL-ban is)
- a megadott című utasításon folytatódik a program végrehajtása

I/O utasítások

Néhány nyelvben hiányoznak (pl. a PASCAL-ból is hiányzik, hiszen a READ és WRITE standard eljárások és nem utasítások). Más nyelvekben lehetnek I/O utasítások - pl. a PL/I-ben 40-50 db I/O utasítás van.

Feltételes utasítások

a) Egy, illetve kétágú szelekció (elágazás)

Ha feltétel **akkor** utasítás1 **Egyébként** utasítás2

Két lehetőség közötti választásra alkalmas. Ha a feltétel igaz, akkor csináld az utasítás1-et, egyébként az utasítás2-t. Ha ezek elvégzését befejezte, akkor a végrehajtás a feltételes utasítás után folytatódik (hacsak az utasítás1 vagy az utasítás2 másképpen nem rendelkezett).

- a feltételes utasítások egymásba skatulyázhatók, max. 255.
- a csellengő **egyébként** szép problémája: hova kapcsolódik az utasítás végén található **Egyébként**? - általános válasz, az utolsó olyan **akkor**-hoz, amihez még nem volt **Egyébként**(de ez implementáció függő) pl. **Ha akkor Ha akkor Egyébként**... A TP azt mondja (amikor az optimalizáló nem mond mást), az utolsó akkor Egyébként kapcsolódik össze.

Például: **Ha** A<8-2*3 **akkor** b:=1 **Egyébként** b:=2

Ha A<8*2/3 **akkor** Van:=**Igaz**

b) Többágú szelekció (elágazás)

Egyes nyelvekben:

Elágazás feltétel1: utasítás1
[feltétel2: utasítás2][5]
...
[**Egyébként** utasításn]
Elágazás vége

más nyelvekben

Elágazás kifejezés
konstanslista: utasítás1
[konstanslista: utasítás2]
...
[**Egyébként** utasításn]
Elágazás vége

az utasítás formája.

Ha a feltétel1 igaz, akkor az utasítás1, ha a feltétel2, akkor az utasítás2 ... kerül végrehajtásra. Ha egyik feltétel sem igaz, akkor az utasításn.[6] Ha nincs **Egyébként** és egyik feltétel sem igaz, valamint a feltételeknek megfelelő utasítások végrehajtása után a végrehajtás az elágazás után folytatódik (hacsak az utasítás1 vagy az utasítás2... másképpen nem rendelkezett). A második fajta szelekció utasítás esetén a kifejezés értékének és a konstanslistának megfelelő (összetartozó) utasítás kerül végrehajtásra, egyébként a működése megfelel az először leírtaknak.

Például: A:=2
Elágazás A
1: A:=3
2: A:=4
Egyébként A:=A+1
Elágazás vége
Ebben a példában A értékül felveszi a 4-et.

Üres utasítás

Nem nagyon lehet vele hibázni.

Ciklusok

Ha a program valamely pontján bizonyos tevékenységet vagy tevékenységcsoportot többször végre kell hajtani, akkor beszélünk ciklusról.[7] A ciklusszervezésnek különböző eszközei vannak - általában nem egy utasítás tartozik a ciklushoz, hanem több. Ilyen értelemben nincs értelme ciklusutasításról beszélni, csak ciklusról.

---

[5] A [] azt jelenti, hogy a közéjük írt szöveg megadása NEM kötelező!
[6] A feltételek valójában logikai kifejezések.

Terminológia:

- *ciklusfej*: általában szabályozza a végrehajtás mikéntjét (a fejben vannak az erre vonatkozó információk)
- *ciklusmag:* azt a tevékenységcsoportot írja le (tetszőlegesen bonyolult tevékenységet), amit ismételni kell (ezek végrehajtható utasítások, legalább egy - lehet üres utasítás is)
- *ciklusvég*: legtöbb nyelvben külön szerepel, van, amivel ki tudom jelölni, hogy ez a ciklus vége.

Az alábbi ciklusfajtákat különböztetjük meg:

1. Feltételes ciklus

A tevékenységcsoport ismételt végrehajtását egy logikai kifejezés értéke szabályozza. Ennek a feltételnek az értékétől függően kell újra és újra végrehajtani a ciklust vagy befejezni a ciklus végrehajtását. Feltételes ciklusnak két fajtájáról beszélünk, attól függően, hogy az a bizonyos logikai kifejezés, annak értékének megállapítása a tevékenységcsoport végrehajtása előtt, vagy után következik be. Ettől kezdő- vagy végfeltételes.

a) Kezdőfeltételes ciklus: a ciklusmag mindannyiszor végrehajtódik, amíg a feltétel igaz (ha a feltétel hamissá válik, abbahagyja az ismétlést).

b) Végfeltételes ciklus: mindaddig végrehajtódik a ciklusmag, amíg a feltétel hamis (mindaddig végrehajtja a ciklusmagot, mígnem a feltétel igazzá nem válik) - ez az általánosabb.

```
Pl.:  a:=0
Ciklus amíg a<10      ** ez a ciklusfej**
      Ki:a            **10-szer kerül kiírásra a értéke, azaz a 0,1,2,...,9 számok**
      a:=a+1          **a ciklusmagot két utasítás alkotja**
Ciklus vége           ** itt a ciklus vége**
```

```
a:=0
Ciklus
      Ki: a  ** 0-át ír ki**
      a:=a+1
Amíg a>0   ** egyszer hajtódik végre a ciklusmag, mivel a feltétel igaz**
```

2. Üres ciklus fogalma (terminológia)

A ciklust üres ciklusnak hívjuk, ha a mag egyetlen egyszer sem hajtódik végre (létezhet a mag). Tehát a végrehajtások számától tesszük függővé. Kezdőfeltételes ciklus lehet üres. Végfeltételes ciklus soha nem lehet üres, a mag legalább egyszer végrehajtódik.

Pl.: **Ciklus amíg Hamis**
Ki: "Én egyszer sem hajtódok végre"

Ciklus vége

3. Végtelen ciklus fogalma (terminológia, tárgyalása később)

A fentivel ellentétes, amikor az ismétlés (ciklus) nem áll le. Pl. a kezdőfeltételes ciklusnál igazra állítom a feltételt, akkor az végtelen ciklus; vagy végfeltételesnél hamisra állítom, akkor az végtelen ciklus - ettől sokkal elegánsabb végtelen ciklusok léteznek. Nagyon kellemes, mert a gép nem csinál semmit, csak végtelen ciklust, tehát hibát sem csinál. Szokás ezt a ciklust várakoztatásra használni.

4. Előírt lépésszámú ciklus:

Olyan ciklus, amelynek van egy ciklusváltozója - ez a ciklushoz tartozik. Ezen túlmenően tartozik hozzá egy tartomány, amely tartományból a ciklusváltozó fölveheti az értékeit. A ciklusváltozót és a tartományt is a ciklus fejével adjuk meg. A nyelvek a következőket szabályozzák és nyelvenként meglehetősen tömény eltérések vannak: Megmondják, hogy a ciklusváltozó a tartományból mely értékeket és milyen sorrendben vegyen föl. A sorrenden azt kell érteni, hogy növekvőleg, vagy csökkenőleg vegye fel az értékeket a tartományból.

Lehetőségek a következők:

1. Elképzelhető, azt írom elő, hogy a tartomány összes értékét föl kell venni (ilyenkor megmondhatom, hogy növekvőleg vagy csökkenőleg vegye fel a tartomány minden értékét).

2. A tartományból csak bizonyos értékeket vegyen föl, azonban ezek az értékek a tartományon belül egyenletesen helyezkednek el. Ebben az esetben, amikor nem minden értéket akarok felvenni, akkor az én feladatom, hogy a ciklusfejben megadjak egy olyan paramétert, előírást, amit úgy hívunk majd, hogy lépésköz - amely megmondja, hogy ezek a szabályosan elhelyezkedő értékek egymástól milyen távol vannak (tehát, hogy mely értékeket kell felvenni). A tartományt pedig úgy adom meg a ciklusfejben (minden nyelvben igaz), hogy megadom az alsó határát a tartománynak (kezdőértékét) és megadom a felső határát a tartománynak (végértékét). Az, hogy ez hogyan történik, az már nyelvenként különböző. Akkor előírt lépésszámú ciklus, ha megadtam a tartományt, megadtam, hogy abból hogy lehet felvenni az értékeket, akkor a fejnek az ismeretében meg tudom mondani, hogy hányszor fog végrehajtódni a ciklus. A fejet kiértékelve meg tudom mondani, hogy hányszor fog lefutni - a ciklusváltozó ugyanis az általam megfelelően paraméterezett módon fölveheti a tartomány minden értékét és mindannyiszor végrehajtódik a ciklusmag.

A ciklusfejben van

- változó
- van tartomány kezdőérték és végérték
- lehet lépésköz
- és meg kell mondanom az irányt, illetőleg vagy meg tudom mondani, vagy a nyelv definiálja.

Ilyen szabályosan csak kevés nyelvben fordul elő.

Felvetődő kérdések:

a) Milyen típusú lehet a ciklusváltozó? Ennek megfelelően milyen típusú lehet a kezdőérték, végérték és lépésköz? Általános válasz: minden nyelvben lehet egész típusú, minden nyelv ismeri. Maximális (legáltalánosabb) válasz: sorszámozható lehet a típus. (A legtöbb nyelvben nem lehet a típus valós.)

b) Hogyan lehet megadni a kezdőértéket, a végértéket és a lépésközt? Milyen formában, milyen módon lehet megadni? Minimális válasz: konstanssal. Maximális válasz: (megfelelő típusú) kifejezéssel. A nyelvek többsége ez utóbbit vallja. A kifejezés kiértékelésével, futás közben derülnek ki a paraméterek.

c) Meg lehet-e változtatni a ciklusmagban a ciklusváltozó értékét? A ciklusváltozónak az a szerepe, hogy vezérli a ciklus végrehajtását. A általam előírt módon felveszi az előírt értékeket. Mi van akkor, ha a ciklusváltozót odébb állítom? Nyelvenként különböző (igen, vagy nem és a kettő között az összes lehetséges) válasz lehet.

d) A ciklus lefutása után mi lesz a ciklusváltozónak az értéke? A változó nem köthető a ciklushoz (a ciklus előtt és után is lehet, itt most ciklusváltozóként használjuk). A korábbi nyelvek azt mondják, hogy a ciklus lefutása után a ciklusváltozónak van értéke. Mégpedig:

- vagy az az érték, amelyikkel utoljára lefutott a ciklus (a TP ezt vallja).
- vagy az az érték, amelyikkel éppen nem futott le a ciklus.

Általában ennek a kérdésnek a megválaszolása implementációfüggő. Egyes nyelveknél viszont a ciklus lefutása után a ciklusváltozó értéke meghatározatlan (számomra nincs értéke), tehát ha újból szabályosan akarom használni, akkor előbb értéket kell neki adnom.

Hogyan működik igazában az előírt lépésszámú ciklus?

1. Odaérünk a ciklusfejhez. Ha kifejezéssel vannak adva a paraméterek, akkor ezeket kiértékeljük. Innentől kezdve konkrét értékekkel rendelkezik a három paraméter: kezdőérték, végérték és lépésköz.

2. Ezek után a ciklusváltozó felveszi a kezdőértéket. (A TP-ben csak akkor, ha a kezdőérték, végérték és az irány értékei egymásnak nem mondanak ellent.)

3. Innentől kezdve kérdés az (és ez nyelvenként eltérő), hogy a következő lépés micsoda? A válasz itt is kettős.

Az előírt lépésszámú ciklus is lehet:

- előltesztelő
- hátultesztelő.

Kérdés még, mi az a vizsgálat, tesztelés? Az általános válasz az, hogy a tesztelés (vizsgálat) arra vonatkozik, hogy a ciklusfejben előírt paramétereknek (lépésköz, irány) megfelelően a kezdőérték benne van-e a tartományban?

1. Ha a kezdőérték nincs benne a tartományban, akkor elöltesztelő ciklus esetén nem hajtódik végre a ciklusmag (üres ciklus), hátultesztelő esetén pedig egyszeri lefutás után befejeződik.

2. Ha a kezdőérték benne van a tartományban, elöltesztelő esetben végrehajtja a ciklusmagot, hátultesztelő esetben pedig már végre hajtotta a ciklusmagot, tehát a ciklusmag egyszer lefutott. Ezután a lépésköznek és az előírt iránynak megfelelően veszi a tartomány következő elemét. A lehetőség megint kettős:
   a) Ha van még ilyen elem a tartományban, akkor ezt adja értékül a ciklusváltozónak.
   b) Ha nincs tovább ilyen a tartományban, akkor befejeződik a ciklus.

Pl.: **Ciklus** i:=1-**től** 12-**ig** ** a lépésszám 1, ha nem írunk semmit**
Ki:i
Ciklus vége ** Kiírja i értékét, vagyis 1-től 12-ig a számokat**

k:=8-2*1; n:=k+10
Ciklus i:=k-**tól** n-**ig**
  a[i]:=a[i]+1; n:=n+1
Ciklus vége

A ciklus fejében i felveszi k értékét, azaz a 6-ot, n értéke pedig 16 lesz. Ez után, mivel i értéke 6, az a[] nevű vektor 6. eleme (a[6]) 1-gyel nő és n értéke 17 lesz. A következő lépésben i felveszi a 6..16 intervallum következő elemét, vagyis a 7-et. Bár n értéke 17, az intervallum határai már a ciklusfej kiértékelése után rögzítettek. Így a[7] nő eggyel, majd a[8],a[9],...,a[16], i értékének változását követve.

5. Felsorolásos ciklus

Ha olyan értékcsoportom van, amelyben nincs szabályosság, akkor ezt az előírt lépésszámú ciklussal nem tudom leírni, mert nem tudom megadni a lépésközt, mert összevissza helyezkednek el az értékek. Erre szolgál a felsorolásos típus. Az előbbi ciklus valamilyen értelemben vett általánosítása. Van ciklusváltozója, azonban nem a tartományt adom meg, hanem konkrétan azokat az értékeket, amelyeket a ciklusváltozónak fel kell venni - annyi értéket sorolok fel, amennyit a ciklusváltozónak fel kell venni és ebben nincs szabályszerűség (a PASCAL nem ismeri).

6. Végtelen ciklus

Van egy kezdete, semmiféle feltétel és ciklusváltozó nincs, és van egy vég. Definíció szerint a végtelen ciklusnak a magja végtelenszer ismétlődik. Tehát az elején (fejben) és a végén nincs korlátozó feltétel. Természetesen azokban a nyelvekben, amelyek a végtelen ciklust ismerik mint fogalmat, azokban a ciklusmagban kell lenni olyan lehetőségnek,

amely végül is befejezteti a ciklust - ez általában egy külön utasítás szokott lenni, tehát a ciklusmagban fejeztetem be valamilyen módon a ciklust (a PASCAL nem ismeri, és a Turbo Pascal sem, de az utóbbinál van lehetőség végtelen ciklus készítésére, amelyből a Break utasítás használatával lehet kilépni).

7. Középfeltételes ciklus

Néhány nyelvben van egy olyan cikluskonstrukció, amelyet középfeltételes ciklusnak lehetne leginkább nevezni. Van olyan eszköz, hogy a ciklusmagban is, valamilyen feltételtől függően, ott is be tudja fejeztetni a ciklust. A ciklusmagban is van ciklusszervező utasítás (a PASCAL nem ismeri).

8. Összetett ciklusok

Az eddigiek voltak a tiszta ciklus válfajok. Vannak nyelvek, amelyek tudják kombinálni a ciklusokat. Pl. az előírt lépésszámú és a feltételes ciklust (van ciklusváltozója és van feltétele is). Ezek az összetett ciklusok.

A PL/I ismeri az összes ciklust, mind benne van (van benne vagy 30 féle ciklus, ami szintaktikailag mind különböző). A PASCAL nem ismeri az összetett ciklust.

III. PROGRAMEGYSÉGEK

Az eljárás-orientált nyelvek nem véletlenül viselik ezt a nevet. Ezen nyelvek mindegyike lehetővé teszi azt, hogy a teljes program szövegét feltördeljük többé-kevésbé önálló részekre. Ez az utasítás és a program közötti logikai egység. Többé-kevésbé önálló programrészek, amelyekből a program összeáll. Ezzel kapcsolatosan a következő kérdéseket kell a nyelveknek megválaszolni:

1. Az adott nyelvben milyen programegységek léteznek?
2. A programegységeket lehet-e külön fordítani, vagy csak együtt az egész programot? (Ha igen, akkor a tesztelés sokkal egyszerűbb.)
3. Hogyan kommunikálnak a programegységek egymással? Hogyan cserélnek információt?

Az alprogram

A programegység egyik formája. Sok más elnevezése is van. Ezek lényegében olyan programegységek, amelyek valamilyen bemenő adategyüttesből egy kimenő adategyüttest produkálnak. Az input adatokat valamilyen módon áttranszformálják output adatokká. Az alprogramok használatánál általában nem szoktunk azzal foglalkozni, hogy a transzformáció hogyan megy végbe. Tehát azt tudjuk, hogy milyen input adatokat kell beadni és azt, hogy milyet kapunk. Ezt úgy szokták mondani, hogy az alprogramok specifikációja érdekel bennünket, és nem érdekel az implementációja. A specifikáció, hogy miből mit csinál, de nem érdekel az, hogy hogyan csinálja.
Két fajtájuk van az alprogramoknak:

1. Az *eljárás*: olyan program, melynek a lényege, hogy valamilyen tevékenységet, vagy tevékenység- csoportot hajt végre. A tevékenység maga a lényeg (mit csinál és nem az, hogy hogyan).

2. A *függvény*: ezzel szemben egy olyan alprogram, amelynek a feladata egy értéknek a meghatározása (matematikai függvény értelemben). Nem a tevékenységen, hanem az érték kiszámításán van a hangsúly.

Egy alprogram olyan programozási objektum, melynek 4 összetevője van:

1. Név: Egy speciális azonosító
2. Paraméterek
3. Törzs
4. Környezet

A paraméterek vagy formális paraméterek lényegében egy azonosítólistaként adhatók meg, amely azonosítók a törzsben változónevek, nevesített konstansnevek, állománynevek vagy más alprogramok nevei lehetnek.

Lényegében a formális paraméterek az alprogram tevékenységének a leírásánál használhatók fel, és a kommunikációt szolgálják. Helyettesíteni kell őket - amikor fölhasználjuk majd az alprogramot - konkrét objektumokkal. Olyanokkal amelyek nevekként szerepelnek a törzsben. Tehát amikor felhasználjuk az alprogramot, akkor formális paraméterek helyére konkrét objektumokat kell majd helyettesíteni (majd meglátjuk, mit jelent az, hogy helyettesíteni). A működés leírását szolgálják. Ettől paraméteres nyilván, mert más-más felhasználásnál egészen más-más objektumokat helyettesíthetünk.

Alprogramok szintaktikája nyelvenként általában úgy néz ki, hogy van egy fej. Ebben van valamilyen speciális alapszó, mellyel eldönti, hogy eljárásról, vagy függvényről van-e szó, egyáltalán alprogramról van-e szó? A függvény fejrészében van megadva a függvény neve és a formális paraméterek. Nyelvenként változik, hogy ennek pontosan mi a szintaktikája. Függvény esetén a fejben még egy információ van: a kiszámított értéknek a típusa. Ez lényegében az alprogram specifikációs része.

Ez után van a törzs. A törzs tartalmazhat deklarációs részt, de ez nem kötelező. A törzs általában tartalmazza a végrehajtható utasításokat. (Eljárásnál azt a tevékenységet, amit produkál, függvénynél pedig azt, hogy hogyan kell kiszámítani az értéket.) A törzs ilyen értelemben lényeges, hiszen ez az implementációs rész. A legtöbb nyelvben valamilyen végutasítás zárja az alprogramot.

Környezet: azokat a neveket, amelyeket egy alprogram törzsében deklarálunk, az alprogram lokális nevei. Ezzel szemben van a neveknek az a csoportja, amelyeket nem az alprogramban deklaráltunk, hanem rajta kívül, de amely neveket az alprogram ismer. Ezek az alprogram globális nevei. A globális változók alkotják egy alprogramnak a környezetét. Egy eljárás ezek után a hatását általában a paramétereinek, vagy a környezetének a megváltoztatásával fejti ki. Ennél több igaz, mert végezhet még I/O műveleteket, stb.

Mellékhatás

Egy függvénynek is vannak paraméterei és létezik környezete. A függvénynek viszont az a feladata, hogy egy bizonyos értéket szolgáltasson. A függvény is megváltoztathatja a paramétereit és megváltoztathatja a környezetét, ezt azonban mostanában károsnak szokták minősíteni, mert nem ez a feladata a függvénynek. Éppen ezért, ha egy függvény megváltoztatja a paramétereit, vagy környezetét, akkor erre azt szokták mondani, hogy mellékhatás.

A függvény értékét a függvény neve hordozza. Kérdés, hogy a függvény nevéhez hol, hogyan és mikor rendelődik hozzá az érték? Többféle válasz lehetséges:

Vannak olyan nyelvek, amelyek megengedik, hogy a függvény neve mint változó szerepeljen. Ilyen funkcióban szerepeltessük a függvény nevét (szabályos változót).

2. Nyelvek másik csoportja azt mondja, hogy a függvény nevének a függvény törzsében értéket kell kapnia. Tehát nem használhatom változóként, de értéket kell kapnia. Hogyan kaphat a függvény neve értéket? Inputtal vagy értékadó utasítással. Az utoljára kapott érték fogja képviselni a függvényértéket.

3. A nyelvek harmadik csoportjában a függvény értékének a meghatározásához egy speciális utasítás szükséges. Külön utasítás van arra, hogy meghatározzuk a függvény értékét (függetlenül a függvény nevétől). Szinte kizárólagosan ez az utasítás:

RETURN kifejezés

Valamilyen módon meg kell adni egy kifejezést, mely ténylegesen az értéket adja majd.

Kérdés továbbá, hogy mikor ér véget egy függvény?

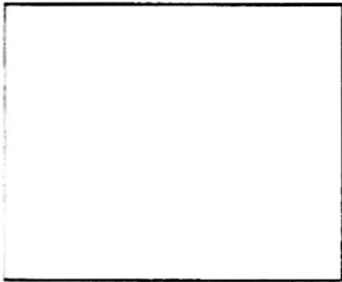
1. RETURN esetén egyértelmű. Csak egy RETURN hajtható végre egyidőben, mert az első RETURN befejezteti a függvényt.

2. Az első két típus esetén a következő a helyzet: a programozási nyelvekben egy függvény akkor ér véget, mikor elértük a végét. Ilyen értelemben kell valamilyen végjelzés. Nyilvánvaló a végigmenés nem a felírás sorrendjét, hanem a logikai (végrehajtás) sorrendjét jelenti. Közben akárhányszor értéket kaphat, a legutolsó érték fogja képviselni a függvényértéket. Ezek a normális befejeződések. Általában a nyelvek megengedik azt, hogy nem normálisan fejezzük be a függvényt. Pl. GOTO-val kilépünk. Valamint a legtöbb nyelvben vannak olyan utasítások, amelyek a program befejeződését jelentik. Ha alprogramban egy ilyen utasítást kiadok, akkor az alprogram is befejeződött. Vannak olyan utasítások, melyek befejeztetik az eljárást (RETURN), csak nem ad vissza értéket.

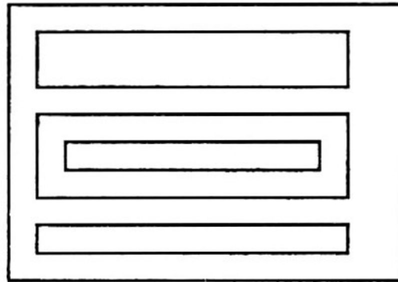
Visszatérve az eljárás specifikációra, ezen típusú nyelvekben a formális paramétereket kerek zárójelek között kell megadni. A formális paraméterek pedig azonosítók. Vannak olyan nyelvek, ahol a formális paraméterlistán nincs is más, csak formális paraméterek, más nyelveknél plusz információkat lehet vagy kell itt elhelyezni. Ezek olyan információk lehetnek, hogy az adott paraméternek milyen a típusa, vagy pedig meg lehet adni a paraméternek a futás közbeni viselkedésére vonatkozó információit. A PASCAL ezen utóbbi nyelvek közé tartozik, azaz még plusz információkat meg lehet, sőt meg is kell adni.

Egy program szerkezet a következőképpen nézhet ki:

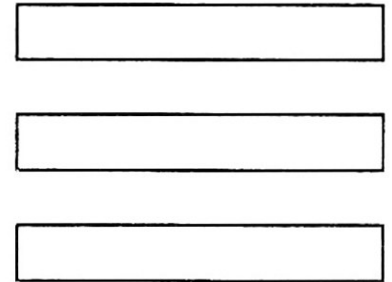
1)



2)



3)



1) Nincs a programnak semmilyen struktúrája (nincsenek programegységek), csak van maga a program.

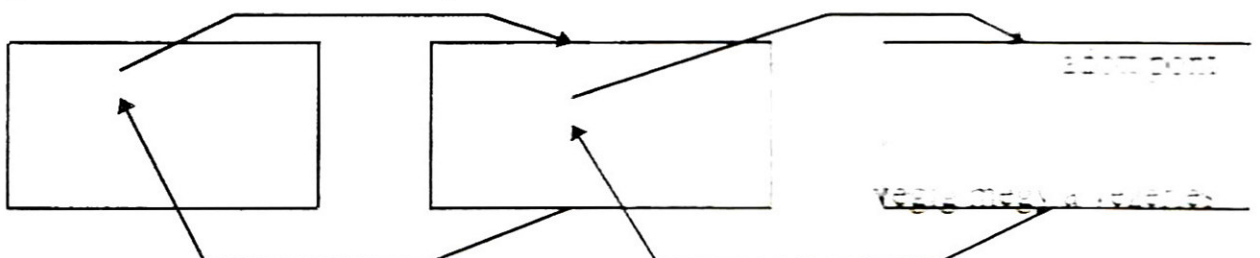
2) A programot úgy bontom fel programegységekre, hogy ezek a programegységek befelé strukturálják a programot. Van a program és a programban vannak programegységek, amely programegységeken belül ismét lehetnek programegységek stb. A programegységeket egymásba skatulyázzuk. Ennél a belső részek alprogramok, ill. majd lesz még más programegység is mint majd látni fogjuk. Van egy olyan programegység is, mely tartalmazza az egészet: ezt szokás *főprogramnak* hívni.

3) A programegységek fizikailag függetlenek egymástól, azaz egymás mellett léteznek. Ebben az esetben lehet ezeket az egységeket egymástól függetlenül fordítani, hiszen fizikailag önállóak. Ezek között is van egy kitüntetett programegység, melyet *főprogramnak* hívunk, amely logikailag összefogja az összes többit. Valamilyen módon egységbe kell foglalni ezeket a részeket. A főprogrammal kezdődik a program és vannak alprogramok. Ez utóbbi két eset kombinációja is elképzelhető. Pl. a standard PASCAL a 2-es esetet vallja. A TP ezen minimálisan túllép. Valamilyen értelemben vett fizikailag különböző strukturáltságot is meg tud csinálni. Ezeket a különálló struktúrákat úgy hívják, hogy UNIT-ok.

Alprogram hívása vagy aktivizálása:

Hogyan tudjuk használni az alprogramot? Miért jó az alprogram? Azért, mert bizonyos tevékenységcsoportot kiemelek, egyszer írom meg, felparaméterezem, flexibilissé teszem, és a program különböző pontjain valamilyen módon felhasználom. Az alprogram felhasználását hívásnak nevezzük, vagy másik terminológia szerint aktualizálom az alprogramot. Meg van írva az alprogram, és a program bármely pontjáról hívhatom

Az alprogram hívása a következőket jelenti:



Hívjuk meg az alprogramot egy adott pontról! Innen átadódik a vezérlés az alprogramra. Azon végigmegy a vezérlés. Amikor befejeződik, akkor visszakerül a vezérlés a hívó pontra. Tulajdonképpen ez egy vezérlésátadás. Természetesen a meghívott alprogram hívhat egy újabb alprogramot, és így tovább. Ezáltal előáll az ún. hívási lánc. Ennek mindenképpen van egy kezdeményezője, és a hívási láncban tetszőlegesen sok alprogram résztvehet. Egy függvény hívása csak kifejezésben történhet. Beírom a függvény nevét, és utána megadom az aktuális paramétereket.

Az eljárás hívására kétféle módszert használnak a nyelvek:
1. Külön utasítással történik az eljárás hívása. Ezen nyelvek többségében ez a CALL szó. Ez egy külön utasítás alapszó. Ezután megadom az eljárás nevét és az aktuális paramétereket.

2. A nyelvek másik részében nincs külön alapszó, hanem az az elv, hogy egy eljárást mindenütt hívhatunk, ahol utasítás szerepelhet. Tehát utasításszerűen hívjuk, vagy hívhatjuk. Pl. a PASCAL az utóbbit vallja.

Paraméterkiértékelés

Kérdés, hogy mi közük van egymáshoz a formális és az aktuális paramétereknek és hogyan megy végbe itt az információcsere? A paraméterkiértékelés az a folyamat, amikor egy alprogram hívása után egymáshoz rendelődnek az aktuális és formális paraméterek, miután meghatározódnak azok az aktuális paraméterjellemzők, amelyek majd a paraméterátadáshoz kellenek. Az általános szabály a következő: az aktuális és formális paramétereknek meg kell egyezniük sorrendben, számban és típusban. Ez az általános szabály, ám a kivételek az érdekesek.

1. A sorrendben való megegyezés annyit jelent, hogy az első aktuális paraméterhez valamilyen módon helyettesíti az első formális paramétert, a másodikhoz a másodikat stb. Ezt a szabályt úgy hívják, hogy *sorrendi kötés*. Ezt általában a nyelvek megkövetelik.(A PASCAL is ilyen).

2. Számban való megegyezés: annyi aktuális paraméternek kell lenni, ahány formális paraméter van. Kivétel az ADA, C, PASCAL. Általában a PASCAL-ban megvan a számbeli egyeztetésnek a szabálya. Ha én írok alprogramot, akkor ez igaz is, de léteznek a nyelvben standard, a nyelv által definiált alprogramok is (pl. READ). Tehát a standard eljárások, függvények esetén a PASCAL-ban sincs számbeli kötés. Ez annyit jelent, ha nincs számbeli kötés, akkor tetszőleges számú paramétert adhatok meg.

3. Típusban való megegyezés: Itt kell visszautalni a típus-egyenértékűség, típus-kompatibilitás és konverzió problémakörre (A PASCAL-nál típus-kompatibilitás van). Tehát valamilyen keveredés megengedett azoknál a nyelveknél, ahol a vegyes típusú kifejezések megengedettek, ott a típusegyeztetés annyit jelent, hogy a formális paraméter típusává konvertálhatónak kell lenni az aktuális paraméternek. Ezt a nyelv definiálja.

Paraméterátadás

Azt definiálja, hogy az aktuális paraméternek milyen komponense kerül át az alprogramhoz. Többféle formája létezik a paraméterátadásnak. Minden nyelvben többféle paraméter átadási mód lehet.

1. Érték szerinti paraméterátadás

A formális paraméterhez az alprogram területén van hozzárendelve tárterület. Tehát a formális paraméter rendelkezik címkomponenssel. A paraméterátadásnál az aktuális paraméter értéke kerül át a formális paraméter területére. Tehát aktuális paraméter olyan valami lehet, aminek van értékkomponense. És itt az érték kiszámítása a lényeg, tehát az aktuális paraméternek meghatározott az értéke és ez kerül át. Ez egyirányú információátadás. A hívótól a hívott felé. Az értékkomponens kerül át és az alprogram ezzel az értékkel a saját területén dolgozik.

2. Cím szerinti paraméterátadás

Ebben az esetben nem létezik címkomponense a formális paraméternek az alprogram területén. Az aktuális paraméternek a címe kerül át a formális paraméterhez. Aktuális paraméter olyan valami lehet, aminek van címkomponense, ezt meghatározzuk és ezt a címkomponenst adjuk át. Ebből az következik, hogy az alprogram nem a saját tárterületén fog dolgozni, hanem a hívó területére átnyúlkál a hívott alprogram. Ez kétirányú információ, mert amikor meghívtuk, és a hívó területén van valamilyen érték, akkor azzal az értékkel dolgozik, bármikor felülírhatja a hívott. Ez egy kétirányú információ átadás.

3. Érték-eredmény szerinti paraméterátadás

A formális paramétereknek van címkomponensük az alprogram területén. Az alprogram ott dolgozik a saját területén, nem nyúlkál át. A hívás pillanatában átkerül az aktuális paraméternek az értéke arra a területre, tehát indul, mint egy érték szerinti paraméterátadás. Amikor vége van az alprogramnak, akkor a formális paraméter értéke visszamásolódik a hívó területére. Ez is kétirányú információmozgásra jó, de közben nem áll fenn az a veszély, hogy a másik területére átmászkálok. Az alprogram saját területén dolgozik. Jól definiáltan átadok egy értéket és jól definiáltan visszakapok egy értéket.

4. Név szerinti paraméterátadás

A hívás pillanatában a hívótól átadódik az aktuális paraméternek a neve, és ezek után formálisan úgy látjuk, hogy az alprogram szövegében a formális paraméter nevét mindenütt felülírjuk az aktuális paraméter nevével. A hívás pillanatában, a szöveget írjuk felül olyan értelemben, hogy a formális paraméter nevét kicseréljük (helyettesítjük) az aktuális paraméter nevére. Ez szövegparaméterezés.

5. Szöveg szerinti paraméterátadás

A név szerintinek egy speciális esete. Nem a hívás pillanatában történik meg a felülírás (megint csak a név mozog, az aktuális paraméternek a neve fog átkerülni), hanem abban a pillanatban, amikor először találkozik a (fordító) szövegben azzal a

bizonyos formális paraméter névvel. Tehát működik az alprogram, és amikor először van hivatkozás az alprogramban a formális paraméter nevére, akkor történik meg a felülírás.

Blokk (a programegységek másik, speciális formája)

Van kezdete és vége, ami egy alapszó. Egyes nyelvekben van neve a blokknak. Van egy deklarációs és egy végrehajtható része. Egy blokk mindig csak egy másik programegység belsejében helyezkedhet el. Egy blokkot úgy lehet aktivizálni, hogy rákerül a vezérlés. Majd végrehajtódik és a végén kilépünk belőle. Nem lehet külön hívni a program más részéről. Viszont tetszőleges mélységben egymásba skatulyázhatók, egymás mellett megjelenhetnek stb.
Vannak kimondottan blokkszerkezetű nyelvek.

Nézzünk néhány példát!

1) **Eljárás** összead (a,b,c)
a:=b+c
Eljárás vége

Hívása: összead(10,8,2)

A 10,8,2 konstansokat nevezzük most *aktuális paramétereknek*, az *a,b,c* változókat pedig formális paramétereknek. A a,b,c felveszi a 10,8,2 numerikus konstansokat, majd az eljárás összeadja b-t és c-t és az eredmény az a-ban keletkezik. Ez egy nagyon szép példa, a probléma csak az, hogy az a változó csak az összead eljárásban ismert (ez a programrész a hatásköre, hiszen itt lett először deklarálva). Ez azt jelenti, hogy sikerült a két számot összeadni, de nem tudom az eredményt "megszerezni".
Erre találták ki (többek között) a cím szerinti paraméterátadást. Hiszen ha változik a formális paraméter (most az a változó) értéke, akkor a közös cím miatt az aktuális paraméter értéke is, vagyis megegyeznek. Ezért a 10 esetében használni kellene a cím szerint átadást. Sajnos azonban azt is tudjuk, hogy egy konstansnak (most a 10-nek) nincs címe. Így nincs is mit átadni, azaz ebben az esetben nem lehet az aktuális paraméter konstans (egyébként kifejezés sem). Így a megoldás a következő:

2) **Eljárás** összead (a,b,c) ** a,b,c egész, a esetén címszerint átadás, hívás történik**
a:=b+c
Eljárás vége[7]

Hívása: segit:=0;
összead(segit,8,2)

Az eljárás befejezése után a segít értéke 10 lesz, a korábbi 0 helyett

[7] A TP-ben cím szerint átadás esetén a Var szót kell a formális paraméter elé írni

3) Használhatunk függvényt is ebben az esetben:

Függvény összead (a,b,c) ** a,b,c továbbra is egészek **
a:=b+c
Függvény vége (Visszaad:a)

Hívása: eredmény:=összead(10,8,2)

A függvény átveszi a 3 db konstanst, majd az a változóba kerül a 10. A függvény visszaadja ezt az értéket a hívás helyére. **Erre azonban fel kell készülni!** Azaz fogadni kell a kapott értéket. Ezért használjuk az eredmény nevű változót, amibe a kapott összeg, vagyis a 10 kerül.

IV. ADATTÍPUSOK, ADATSZERKEZETEK

A programban mindenféle adatokat dolgozunk fel. Feldolgozás során lényeges, hogy milyenek lehetnek az egyes értékek, amelyekkel dolgozunk. Az adattípus valamilyen értelemben erre válasz, ez a fogalom a fejezet témája.

Az *adattípus* fogalmát a következőképpen definiálhatjuk: az adattípus megadása a típus értékkészletének, a rajta végzett műveleteknek, az értékek jelölésének és a tárban való ábrázolásának a rögzítését jelenti.

Az adattípusaink kétfélék lehetnek: *elemiek*, amelyek felhasználói szempontból nincs belső szerkezetük, valamint *összetettek*, amelyek elemiekből épülnek fel: ezek az adatszerkezetek.

Egy adattípust három dolog határoz meg:

1. Milyen értékekkel rendelkezhet, mi a tartománya?
(A tartományt nevezhetjük értékkészletnek is.)

2. Milyen műveleteket lehet vele végezni?

3. Minden egyes adattípus mögött van egy belső ábrázolás (adattípusra jellemző ábrázolási mód).

A programozás folyamán az utóbbi időben az terjedt el, hogy ne vegyük figyelembe, hogyan néz ki a belső ábrázolása az adattípusoknak. Hozzátartozik ugyan, de a programozás során nem illik foglalkozni vele (ezért van megadva a tartomány és a vele végezhető műveletek). Azért lényeges, hogy mi van mögötte, mi az adatábrázolási forma, mert korábbi nyelveknél nagyon lényeges volt az ábrázolás. Új nyelveknél irányvonal, hogy ezt ne tudjam, ne foglalkozzam vele - benne van, a rendszer dolga. Később sok szó fog esni a specifikációról és az implementációról. Adattípus esetén az alábbiakat jelenti a specifikációs és implementációs szint.

Specifikáció (specifikációs szint):
Mi az adattípus tartománya? Milyen műveleteket lehet vele végezni? Mire és hogyan lehet használni az adattípust?

Implementáció:
Hogy néz ez ki a gépen belül? Hogy jelenik meg az érték? Hogy van *megvalósítva* az adatábrázolás? Manapság az a törekvés, hogy a specifikációs rész ismert, az implementációs rész pedig nem.

Típusokkal kapcsolatban a nyelveknél a következő kérdéseket szokás megválaszolni:

1. Melyek az ún. *standard típusok* a nyelvekben? (Melyek azok a típusok, amelyek be vannak építve a nyelvekbe?)

Van-e olyan standard típus, amely minden nyelvben létezik? (Van, pl. az egész vagy valós - nem biztos, hogy így hívják, de ez a megfelelő típus.)
A standard típusok nyelvenként meglehetősen különböznek. Van egy közös mag, mely minden nyelvben van (nyilván az adott nyelvet kell megnézni, hogy milyen típusokkal dolgozik.)

2. Az adott nyelvben a programozó tud-e definiálni *(saját) típust*? A nyelvek egy része azt mondja: nincs, a nyelvek másik része azt mondja, van ilyen eszköz (pl. a PASCAL-ban, C-ben van).

Az alapvető kérdés az, hogy mit tekintünk elemi adatnak és mit összetettnek? Az elemi adatnak a program szempontjából nincs szerkezete, nem tudjuk egyes részeit külön kezelni, az összetett adatoknál erre van lehetőség. E definíció alapján a kérdésre nem tudunk egyértelmű választ adni. A válasz elsősorban a használt programozási nyelvtől függ.
A másik kérdés az, hogy miért kell foglalkozni az adatszerkezetekkel, az adatszerkezetek ábrázolásával, a rajtuk végzett műveletek megvalósításával? A kérdésre tulajdonképpen már adott a válasz első része: azért, mert az egyes nyelvekben nem létezik minden, számunkra szükséges adatszerkezet. A válasz másik része az adatszerkezeteket a programozási tételekhez (algoritmustípusokhoz) hasonlítja. Legtöbbször ugyanis nem akármilyen adatszerkezetet kell használnunk, hanem valamilyen nevezetes, gyakran használt típust (egész szám, valós szám, tömb ...), így az adatszerkezetek körében is elvégezhetjük a tipizálást.
Az adatszerkezeteket most mégis elemi és összetett adatokra bontjuk, s ezt a magas szintű nyelveknél szokásos felosztás alapján végezzük el.
Az adatok lehetnek egymástól függetlenek, és akárhány adattal dolgozhatok, amelyek egymástól függetlenek. Egy probléma megoldásán belül lévő adategyüttesek nem egymástól független, egyedi adatokból állnak, hanem adatcsoportokból - valamilyen logikai összefüggés érvényesül közöttük.
Az adatszerkezetek adják meg azokat a logikai összefüggéseket, amelyekkel le lehet írni a valós világ adatainak viselkedését, a valós világ adatai közötti összefüggéseket.
Az adatszerkezetek mindegyike adatelemekből áll. Más-más adatszerkezet más-más típusú adatokból építhető fel, de egy adatszerkezetben csak egy adattípus szerepelhet.
Az adatszerkezetek két csoportja:

- *statikus adatszerkezet*: adatelemek száma rögzített.
- *dinamikus adatszerkezet*: adatelemek száma változtatható.

Az adatjellemzők összefoglalása

Azonosító

Az a jelsorozat, amellyel hivatkozhatunk a program egyes részeire, így az adatokra is, azok tartalmára, amely által módosíthatjuk.

Pl.:

1. j, n, a - a változók nevei. A programban a változókra a nevükkel hivatkozhatunk. ami végül is egy azonosító

π, -128, 3.14, "Eredménye=", Igaz - konstansokat azonosító jelsorozat

Kezdőérték

A születéskor hozzárendelt érték

Konstansoknál nyilvánvaló, változók értéket kaphatnak deklarációnál, futás közben - a nyelv szabályozza.

Hatáskör

A hatáskör a program szövegének azon része, ahol egy név ugyanazokkal a jellemzőkkel rendelkezik. Más szóval: a programszöveg azon tartománya, amelyben az adathoz hozzáférés megengedett (nem független). Például az egyes eljárások használják egymás változóit, konstansait.

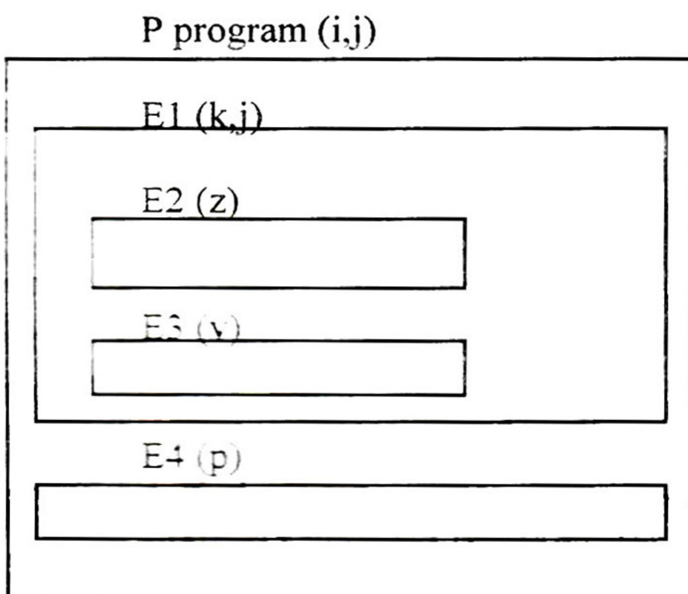
Lényeges, hogy névnek van hatásköre és lényeges, hogy ez egy szövegrész, a programon belül.

Ezzel kapcsolatban a nyelvek a következőket mondják:

Egy *lokális név* hatásköre az a programegység, amelyben az deklarálva lett, beleértve mindazon tartalmazott programegységeket is, ahol az adott nevet nem deklaráltuk újra. (Ha a programegység maga főprogram, akkor *globális* névről beszélünk.) Ha ott ugyanazzal a névvel deklaráltunk egy változót, akkor a két névnek semmi köze egymáshoz (újradeklaráltam a nevet, ugyanaz a név ott más jellemzőkkel ismert).

Hatáskör tehát szövegrész, pontosan meg kell mondanom, hogy ettől a szövegrésztől eddig a szövegrészig terjed. Így tehát a programegységek azok a szövegrészek ezekben a nyelvekben, amelyek a hatáskört igazából definiálják. Ilyen értelemben a blokk az egy olyan programegység, amely hatáskör elhatárolására, definiálására szolgál, szűkebb értelemben szokás ezt statikus hatáskörnek nevezni.

Nézzünk erre egy példát, ahol P az úgynevezett főprogram, E1,E2,E3,E4 programegységek!



- a P program deklarálunk egy i és egy j *nevű* változót, amelyek közül az i-t nem deklarálom újra. Ekkor ez az i változó lokális a P programra nézve és a P által tartalmazott programegységekre is. Vagyis erre a változóra a P program teljes területén tudok hivatkozni (hatásköre P). Mivel P a *főprogram,* ezért ez a változó globális a P programra nézve. Egy másik programban vagy programrészben, amelyet a P már nem tartalmaz természetesen ez az i nevű változó ismeretlen
- az E1 alprogramban deklarált k változó nem lesz ismert az E1 által nem tartalmazott részekre, például E4-ben sem (hatásköre: E1,E2,E3). Az E1-ben deklarált j-nek viszont semmi köze a P-ben megadott j-nek. Azaz újradeklarálásra került egy j nevű változó (hatásköre:E1,E2,E3). Mindaddig, amíg az E1-ből ki nem lépünk, addig az E1-ben deklarált lesz ismert, majd a kilépés után a P-ben megadott(ennek a hatásköre: E4 és P-nek E1-en kívüli része)[8].
- E2-ben ismert lesz i, z, k és j (az E1-ben deklarált).
- E3-ban ismert i, k, j (E1) és y (z NEM!).
- E4-ben ismert i,j (a P-ben deklarált) és p.

Élettartam

A futási időnek azt a részét nevezzük egy név élettartamának, amikor egy adott névhez van hozzárendelve tárrész, tehát van címkomponense, azaz a futási időnek az az intervalluma, amelyben az adat azonosítója végig ugyanazt az objektumot jelöli.
Csak olyan objektumok neveiről lehet szó, amelyeknél van értelme címkomponensről beszélni. A változóknál beszéltünk a tárkiosztásról. Ez ugyanaz, csak most már több fajta objektumot ismerünk, aminek van neve és címkomponense.

A lehetőségek a következők:

Statikus tárkiosztásnak megfelelően vannak olyan nevek, amelyek a program teljes futási idejében rendelkeznek tárterülettel. Az igazán izgalmas a következő:
A dinamikus tárkezelés a következőképp néz ki általánosan: egy programegység lokális neve a programegység aktivizálásának pillanatától rendelkezik tárterülettel, és ezt a tárterületet felszabadítja a rendszer akkor, amikor az illető programegység véget ér. Ez a teljes dinamikus tárkezelés (programegységhez kötődik a tárterület hozzárendelése - amikor él a programegység, van tárterület, amikor nem él, nincs tárterület).
Dinamikus tárkezelés tehát annyit jelent, ha írtam egy alprogramot, amikor aktivizálom először, akkor a benne szereplő neveknek itt van tárterület rendelve, mikor másodszor aktivizálom, akkor pedig ott van tárterület rendelve. Ez változik, pontosan ettől dinamikus.
Ennyiben lényeges a hatáskör, mert azt is a programegységhez kötöttük.

Például:
- a *globális változók* születésüktől a program teljes futásidejében élnek.

[8] P-nek j még "létező " változó, csak éppen hatásköre nem terjed ki E1-re

- a *lokális változók* csak addig élnek, amíg az az eljárás (programrész) aktív, amelyben deklarálták, ill. ezek is a kóddal együtt születnek meg fordításkor, hozzátapadnak, de nem "aktivizálódnak" amíg a kód megfelelő része nem "mozgatja".

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

Az értéktípus

az adatoknak az a tulajdonsága, hogy értékei mely halmazból származnak és tevékenységeknek (függvények, operátorok, utasítások) mely "készlete, amely létrehozza. felépíti, lerombolja és részekre bontja", alkalmazható rá.

Egyszerű típusok

1. Egésztípus

Értékhalmaz:	-32768...32767 (*Min'Egész...Max'Egész*)[9]
Műveletek:	+ , - , * , DIV (egész osztás) , ^ (pozitív egész kitevőjü hatványozás), MOD (maradékképzés), - (uniáris mínusz). Ezen műveletek végezhetők egész típussal
Relációk	= , <> , ≤ , ≥ , > , ≠ (nem egyenlő)
Ábrázolás:	(két byte-os) 2-es komplemens kód

Hagyományosan az egész számokat valahány bites (byte-os) 2-es komplemens kódban ábrázolják. (az egyes programnyelvek nemcsak 2 byte-os egészeket ismernek, így természetesen az értékhalmaz az egésztípustól függően változhat!)

A lényeg a következő:

1. az első bit az előjelbit, 0, ha a szám pozitív, 1, ha negatív;
2. a pozitív számok szokásos bináris számként tároljuk;
3. a negatív számokat - , hogy a lehető legegyszerűbb legyen velük a műveletvégzés - úgy tároljuk, hogy additív inverzét hozzáadva - elfelejtkezve az előjelbit különleges jelentéséről - 0-t (azaz csupa 0-bitből álló számot) kapjunk. Ezt így adhatjuk meg:

negatív szám=a szám abszolút értékének 1-es komplemense +1

Az 1-es komplemens nem más, mint bitről bitre komplementált szám (azaz bitenként 0 helyett 1-et írva, és fordítva). Példaként 1-byte-os 2-es komplemens kódban a

[9] Természetesen itt és később is az ilyen természetű adatokat egy konkrét ábrázolással összefüggésben kell érteni. Azért, hogy ilyen értékeket ne kelljen megtanulni, kerül bevezetésre a Min, Max konstans-függvények: ennek típushoz tartozó példánya fog majd minden egyes típusnál szerepelni.

$-1=11111111_2$, hiszen $1=00000001_2 \Rightarrow 11111110_2+1=11111111_2$

A fentiek alapján egy példa:

$$-1+1=11111111_2 + 00000001_2 = 10000000_2 = 0\ )$$

túlcsorduló bit

$-128=10000000_2$, hiszen $128=10000000_2 \Rightarrow 01111111_2+1=10000000_2$

Ennél egyszerűbb képzési mód a következő: jobbról az első egyesig leírjuk változatlanul a számot, utána a számjegyek ellenkezőjét írjuk.

Pl.: **Változó** i,j: *Egész*
Konstans N : *Egész*=35, vagyis N egész és értéke M

2. Valós típus

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

Műveletek: + , - , * , / (osztás) , ^ , - (uniáris mínusz)

Relációk: =, < , ≤ , > , ≥ , ≠

Ábrázolás : lebegőpontos normalizált alakú

Az ábrázolás általában a következőt jelenti:

valahány byte-os mantissza	1-byte-os kitevő (karakterisztika)

A mantissza a valós szám pontosságát, a kitevő a nagyságrendjét határozza meg.

Pl.: **Változó** P: *Valós*
Konstans pi: *Valós*=3.141592

3. Logikai típus

Értékhalmaz: Hamis..Igaz (csak ez a két értéke lehet)

Műveletek: Nem, és, vagy (a szokásos logikai műveletek

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

Ábrázolás: közönséges egész számként, Hamis=0, Igaz=-1 konvencióval (általában ... elképzelhető, hogy 1 biten (0= Hamis, 1=Igaz)

Pl.:

Változó Van_e: *Logikai*
Konstans Nem: *Logikai* =hamis

4. Karaktertípus

Értékhalmaz : 0..255 kódú jelek
(*Min'Karakter*.. *Max'Karakter*: 0, ill. a 255 kódú karakter)

Műveletek karakter-specifikus: nincs

Relációk $=, <, \leq, >, \geq, \neq$ (belső ábrázolásuk alapján)

Ábrázolás: valamely kódkészlet szerint történik a jel-kód egymáshoz rendelése alapján (pl. ASCII)

Az ASCII néhány kódjáról:
- 0-31: ún. kontroll karakterek
- 32-64: szóköz, szokásos írásjelek, illetve számjegyek (48-57="0"- "9");
- 65-91: nagybetűk;
- 92- : kisbetűk, grafikus jelek (erősen implementációfüggő)

Elképzelhető lenne ettől lényegesen eltérő hozzárendelés is, sőt az sem feltétel, hogy minden karakternek éppen 1 byte-os feleljen meg.(Pl.: Morze-ABC, Huffman-kód)

Pl.: **Változó** Betű: *karakter*
Konstans Space: *karakter*=" " vagy
Konstans Szóközök: *karakter*=" "

5. Mutató típus

Segítségével úgy tudunk hivatkozni egy változó (ami nem is igazán változó, hiszen nincs neve) értékére, hogy nem a nevét adjuk meg, hanem egy mutatót, ami az értéket tároló memóriabeli címre mutat.

Értékhalmaz: a memória-tartománybeli címe egy adott típusú (!) objektumnak
(*MinMutatottípus*..*MaxMutatott* típus értelmetlen!)

Miért nem akármire mutató? A mutató nem szám? Válasz: nem szám a szokásos értelemben, hiszen nem lehet vele aritmetikai műveleteket végezni, hanem cím. S mivel a mutatóval közvetlenül van módunk a memóriában matatni, ezért a rendszer a saját érdekében - és ezáltal a program írójáéban is- többszörös kontrollal várja a

mutatók használatát; másrészt persze ezt a tényt ki is lehet használni.

Műveletek:

konstrukciós függvény[10], amelynek szintaxisa: *Típus* (mutató típusú objektum),

Jelentése: az a (többnyire) összetett objektum, amire a mutató éppen mutat, természetesen akár az értékadás bal, akár jobb oldalán előfordulhat;

És egy másik konstrukciós eljárás, amelynek feladata, hogy létrehozzon dinamikusan - futási időben - egy adott típusú objektumot:

Létrehoz (mutató típusú változó), illetve egy "teljesebb" alakja:

Létrehoz (mutató típusú *változó*, mutatott típusú *kifejezés*);

ez a memóriából elfoglal a típusnak megfelelő méretű összefüggő tartományt, és ezt hozzárendeli az objektumhoz; majd ha kell, feltölti a tartományt kezdőértékekkel a kifejezés értéke alapján. Ha a helyfoglalás sikertelen volt, akkor a mutató paraméterében Sehova(Nil) cím kerül (ez egy előredefiniált konstans, aminek "fölismerhető" jelentése van);

Ahogy konstruálni lehet egy objektumot, úgy "lerombolni", megszüntetni is; pontosabban a helyét felszabadítani és visszaadni a szabad területhez:

Felszabadít (mutatótípusú változó),

ez a "destruálás" befejezésével a paraméterét "sehová" értékűvé állítja.

Relációk :

=, ≠ , (> , ≥, < , ≤) is elképzelhetők a belső ábrázolásuk alapján, bár nem sok értelme van)

[10] Ebben az esetben a konstrukciós függvény nem csak egyszerűsíti a leírást (amint [illegible] a később tárgyalandó összetett típusúaknál), hanem nélkülözhetetlen is. Ugyanis az ilyen dinamikus objektumoknak - mint látható - azonosítója nincs, amit a hozzáférésükhöz felhasználhatnánk, csak a címe. E címmel paraméterezett fenti függvény teszi lehetővé a mutatott objektumra való hivatkozást.

Ábrázolás: a címnek megfelelő szélességű tartomány (általában: 2-3 byte)

Pl.: **Változó** Mutat: ***Egész** (egy egészre fog mutatni)

Létrehoz (Mutat)
Mutat*:=12 (azon a címen, ahová a Mutat mutat 12 található)
Felszabadít(Mutat) (a Mutat a Nil-re mutat, így a tár egy része felszabadult))

6. Felsorolástípus

Mindazon típusokat, amelyek értékkészletét konstansainak egyszerű felsorolásával adhatjuk meg, felsorolástípusnak nevezzük. Speciálisan tehát ilyen az egész, logikai, karakter, de lehet bármilyen - a program írója által kreált - absztrakt(elvont) konstansokat tartalmazó ún. absztrakt felsorolás típus 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 nem definiált az értékük)
Következő(*típusbeli kifejezés*),
Előző(*típusbeli kifejezés*),
Rend(*típusbeli kifejezés*).

Relációk: =, ≠ , < , ≤ , > , ≥ (a felsorolás sorrendje egyben a rendezés is)

Ábrázolás : annyi biten, amennyi szükséges **Rend**(MAX'típus) ábrázolásához.[11]

Pl.:
Típus: *Évszak* = (Tavasz, Nyár, Ősz, Tél)
Változó június : *Évszak*
Értékadás: június:=Nyár

[11] Mivel a legtöbb fordítóprogram a kódba nem illeszti bele az absztrakt konstansok azonosítóit (csupán belső ábrázolású megfelelőik maradnak meg), ezért nem engedik az I/O műveletekben az ilyen típusú objektumok használatát.

Összetett adattípusok, adatszerkezetek

Az összetett adatok nem összefüggéstelen halmazai az adatelemeknek, hanem valamilyen sorrendi, szervezeti összefüggés van köztük.
Háromféle összetételi módot ismerünk:
- összetett adat azonos típusú elemek sokasága (tömb, szöveg, sor, lista, direkt állomány, indexelt állomány, bináris fa, és halmaz)
- az összetett adat különböző típusú részekből áll, mely részeket minden esetben egyenként kell megadni (rekord),
- az összetett adat itt is többféle típusú részekből állhat, de a részekre osztás is többféleképpen történhet, a részeket most is minden esetben egyenként kell megadni (alternatív szerkezet)

Az összetett adattípusok többsége azonos típusú elemek valamilyen sorozata (tömb, szöveg, verem, sor, lista, szekvenciális állomány, direkt állomány és indexelt állomány), melyeket a velük végezhető műveletek alapján különböztethetünk meg. Néhány lehetséges művelet:
- tetszőleges sorszámú elem értékének felhasználása, vagy megváltoztatása
- a sorozat utolsó eleme értékének felhasználása, vagy megváltoztatása,
- a sorozat elemszámának meghatározása,
- új elem felvétel a sorozat elejére,
- új elem felvétele a sorozat két adott eleme közé,
- a sorozat első elemének kivétele a sorozatból,
- a sorozat adott elemének kivétele a sorozatból,
- sorozat ürességének vizsgálata,
- a sorozat részsorozatának felhasználása vagy megváltoztatása.

Az adatszerkezetek elemeinek térbeli sorrendje és az adatelemek valódi sorrendje eltérő lehet. Az előbbit a fizikai, az utóbbit a logikai sorrendnek nevezzük. Az adatszerkezet ábrázolásánál arra kell törekednünk, hogy az elemekkel végzett műveletek során könnyen felhasználhatók legyenek a köztük fennálló szerkezeti összefüggések. Az adatszerkezeteket tehát úgy kell elhelyezni a tárban, hogy ne csak az adatelemeket, hanem a szerkezeti összefüggéseket is ábrázoljuk

Tömb

Általában statikus adatszerkezet, ami annyit jelent, hogy a tömb elemeinek száma rögzített (elemek száma nem változtatható). A tömbnél a logikai összefüggést az adatelemek között azok egymáshoz viszonyított elhelyezkedése adja (ez adja a szerkezetet). Beszélhetünk egydimenziós, kétdimenziós stb. tömbről. Minden esetben van a tömbnek legelső eleme (egy kitüntetett eleme) és valamilyen értelemben ehhez viszonyítjuk a többinek a helyét. Ez adja a szerkezetet és ez a szerkezet kötött.

Pl.: Egydimenziós tömbben meg tudom mondani, hogy az elsőhöz képest hányadik az illető elem.

Pl.: Kétdimenziós tömb (mátrixnak nevezik) esetén szintén az elsőhöz képest mondjuk meg, hogy az elsőhöz képest hányadik sor, hányadik oszlop - ha megmondom, hogy hol van, azt úgy hívjuk, hogy indexelés - adott szerkezetben adott elemre az indexével hivatkozom, az indexet pedig valamilyen értelemben mindig az elsőhöz viszonyítom - ilyen értelemben index lehet bármi (bármilyen jellegű hivatkozás) ami egyértelmű. Az elsőt kell rögzítenem valamilyen értelemben.

Egydimenziós esetben, ha a tömb neve A, akkor az A[2] alatt a tömb 2. elemét értem. Kétdimenziós esetben A[3,2] jelenti a mátrix 3. sorának, 2. oszlopában lévő elemét.

Olyan programozási eszköz tehát, melyet neve, dimenziója vagy dimenziói, elemeinek típusa, indexeinek típusa és tartománya jellemez. A tömb neve egy tetszőleges azonosító, a programozó választja, akárcsak a dimenziók számát. A dimenziók számát illetően a nyelvek meg szoktak állni valahol, ez általában 255, néha 7. Háromdimenziós tömbbel szinte minden probléma megoldható, ennek részei: sor, oszlop, lapok. Tovább már szemlélet nincs, nem is kell, de lehet (pl. idő).
Lényeges, hogy a tömb elemei milyen típusúak lehetnek. Nem minden nyelv engedi meg, hogy bármilyen tömb tetszőleges elemekből álljon, a PASCAL nagyjából igen. A PASCAL-ban tetszőleges típusú tömböket lehet felépíteni a megadott elemekből, kivétel a Fájl.
Lényeges az is a nyelvekben, hogy az indexek típusa milyen lehet. Szinte minden nyelv tudja kezelni az egész típusú indexekkel rendelkező tömböket. A PASCAL-ban az indexek típusa bármilyen sorszámozott típus lehet.
Nyelvfüggő, hogy az indexek tartománya mekkora lehet, különös tekintettel arra, hogy az indexek tartományát hogyan lehet előírni. Egyes programozási nyelveknél az indexek tartományának alsó határa rögzített a nyelv által, nem befolyásolhatom. Más nyelvek esetében a programozónak kell megadnia az index típusát és tartományát, vagyis az indexeknél meg kell adni egy alsó és egy felső határt, amivel definiálhatjuk az indexek tartományát. Ilyen pl. a PASCAL is.

A tömb egészére a nevével hivatkozhatunk. A tömb egy elemére pedig úgy, hogy a név mellett megadjuk az illető elem indexét vagy indexeit.

név[index(ek)]

Ha kifejezéssel adom meg az indexet, akkor a kifejezés értékének a deklarált indexhatárok tartományába kell esni, valamint a kifejezés típusának meg kell egyezni az index definícióban megadott típussal. Vannak olyan nyelvek, amelyek lehetővé teszik, hogy többdimenziós tömbnél valamelyik dimenzióban elhelyezkedő összes elemre egyszerre lehessen hivatkozni. Pl. kétdimenziós tömbnél egy sorra lehet hivatkozni. A PASCAL is ilyen.
Lényeges még - a hivatkozási nyelv vagy az implementáció dönti el -, hogy a tárban hogyan jelenik meg a tömb, hogyan tárolódik a tömb? Válasz: sorfolytonosan vagy oszlopfolytonosan. Sorfolytonos akkor, ha felírásnál a legutolsó index változik a leggyorsabban. PASCAL-nál az elhelyezés sorfolytonos. Ez nyelvfüggő, legelőször

mindig a legelső elem tárolódik. A tárolás többdimenziós tömbnél lényeges. Lényeges tudni, hogy sor, vagy oszlopfolytonos a tárolás, mert a tömb nevének leírása az összes elemre való hivatkozást jelenti és lényeges, hogy milyen sorrendben vannak az elemek. Ez többdimenziós tömb egydimenziós tömbre való leképezése.

Bizonyos nyelvek tudnak olyan kifejezéseket kezelni, melyekben tömbneveket szerepeltethetek. Vannak olyan nyelvek, amelyek képesek a tömbökkel úgy műveleteket végezni, hogy a tömb minden elemével műveletet végeznek. Ilyen pl. a PL/I. A PASCAL nem ilyen, csak a tömbelemekkel lehet műveleteket végezni.

Ha a tömb statikus, akkor az azt jelenti, hogy a fordítás pillanatában eldől a mérete. A deklarálásnál fix alsó és felső határokat adok, fix a méret és ez nem változtatható meg a futás során. A PASCAL-ban az indexek határa hozzátartozik a tömb típusához.

Strukturálás :	Tömbtípus = *Tömb* (Indextípus: Értéktípus)
Értékhalmaz :	az alaphalmaz "Indexterjedelemszeres" hatványa (indexterjedelemnyi értéktípusú elemek "egyesített" halmaza)
Műveletek:	szelekciós függvény, amellyel az egyes elemek kiválaszthatók: "(.)", ahol a "." egy indextípusú kifejezést jelöl
Relációk :	= (tömbelemenkénti egyezés), ≠
Ábrázolás :	Indexterjedelemnyi értéktípusú elem egy folytonos tártartományon (általában!)

Pl.: **Konstans** n : *Egész*=(99

Típus Elem=*Valós*

Vektor = *Tömb[*1..n] *Elem* ** *n elemű vektor, elemei Valósak*

Változó v: *Vektor*

i: 1..n

Be: v[]

Ciklus i:=2-**től** n-1-**ig**

Ha v[i] T tulajdonságú **akkor** v[i]:=(v[i-1])−v[i−1])/2

Ciklus vége

Vagy

Változó Mátrix: *Tömb*[1..10,2..20] *Egész*

10*19 elemű mátrix, elemei Egészek. 10 sora van és 19 oszlopa. A Mátrix[2,3]:=1 egy érvényes értékadás, de a Mátrix[1,1] már nem, hiszen az oszlopindex számozása 2-től kezdődik.

Szöveg

A szövegtípus hasonlít a tömbhöz. Elemei csak karakterek lehetnek, viszont az elemek számát (a szöveg hosszát) több nyelv nem tartja állandónak. Tehát ez egy olyan sorozat, amelynek az elemszáma változhat is. A másik különbözőség, hogy egyszerre nemcsak egy elemmel lehet dolgozni, hanem többel, akár az összessel is. Azaz a szövegtípusra létezik az egymás után írás művelete, ill. a részképzés. Ez utóbbit néha függvényekkel valósítják meg (bal oldali valahány karakter, jobb oldali valahány karakter, tetszőleges részből valahány karakter)

Ha egy szövegtípusú változó hossza (karakterszáma) futás közben változhat, akkor a tárolása kicsit bonyolulttá válik. Meg kell adni minden esetben, hogy hol található egy ilyen változó, valamint azt, hogy aktuálisan hány karakterből áll. Ezt a két adatot a programnyelv elrejti előlünk és csak speciális függvényeken keresztül teszi elérhetővé.

Strukturálás: annak ellenére, hogy egy nyilvánvalóan összetett típusról van szó, nincs értelme strukturálásról 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: Maxhossznyi karaktersorozatok halmaza (karaktertípus Maxhossz-szoros direktszorzata)

Műveletek : Hossz, + , Balrész , Jobbrész , Jele (adottadik jele), Üres ("")

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

Ábrázolás : általában egy "kvázirekordként" valahogy így:
Rekord(hossz : 0.. Maxhossz, jele: tömb(1..Maxhossz: *Karakter*))
Más nyelvek a szöveg hosszát nem a szöveg elején, a nulladik byte-on tárolják, hanem a szöveg végét egy terminális[12] jel jelzi, pl. "/0"

Pl.: **Típus** NÉV = *Szöveg*[30]

---

[12] Terminális: valamilyen sor -v. láncolat utolsó tagját alkotó , valaminek a végén lévő.

Konstans Utolsóutáni: *Név*="zzzzz" ** a konstans *Név* típusú, vagyis 30 hosszú szöveg.

Verem (stack)

Úgy definiálandó, hogy hogyan kell kezelni. Nem a szerkezete definiálja, hanem a kezelési módja.

A verem olyan adatszerkezet, melybe bevinni elemet az eddig már bentlévő elemek után lehet. Az első elemet a verem aljára teszem, a következőt ennek a tetejére és így tovább. Veremről lévén szó, mindig csak a legfelsőt látom. Egy veremből mindig csak a legutoljára bevitt adatelemet tudom kiemelni. Az, hogy kiolvasok egy elemet a veremből, az egyben az adott adatelemnek az eltávolítását is jelenti. Ebből látszik, hogy dinamikus szerkezet a verem (ha pakolok bele, nő, ha kiolvasok belőle, csökken).

Az adatszerkezet rövidített neve: LIFO (utoljára be, először ki).

Strukturálás:	Veremtípus = *Verem* (Elemtípus)
Értékhalmaz:	az alaphalmaz iteráltja (reprezentációfüggő mennyiségű elem sokasága)
Műveletek:	Verembe, Veremből, Üresre állítás, Üres?
Relációk:	= (pillanatnyi mélysége és elemei tartalma alapján) ≠
Ábrázolás:	folytonos, láncolt sorozata az elemeknek, kiegészítve egy a verem tetejét kijelölő mutatóval

Ez a gyakran használt struktúra, amely a gépi kód kivételével nagyon kevés nyelvben fordul elő. Ezért ha szükségünk van rá, akkor nekünk kell megvalósítani. A verem adatok sorozatát tartalmazza, de csak speciális műveletet engedünk meg velük kapcsolatban. Ezek a következők:

Be(X) - egy x értéket a verem tetejére, a sorozat végére helyez
Ki(X) - a verem tetején, a sorozat végén lévő értéket az X változóba teszi, majd a veremből elhagyja.

A verem tehát egy olyan sorozat, amelynek csak egyik végét tudjuk kezelni, oda tehetünk be új elemet és onnan vehetünk ki elemet.

Megvalósítás egy vektor (V) segítségével történhet. Ha ennek dimenziója N, akkor nyilván max. N eleme lehet a veremnek, és kell első elemnek lennie.

Műveletek:

Eljárás Üresre_állítás
    VeremMutató:=1
Eljárás vége

Eljárás Be(X)
Ha VeremMutató>N **akkor Ki**:" Betelt a verem"
Egyébként
V(VeremMutató):=X
VeremMutató:=VeremMutató+1
Elágazás vége
Eljárás vége

Eljárás Ki(X)
Ha VeremMutató=1 **akkor Ki**: "Üres a verem"
Egyébként
VeremMutató:=VeremMutató-1
X:=V(VeremMutató)
Elágazás vége
Eljárás vége

A veremszerkezet nélkül nincs számítástechnika ⇒ számos alkalmazása van. Ilyen például az eljárások hívásának szervezése. Ha egy A eljárás meghívja a B eljárást, akkor tárolni kell azt a címet, amelyre a B eljárás végrehajtás után vissza kell térni. Ha a B eljárás még egy C eljárást is hív, akkor a B-beli folytatás címét szintén tárolni kell. A verem alkalmazásával ez egyszerűen megoldható. Minden eljáráshívásnál a verembe tesszük az a címet, ahol folytatni kell az eljárás befejezése után a végrehajtást. Az eljárás végén pedig ezt a címet ki kell venni a veremből, majd ezen címen kell folytatni a végrehajtást.

Sor

Strukturálás:	Sortípus = *Sor* (Elemtípus)
Értékhalmaz:	az alaphalmaz iteráltja (a megvalósítástól függő mennyiségű elem sokasága)
Műveletek	Sorba, Sorból, Üresre állítás, Üres?
Relációk:	= (pillanatnyi hossza és elemei tartalma alapján) ≠
Ábrázolás:	folytonos, láncolt sorozata az elemeknek kiegészítve egy az első és az utolsó elemet kijelölő mutatóval

A sor egy kicsit hasonlít a veremre. Ez olyan sorozat, amelynek az egyik végére lehet tenni új elemeket, a másik végéről pedig el lehet venni őket. (Gondoljunk például egy mozi előtti sorra.) Így a művelet:

Sorba(X) - berakja a sor végére az X elemet
Sorból(X) - kivesz a sor elejéről egy értéket X-be

A legegyszerűbb megvalósítás szintén egy vektorral történik (SOR(N)), de itt két mutatóra van szükség: tudni kell, hogy hová lehet tenni új elemet (HOVA), illetve honnan

lehet kivenni (HONNAN). Itt is szükség lesz egy új műveletre, a kezdőértékadásra, valamint vizsgálni kell, hogy a vektor végére értünk-e, illetve hogy kiürült-e a sor.

Műveletek:

```
Eljárás Üresre_állítás
    Hova:=1; Honnan:=1; Tartalom:=0
Eljárás vége

Eljárás Sorba(X)
    Ha Tartalom=N akkor Ki:"Betelt a sor"
    Egyébként S(Hova):=X
            Hova:=Hova+1
            Tartalom:=Tartalom+1
            Ha Hova >N akkor
                    Hova:=1** kezdem előről **
             Elágazás vége
    Elágazás vége
Eljárás vége

Eljárás Sorból(X)
Ha Tartalom=0 akkor Ki:"Üres a sor"
Egyébként
    X:=S(Honnan)
    Honnan:=Honnan+1; Tartalom:=Tartalom-1
    Ha Honnan>N akkor
            Honnan:=1       **kezdődik minden előről **
    Elágazás vége
Elágazás vége
Eljárás vége  ** a Sor ciklikusan lett tárolva!!**
```

Lista

Olyan adatszerkezet, ahol minden listaelem két részből áll:

1. Adatrész: ez tartalmazza az adatelemet.

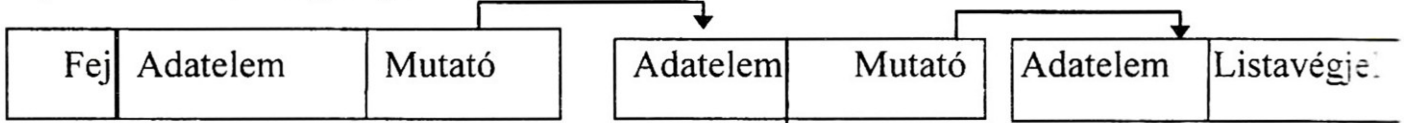
2. Mutatórész: feladata, hogy megmondja, az adott listaelem után melyik másik listaelem következik (a mutatórész a szerkezetét adja a listának).

A lista olyan adatszerkezet (más megfogalmazásban), amelynek van első és utolso eleme, és az utolsó elemtől eltekintve minden elemnek van rákövetkezője. Ilyen értelemben a mutatórész feladata, hogy megmutassa, az adott elemnek mi a rákövetkezője. Ahhoz, hogy a listát kezelni tudjam, kell még két információ:

1. A legutolsó listaelem mutatóját, aminek nincs rákövetkezője (valamilyen speciális jel, pl. csillag karakter, vagy speciális mutató érték, amely nem mutat tovább, hiszen itt a lista vége - megegyezés, hogy a két jelzés közül melyiket használják)

2. Kell egy olyan információ, hogy hol kezdődik a lista (hol van az első elem, hiszen erre nem mutat semmi, nincs előzője) - ezt hívjuk a lista fejének, mely nem listabeli elem - a fej a lista kezeléséhez szükséges listán kívüli információ.

A lista lerajzolható, mely a lista egyfajta reprezentációja (implementációja). Implementáció, ha így rajzolok.



Szűkebb értelemben szokás ezt a szerkezetet egy irányban láncolt listának nevezni. A továbbiakban a jelző nélküli lista ez a lista lesz.

A lista más változatai:

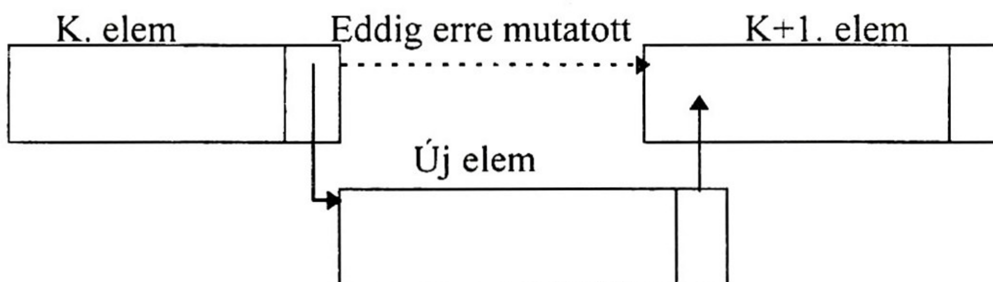
- *két irányban láncolt lista:* a listaelem mellett két mutatórész van, az adott listaelem nemcsak a rákövetkezőre, hanem az előzőre is mutat

-*ciklikus lista*: az utolsó elem visszamutat az elsőre - listafej változatlanul van, mert valahol kezdenem kell a listát

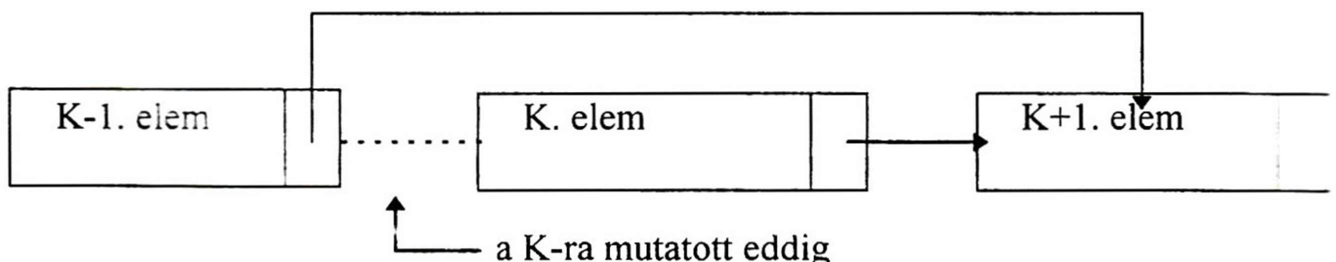
-*multilista*: adatelemeknek egy együttesét több listába fűzöm össze, és ezek a listák tetszőlegesen keresztezik egymást - egy adott adatelem több listának is eleme. Ebben az esetben több mutató van, egy-egy konkrét listához tartozik egy-egy mutató.

Az eddigi adatszerkezeteknek a szerkezetét ténylegesen a szerkezet adta. Valamilyen értelemben az elemeknek az egymáshoz való viszonya adta meg a szerkezetet (Hol van? Melyikre mutat?)

Ebből pedig az látszik, hogy a láncolt ábrázolásnál az adatszerkezet helyfoglalása megnövekszik. A láncolt ábrázolás előnyei az adatszerkezet módosításakor mutatkoznak meg. Egyszerű a listába új elemet *beilleszteni*, pl. a K és a K+1, közé (a K-adik elem mutatóját át kell állítani!):



A K. elem *törlés* csak a (K-1). elem mutatójának megváltoztatását jelenti:



Ábrázolás: az adatokat az ADAT(N) vektorban tároljuk, a következő elemre mutatókat pedig a KOV(N) vektorban. Tartalmazza a logikailag első elem sorszámát a KOV(0)

Műveletek:

Az első elem címének meghatározása:

Eljárás Első(Mutató):
Mutató:=KOV(0)
Eljárás vége

A következő elem címének meghatározása:

Eljárás Következő(Mutató)
Mutató:=KOV(Mutató)
Eljárás vége

Új elem beillesztése a listába, a K. után a fizikailag H. helyre (ADAT(H) az érték, KOV(H) lesz a mutatója):

Eljárás Újelem(k,H,x):
Első(Mutató); i:=1
Ciklus amíg i<k
Következő(Mutató); i:=i+1
Ciklus vége
KOV(H):=KOV(Mutató): KOV(Mutató):=H
ADAT(H):=x
Eljárás vége

A K. elem törlése a listából:

Eljárás Törlés(K)
Mutató:=0; i:=1
Ciklus amíg i<k
Következő(Mutató): i:=i+1
Ciklus vége
KOV(Mutató):=KOV(KOV(Mutató))
Eljárás vége

Kétirányú lista esetén az adatelemekhez még egy mezőt illesztünk, amely az őt megelőző elem címét adja meg. Az egységesség kedvéért célszerű a listafejet megduplázni.

Feladat:Adjuk meg a műveleteket!

Strukturálás: Listatípus = *Lista* (Elemtípus)

Értékhalmaz: az alaphalmaz iteráltja (a megvalósítástól függő mennyiségű elem sokasága)

Műveletek: Elejére, Következőre, Előzőre (kétirányú lista esetén), Töröl, Üresre állítás, Elem (az aktuális elem elérése függvény), Üres?

Relációk: = (pillanatnyi hossza és elemi tartalma alapján) ≠

Ábrázolás: folytonos, láncolt sorozata az elemeknek kiegészítve egy az aktuális és az első elemet kijelölő mutatóval

V. ALGORITMUSLEÍRÓ ESZKÖZÖK

Az első fejezetben, a feladatmegoldás lépései között már találkoztunk az algoritmuskészítés fogalmával. Most az algoritmus leírásának eszközei közül kerül bemutatásra néhány, természetesen a teljesség igénye nélkül. Ezeknek az algoritmusleíró eszközöknek célja a megoldás menetének géptől független, szemléletes, a logikai gondolatmenetet, a szerkezeti egységeket világosan tükröző megjelenítése.

Előnyei:

- a szemléletes algoritmusleírás könnyebben áttekinthető és követhető bárki számára,
- a világos algoritmusszerkezet gyakran sugall korábban nem is sejtett egyszerűsítő megoldásokat, esetleg egy adott probléma valamilyen irányba történő továbbfejlesztési lehetőségeit is.

1. Mondatszerű leírás

Amint ezt a név is mutatja, az algoritmus egymást követő lépéseit mondatokkal vagy mondatszerű szerkezetek egymásutánjával írjuk le. A mondatszerű leírás két fajtája különböztethető meg.

a) Egyszerűbb az, amikor mondatok sorozata írja le a feladat megoldását. Ezeket a mondatokat sorszámmal látjuk el. Erre azért van szükség, mert a programok legnagyobb részénél feltétel nélkül vagy valamilyen feltételtől függően el kell térnünk a szekvenciális végrehajtástól és ilyenkor hivatkozhatunk a megfelelő sorszámra.

b) Szemléletesebb az, amikor mondatok helyett csak ún. mondatszerű szerkezetek egymásutánjával írjuk le a feladat megoldását. Ennél a leírási módnál az algoritmus szerkezeti egységeit bekezdéses struktúra segítségével szemléltetjük. Nézzük meg, milyen utasításokat tartalmaz az így készített algoritmusleíró nyelv, és utasításfajtánként hogyan definiáljuk ezeket a mondatszerű szerkezeteket!

Beolvasó és kiíró utasítás

Formája:

Be:	(..
változók felsorolása	(az adatokkal szemben támasztott követelmények)
Ki:	(..
kifejezések felsorolása	a kiírás formájára vonatkozó követelmények

Értékadó utasítás

Formája:

................:=..................
Változó kifejezés

Az utasítás hatására a változó felveszi a kifejezés aktuális értékét. A változó típusának és a kifejezés értékének a típusának kompatibilisnek kell lennie egymással, különben futási hiba, illetve értékvesztés következhet be. (Azt, hogy melyik típus kompatibilis a másikkal, azt az adott programnyelv illetve annak reprezentációja definiálja, vagy ha mégsem, akkor valószínűleg konvertálást végez, ami szintén a nyelvben van definiálva)[13]

Elágazások (feltételes utasítások)

Formája:

a) **Ha** .... **akkor** ....
valamilyen feltétel utasítás(ok)
Elágazás vége

Jelentése:
Ha az adott feltétel teljesül, akkor az utasítás(ok) kerül(nek) végrehajtásra, és a feladat megoldása az **Elágazás vége** után folytatódik. Ha a feltétel nem teljesül, a program egyből az **Elágazás vége** után folytatódik.

b) **Ha** .... **akkor** ....
valamilyen feltétel utasítás(ok)1

Egyébként ....
utasítás(ok)2

Elágazás vége

Jelentése:
Ha a feltétel teljesül, akkor az utasítás(ok)1 kerül(nek) végrehajtásra, különben az utasítás(ok)2. A feladat megoldása minden esetben az **Elágazás vége** után folytatódik. Ha az elágazás ágain egyetlen utasítás van, akkor az "Elágazás vége" szöveget nem kötelező kiírni, bár ez nyelvfüggő.

[13] Megj.: Egyes nyelvek a "=" szimbólumot használják a " := " helyett értékadásra.

Elágazás (többágú szelekció)

Elágazás

.... : ....
feltétel 1 utasítás(ok)1

.... : ....
feltétel 2 utasítás(ok)2

.
.
.

.... : ....
feltételn utasítás(ok)n

Egyébként ....
utasítás(ok)n+1

Elágazás vége

Jelentése:

Ha a feltétel i (i=1,2,...,n) teljesül, akkor a neki megfelelő utasítás(ok)i hajtódik végre, majd a végrehajtás az elágazás vége után folytatódik.[14] Ha egyik feltétel sem igaz, akkor az egyéb esetnek megfeleltetett utasításokn+1 hajtódnak végre, majd a végrehajtás ismét csak az elágazás vége után folytatódik. Ha "egyéb esetben" nem létezik és egyik "esetén" sem teljesül, akkor az utasítás nem csinál semmit.

Ciklusszervező utasítások

Előírt lépésszámú ciklus

Formája :

Ciklus ....:= ....**-től** ....**-ig** ....**-sével**
ciklus változó kezdőérték végérték lépésköz

a ciklusmag utasításai

Ciklus vége

Jelentése:

A ciklusmag utasításait a ciklusváltozó kezdő- és [illegible] a lépésköznek megfelelő számszor kerülnek végrehajtásra.[15]

---

[14] A C++-ban például minden feltétel kiértékelődik egymás után.

[15] Megj.: Vannak nyelvek, amelyek nem támogatják a lépésköz megadását [illegible] mások esetében ha a lépésköz 1 vagy -1, akkor elhagyható a megadás.

Elöltesztelős ciklusfajta, azaz ha a ciklusváltozó kezdőértéke a megadott lépésköz esetén sohasem lesz képes elérni a végértéket, akkor a ciklusmagban lévő utasítás(ok) egyszer sem kerülnek végrehajtásra. Természetesen a ciklusváltozó értéke a cikluson belül is változtatható, de ez sok esetben nem szerencsés.

Feltételes ciklus

Ciklus amíg ....
feltétel

.
. a ciklusmag utasításai
.

Ciklus vége

Jelentése:
A ciklusmagot addig kell végrehajtani, amíg a feltétel igaz, majd a ciklus végét követő utasítás lesz a következő végrehajtandó utasítás. Ha a feltétel nem teljesül, a végrehajtás a ciklus vége után folytatódik. A ciklus elöltesztelős, azaz ha a feltétel már a ciklusra történő "rálépés" esetén sem teljesül, akkor a ciklusmag utasításai egyszer sem hajtódnak végre.

Ciklus

.
. a ciklusmag utasításai
.

amíg ....
feltétel

Jelentése:
A ciklusmagot addig kell végrehajtani, amíg a feltétel hamis. A ciklusmag utasításai legalább egyszer biztosan végrehajtódnak, majd csak ezután kerül megvizsgálásra a feltétel teljesülése (hátultesztelős ciklus)[16]

Eljárások

Bonyolultabb, összetett feladat esetén a teljes algoritmust a feladat struktúrájának megfelelően többszörös mélységben készítjük el. Az egyes feladatmegoldási szinteken a következő szint algoritmus- részeire eljáráshívásokkal hivatkozunk.

[16] Megj. Ezt a ciklusfajtát egyes nyelvek nem támogatják, mások esetében a ciklusmag utasításai addig hajtódnak végre, amíg a feltétel igaz.

Formája:

Eljárás .... (........................)
eljárásnév paraméterei
.
. az eljárás utasításai
.
Eljárás vége

A teljes programot a következőképpen írjuk:

Program ....
a program neve
.
. utasítások
.
Program vége

Ha az eljárás nem kap paramétert, akkor nem kell a zárójelet kiírni. Az eljárás hívása nevének megadásával történik.
Ha az eljárás függvény, akkor az eljárás vége után meg kell adni zárójelben a visszaadott értéket tartalmazó kifejezést.

Pl.
Függvényeljárás Fv (paraméter1, paraméter2, ..., paraméterrn)
.
.
.
Eljárás vége (Visszaad:egyik-masik*2)[17]

Egyéb jelölések

Ha egy sorba több utasítást írunk, akkor közéjük pontosvesszőt kell tenni.
Az algoritmusleírásba bárhová elhelyezhetünk megjegyzéseket (el is kell helyezni!) a megértéshez és a kódoláshoz szükséges magyarázatokat (Pl. milyen típusú az a paraméter) két csillag után. (a komment jelzésére használható kapcsos zárójel és egyéb jel is, a lényeg, hogy jól elkülönüljön az utasításoktól.)[18]

Az algoritmus elkészítése során célszerű az alábbi elveket betartani:

[17] TP-ben azonban a függvény nevének legalább egyszer szerepelni kell egy [illegible] vagyis a névben lesz majd a visszaadott érték. Söt a formális paraméterek [illegible] megadni és nem megjegyzésként.
[18] Megj. Vannak nyelvek, amelyek nem támogatják az egy-sorba-több-utasítás [illegible] esetében az utasításokat kettősponttal választják el.

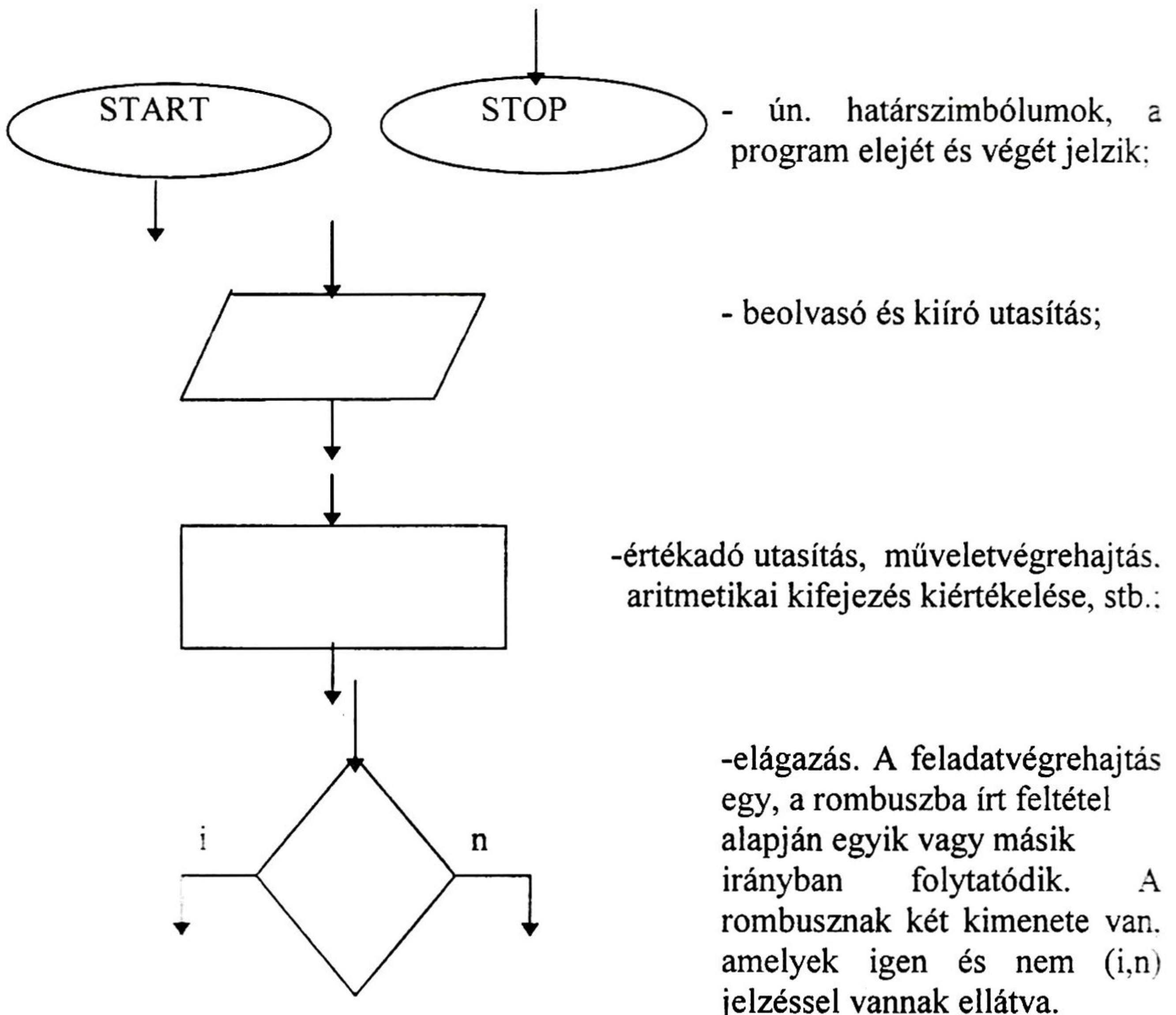
1. A bekezdéses leírás elve arra utal, hogy az egyes struktúrák belsejét formailag is különítsük el a struktúrán kívüli utasításoktól!! Ezt úgy tehetjük meg, hogy a struktúra belsejébe tartozó utasításokat beljebb kezdjük. (legalább két karakternyivel, de többel nem érdemes)

2. A kevés algoritmusleírási szabály elve azt mondja ki, hogy korlátozzuk az algoritmus-leírásban használható utasítások számát. Minél többfajta utasítást használunk ugyanis, annál nehezebben lesz megtanulható, használható az algoritmusleíró nyelv.

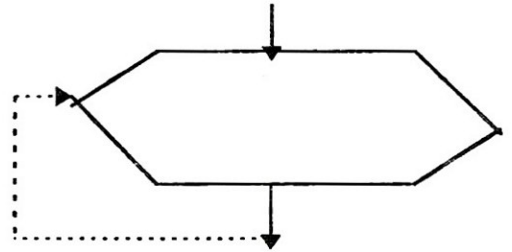
3. A beszédes azonosítók elve az azonosítók (változók, eljárások nevei stb.) elnevezésének fontosságára hívja fel a figyelmet. Ha értelmes (nem egyenértékű azzal, hogy hosszú) neveket választunk, akkor programunk sokkal olvashatóbb lesz, s így könnyebb megérteni, módosítani, hibát keresni benne (úgyis lesz).

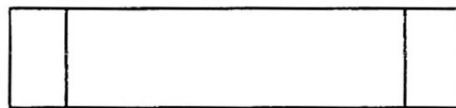
2. Folyamatábra

A folyamatábrákat a feladat megoldási lépéseinek sorrendjét - utasítástípusonként különböző geometriai alakzatok felhasználásával - szemléltető ábra. A felhasznált szimbólumok és jelentésük a következő:



Gyakran találkozunk még a számlálásos ciklus jelölésére a következő szimbólummal is:





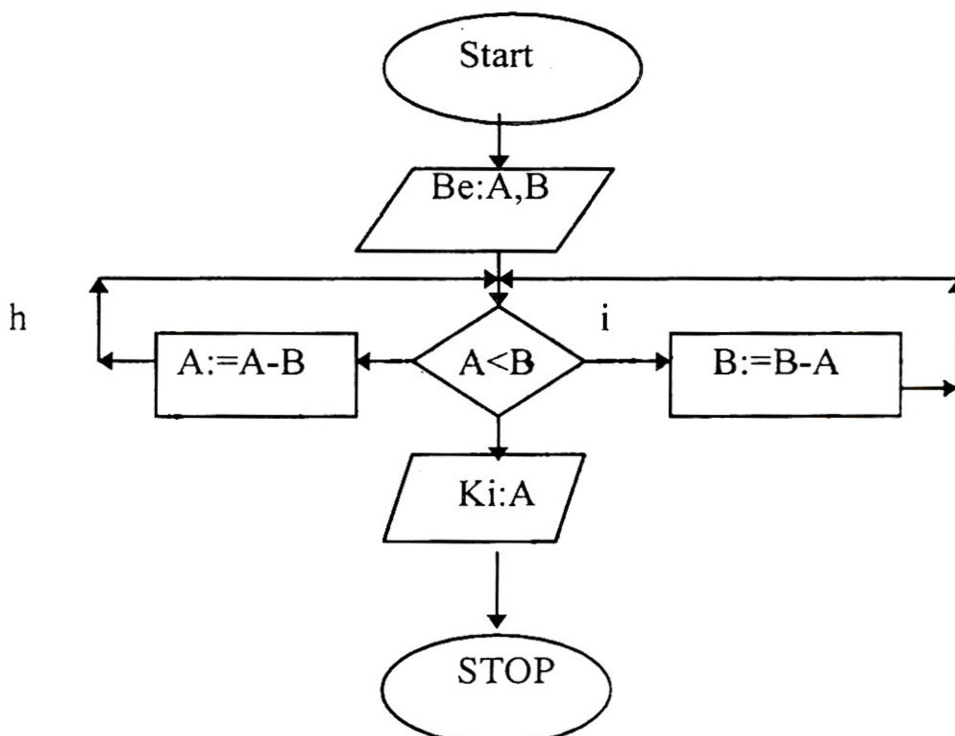
-eljárás hívása, az eljárás nevének megadásával. Ekkor célszerű a kezdő szimbólumba beírni az eljárás nevét.

A folyamatábrában a haladás irányát nyilakkal jelöljük. Ha a nyilakat elhagyjuk, a folyamatábrát felülről lefelé, illetve balról jobbra haladva követjük.

A fenti szimbólumok segítségével utasítássorozatokat (szekvencia), elágazások (szelekciók), ciklusok (iterációk), eljárások szervezhetők.

A folyamatábra hátránya, hogy elemei nem felelnek meg az algoritmuskészítés során felhasználható struktúráknak (szekvencia, elágazás, ciklus). Használatukkal igen könnyen előállíthatók olyan bonyolult szerkezetek is, amelyek megértése azután nagyon nagy problémát jelent.

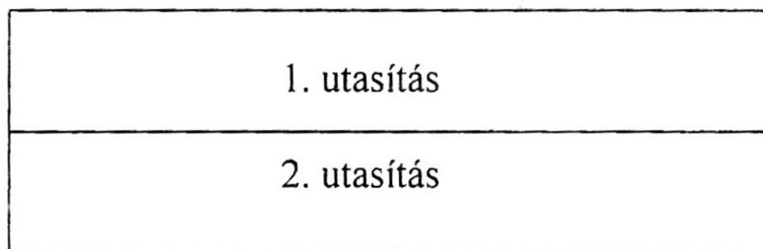
Például az alábbi algoritmus két szám legnagyobb közös osztóját számítja ki:



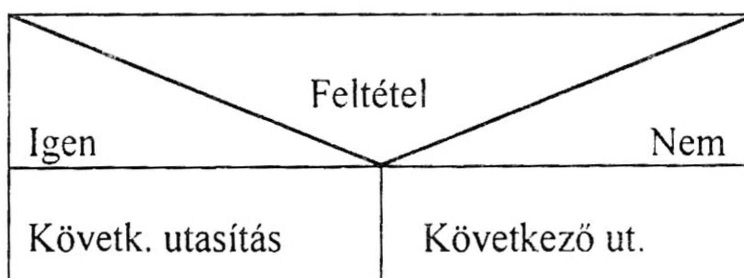
3. Struktogram

A második fajta algoritmusleíró nyelv az ún. struktogram. Ez valójában mondatszerű leírás és a folyamatábra egyfajta keveréke. Magában hordozza a mondatszerű leírásból a struktúrára jellemző bekezdéses ábrázolás bizonyos jegyeit, de megtalálhatók benne a folyamatábrára emlékeztető geometriai elemek, elágazások is.

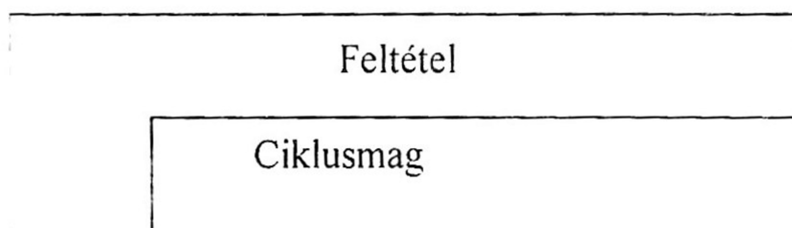
Magát az algoritmust egy téglalapba írjuk. A feladatmegoldás struktúrájának megfelelően ebbe a téglalapba további téglalapokat illesztünk és a végrehajtandó utasításokat ezekbe írjuk bele. Egymás utáni végrehajtás esetén a téglalapot két (több) téglalapra osztjuk:



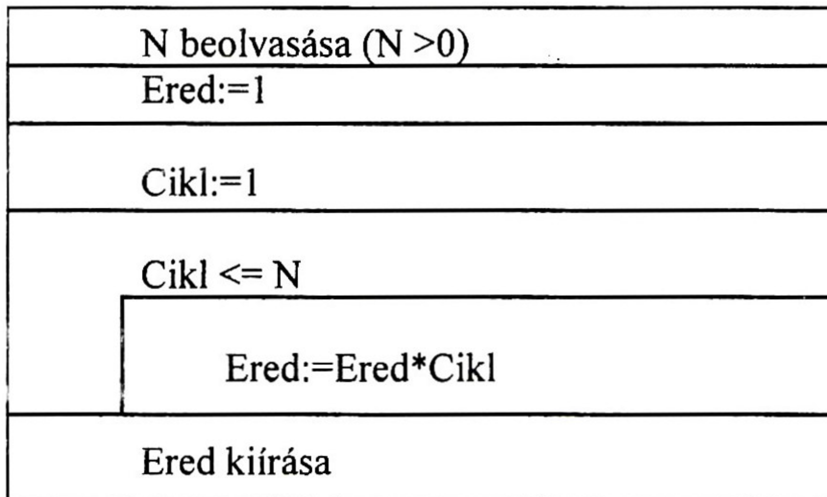
Elágazás esetén a téglalapot kettévágjuk és a megoldást a feltételeknek megfelelően folytatjuk:



A ciklusutasítást egy másfajta téglalappal adható meg:



Befejezésképpen nézzük meg egy feladat struktogramját, melynek elemzése az olvasó feladata marad.



VI. PROGRAMOZÁSI TÉTELEK

Egy sorozathoz egy érték rendelése

Az összegzés tétele

Adott egy számsorozat. Számoljuk és írassuk ki az elemek összegét. A sorozatot most és a továbbiakban is az N elemű A[N] vektorban tároljuk.
Az algoritmus:

Eljárás Összegzés
s:=0
Ciklus I:=1-**től** N-**ig**
s:=s+A[I] ** az s-be tegyük be s korábbi értékének és az A[I] -nek az összegét**
Ciklus vége
Ki: s ** kiíratjuk az i változó értékét**
Eljárás vége

Nem csinálunk tehát mást mint végig olvassuk az összes számot és összeadjuk őket az s nevű változóba. Vigyázni kell arra, hogy az összegzés megkezdése előtt az s-nek 0-t adjunk, hiszen nem tudhatjuk, hogy korábban milyen értéke volt. Ellenkező esetben rossz eredményt kaphatunk!
Feladat: N napon keresztül, naponta egy alkalommal megmértük testünk hőmérsékletét. Adjuk meg az N napos időszak átlaghőmérsékletét!
(Ha a programokat valamilyen nyelven szeretnénk kódolni, figyeljünk arra, hogy az egyes objektumokat, például változókat, nevesített konstansokat a programban általában DEKLARÁLNI kell!)

Program Összegzés
Ciklus
Ki: 'Kérem a napok számát: '
Be: n
Amíg n<=366 ** csak egy évvel rövidebb időszakkal foglalkozunk **
Ciklus i:=1-**től** n-**ig** ** n db számot kérünk be **
Ki: 'Kérem az adatokat'
Be: a[i] (bizonyos határok között, pl. 35 és 45)** itt is lehetne vizsgálni a bevitelt, mint n-nél**
Ciklus vége
s:=0 ** innen kezdődik az összegzés tétele **
Ciklus i:=1-**től** n-**ig**
s:=s+a[i]
Ciklus vége
Ki: 'Az átlag:', s/n
Program vége

Feladat: Irjuk úgy át az algoritmust, hogy az gyorsabb legyen (ne legyen benne ennyi ciklus) !

Az eldöntés tétele

Adott egy N elemű sorozat és egy, a sorozat elemein értelmezett T tulajdonság (pl. a kettővel való oszthatóság, vagy a számjegyek összeg prímszám, stb.). Az algoritmus eredménye: annak eldöntése, hogy van-e a sorozatban *legalább egy* T tulajdonsággal rendelkező elem.
Algoritmus:

Eljárás Eldönt
I:=1
Ciklus amíg I<=N **És** A[I] nem T tulajdonságú ** ha nincs több elem, vagy megtalálta, akkor vége a ciklusnak **
    I:=I+1
Ciklus vége
VAN:= I<=N ** Van értéke igaz, ha I<=N igaz, egyébként Hamis**
Eljárás vége

Pontosan úgy csináljuk mint a valóságban. Vesszük sorban az elemeket (ezért nő I mindig eggyel), és ezt addig csináljuk, amíg nem találunk megfelelő tulajdonságú elemet vagy már nincs több elem (éppen ez van leírva a ciklus fejében).

Feladat: N napon keresztül, naponta egy alkalommal megmértük testhőmérsékletünket. Adjuk meg, hogy volt-e olyan nap, amikor legalább hőemelkedésünk volt! (azaz a T tulajdonság az, hogy "nagyobb vagy egyenlő 37")

Program Eldöntés
Ki: '0-ra befejeződik a felvitel '
i:=1: **Ki:** 'Kérem az adatokat'
Be: a[i] (35 és 45 között) ** itt is lehetne valamilyen ellenőrzés **
Ciklus amíg a[i]<> 0
    Ki: 'Kérem az adatokat';i:=i+1
    Be: a[i]
Ciklus vége
van:=**Hamis** ** feltesszük, hogy nincs ilyen **
i:=1 **értéket kell adni, mert feltételes ciklust használunk**
Ciklus amíg i<=n **És Nem** van ** azaz van=**Hamis****
    Ha a[i]>=37 **akkor** van:=**Igaz**
    i:=i+1
Ciklus vége
Ha van **akkor Ki:** 'Igen, volt.' **Egyébként Ki:** 'Nem.' ** van=Igaz ekvivalens a van-nal**

Program vége

A kiválasztás tétele

Adott egy N elemű sorozat és egy, a sorozat elemein értelmezett T tulajdonság, valamint azt is tudjuk, hogy a sorozatban van legalább egy T tulajdonságú elem. A feladat ezen elem sorszámának meghatározása.

Algoritmus:

Eljárás Kiválasztás
i:=1
Ciklus amíg A[i] nem T tulajdonságú, ** nincs szükség a i<=N vizsgálatra, mert biztosan van ilyen elem**
    i:=i+1
Ciklus vége
SORSZ:=i ** a T tulajdonságú elem sorszáma kerül a SORSZ nevű változóba
Eljárás vége[19]

Itt tehát nincs szükség arra, hogy megvizsgáljuk: van-e még elem. Ha megtaláltuk a T tulajdonságú elemet, akkor a ciklus megáll, és már csak az i értékét kell lekérdeznünk. Az eredmény a SORSZ-ban lesz.

Feladat: N napon keresztül, naponta egy alkalommal megmértük testhőmérsékletünket. Adjuk meg, melyik napon volt hőemelkedésünk! (Tegyük fel, hogy van ilyen nap!)

Program Kiválasztás
Ki: '0-ra befejeződik a felvitel '
i:=1; **Ki:** 'Kérem az adatokat'
Be: a[i] (35 és 45 között) ** itt is lehetne valamilyen ellenőrzés **
Ciklus amíg a[i]<> 0
        Ki: 'Kérem az adatokat';i:=i+1
        Be: a[i]
Ciklus vége
i:=1
Ciklus amíg a[i]<37
    i:=i+1
Ciklus vége
Ki: i, '.napon volt hőemelkedésünk!'
Program vége

[19] Az algoritmus nem foglalkozik azzal az esettel, ha több T tulajdonságú elem is van!

A keresés tételei

a) A lineáris keresés tétele

Rendelkezésre áll egy N elemű sorozat, és egy, a sorozat elemein értelmezett T tulajdonság. Olyan algoritmust kell írni, amely eldönti, hogy van-e T tulajdonságú elem a sorozatban, *s ha van*, akkor megadja a sorszámát (ez utóbbival több, mint az eldöntés).

Algoritmus:

```
Eljárás Lin_keresés
i:=1
Ciklus amíg i<=n És a[i] Nem T tulajdonságú
    i:=i+1
Ciklus vége
VAN:= i<=n
Ha VAN akkor Ki: i
Eljárás vége  ** csak egy elemet keres**
```

Feladat: N napon keresztül, naponta egy alkalommal megmértük testhőmérsékletünket. Adjunk meg egy olyan napot, amikor hőemelkedésünk volt!

```
Program lin_keres
Ki: "Napok száma?"
Be: n (n>0 és n egész)
Ciklus i:=1-től n-ig
      Ki: 'Kérem az adatokat'
      Be: a[i]
Ciklus vége
van:=Hamis: i:=1
Ciklus (i<=n) És (Nem van)
      Ha  a[i]>=37 akkor  ** ha T tulajdonságú, akkor lesz a Van értéke Igaz **
            van:=Igaz
            sorsz:=i
      Elágazás vége
      i:=i+1
Ciklus vége
Ha van akkor Ki: sorsz,'.napon volt hőemelkedésünk.' Egyébként Ki: 'Nem volt probléma.'
Program vége
```

Vegyük észre, hogy eltértünk az alapalgoritmustól, hiszen a T tulajdonság vizsgálata nem a ciklus fejében szerepel, hanem a ciklusban, egy elágazó utasításban.

Abban az esetben, ha ezt a fajta keresést alkalmazzuk egy rendezett sorozatban (pl. ha az adatok a hőmérséklet szerint növekvő sorrendbe rendezve lennének), akkor nem kell a

sorozat elemein végig menni. Ugyanis, ha az összehasonlításoknál a keresett értéket egy tőle nagyobb értékkel hasonlítom össze, akkor - a rendezettség miatt- már nem lehet vele egyező érték a sorozatban. Pl. ha a sorozat : 1, 3, 11, 22, 45, 67 és a keresett elem 6, akkor a 11-gyel való összehasonlítás után már nem kell tovább keresnem a sorozatban

b) A logaritmikus keresés tétele

Rendelkezésre áll egy N elemű *növekvő sorrendbe rendezett*(!!) sorozat és egy keresett elem (X). Olyan algoritmust kell írni, amely eldönti, hogy szerepel-e a keresett elem a sorozatban, s ha igen, akkor megadja a sorszámot.

Most kihasználjuk, hogy a vizsgált sorozat rendezett. Ez alapján bármely elemről el tudjuk dönteni, hogy a keresett elem előtte vagy utána van-e, esetleg megtaláltuk. Az eljárás lényegének megértéséhez tudni kell, hogy az A és az F változóknak kiemelt szerepük van: mindig annak a részintervallumnak az alsó és felső végpontjai, amelyben a keresett elem benne van.

Algoritmus:

Eljárás Bináris_Keresés
A:=1: F:=N
Ciklus ** K-ban lesz az intervallum közepe **
  K:= (A + F) DIV 2 **DIV továbbra is egész osztás**
  Ha A[K]<X **akkor** A:=K+1 ** ha X az intervallum (sorozat) középső elemétől nagyobb**
  Ha A[K]>X **akkor** F:=K-1** ha X az intervallum (sorozat) középső elemétől kisebb**
amíg A>F **És** A[K]=X ** ha összeért a két részintervallum vagy megtaláltam, akkor vége **
VAN:=A<=F
Ha VAN **akkor** SORSZ:=K
Eljárás vége

Az algoritmus megértéséhez nézzünk egy példát:
Legyen a sorozat 1, 4, 6, 8, 12, 22, 27 és a keresett elem a 22
Mivel A=1 és F=7 ezért K=4, ezért az első lépésben a sorozat (1+7) Div 2, azaz a 4. elemét, a 8-at hasonlítjuk a keresetthez, vagyis a 22-höz. Mivel a 8<22 ezért A egyenlő lesz 5-tel (K-1), így most már csak az 5. és a 7. elem között keressük a 22-t, vagyis A=5 és F=7. Az 5. elem előtti elemek között már nem keressük a 22-t, vagyis egy hasonlítással "átvizsgáltuk" a sorozat felét.

A megszámlálás tétele

Általános feladat:
Rendelkezésre áll egy N elemű sorozat és egy, a sorozat elemein értelmezett T tulajdonság. Most a T tulajdonsággal rendelkező elemek megszámlálása és kiírása a feladat.

Algoritmus:

Eljárás Megszámlálás
s:=0** nem szabad elfelejteni**
Ciklus i=1-**től** n-**ig**
    Ha A[i] T tulajdonságú **akkor** s:=s+1 ** s értékét növeljük eggyel**
Ciklus vége
Ki: s
Eljárás vége

Feladat: Töltsünk fel egy N*M-es mátrixot véletlenszámokkal! Számoljuk meg, hogy hány olyan eleme van a mátrixnak, amely páros szám, vagy ahol az indexek összege megegyezik az elemmel.

Program Számlál
Konstans N=10
M=20
Ciklus i:=1-**től** N-**ig**
    Ciklus j:=1-**től** M-**ig**
    a[i,j]:=**Véletlen**(256) ** a [0,255] intervallumba eső egész számokat generál a **Véletlen** nevű függvény**
    Ciklus vége
Ciklus vége ** a mátrix feltöltése számokkal**
s:=0
Ciklus i:=1-**től** N-**ig**
    Ciklus j:=1-**től** M-**ig**
        Ha a[i,j] MOD 4 = 0 **Vagy** i+j=a[i,j] **akkor** s:=s+1 **a MOD továbbra is maradékképzés**
    Ciklus vége
Ciklus vége
Ki: 'Az elemek száma : ',s
Program vége

Vegyük észre, hogy a feladatot megoldhattuk volna úgy is, hogy csak két ciklust használunk, vagyis az utolsó két ciklus felesleges. Így már a generáláskor eldöntjük, hogy az elem a keresett tulajdonságú-e.

A szélsőérték-kiválasztás tételei

a) A maximumkiválasztás tétele

Ebben a feladatban egy N elemű sorozat legnagyobb elemét kell megadni

Algoritmus:

Eljárás Maximum
Max:=A[1]
Ciklus j:=2-**től** N-**ig**
    Ha Max<A[j] **akkor** Max:=A[j]
Ciklus vége
Ki: Max
Eljárás vége

Éppen úgy járunk el, mint a valóságban. Vesszük a sorozat első elemét. Ezek után ehhez az elemhez hasonlítjuk a többi elemet. Ha azonban találunk ettől az elemtől nagyobb számot, akkor innentől kezdve már a megtalált nagyobb számhoz hasonlítjuk a többi elemet. Vagyis amikor találunk az eddig megvizsgáltak közül kiválasztott legnagyobbtól (ez van a Max változóban) nagyobbat (A[j]), akkor azonnal berakjuk a Max-ba, és innentől kezdve ez a legnagyobb.

b) A minimum kiválasztás tétele

Ebben a feladatban egy N elemű sorozat legkisebb elemét kell megtalálni.

Algoritmus:

Eljárás Minimum
I:=1
Ciklus j:=2-**től** N-**ig**
    Ha A[I]>A[j] **akkor** I:=J
Ciklus vége
Ki: A[I]
Eljárás vége

A megoldás majdnem azonos az előzővel, de most a legkisebb elem *indexét* tároljuk és nem magát az elemet. Ez azért lehet előnyös, mert ha például a sorozat elemei nagy számok, vagy valós számok, akkor ezek tárolása több helyet foglal a memóriában mint az index tárolása, ami mindig egész. Természetesen a **Ha** utasítás feltételében a relációjel megfordult (>), hiszen most a legkisebb elemet keressük.

Ezen tételekre épül a következő feladat megoldása

Írassuk ki az A nevű négyzetes (n*n-es) mátrix nyeregpontjait! Azok az elemek a nyeregpontok, amelyek a sorukban a legnagyobbak, de az oszlopukban a legkisebbek vagy éppen fordítva. A nyeregpontok zölden villogjanak és a mátrix többi elemét is írassuk ki mátrix alakban, de az elemek kiírásának sorrendje véletlenszerű legyen!

Bevezetjük a MenjXY eljárást, amely a kurzort a paraméterként megadott (oszlop, sor) koordinátájú pontra viszi.

A Kiir az A[]-val megegyező méretű mátrix, elemei logikai típusúak! Ha Kiir[i,j]=Igaz, akkor A[i,j] nyeregpont.

```
Program nyereg
Ciklus
    Ki: 'Kérem a mátrix méretét'
    Be:n
Amíg n<11  ** maximum 10*10-es lehet**
Ciklus cikl1:=1-től n-ig
    Ciklus cikl2:=1-től n-ig
        seged:=Véletlen(32767)
        a[cikl1,cikl2]:=seged-Véletlen(32767) **-32767 és 32766 közé eső véletlen
                                                                        számok**
        kiir[cikl1,cikl2]:=Hamis  ** nincs nyeregpont**
    Ciklus vége
Ciklus vége
Ciklus cikl1:=1-től n-ig
    ny:=32767;nyereg:=Igaz   ** ettől nem lehet nagyobb!**
    Ciklus cikl2:=1-től n-ig
        Ha tomb[cikl1,cikl2] < ny akkor
            ny:=tomb[cikl1,cikl2];cikl3:=cikl2 **minimumot keresünk**
        Elágazás vége
    Ciklus vége
    Ciklus cikl2:=1-től n-ig
        Ha ny < tomb[cikl2,cikl3]  akkor nyereg:=Hamis **a minimum nem
                                                        maximum az oszlopban**
    Ciklus vége
    Ha nyereg akkor
        Szín(Zöld)
        MenjXY(cikl3*7,cikl1)
        Ki: tomb[cikl1,cikl3]:7 ** 7 hosszon kiíratjuk**
        kiir[cikl1,cikl3]:=Igaz;m:=m+1  ** nyeregpont
    Egyébként
        ny:=-32768;nyereg:=Igaz
        Ciklus cikl2:=1-től n-ig
            Ha tomb[cikl1,cikl2] > ny akkor **maximumot keresünk**
                ny:=tomb[cikl1,cikl2];cikl3:=cikl2
            Elágazás vége
        Ciklus vége
        Ciklus cikl2:=1-től  n-ig
            Ha ny > tomb[cikl2,cikl3] akkor nyereg:=Hamis ** nem [illegible] az
                                                                [illegible]
        Ciklus vége
        Ha nyereg akkor
            Szin(Zöld);MenjXY(cikl3*7,cikl1);
            Ki: tomb[cikl1,cikl3]:7  **hosszon iratjuk ki**
            kiir[cikl1,cikl3]:=Igaz;m:=m+1
        Elágazás vége
```

```
        Elágazás vége
    Ciklus vége
    Ciklus cikl3:=1-től n*n-m -ig  ** m db nyergpont volt **
        Ciklus
            cikl1:=Véletlen(n)+1
            cikl2:=Véletlen(n)+1  ** koordináták generálása**
        Amíg kiir[cikl1,cikl2]=Hamis
        MenjXY(cikl2*7,cikl1);Szín(Fehér)
        Ki: tomb[cikl1,cikl2]
        kiir[cikl1,cikl2]:=Igaz
    Ciklus vége
    Program vége
```

Egy sorozathoz egy sorozat hozzárendelése

A kiválogatás tétele

Általános feladat:

Egy N elemű sorozat összes T tulajdonsággal rendelkező elemét kell meghatározni. Gyűjtsük a kiválogatott elemek sorszámait a B vektorban!

Algoritmus:

```
Eljárás Kiválogat
j:=0 ** nem szabad elfelejteni **
Ciklus i:=1-től N-ig
      Ha a[i] T tulajdonságú akkor
            j:=j+1
            b[j]:=i             **a sorszámot tároljuk**
      Elágazás vége
Ciklus vége
Eljárás vége
```

Itt is azért tároljuk az indexeket, mert így kevesebb helyet foglal a b[] vektor a memóriában. Ha ki is kell íratni az a[] vektor T tulajdonságú elemeit, akkor az a következőképpen történhet:

```
Ciklus i:=1-től j-ig  ** Azért megy j-ig a ciklus, mert b-nek j db
                                                        eleme van!*
      Ki: a[b[i]]
Ciklus vége
```

Az a[b[i]] az a vektor b[i]-edik elemét jelenti. Ha pl. i=1 és a b[i] (a b vektor i-edik eleme] egyenlő 5, akkor a[5]-öt, azaz az a vektor 5. eleme kerül kiíratásra.

Feladat: *Adott* egy N*M-es mátrix. Gyűjtsük azon elemeit egy másik vektorba, amelyeknél a számjegyek összege nem osztható 5-tel!

```
Eljárás Kiválogat
i2:=0 ** fontos **
Ciklus sor:=1-től N-ig
      Ciklus oszlop :=1-től M-ig
            Ha Jegyek(a[sor,oszlop]) MOD 5 <> 0 akkor  ** ha a számjegyek összege ... **
                  i2:=i2+1:b[i2]:=a[sor,oszlop]     ** b-ben lesznek az elemek**
            Elágazás vége
      Ciklus vége
Ciklus vége
Ciklus sor1:=1 -től i2-ig ** i2 eleme van b[]-nek **
      Ki: b[sor1]** vektor elemeinek kiírásához ciklust kell szervezni**
Ciklus vége
Program vége
```

Ezúttal nem célszerű az indexeket tárolni, hiszen mátrixról lévén szó kettő is van. Ráadásul el kellene készíteni azt a függvényt, amely kiszámítja a paraméterként átadott szám (mátrix elem) számjegyeinek összegét. A függvény a következő lehet:

```
Függvény Jegyek(megkap)** a megkap változó típusa feleljen meg a mátrix
elemeinek a típusával *
sz:=0
Ciklus amíg megkap<>0
      ered=megkap Div 10
      sz:=sz+megkap-ered*10
      megkap=ered
Ciklus vége
Függvény vége (Visszaad:sz)
```

Érdemes megvizsgálni a függvény működését. A kapott számot (megkap) elosztjuk 10-zel, de az egész osztást alkalmazzuk. Így pl. 123 Div 10 éppen 12. Ha a 123-ból (megkap) kivonjuk a kapott eredmény 10-szeresét, ami most 120 (ered*10), akkor 3-at kapunk, azaz az utolsó számjegyet. Ezzel az értékkel növeli a függvény az sz változó értékét, ami az elején természetesen 0. Ezután a megkap változó felveszi az ered értékét (12) és tovább folytatódik az 10-zel való osztás, de most már az ered-be a 12 Div 10 értéke kerül. A ciklus befejeződése után a függvény visszaadja a paraméterként átadott szám számjegyeinek az összegét, vagyis sz-et.

Több sorozathoz egy sorozat hozzárendelése

A metszetképzés tétele

Általános feladat

Rendelkezésünkre áll egy N és egy M elemű halmaz az A[] és B[] vektorokban. Készítsük el a két halmaz metszetét a C[] vektorba. (Két halmaz metszetébe azok az elemek tartoznak, amelyek mindkettőben szerepelnek)

Algoritmus

```
Eljárás Metszet
szamol:=0
Ciklus I:=1 -től N-ig ** vesszük az A[] elemeit **
    j:=1
    Ciklus amíg j<=M És A[I] <> B[j]  **addig, amíg nincs a A[I]-vel egyező,
                                                         vagy van még elem**
        j:=j+1 ** vesszük a B[] elemeit
    Ciklus vége
    Ha j<=M akkor ** ha volt egyező eleme **
        szamol:=szamol+1;C[szamol]:=A[I] ** betesszük a C vektorba ***
    Elágazás vége
Ciklus vége
Eljárás vége
```

Az algoritmus pontosan úgy működik, ahogyan a mindennapi életben végezzük két halmaz metszetének megálapítását. Vesszük az egyik halmaz (A) elemeit sorban, először az elsőt, majd a másodikat, stb.(ezért indul az első ciklus). Majd a másik halmaz (B) elemeit olvassuk, de csak addig, amíg van a halmaznak meg nem vizsgált eleme vagy nem találtuk az A-belivel egyezőt. Ellenkező esetben befejezzük az olvasást, azaz vége a ciklusnak. Ha még nem olvastuk végig a B halmazt(j<=M) az csak azért lehet, mert találtunk közös elemet. Ezt az elemet rakjuk be a C[] vektorba.

Feladat: Számoljuk meg, hogy egy N elemű A[] vektornak hány különböző eleme van.

Itt nem lehet azonnal a megszámlálás tételét alkalmazni, hiszen nehéz a T tulajdonságot az algoritmusba beépíteni. Célszerű a metszetképzést használni, de ezúttal a B[] vektor üres, majd folyamatosan bővül.

```
Eljárás Különböző
szamol:=0
M:=0
Ciklus I:=1 -től N-ig ** vesszük az A[] elemeit **
    j:=1
    Ciklus amíg j<=M És A[I] <> B[j]        addig, amíg nincs a A[I]-vel
                                            egyező, vagy van még elem**
        j:=j+1 ** vesszük a B[] elemeit
    Ciklus vége
    Ha j>M akkor ** ha nem volt ilyen elem a B[]-ben **
        M:=M+1;B[M]:=A[I] ** betesszük a B vektorba ***
    Elágazás vége
Ciklus vége
Eljárás vége
```

Az eljárás csak akkor működik helyesen, ha a B[] le lett nullázva, vagy olyan elemekkel lett feltöltve, amelyek *biztosan* nem fordulhatnak elő A[]-ban. Az algoritmus érdekessége, hogy kezdetben B[] üres, majd akkor rakunk bele A[]-ból elemet, ha az még nincs a B[]-ben. A végén az M mutatja, hogy B[] -nek hány eleme van, vagyis az A[] vektor különböző elemeit.

Ha pl. az A[] elemei 2,4,5,5,6,3,2, akkor először a 2 kerül be a B[]-be, majd a 4 és az 5. Ezután a következő ötöshöz keresünk párt a B[]-ben, mivel azonban találunk, azaz a j nem lesz nagyobb mint M, így ez az 5 nem kerül be a B[]-be. Erre a sorsra fog jutni az A[] vektor utolsó eleme, a 2 is.

Az egyesítés (unióképzés) tétele

Általános feladat

Rendelkezésre áll egy N és egy M elemű halmaz, az A[] és a B[] vektorban ábrázolva. Készítsük el a két halmaz egyesítését a C vektorba!

(Két halmaz egyesítésében azok az elemek tartoznak, amelyek legalább az egyikben szerepelnek)

Algoritmus:

```
Eljárás Unió
Ciklus i:=1-től N-ig
    C[i]:=A[i]          ** először az A[] elemei átkerülnek a C[]-be **
Ciklus vége
szamol:=N
Ciklus j:=1-től M-ig
    i:=1
    Ciklus amíg i<=N És A[i]<>B[j]  ** B[]-beli elemhez keresünk A[]-ból **
            i:=i+1 ** vesszük az A[] elemeit **
    Ciklus vége
    Ha i>N akkor ** ha nincs közös elem**
            szamol:=szamol+1
            C[szamol]:=B[j] **akkor berakjuk az elemet a C[]-be**
    Elágazás vége
Ciklus vége
Eljárás vége
```

Ha ki is szeretnénk íratni C[]-t, akkor ahhoz célszerű ciklus szervezni:

```
Ciklus cikl:=1-től szamol-ig ** számol eleme van a C-nek**
    Ki:C[cikl]
Ciklus vége
```

Az algoritmus csak abban tér el a metszetképzéstől, hogy először az A[] vektor tartalmát bemásoljuk a C[]-be (így annak már N eleme van) majd B[] minden egyes eleméhez (ezért megy a ciklus M-ig) keresünk A[]-beli egyezőt. Ha *nem* találunk (azaz I>M), akkor rakjuk C[]-be B[j]-t.

Az összefuttatás tétele

Általános feladat

Rendelkezésre állnak két rendezett sorozat elemei. Állítsunk elő belőlük egy sorozatot úgy, hogy az eredeti sorozatok minden eleme szerepeljen benne, és ez a sorozat is rendezett legyen! (Például növekvő sorrendű sorozatokat vizsgálunk.)

A feladat tulajdonképpen az uniótétel speciális esete: uniót kell előállítani úgy, hogy a rendezettség megmaradjon. A két sorozat: A[N], B[M].

Az eredmény: C[N−M]. A megoldásban +∞ jelöli a számítógépen ábrázolható legnagyobb értéket. Ezt alkalmazzuk végjelként a megoldás egyszerűsítése érdekében.

Algoritmus:

Eljárás Összefuttat
i:=1: j:=1: k:=0
A[N+1]:=+∞: B[M+1]:=+∞
Ciklus amíg i<N+1 **Vagy** j<M+1
    k:=k+1
    Elágazás
      A[i]<B[j] esetén C[k]:=A[i]: i:=i+1** venni kell majd A[] következő elemét**
      A[i]>B[j] esetén C[k]:=B[j]: j:=j+1** venni kell majd B[] következő elemét**
      A[i]=B[j] esetén C[k]:=A[i]: i:=i+1: j:=j+1 ** mindkét vektorból venni kell a köv. elemet**
    Elágazás vége
Ciklus vége
Eljárás vége[20]

Mivel elképzelhető, hogy az egyik vektor elemei elfogynak, így csak a másikból kell olvasni az elemeket és rakni át a C[]-be. Ezért rakjuk a vektorok utolsó értékes eleme után a lehető legnagyobb értéket, hiszen így a másik vektorban minden elem kisebb lesz ettől a nagy értéktől, vagyis már csak az értékes elemeket taralmazó vektort olvassuk.

Feladat:
Ismerjük egy osztály tanulóinak a névsorát, külön a lányokat és külön a fiúkat névsor szerint sorrendben. Adjuk meg az egyesített névsort!

Megfeleltetés:

lsz - lányok száma
fsz - fiúk száma
L[lsz] - lányok nevei névsorba rendezve
F[fsz] - fiúk nevei névsorba rendezve
N[lsz+fsz] - az osztálynévsor

A fiúk és a lányok névsorában azonos nevű nem lesz, így az összefuttatás algoritmusa egyszerűsíthető.

[20] Az összefuttatás az alapgondolata az ún. külső rendezésnek, [illegible] nem fér el egyszerre az operatív tárban.

Algoritmus

```
Eljárás Névsor
I:=1: j:=1: k:=0
L[lsz+1]:="ZZZZZZZZ"
F[fsz+1]:="ZZZZZZZZ"  **a lehető legnagyobb név**
Ciklus amíg i<=lsz És j<=fsz
      k:=k+1
      Elágazás
            L[i]<F[j] esetén N[k]:=L[i]:i:=i+1
            L[i]>F[j] esetén N[k]:=F[j]:j:=j+1
      Elágazás vége
Ciklus vége
Eljárás vége
```

Rendezések

Buborékos rendezés

A módszer lényege:

Végigmenve az adatsort tartalmazó vektoron minden _szomszédos_ elempárt növekvő sorrendbe rakunk, azaz meghagyjuk helyükön, vagy megcseréljük őket aszerint, hogy jó sorrendben voltak vagy sem. Egy ilyen menet végén a legnagyobb elem a vektor végére kerül. Ugyanezt a páronkénti cserét végrehajtjuk a még rendezetlen a[1..(n-1)] részvektoron. Ezzel a második legnagyobb elem az a[n-1] elem helyére kerül. Addig folytatjuk az egyre rövidebb rendezetlen sor párcseréit, míg minden elem a helyére kerül. Egy menetben a maradék sor legnagyobb eleme, mint egy buborék halad végig az adatsoron. Innen származik a módszer neve. Ha a rendezetlen adatsor hosszát r-rel jelöljük, akkor az eljárás r=n értékkel indul, és minden menet után r értéke 1-gyel csökken. Az utolsó végrehajtandó menetben r=2, hiszen 1 elemet már nem kell rendezni.

A buborékrendezés nagyon lassú, az ismert helyben rendezések közül a legnagyobb az időigénye. Az r hosszúságú adatsoron végighaladva ugyanis r-1 számú összehasonlítást kell elvégezni ahhoz, hogy a csere szükségességét eldöntsük. Ha a sorozat már eleve rendezett volt, akkor egyetlen cserére sem kerül sor. Ha azonban a legrosszabb eset áll fenn, azaz éppen fordított sorrendben volt az adatsor, akkor minden párnál csere szükséges, és így egy menetben a cserék átlagos száma (r-1)/2. Az összes menetben tehát a vizsgálatok és a cserék száma:

$$v:=\sum_{r=2}^{n}(r-1) = n/2\cdot(n-1) \qquad c:=\sum_{2}^{n}(r-1) = n/4 * n(-1)$$

Mindkettő az adatsor hosszának négyzetével arányosan növekszik: $o(v)=O(c)=n^2$. Ezer adatnál ez majdnem félmillió összehasonlítás és negyedmillió adatcsere! Ehhez még a gyors elektronikus számítógépeknek is jelentős időre van szüksége.

A buborékrendezési algoritmust kétféleképpen javíthatjuk. Az egyik azon az észrevételen alapul, hogy egy menetnek az eredményeként a maximális elem a sor végére kerül. Eszerint a menet közbeni adatcseréket megtakaríthatjuk, ha a maximális elemet és ennek helyét megállapítjuk és csak a menet végén egyetlen cserével tesszük a helyére. Ezt a módszert maximum-kiválasztásos módszernek nevezik. Tovább javíthatjuk az algoritmust, ha egy menetben nemcsak a maximumot, hanem a minimumot is kiválasztjuk és ezt az első elemmel cseréljük ki.

A másik jobbítási ötlet különösen akkor ad kézzelfogható eredményt, ha az eredeti adatsor nem nagyon rosszul rendezett, azaz kicsi a szükséges cserék (inverziók) száma. Ehhez egy speciális változót kell használni, amely egy menetben számlálja, vagy csupán jelzi (logikai változó) a cseréket. Ha egy menet úgy fut le, hogy közben már nem kellett cserét végrehajtani, akkor az előző menetben már minden elem a helyére került, további menetre nincs szükség. Ha a sor ellenkező rendezésű volt, akkor ez az ötlet nem segít, minden menetre szükség van. Ez a módosítás nem garantálja a gyorsítást minden adatsornál. Az így javított buborék rendezés és a maximum-kiválasztásos rendezés algoritmusa a következő:

```
Eljárás Buborék
r:=n; csere:=Hamis
Ciklus amíg r>1 És Nem Csere
      csere:=Igaz
      Ciklus i:=1-től r-1-ig
            Ha a[i]>a[i+1] akkor ** itt a páronkénti összehasonlítás **
                  csere(a[i],a[i+1]) **eljárás, amely megcseréli a két értéket[21] **
                  Csere:=Hamis ** megjegyezzük, hogy volt csere **
            Elágazás vége
            i:=i+1
      Ciklus vége
      r:=r-1** már eggyel kevesebb elemet kell vizsgálni, hiszen a legnagyobb már
                                                                    meg van **
Ciklus vége
Eljárás vége
```

Nézzük meg egy példán, hogyan is működik az algoritmus. Legyen az elemek
4,1,0,3,6,2,9,7 Majd az első menet után (itt még r=n, azaz 8 volt):
1,0,3,4,2,6,7,9 hiszen cserélni kellett a következő párokat:(4,1),(4,0),(4,3),(4,2),(9,7)
Azaz a 4 addig vándorolt, amíg egy tőle kisebb elemmel nem találkozott. A második menet után:
0,1,3,2,4,6,7,9 csak az (1,0) és a (4,2) párok cseréltek helyet.

[21] Pl. seged:=a[i]:a[i]:=a[i+1]:a[j]:=seged

0,1,2,3,4,6,7,9 a (3,2) csere történt és sajnos még egyszer végig nézi az algoritmus a sort, mivel volt csere!

0,1,2,3,4,6,7,9 itt nem volt csere így a rendezettség elő állt!

Vegyük észre, hogy r még csak 4, de már elő állt a rendezettség és így nincs értelme tovább vizsgálni a vektort!

Az olvasó feladata marad az algoritmus átírása utófeltételes ciklus használatával

```
Eljárás MaxkivRend  ** Maximum kiválasztással **
r:=n
Ciklus amíg r>1
        i:=1;m:=a[1];k:=1
        Ciklus amíg i<r
                Ha a[i]>m akkor
                        m:=a[i]; k:=i ** maximum és indexenek tárolása **
                Elágazás vége
                i:=i+1
        Ciklus vége
        a[k]:=a[r];a[r]:=m **az utolsó elem(a[r]) kerül a maximum helyére, az utolsó
                                                        helyre(r) a maximum**
        r:=r-1
Ciklus vége
Eljárás vége
```

A maximum-kiválasztásos rendezésnél a szükséges összehasonlítások száma ugyanannyi, mint a buborékos módszernél, azonban a cserék száma pont ellenkezőleg alakul, és ezzel az értékadási lépések átlagos száma ugyanannyi marad.

Ha ugyanis az adatsor eleve növekvően rendezett, akkor a maximum már a helyén van minden menetben. Ezért az algoritmus belső ciklusában minden lépésben sor kerül az új maximumnak és indexének beállítására. Ezzel szemben, ha az adatsor eleve jó irányban rendezett, akkor ezekre az értékadásokra nincs szükség, csak egyetlen cserére kerül sor a menet végén. Ebből következik, hogy a maximum-kiválasztáson alapuló rendezési algoritmusba is beépíthetünk egy olyan jelzőt, amelyikkel leállíthatjuk a felesleges menetek végrehajtását. Ez a módosítás legyen az olvasó feladata.

A maximum-kiválasztáson alapuló rendezés igazi előnye a buborékos rendezéssel szemben akkor mutatkozik meg, ha az adatsor elemei nem egyszerű adatelemek, hanem több komponensből állnak. Ilyenkor a buborékos algoritmusnál minden komponenst át kell helyezni, hogy az összetartozók együtt maradjanak. Az új maximum és az index feljegyzése csak két egyszerű adatelemre vonatkozó értékadás, míg az adatcsere komponensenként 3-3 elemi lépést igényel.

A nagyméretű információs rendszerek esetében nem mindegy, hogy milyen rendezési módszert alkalmaz a (profi) programozó

Feladat: Irjuk úgy át az algoritmust (két módszer is lehetséges), hogy az elemek a vektorban csökkenő sorrendbe legyenek

Rendezés minimum-kiválasztással

A módszer lényege

A felesleges cserék kiküszöbölés érdekében két segédváltozó bevezetésére van szükség. Az ÉRTÉK nevű változó tartalmazza az adott menetben addig megtalált legkisebb elemet, az INDEX pedig annak vektorbeli sorszámát, indexét. Az A vektor elemeit mindig az ÉRTÉK változó tartalmával hasonlítjuk össze. Ha ÉRTÉK-nél kisebb elemet találunk, azt betesszük az ÉRTÉK nevű változóba és az INDEX-ben megjegyezzük a szóban forgó elem indexét. A menet végére az ÉRTÉK a vektor soronkövetkező legkisebb elemét tartalmazza, az INDEX pedig azt a sorszámot, ahol ezt az elemet találtuk. Csak a menet utolsó lépésében van szükségünk cserére, amikor az ÉRTÉK-ben levő legkisebb elemet a helyére tesszük

A rendezés algoritmusa

Eljárás REND2
    Ciklus cikl:=1-**től** N-**ig**
        index:=cikl: ÉRTÉK:=a[cikl]
        Ciklus j:=cikl+1-**től** N-**ig**
            Ha ÉRTÉK >a[j] **akkor** ÉRTÉK:=a[j]; index:=j
        Ciklus vége
        a[index]:=a[cikl]; a[cikl]:=ÉRTÉK
    Ciklus vége
Eljárás vége

Vegyük észre, hogy az algoritmusban a korábban említett minimum-kiválasztást [illegible] alkalmaztuk N-1-szer!
Itt az összehasonlítások száma . n/2*(n-1).
A mozgatások száma legjobb esetben 3*(n-1), legrosszabb esetben [illegible]

Rendezés közvetlen elemkiválasztással

A módszer lényege:

A rendezendő számok legyenek az A vektor elemei. Az [illegible] vektor legkisebb elemét úgy, hogy az a(1) -et összehason[illegible] mindegyikével. Ha ezt a legkisebbet tesszük a(1)-be, így [illegible] vektor legkisebb elemét tartalmazza majd. Az eljárást a(2)-vel [illegible] össze az a(3),...,a(n) elemekkel, és így tovább. menetenként [illegible] elem kiválasztásával N-1 menet után a vektor rendezett lesz

A rendezés algoritmusa:

Eljárás REND1
Ciklus i:=1-**től** N-1-**ig**
Ciklus j:=i+1-**től** N -**ig**
Ha a[j]<a[i] **akkor** csere(a[i],a[j]) **eljárás, amely megcseréli a két értéket**
Ciklus vége
Ciklus vége
Eljárás vége
Ha a számok 5,2,7,4,1, akkor a rendezés során:
1. menet 1,5,7,4,2
2. menet 1,2,5,7,4
3. menet 1,2,4,5,7
4. menet 1,2,4,5,7

Sajnos az algoritmus nem "veszi észre", hogy már a harmadik menetben a rendezettséghez jut és vannak felesleges cserék is.
Ennél a módszernél az összehasonlítások száma n/2*(n-1). A mozgatások száma természetesen függ az elemek előrendezettségétől. Ha a rendezendő elemek eredetileg monoton növekvő sorozatot alkottak, akkor 0, monoton csökkenő előrendezettség esetén 3*n *(n-1)/2

Feladat: valamelyik rendezési algoritmus segítségével rendezzük egy N*M-es mátrix páratlan soraiban lévő elemeit növekvő, a páros soraiban lévőket pedig csökkenő sorrendbe, és írassuk ki az elemeket képernyő közepére mátrix alakban.

Három problémát kell megoldani az algoritmus elkészítéséhez:
1) A mátrix feltöltése
2) Rendezés
3) Kiíratás

A kiírás során figyelni kell arra, hogy minden sor utolsó elemének kiírása után kell egy sort emelni, egyébként NEM. Meg kell még oldani a középre írást (feltesszük, hogy N és M mérete lehetővé teszi ezt). Mivel 80 oszlopos egy karakteres képernyő (alapesetben), ugyanakkor minden elem kiírása 4 oszlopot igényel (3 számjegy és egy szóköz). Így összesen M*4 oszlopot vesz igénybe egy teljes sor kiírása. Marad tehát 80-M*4 oszlop üresen. Így ki kell hagyni az elején ennek a felét, a végén pedig a másik felét. Hasonlóan történik a sor meghatározása, de egy sor kiírása egy sort vesz igénybe, és 25 oszlopos a képernyő.

```
Program Rendezget
Az A[] mátrix, N és ,M globálisak!
N:=10;M:=8

Eljárás Generál
N:=10;M:=8
Ciklus SorIndex:=1-től N-ig
    Ciklus OszlopIndex:=1-től M-ig
      A[SorIndex,OszlopIndex]:=Véletlen(100)+10 **10 és 109 közé eső
                                véletlen számot generál a Véletlen függvény**
    Ciklus vége
Ciklus vége
Eljárás vége

Eljárás Rendez
Ciklus SorIndex:=1-től N-ig
    Ciklus OszlopIndex:=1-től M-1-ig
        Ciklus Oszlop:=OszlopIndex+1-től M-ig
            Ha SorIndex MOD 2 = 0 akkor
                Ha A[SorIndex,OszlopIndex]<A[Sorindex,Oszlop] akkor
                    seged:=A[SorIndex,OszlopIndex]
                    A[Sorindex,OszlopIndex]:=A[SorIndex,Oszlop]
                    A[SorInde.,Oszlop]:=seged
                Elágazás vége
            Egyébként
                Ha A[SorIndex,OszlopIndex]>A[Sorindex,Oszlop] akkor
                    seged:=A[SorIndex,OszlopIndex]
                    A[Sorindex,OszlopIndex]:=A[SorIndex,Oszlop]
                    A[SorIndex,Oszlop]:=seged
                Elágazás vége
            Elágazás vége
        Ciklus vége
    Ciklus vége
Ciklus vége
Eljárás vége

Eljárás Kiírás
MenjXY((80-M*4) DIV 2,1+(24 - N) DIV 2) ** magyaráz[illegible]
Ciklus SorIndex:=1-től N-ig
    Ciklus OszlopIndex:=1-től M-ig
        Ki:A[SorIndex,OszlopIndex] ** 4 hosszon [illegible]
    Ciklus vége
    Soremelés
Ciklus vége
Eljárás vége
```

Generál
Rendez] Eljárások hívás
Kiírás
Program vége

Vegyük észre, hogy a párosság vizsgálat mennyire rontja a hatékonyságot és lehetne máshová is tenni. Erről - a hatékonyságról - is lesz szó a következő fejezetben

Feladatok

1) Barátságosnak mondunk két természetes számot, ha az egyik (nála kisebb) osztóinak összege megegyezik a másik számmal és viszont. Döntsük el két természetes számról, hogy barátságosak-e!

2) Keressük meg az első N barátságos számot.

3) Döntsük el egy számról, hogy tökéletes szám-e? (valódi osztóinak az összege+1 a számot adja)

4) Generáljunk egy sakkfelállást és döntsük el, hogy a világos király sakkban van-e!

5) Írjunk algoritmust, amely egy 11 jegyű számról eldönti, hogy személyi szám-e!

6) Állapítsuk meg, hogy az N természetes szám binárisan felírt alakjában hány darab egyes szerepel.

7) Számoljuk meg, hogy egy adott szöveg hány betűt tartalmaz, ha az ékezetes betűket (á,é,ó,ő,ú,ű) két jellel A,E,O,U, betűvel és aposztróffal (vagy idézőjellel) ábrázoljuk. (Pl.: a' vagy o")

8) Adott C valós szám és az A(N) valós vektor. Számoljuk meg, hány olyan eleme van a vektornak, amelyre a X*X-2*A(I)*X+C=0 egyenletnek két különböző valós gyöke van.

9) Állapítsuk meg, hogy a növekvően rendezett A(N),B(M) vektornak hány olyan közös eleme van, ami mindkettőben csak egyszer fordul elő.

10) Állapítsuk meg, hogy egy sorozatnak hány egymástól különböző eleme van.

11) Adott az A(N) vektor. Számoljuk meg, hogy a vektorban hány olyan számpár van, amelyek relatív prímek.

12) Az R sorozat az első N természetes szám egy permutációját tartalmazza. Állapítsuk meg, a sorozatban az inverziók számát (az 1,...,N sorrend eléréséhez hány csere szükséges)

13) Számoljuk meg, hogy egy négyzetes mátrixnak hány olyan, a főátlóval párhuzamos átlója van, amely tartalmaz negatív elemet.

14) Adott egy háromdimenziós csillagtérkép, amelyben a csillagokat koordinátáikkal adjuk meg. Csillagsűrűsödésnek nevezzük azokat a legbővebb csoportokat, amelyekben bármelyik két csillag távolsága kevesebb egy T értéknél, és a csillagok száma legalább N. Határozzuk meg térképpontokon a csillagsűrűsödések számát.

15) Állapítsuk meg, hogy az R szósorozatban hány magas-, mély-, vegyes hangrendű szó van.

16) Adott N város (X(1),Y(1)), ..., (X(N)Y(N)) koordinátákkal. Melyik városba telepítsük rádióadót, ha az a cél, hogy minden városban tudják fogni az adást, és a lehető legkisebb legyen az adó hatósugara.

17) Adott az A(N,M) mátrix és K szám. határozzuk meg a B(N) logikai vektor elemeinek értékét, ahol B(I) igaz. ha az A(N,M) i-edik sorának maximális eleme kisebb K-nál, különben hamis.

18) Adott az A(N,M) mátrix, amelynek elemei 0-k és 1-esek. Határozzuk meg a - vízszintesen, függőlegesen, átlósan - leghosszabb, csak egyeseket tartalmazó szakaszt!

19) Adott egy pozitív és negatív egész számjegyeket tartalmazó sorozat. Határozzuk meg azt a részsorozatát, ahol a számjegyek összege maximális!

20) Számoljuk meg, hogy egy számsorozatban hány különböző számjegy van

21) Adott az A(N,M) mátrix. Határozzuk meg a mátrixnak egy legnagyobb részmátrixát, amely nem tartalmaz 0 értékű elemet.

22) Képzeletbeli rácsot fektetünk egy hegyvidékre. A rácspontokban megmérjük a felszín tengerszint feletti magasságát. Határozzuk meg azt a két szomszédos rácspontot, amelyeknél az összekötő út a legmeredekebb.

23) A Vadása tóra egy négyzethálót fektetünk, és minden rácspontban megmérjük a víz mélységét. Feltételezzük, hogy a tó felszínén minden mérés eredménye nulla, a vízmélység pedig mindenhol nagyobb mint 0. A négyzetháló szélső rácspontjai a szárazföld felett vannak. Határozzuk meg a leggyorsabban mélyülő partmenti helyet.

24) Adottak az X,Y,Z halmazok. H jelöli az X,Y,Z közül pontosan két halmazban szereplő elemek halmazát. Írjunk algoritmust a H halmaz előállítására.

25) Adott egy iskola tanulóinak névsora a tanulók személyi számával. Válogassuk szét a hétfőn, kedden, ... születetteket.

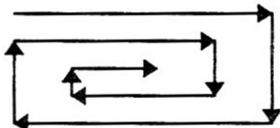
26) Rendezzük az A(N,M) elemeit úgy, hogy A(I,J)<=A(K,L) legyen, ha I<=K és J<=L.

27) Töltsünk fel egy N*M-es mátrixot 10 és 1000 közé eső véletlenszámokkal. Írassuk ki a mátrix elemeinek összegét, átlagát, de a megjelenítés során az azonos sorban, illetve azonos oszlopban lévő elemek a képernyőn is azonos sorban, illetve oszlopban legyenek.

28) Töltsünk fel egy N*M-es mátrixot 10 és 100 közé eső véletlenszámokkal. Válogassuk ki egy B vektorba azokat az elemeket, amelyek oszthatóak a nekik megfelelő indexek összegével. Pl. i=1, j=4 és A[i,j]=35, akkor az A[i,j]-t a B[]-be kell rakni, hiszen a 35 osztható 5-tel. Írassuk is ki a B vektort!

29) Egy N elemű vektor eleme 1 és 1345 közé eső egész számok. Írassuk ki:
- ezen számok számjegyeinek összegét,
- a számjegyek összegének az összegét
- a számjegyek számtani átlagát
Döntsük el, hogy
- van-e olyan eleme a vektornak, ahol a számjegyek szorzata maga a szám (a vektor elem)
- a számjegyek összege osztható a vektor elemmel!

30) Írassuk ki egy N*M-es mátrix elemeit csigavonalban.



31) Ismert a SuperBike világbajnokság 12 idei futamának beérkezési sorrendje. Adjuk meg azok névsorát, akik minden futamon célba érkeztek!

32) Ismertek 3 metróvonal állomásai. Adjuk meg - ha vannak - az egyes metróvonalak közös állomásait!

33) Írassuk ki a P természetes számhoz legközelebb álló kettőhatványt!

34) Egy áruházban a liftet csak úgy lehet használni, ha maximum N ember akar vele a földszintről felmenni valamelyik emeletre, s ekkor T időegység múlva tér vissza a földszintre. Készítsünk programot, amely rendszertelenül érkező emberek esetén szimulálja a lift működését!

35) Egy úszóversenyen 64-en vesznek részt. A versenyzőket 8 fős csoportokba osztják. Minden csoportból a legjobbak kerülnek a döntőbe. A nevek és az eredmények ismeretében írassuk ki az eredmények növekvő sorrendjében a döntőbe jutottak nevét és időeredményét!

36) Ismerjük egy tanulmányi versenyen résztvevők nevét és pontszámát. Írassuk ki a versenyzők nevét és elért pontjaikat a pontok csökkenő sorrendjében és készítsünk listát a nevek alfabetikus sorrendjében is!

VII. A PROGRAM HELYESSÉGE

Egy elkészült program tesztelésekor csak annyit mondhatunk ki, hogy az hibás, vagy hogy hiba a tesztelés során nem került elő. Ez utóbbi még nem jelenti azt, hogy a program feltétlenül jó, hibamentes. Egy program jóságáról megnyugtatóan csak programhelyesség-bizonyítással győződhetünk meg. A bizonyítás szigorú matematikai, logikai bizonyítást jelent.

A bizonyítás során az algoritmust bemeneti (elő-) és kimeneti (utó-) feltételekkel egészítjük ki, vagyis megfogalmazzuk a bemeneti adatokkal (kezdeti állapottal) és az eredményekkel (végállapottal) kapcsolatos elvárásainkat, feltételeinket. Ezt követően a program különböző részeinél odaillő elemi állításokat helyezünk el, majd ezen elemi állításokból logikai műveletekkel és a programváltozók különböző értékeivel összetett logikai állításokat, következtetéseket mondunk ki, a következő elemi állításra, amelynek igaznak kell lenniük, ha a két elemi állítás közötti programrész helyesen működik. Így lépésről-lépésre bizonyíthatjuk a program (algoritmus) helyességét, annak minden egyes szintjén.

Az egyes programrészek működésére már kimondott axiómák, bizonyított szabályok léteznek. Ilyen az értékadás, és az üres utasítás axiómái, vagy a szekvencia, az elágazás, a ciklus, a következmény, a befejeződő ciklus következtetési szabályai. A feladat meglehetősen bonyolult, ezért szokás a program (algoritmus) helyességének bizonyítását részfeladatokra bontani, annak befejeződését részleges (parciális) és teljes helyességét külön lépésben bizonyítani. A program befejeződése azt jelenti, hogy annak végrehajtása befejeződik valamilyen módon. A parciális helyesség fogalma azt jelenti, hogy olyan bemeneti adatra, amelyre a előfeltétel igaz és a program megáll, az eredmény is helyes lesz. A teljes helyesség már nyilvánvaló: a program minden érvényes bementi adatra megáll és az eredmény is helyes lesz.

Mivel ez az eljárás nehéz, nagy programra nem végezhető el, más megoldást kell keresni. Ez lesz a tesztelés. A tesztelés nem fogja garantálni a program hibamentességét, csupán valószínűsíti azt.

Egy programot csak akkor fogadhatunk el jónak, ha meggyőződünk helyességéről.

Tesztelés: először meg kell vizsgálnunk, hogy a program azt teszi-e, amit várunk tőle.

Hibakeresés: ha a tesztelés eredményeként azt kapjuk, hogy a program hibásan működik, akkor meg kell keresnünk a hibát.

Hibajavítás: végül, ha megtaláljuk a hibát, akkor azt ki kell javítanunk.

A hiba kijavítása után ez a folyamat újrakezdődik, hiszen meg kell győződni róla hogy a javítás sikeres volt, illetve arról is, hogy nem keletkezett miatta más hiba.

Tesztelési módszerek

A programok tesztelésének két logikailag eltérő módszere van. Az egyikben a program szövegét vizsgáljuk, ezt nevezzük *statikus tesztelésnek*. A másikban a program végrehajtását vizsgáljuk: ez a *dinamikus tesztelés*. Egy program ellenőrzését célszerű mindig statikus teszteléssel kezdeni, mert a tapasztalat azt mutatja, hogy a hibák jelentős

része már így kiszűrhető és ehhez nincs szükség a program sok időbe kerülő futtatatásához.

Statikus tesztelés

A kódellenőrzés során a program szövegét hasonlítjuk össze az algoritmussal. A formai ellenőrzést legtöbbször a fordító vagy értelmező programra bízhatjuk. Például kigyűjthetik a programban használt változókat, megmondhatják, hogy melyiket hol használtuk, értékét hol változtattuk meg.

A tartalmi ellenőrzés, ellentmondáskeresés sokkal nehezebb feladat. Ilyen tartalmi hiba, ha egy változónak nem adunk értéket, és mégis használjuk, vagy ha értéket adunk neki, de semmire sem használjuk, vagy amikor egy változóban halmozunk értékeket, majd nem azt íratjuk ki Ide tartoznak a vezérléssel kapcsolatos hibák: a végtelen ciklus, illetve amikor egy utasítást sohasem hajtunk végre.

Dinamikus tesztelés

Lényegük, hogy a programot különböző tesztadatokkal kipróbáljuk, majd az eredményekből következtetünk a program helyességére. Elvileg elérhető lenne, hogy így garantáltan meggyőződjünk a program helyességéről. Erre két módszer is kínálkozik. Az egyik szerint hajtsuk végre a programot az összes lehetséges bemenő adattal, majd vizsgáljuk az eredményt! A másik, hogy a programot hajtsuk végre, utasításai összes lehetséges sorrendjében! A gyakorlatban persze mindkét ötlet kivitelezhetetlen. Ha csak egy egész számot olvas be a program, akkor is annyiszor kellene kipróbálni az első elv szerint, ahány egész szám lehetséges, nem is szólva a valós számokról. Ha két egész szám van, akkor már az előző mennyiség négyzete a kipróbálások száma. A másik elv szerint, ha a programban egy ciklus található, akkor ki kell próbálnunk úgy, hogy a ciklusmag egyszer sem fut, egyszer fut, kétszer fut,.... Ez persze szintén járhatatlan út. Mégis e két elv határozza meg, a dinamikus tesztelés két fő osztályát: a fekete és a fehérdoboz módszereket.

Feketedoboz módszerek

Ebben a program adatait vizsgáljuk, s ez alapján próbálunk tesztadatokat készíteni, s magát a programot, mintegy fekete doboznak tekintjük, éppen ezért nevezik adatvezérelt tesztelésnek is. Magát a program szövegét nem is nagyon ismerjük. Itt tulajdonképpen arról van szó, ha a program két bemenő adatra egyformán működik, akkor elég az egyikkel kipróbálni. Ez alapján a bemenő adatok csoportosíthatók, osztályba sorolhatók. A tesztelés lényege ezen osztályok meghatározása

Ekvivalencia osztályokat kell keresni. A bemenő adatokat soroljuk osztályokba a következő elv szerint.

Ha az osztály egyetlen tagjával kipróbálva a programot jó eredményt kapunk, akkor várhatólag a többi elemre is jó eredmény kapunk! Ha a program az osztály egy tagjára hibásan működik, akkor a többire is működjön hibásan. Ezeket az osztályokat a feladat elemzése alapján határozhatjuk meg. Az érvényes bemenő adatokra is kell létrehozni osztályokat és az érvénytelenekre is! Például, ha a feladatunk egy max. 50 karakteres

szöveg magánhangzóinak megszámlálása, akkor a következő ekvivalencia osztályokat vehetjük fel:

- 0 hosszúságú szöveg,
- több, mint 50 hosszúságú szöveg,
- a szövegben nincs magánhangzó
- a szövegben van magánhangzo, az utóbbi osztályt esetleg további részekre oszthatjuk:
- a szövegben van A betű
- a szövegben van Á betű

.....

Ha választottunk ekvivalencia osztályokat, akkor el kell döntenünk, hogy mikor melyikből választunk kipróbálandó adatot. Ezt megtekinthetnénk úgy is, hogy olyan bemenő adatokat választunk, amelyek egyszerre pontosan egy ekvivalencia osztályba tartoznak, ennél azonban van jobb stratégia is. Az előbbi feladatra gondolva, azt a programot várhatóan felesleges úgy kipróbálni, hogy egyszer csak A betű van a szövegben, egyszer csak Á... Először válasszunk olyan bemenő adatot, amely minél több érvényes ekvivalenciaosztályt fed le! Ha az érvényes osztályokkal végeztünk, akkor vehetjük egyenként az érvényteleneket. Ezeket mindenképpen külön kell vizsgálnunk, hiszen egy hibás adat miatt elképzelhető, hogy a következőt nem vizsgáljuk meg.

Célszerű az egyes osztályokból határon levő elemeket választani! Az előbbi példára gondolva, vizsgáljunk 51 betűs szöveget, 50 betűst, 1 betűst. 0 betűst.

Fehérdoboz módszerek

Most elsődlegesnek a program szerkezetét, logikáját tekintjük. Itt az utasítások összes lehetséges végrehajtási sorrendjénél szeretnénk valami okosabbat kitalálni. Az utasítások egyszeri lefedésének elve alapján olyan tesztadatokkal kell kipróbálni a programot, amelyek esetén a program minden utasítását végrehajtjuk legalább egyszer. Ezzel legalább azt elérjük, hogy ha a formai, szintaktikus ellenőrzés nem történt meg, akkor a végrehajtás során kiderüljenek a szintaktikus hibák.

Arra kell törekedni, hogy a programban szereplő összes döntés, elágazási feltétel legyen valamikor igaz és valamikor hamis értékű is! Azaz a program valamennyi ágára rákerüljön a vezérlés legalább egyszer.

Hibakeresés

Hibakeresésről akkor beszélünk, ha már tudjuk, hogy van hiba a programban, csak még nem tudjuk, hogy hol. A hibakeresés célja a hiba helyének meghatározása. A hiba okozóját különböző módszerekkel kereshetjük.

Hibakeresési módszerek

Próbáljuk rendszerezni a rendelkezésünkre álló tesztadatokat! (indukciós módszer) Ebből próbáljuk valamilyen feltételezést tenni a hiba okára vonatkozóan! Ha ezt a feltevést igazolni tudjuk, akkor a hibát megtaláltuk, következhet a kijavítása, ha nem, akkor új feltevéssel kell próbálkozni. Például, ha egy program jól működik a 4, 8, 20 bemenő adatokra és hibásan a -1, -7-re, akkor tételezzük fel először, hogy a pozitívakra

jól működik, a negatívokra pedig nem. Ha ezt sikerül igazolni, akkor készen vagyunk, ha nem, akkor vizsgáljuk meg azt a feltevést, hogy a párosakra működik jól és a páratlanokra hibásan! Ha ez sem sikerül, akkor még mindig feltehetjük, hogy az abszolútértékben prímszámokra működik hibásan. A példából egy fontos dolog kiderült: a hipotézisek felállításához nincs szükség a program ismeretére, az csak a hipotézis igazolásához kell.

A dedukciós módszer lényege a hiba lehetséges okai körének szűkítése. Először minden lehetséges okot feltételezünk, majd ezek közül ki kell küszöbölni azokat, amelyek nem okozhatnak hibát.

Vizsgáljuk meg így az előző példát! Első feltevésünk az, hogy negatív, vagy páratlan, vagy prím abszolútértékű, vagy 2-vel nem osztható, vagy ... adatokra a program hibásan működik. Várhatóan ennyi hiba nincs a programban, ezért ezt a kört szűkíteni kell mindaddig, amíg meg nem találtuk a hibát.

A visszalépéses módszer a hiba előfordulásának helyetől (kiír vagy futtatási hibaüzenet) indul ki, s követi visszafelé a program végrehajtását mindaddig, amíg az adatok hibásak. Amint egy olyan ponthoz érünk, amikor még nincs hiba, akkor megtaláltuk a hiba helyét.

Mindhárom módszert segíthetjük a teszteléssel, csak ilyenkor a tesztelés célja nem új hiba felfedezése, hanem egy meglévő lokalizálása. Így tehát olyan adatokkal kell kipróbálni a programot, amelyek közelebb visznek a hiba felfedéséhez.

Hibakeresési eszközök

A hibák megkereséséhez a legtöbb programozás nyelv biztosít valamilyen segédeszközt. Ha egy eszköz az adott esetben nem elérhető, általában akkor is van mód elkészítésükre. Ezek között találhatunk egyszerű eszközöket és bonyolultakat is.

A lehető legegyszerűbb eszköz a *kiírás*. A program alkalmas pontjain elhelyezett kiíró utasításokkal sok hasznos információt nyerhetünk programunkról. Ha nem történt kiírás, akkor erre az ágra nem került a vezérlés. Ha például két adatnak egyformának kellene lennie, akkor az egyenlőség helyén kiírathatjuk a két adatot, így eldönthető az egyenlőség. Arra kell csupán vigyázni, hogy ha kiírást ciklus belsejébe teszünk, akkor annyi adatot kaphatunk, amennyit nehéz lesz feldolgozni, bár a ciklus belsejébe tehetünk még egy várakoztató utasítást is.

A *töréspontok* elhelyezése lehetővé teszi, hogy a végrehajtás adott [illegible] a program, ott le lehessen kérdezni a változók értékeit, esetleg meg lehessen [illegible] azokat, majd tovább lehessen folytatni a programot vagy esetleg újraind[illegible]

A *nyomkövetés* nyújt lehetőséget arra, hogy a program [illegible] követhessük (így megtudjuk, hogy programunk merre jár). Ezt esetleg [illegible] utasításokra vagy a program egyes részeire is kérhetjük. Van [illegible] végrehajtott utasítás eredményéről is tájékoztatást kapunk. Pl. [illegible] logikai feltétel igaz vagy hamis volta).

A *változófigyelés* arra szolgál, hogy kiválasztott [illegible] vagy [illegible] változásait észrevegyük.

A *lépésenkénti végrehajtás* a töréspontok speciális esete. Töréspontot helyezünk a program belsejébe, majd a megállás után lépésenként (utasításonként) folytathatjuk a végrehajtást.

A *memóriakiírás (dump)* gépi kódú programozásnál használható eszköz, de nagyon óvatosan kell bánni vele, mert rengeteg kiírást eredményezhet, melyben nehéz megtalálni a számunkra szükségeseket.

Az *állapotellenőrzés* lényege a program változóira felírt állítások elhelyezése a programban, s ha ezek az állítások nem teljesülnek, akkor valamilyen tevékenység (kiírás, leállás, egyéb) elvégzése.

A *szemantikus (tartalmi) ellenőrzésre* akkor van szükség, ha a program futásakor az ilyen hibákat automatikusan nem figyeli (a gyors futás érdekében), s e lehetőség beépítését a fordítóprogramtól külön kell kérni. Ilyen például egy vektor indexének túlfutása utáni hivatkozás egy nem létező elemre. (I>N, ahol N a vektor elemeinek a száma, így A[I] nyilván hibát eredményez)

Bármelyik hibakeresési eszközt is használjuk, a program szövegében valamilyen utasítást beépítettünk, akkor azt a hiba kijavításakor után el is lehet onnan távolítani. Igen ám, de ha újabb hibát találunk, akkor valószínűleg újra be kell írni őket. Ezért az a jó rendszer, ahol ezeket nem kell kitörölni, hanem a fordítás/végrehajtás előtt megmondhatjuk, hogy kellenek-e vagy sem. Ha erre nincs lehetőségünk, akkor is célszerű ilyenkor megjegyzés sorokba tenni őket, semmint kitörölni őket.

Hatékonyság vizsgálat

A hatékonysággal kapcsolatos kérdésekkel szemben az első érv az szokott lenni, hogy ez a professzionális programozásban érdekes dolog, de nekünk lényegében semmi dolgunk vele. Ez így van. A programozási tételek algoritmustípusok alkalmazása ugyanis csak a feladatok megoldására adnak receptet, de ettől még nagyon rossz megoldást eredményezhetnek. Nézzük meg azt a programot, amely az összegzést alkalmazza az első N szám összegének kiszámítására!

Eljárás Egy
S:=0
Ciklus cikl:=**1-től** N-**ig**
    s:=s+cikl
Ciklus vége
Eljárás vége

Ezt a feladatot sokkal egyszerűbben is meg lehet oldani, némi matematikai ismeretek birtokában:

Eljárás Jobbegy
    S:=N*(n+1)/2
Eljárás vége

Mielőtt belekezdenénk a módszerek tárgyalásába, meg kell állapítanunk, hogy micsoda valójában a hatékonyság, mit mérünk, amikor a hatékonyságot mérünk. Három

dolog jön számításba: a végrehajtási idő, a helyfoglalás (programszöveg és adatok), valamint a bonyolultság (elkészítési/megértési idő, szükséges tudás)

Más osztályozásra jutunk, ha a vizsgálandóból indulunk ki. Globális hatékonyságról beszélünk, ha az algoritmusleíró nyelven írt algoritmust vizsgáljuk (esetleg kódolási szempontokat is figyelembe véve), lokális hatékonyságról pedig akkor, ha a program szövegét. Az elsőhöz érteni kell a program működését, a másodiknál az adott részlet szerkezetét kell csak ismerni, de a program globális működését nem. Mi először a globális hatékonysággal fogunk foglalkozni

A végrehajtási idő csökkentése

A program végrehajtási idejét elsősorban a ciklusok (és az elágazások száma) befolyásolják. Ha egy utasítást ugyanis cikluson kívül hajtunk végre, a végrehajtási idő nagyon-nagyon kis százalékát jelenti csupán. Nyilvánvaló tehát, hogy a ciklusokkal kell valamit tenni. A végrehajtási idő csökkentésére szolgáló módszerek egy része a ciklusok végrehajtási számát próbálja csökkenteni, más része pedig a ciklusmag egy végrehajtásának idejét

A ciklus végrehajtási számának csökkentése

Ha lényegesen csökkenteni tudjuk egy ciklusmag végrehajtási számát, az a program végrehajtásában jelentős előnyöket hozhat. Erre különböző módszereket találhatunk. A legjelentősebb módszerek közé tartozik az adatok speciális tulajdonságának kihasználása. Ha ugyanis rendelkezünk valamilyen tudással a feldolgozandó adatokról, akkor ha ez kihasználható a program elkészítésében, nagy futásidő-nyereséget érhetünk el.

Mi lehet az a speciális bizonytalanság? Például lehet a rendelkezésre álló adatok rendezettsége. A programozási tételek között két ilyen tulajdonságú feladat is szerepelt. Az egyikben egy rendezett sorozatban egy adott értéket kellett keresni, ez volt a logaritmikus keresés. A másikban két rendezett sorozat unióját kellet megadni, ez volt az összefuttatás. Mindkét algoritmussal jóval kevesebb lépéssel lehetett megoldani az adott feladatot, mint rendezettség figyelembevétele nélkül, keresési,-illetve uniótétellel.

A második módszer a matematikai ismeretek kihasználására épül. Nagyon sokszor előfordul az, hogy valamilyen szempontból vizsgálandó sorozat helyett elég valamely részsorozatát vizsgálni az adott feladat szempontjából. Például ez a helyzet, ha egy számról kell dönteni, hogy prímszám-e. A megoldás alapváltozata azt mondja, hogy N akkor prímszám, ha 2 és N-1 között nincs osztója. A matematikai ismeretek felhasználásával viszont állíthatjuk, hogy egy szám akkor prím, ha kettő és saját négyzetgyöke között nincs osztója. Ilyen jellegű tudást nagyon sok feladatban felhasználhatunk. Például a legnagyobb osztó=N/legkisebb osztó, így a legnagyobb osztó keresésekor elég, ha a legkisebbet keressük, amit gyorsabban lehet megtalálni

Az alábbi függvény eldönti egy számról, hogy az prímszám-e

Függvény prim(A) ** A egész és >1 **
prime:=**Igaz**
Ciklus i:=2 **-től** Gyök(A)**-ig**
  Ha A MOD i =0 **akkor** prime:=**Hamis**
Ciklus vége
Függvény vége (Visszaad:prime)

Itt kihasználtuk a korábban említett matematikai tudást, de a ciklusmag végrehajtása akkor is folytatódik, ha A-ról már az elején kiderült, hogy nem prím. Pl. ha A=10000, akkor a ciklusmag 999-szer fog végrehajtódni, pedig a 2-vel való osztásnál (i=2) már a prime értéke Hamis lett, mivel 10000 osztható 2-vel, azaz nem prím. Csökkenthető a ciklus végrehajtási száma a következő módon:

```
Függvény prim(A) ** A egész és >1 **
prime:=Igaz: i:=2
Ciklus amíg i<= Gyök(A) És prime ** ha prime Hamis, azonnal befejeződik a
                                                                      ciklus
        Ha A MOD i =0 akkor prime:=Hamis
Ciklus vége
Függvény vége (Visszaad:prime)
```

Nézzünk még egy példát!

```
Eljárás Számol
Be: A,B (A,B>0 és egész)
Ciklus amíg A<>B
        Ha A<B akkor B:=B-A Egyébként A:=A-B
Ciklus vége
Eljárás vége
```

Itt a legnagyobb közös osztót határozzuk meg. Azt használtuk ki, hogy A és B közös osztói A-B -nek is, ha A>B. Aki ezzel a matematikai ismerettel nem rendelkezik, az a feladatot így képtelen megoldani, holott ez az egyik leghatékonyabb megoldás. Persze ha a két szám különbsége túl nagy, akkor már nem túl hatékony ez a megvalósítás, inkább a bonyolultságot csökkenti. Ekkor viszont gyorsabban kapunk eredményt az Euklideszi algoritmus megvalósításakor:

```
Eljárás Euklidesz
Be: A,B (A>B>0 és egészek)
Ciklus amíg A MOD B<>0      ** A osztható B-vel **
        seged:=A MOD B
        A:=B
        B:=seged
Ciklus vége
```

Az utóbbi eljárásban a ciklusmag lényegesen kevesebbszer hajtodik végre, ha pl. A=1000, B=5. Nincs viszont nagy különbség ha A=24, B=9. Az utóbbi eljárásban azonban egy segédváltozót is használunk és kettővel több értékadás is szerepel, nincs viszont elágazás A segédváltozó használatát ki lehet küszöbölni, de csak újabb értékadó utasítások használatával. (Pl. A:=A-B: B:=B+A: A:=B-A) Vagyis a ciklus végrehajtási ideje csökken, de ezzel szemben valamelyest nőhet a végrehajtási idő.

Egy ciklus végrehajtási idejének csökkentése

A másik lehetőségünk, hogy a ciklusmag végrehajtási idejét csökkentjük. Ez azért lehet jelentős, mert ha a ciklusmagot sokszor hajtjuk végre, akkor a teljes futási idő is lényegesen nőhet.

Az első módszer az elágazások transzformálására épít. Azt használja ki, hogy egy elágazás feltételének vizsgálata helyett általában egyszerűbb egy indexes változó indexelése. Például, ha meg kell számolni, hogy egy szövegben az angol ABC egyes nagybetűi hányszor fordulnak elő, akkor a következő két megoldás közül a második a rövidebb végrehajtási idejű:

```
Eljárás Betűszámlálás(N, A[], BETŰ[ ] )          **[ ] vektort jelent**
BETŰ[]:=0   ** BETŰ vektor nullázása
Ciklus cikl:=1-től n-ig
      Ha A[cikl]="A" akkor
            BETŰ[Sorszáma("A"]:=BETŰ[Sorszáma("A"] +1
      Egyébként
            Ha A[cikl]="B" akkor ...
Ciklus vége
Eljárás vége
```

```
Eljárás Betűszámlálás(n.a[],betű[])
Betű[]:=0
Ciklus cikl:=1-től n-ig
      x:=Sorszáma(a[i]) ** az a[i]-ben lévő betű sorszáma **
      Betű[x]:=Betű[x]+1
Ciklus vége
Eljárás vége
```

Másik módszer a ciklusok transzformálása az adatszerkezet-választás alapján. Itt általában adatábrázolás-választással érhetjük el, hogy egy ciklus helyett egyetlen műveletet kell elvégeznünk. Itt tehát tulajdonképpen egyes összetett adattípusok létező műveleteit használjuk a végrehajtási idő csökkentésére. Gondoljunk arra a példára, amikor a programozási tételeknél a halmazok metszetének, illetve uniójának elkészítését vizsgáltuk. Mindkét esetben két, egymásba ágyazott ciklust használtunk. Ugyanakkor ezen feladatok megoldását a halmazok jól megválasztott ábrázolásával lényegesen meggyorsíthatjuk.

Sokszor kivételes esetek vizsgálata lassítja a programot, ezért sebességnövekedést érhetünk el a kivételes eset kiküszöbölésével. Gondoljunk egy olyan feladatra, amikor egy számsorozatban kell egy adott tulajdonságú elemet keresni (n elemű, a vektort használunk):

```
Eljárás újabb
i:=1
Ciklus amíg i<=n és a[i] nem a keresett
      i:=i+1
Ciklus vége
Ha i<=n akkor Ki: i
Eljárás vége
```

Itt a kivételes eset az, hogy a vektorban nincs benne a keresett elem. Emiatt van szükség az i<=n feltétel vizsgálatára. Ha tudnánk, hogy biztosan van ilyen elem, akkor ezt az ellenőrzést kihagyhatnánk, amitől a ciklus egyszeri lefutásának ideje kisebb lenne. A kivételes esetet úgy szüntethetjük meg, hogy a vektor végére egy olyan elemet helyezünk, mint amilyet keresünk. Ha a keresés során ezt találjuk meg, akkor nem volt ilyen elem, különben pedig volt.

```
Eljárás Keres ** keresés strázsával **
a[n+1]:=amilyet keresünk
i:=1
Ciklus amíg a[i] nem a keresett
      i:=i+1
Ciklus vége
Ha i<=n akkor Ki:i
Eljárás vége
```

A negyedik módszer az adatok speciális tulajdonságainak kihasználására épít. Már láttuk, hogy ennek alapján a ciklusok végrehajtási számát csökkenthetjük. Ez a módszer néha egy futás végrehajtási idejét is csökkentheti. Például, ha egy szám csak nullákat és egyeseket tartalmaz, és meg kell számolni, hogy melyikből mennyi van, akkor elég a nullák számát számolni, mert egyesek száma az összes számjegyek számából levonva a nullák száma.

Sokszor érhetünk el futásidő-csökkenést az adatok előfeldolgozásával is. Ez azt jelenti, hogy ha egy kifejezést többször kell kiszámolni, akkor ezt tegyük meg egyszer, majd később csak használjuk a kiszámított értéket!

Gyakran a konkrét adatmozgatással veszítünk rengeteg időt. Az adatmozgatást elkerülhetjük, ha listákat, mutatókat használunk az adatok sorrendjének, kapcsolatainak leírására. Ez a módszer a hivatkozások módszere.

A helyfoglalás csökkentése

Kétféle szempontból érdemes foglalkozni a helyfoglalással. Egyrészt csökkenthető a program által használt adatok helyfoglalása, másrészt magának a programszövegnek a helyfoglalása. Az egyes módszerek ezek közül valamelyikkel foglalkoznak.

Az adatok mennyiségének csökkentése

A legtöbb nyereséget hozó módszer az indexes változók kiküszöbölése. Sokszor használunk tömböt az adatok tárolására is, amikor valójában nincs rá szükségünk. Például az n. Fibonacci-féle számot[22] megadó program:

```
Program Csokk
Be:n
f[0]:=1: f[1]:=1
Ciklus i:=2-től n-ig
      f[i]:=f[i-1]+f[i-2]
Ciklus vége
Ki:f[n]
Program vége
```

Így szükségünk van egy n+1 elemű vektorra, pedig minden új érték kiszámításához csupán az előző kettő elemre lenne szükségünk:

```
Program Csokk2
Be:n (n egész és >1 )
Fib1:=1; Fib2:=1
Ciklus i:=2-től n-ig
      f:=Fib1+Fib2
      Fib1:=Fib2; Fib2:=f
Ciklus vége
Ki:f
Program vége
```

Az indexes változók kiküszöbölésére gyakran jó lehetőség van a ciklusok összeolvasztásával. Azaz, ha egy kiszámítandó vektor minden elemére csupán egyszer van szükség, akkor a vektor helyett inkább a szükséges helyen számítsuk ki - egyszer - a megfelelő értéket!

Ha egy indexes változót nem szüntethetünk meg, akkor is hozhat eredményt a változó transzformálása. Ha például egy mátrixnak csak a diagonálisában vannak nem nulla elemek, akkor felesleges a többi helyen levő nullákat tárolni, helyette csak egy elem megadását célszerű módosítani. A diagonális elemeket beírjuk egy vektor[i] tömbbe és az A[] mátrixot a következő függvénnyel helyettesítjük:

```
Függvényeljárás Ama (i,j)  ** i,j egészek **
Ha i=j akkor a:=vektor[i] Egyébként  a:=0
Eljárás vége (visszaad: a)
```

Egy ehhez hasonló transzformáció sok speciális mátrix esetén lehet gazdaságos.

[22] Fibonacci számok: 1 1 2 3 5 8 11 19 ... , azaz egy szám az előző kettő összege, kivéve a két egyest.

Hasonlít az előző módszerre az adatreprezentáció megválasztása. Itt arról van szó, hogy az adatainkat úgy ábrázoljuk, hogy minél kisebb helyet foglaljanak! Ez egyszerű esetben azt jelenti, hogy valós típusú változó helyett használjunk egész típusút, ha csak egész számokra van szükségünk. Ezt ki is használtuk a minimum kiválasztás tételénél.

A programszöveg méretének csökkentése

Az első módszer tulajdonképpen a végrehajtási idő csökkentés egy módszerének megismétlése: az adatok előfeldolgozása. Ha ezt megtesszük, akkor a feldolgozó kódrészt nem kell sokszor megismételni, így a programszöveg is rövidülhet.

A második módszer előforduló részek egyszeri leírásán alapul, ez az azonos funkciók eljárásba foglalása. Lényege, hogy a programban több helyen is szereplő azonos/hasonló részeket önálló programegységekbe foglaljuk, s egyszer írjuk le! (Ne használjuk ki a szövegszerkesztők blokkparancsait, és másolgassuk az azonos részeket, hiszen ha az a bizonyos rész hibás, akkor kezdődhet minden elölről!)

Ha három halmaz metszetének a meghatározása a feladat, ne írjuk le kétszer a metszetképzés tételét, írjuk át inkább az algoritmust olyan eljárássá, amelynek lehet átadni tömböt.

A bonyolultság csökkentése

Ez a jellemző az, amelyik a legnehezebben megfogható, hiszen nincs az előző kettőhöz hasonló mérőszáma. Mégis érdemes vele foglalkozni, mert az egyszerűség jelentősen csökkentheti a program megértésének idejét, ami a hibát keresők, módosítók számára igen nagy előny.

A kivételes eset sokszor amiatt fordul elő, mert egy ciklus lefutásakor az első esetben mást kell csinálni, mint a többiben, ráadásul ez azért fordul elő, mert egy változónak valamilyen kezdőértéket kell adni. Ekkor segít a fiktív kezdőértékadás elve, amely azt mondja ki, hogy adjunk a kérdéses változónak olyan kezdőértéket, amelyet a program a ciklus első lefutásakor biztosan módosít, s így nem kell külön figyelni az első lefutás esetét. Például egy maximum-kiválasztásos feladatnál:

```
Eljárás Csokk
Érték:=a[1]
Ciklus cikl:=2-től n-ig
        Ha a[cik]> Ertek akkor
                érték:=a[cikl]
        Ciklus vége
Eljárás vége
```

Bonyolultságot szokott okozni az is, hogy egyszerre túlságosan sokat akarunk megoldani. Ezen segít a funkciók szétválasztásának elve. Például, ha a feladat egy számsorozat összes maximális értékű elem sorszámának kiírása, akkor ezt ne egy ciklusban oldjuk meg (bonyolult adatszerkezettel), hanem egymas után vegezve a két lépést:

Eljárás MaxMin
érték:=a[1]
Ciklus i:=2-**től** n-**ig**
    Ha a[i] < érték **akkor** érték:=a[i]
Ciklus vége
Ciklus i:=1-**től** n-**ig**
    Ha a[i]= érték **akkor Ki**: i
Ciklus vége
Eljárás vége

Lokális hatékonyság

Most a program teljes megértése nélküli módszerek következnek.

Elvi tanácsok:
- gyors műveletek használata (a^2 helyett a*a)
- kifejezések egyszerűsítése
- részeredmények megszüntetése
- függvényhívások kiküszöbölése (a<GYÖKVONÁS(x) helyett a*a<x)
- kis helyfoglalású adattípusok használata

Kódolási technikák
Ezek egy-egy nyelv kódolási lehetőségeit használják ki:
- többágú szelekció sok egymásutáni kétágú szelekció helyett ("Case" utasítás "If" helyett
- halmaztípus vektor helyett
- stb.

Program-transzformációk

Ezek a módszerek a programszerkezet formális jegyei alapján dolgoznak. Azt adják meg, hogy milyen szerkezetű programot milyen feltételek esetén lehet más szerkezetűvé alakítani.

a) A ciklustól független utasítások kiemelése

Ciklus cikl:=1-**től** n-**ig**
    k:=10
    s:=s+a[i]
Ciklus vége

Hatékonyabb

k:=10

```
Ciklus cikl:=1-től n-ig
      s:=s+a[i]
Ciklus vége
```

Még egy részlet ugyanerre a problémára:

```
Ciklus h:=h0-tól 0-ig -1 -esével
      Eh:=m*g*h
      Em:=m*g*h0-m*g*h
      Ki:eh,em
Ciklus vége
```

Hatékonyabb:

```
mg:=m*g;e0:=mg*h0; eh:=e0
Ciklus h0 >=0
      Ki: eh,em
      h0:=h0-1
      Eh:=eh-mg
Ciklus vége
```

b) Ciklusok összevonása

Ha két ciklus azonos lépésszámú, végrehajtásuk egymástól független, akkor összevonhatók.

```
Min:=a[1]
Ciklus i:=2-től n-ig
      Ha min>a[i] akkor min:=a[i]
Ciklus vége
Max:=a[1]
Ciklus i:=2-től n-ig
      Ha max<a[i] akkor max:=a[i]
Ciklus vége
```

Hatékonyabb (bár bonyolultabb):

```
min:=a[1]; max:=min
Ciklus i:=2-től n-ig
      Ha min>a[i] akkor min:=a[i]
      Ha max<a[i] akkor max:=a[i]
Ciklus vége
```

c) Elágazások összevonása

Ha egymás után két elágazás található, amelynek ugyanaz a feltétele, s a végrehajtás alatt is ugyanúgy értékelődnek ki, akkor összevonhatók.

Ha a<b **akkor** min:=a **Egyébként** min:=b
Ha a<b **akkor** max:=b **Egyébként** max:=a

Hatékonyabb:

Ha a<b **akkor** min:=a ; max:=b
Egyébként min:=b ; max:=a

d) Elágazások felesleges feltételeinek elhagyása

Ha két-vagy többirányú elágazást használunk, és az utolsó ág feltétele egyenértékű a "minden más" esettel , akkor ez a feltétel elhagyható.

Ha x<0 **akkor** Ki: "negatív"
Egyébként Ha x=0 **akkor ki**: "Nulla"
Egyébként Ha x>0 **akkor Ki**:"pozitív"

Hatékonyabb:

Ha x<0 **akkor Ki**: "negatív"
Egyébként Ha x=0 **akkor Ki**:"Nulla"
Egyébként Ki: "pozitív"

e) Összetett feltétel szétválasztása

Ha egy feltétel két részfeltételből áll, s ezek közül az egyik kiértékelése sok időt vesz igénybe, ugyanakkor a másik kiértékelése gyakran egyértelművé teszi a feltételt, akkor célszerű ezeket a részfeltételeket külön vizsgálni. Ennek oka, hogy sok programozási nyelv a logikai műveletek mindkét operandusát kiértékeli, függetlenül attól, hogy igazságértéke sokszor az egyik alapján eldönthető lenne:

Ha p osztója n-nek **És** p prímszám **akkor Ki:** p, " prímosztó"

Hatékonyabb:

Ha p osztója n-nek **akkor**
Ha p prímszám **akkor Ki**: p,"prímosztó"

f) Elágazás ágainak kiemelése

Ha az elágazás mindkét ága egyformán indul és a közös rész nem változtatja meg a feltétel kiértékelését, akkor ez a közös rész kiemelhető az elágazás elé. Ha az elágazások vége azonos, akkor ez mindentől függetlenül kiemelhető az elágazás mögé. Mindkét átalakítás a programszöveg hosszát csökkenti.

Ha a<b **akkor** min:=a ; **Ki**: min **Egyébként** min:=b; **Ki**: min

Rövidebb:

Ha a<b **akkor** min:=a **Egyébként** min:=b
Ki: min

TÁRGYMUTATÓ

É

Élettartam • 45

Ü

Üres utasítás • 28

A

adatok elszigetelésének elve • 12
adattípus • 42
algoritmus • 10, 12, 14, 15, 61, 64, 65, 66, 67, 70, 71, 72, 74, 80, 81, 82, 85, 86, 88, 94, 100
alprogram • 34, 35, 36, 37, 38, 39, 40
attribútumok • 18, 19
azonosító • 15, 16, 17, 18, 25, 34, 43, 44, 52

B

blokk • 19, 40, 44
bottom-up • 13

C

címke • 15, 17, 18, 27
ciklus • 13, 29, 30, 31, 32, 33, 63, 64, 67, 71, 72, 73, 75, 78, 79, 80, 82, 86, 94, 95, 97, 99, 100, 101, 102, 104, 106

D

döntések elhalasztásának elve • 12
döntések kimondásának (el nem mulasztásának, nyilvántartásának) *elve.* • 12
Dinamikus tárkezelés • 19, 45

E

Egésztípus • 46
elöltesztelő • 31, 32
eljárás • 13, 34, 35, 36, 38, 40, 46, 49, 56, 65, 67, 74, 81, 84, 85, 88, 94
Eljárás-orientált nyelvek • 18

F

főprogram • 44, 45
függvény • 34
feltételes utasítások • 27, 62
Folyamatábra • 66
frontális feladatmegoldás • 11

G

globális • 13, 35, 44, 45, 99

H

hátultesztelő • 31
halmaz • 80
Hatáskör • 44
Hibakeresésről • 96

I

imperatív • 15
Implementáció • 42

J

jobbról-balra szabályt • 23

K

Környezet • 34
kötött formátumú nyelvek • 17
Karakter • 15, 48, 54
karakterkészlete • 15, 16
kifejezés • 15, 20, 21, 23, 24, 25, 26, 28, 29, 31, 36, 40, 49, 50, 52, 62, 65, 66
kifejezések • 23, 24, 25, 28, 38, 61, 105
konstans • 15, 21, 22, 23, 25, 26, 40, 46, 49, 55
Konverzió • 21

kritikus részek kiemelésének • 13
kulcsszavak • 15, 16, 17, 18

L

lépésenkénti finomítás elvének • 11, 12
Lexikai egységek • 15, 16
lista • 57
lokális • 13, 19, 35, 44, 45, 46, 99

M

mátrix • 52, 54, 75, 76, 79, 88, 89, 91, 92, 103
megjegyzés • 15, 17, 98
mondatszerű leírás • 61

N

nevesített konstans • 15, 21, 22
nyílt rendszerű felépítés elve • 12
nyomkövetés • 97

P

párhuzamos finomítás elve • 12
paraméterátadás • 24, 39
Paraméterek • 34
Paraméterkiértékelés • 38
precedencia szabály • 23

S

sor • 56
Specifikáció • 42
standard típusok • 42
Statikus tárkiosztás • 19
struktogram • 68
szöveg • 54
szabad formátumú nyelvek • 17
szekvencia, • 13
szelekció • 27, 28, 63, 105

T

típus-egyenértékűség • 24
típuskényszerítés • 24
Típus-kompatibilitás • 24
tömb • 43, 51, 52, 53, 54
töréspontok • 97
tesztelés • 95
tipizált konstans • 15, 21, 25
top-down • 11

U

Utasítások • 15, 27

V

változó • 15, 18, 19, 20, 21, 22, 23, 26, 30, 31, 35, 40, 45, 48, 49, 54, 62, 63, 70, 79, 85, 87, 101, 103, 104
vektor • 78
verem • 55
vissza az ősökhöz elvre • 12

IRODALOMJEGYZÉK

Számítástechnikai feladatok 2000-ig I-II.: Szerkesztette:Dr. Hetényi Pálné

Számítástechnika középfokon: Szerkesztette: Dr. Hetényi Pálne

Niklaus Wirth : Algoritmusok+Adatstruktúrák = Programok

Angster Erzsébet - Kertész László : Turbo Pascal 6.0

Szlávi Péter: Adatok. adattípusok

Benkő Péterné. Benkő László. Tóth Bertalan: Programozzunk C nyelven!

300

GO-0017/2

A sorozat moduljai

1. Frank Pálné	Bevezetés az informatikába
2. Faránki Gyula	Informatikai eszközök
3. Busi Lajos	Számítógépes szoftverek
4. Bánhegyesi Zoltán	Számítógép-hálózatok
5. T. Várkonyi Attila	A természet informatikája
6. Dr. Koncz József	Irodai alkalmazások: szövegszerkesztés
7. Módos Gábor	Word for Windows 2.0 szövegszerkesztő
8. Bánhegyesi Zoltán	Adatfeldolgozás alapjai
9. Bánhegyesi Zoltán	dBase III Plus kezelése
10. Módos Gábor	Works for Windows 3.0 alapjai
11. Nemcsik János	dBase IV kezelésének alapjai
12. Faragó István-Piross László	A szövegszerkesztés alapjai
13. Nemcsik János	A táblázatkezelés logikája
14. Módos Gábor	Excel for Windows 4.0 táblázatkezelő
15. DombováriMátyás	Informatika a technikában
16. Busi Lajos	Szövegszerkesztés egyszerűen: WinWord 6.0
17. Király Sándor	A programozás logikája I.
18. Módos Gábor	Programozás Turbo Pascal nyelven
19. Bánhegyesi Zoltán	Kapcsolat a külvilággal: Internet
20. Király Sándor	A programozás logikája II.
21. Miklósi Viktor	Windows '95 kezdő felhasználóknak
22. Busi Lajos	MS Office for Windows '95. I.
23. Tóth Marton László	Multimédia
24. Sándor Miklós	CAD/CAM alapjai
25. Bánhegyesi Zoltán	Adatfeldolgozás Access 2.0 segítségével
26. Busi Lajos	Középiskolai feladatgyűjtemény
27. Bánhegyesi Zoltán	A számítógépes szimuláció alapjai
28. Busi Lajos	MS Office for Windows '95. II.
29. Frank-Faránki-Busi-Bánhegyesi	Informatika I. (1-4. modulok)
30. Nemcsik-Busi-Bánhegyesi	Informatika II. (13. 16. 25. modulok)
31. Király Sándor	Novell Netware 4.1 felhasználói ismeretek