Kalmár László: Belső gépi nyelvek, beleértve a magasszintű programozási nyelveket, (1973), 1-142.

A különböző magasszintű programozási nyelvek elemzése után a tanulmány javaslatot tesz a Kalmár-féle formulavezérlésű számítógép belső nyelvére.

Ezeket a több mint 30 éves gondolatokat érdekes dolog szembesítenünk a mai valósággal, melyek voltak a helyes "előrelátások" és melyek kevésbé helyesek.

2.6. BELSŐ GÉPI NYELVEK, BELEÉRTVE A MAGASSZINTŰ NYELVEKET

- Tanulmány -

Kézirat

Szeged, 1973.

Tartalommutató

Az MTA Természettudományi I. Főosztályának vezetője, Páris György elvtárs 1973. március 1-én kelt, 3064/6 sz. levelében felkérte Kalmár László akadémikust az ESzR távlati fejlesztésével foglalkozó 2. sz. FT munkacsoport első moszkvai értekezletén résztvevő országok képviselői által összeállitott témajavaslat II. pontja /A rendszer architekturája és komponensei/ alatt szereplő 2.6. /Belső gépi nyelvek, beleértve a magasszintü nyelveket/ téma kidolgozására. A tanulmány -- alábbiakban részletezendő -- célját Kalmár László 1973 március 8-án kelt, Páris elvtársnak cimzett levelében rögzitette /ld. a Bevezetést/.

Kalmár László a téma kidolgozásához munkacsoportot alakitott a következő résztvevőkkel:

Gyurkovics Éva /MTA, Matematikai logikai és automataelméleti Tanszéki Kutató Csoport/

Hunya Péter /JATE, Kibernetikai Laboratórium/

Komor Tamás /Infelor/

Makay Árpád /MTA, Matematikai logikai és automata-

elméleti Tanszéki Kutató Csoport;
jelenleg ösztöndijas aspiráns/

Muszka Dániel /JATE, Kibernetikai Laboratórium/

Révész György /SzKI/

Sára Attila /JATE, Kibernetikai Laboratórium/

Simon Endre /MTA, Matematikai logikai és automata-
elméleti Tanszéki Kutató Csoport/

Székely Sándor /JATE, Kibernetikai Laboratórium/

Varga Antal /JATE, Számitástudományi Tanszék/

Varga Tibor /JATE, Kibernetikai Laboratórium/

A tanulmány Kalmár László közvetlen irányitásával készült. A 2. és a 3. fejezetet Makay Árpád és Hunya Péter összefoglaló tanulmányai alapján állitottuk össze, figyelembe véve a munkába bevontak résztanulmányait.

A tanulmány előzetes szerkesztői munkáját Székely Sándor, a végleges alakba való szerkesztést Kalmár László végezte.

Bevezetés

A tanulmány célja, hogy -- elemezve a gépi utasitás-rendszerek és programozási rendszerek fejlődését, valamint a magasabb szintü speciális célu és univerzális programozási nyelvek legfontosabb tartalmi és formai sajáosságait -- javaslatot tegyünk a korszerü formulavezérlésü számitógép belső nyelvként számitásba jövő magas szintü programozási nyelvre.

A fő célkitüzésnek megfelelően a tanulmány 1. fejezete részletesen nyomon követi a gépi utasitás- és programozási rendszerek fejlődését, rámutatva azokra a -- zömmel a felhasználási területekről származó igények hatására kialakult -- hardware és software fejleményekre, amelyek megszabták ennek menetét. Ugy gondoljuk, hogy ennek során sikerült kiemelnünk azokat a tendenciákat, amelyek figyelembe vétele elvezet a tanulmány fent vázolt céljához.

A 2. fejezetben először a gépi nyelvek és a magasabb szintü programozási nyelvek közötti kapcsolatokat vizsgál-

juk meg. /Gépi nyelven itt -- és a továbbiakban is -- a szokásos, nem formulavezérlésü számitógépek belső nyelvét értjük./ Ezt követően a legfontosabb speciális célu és univerzális programozási nyelvek elemzésével részint rendszerezett képet kisérlünk adni a közös, illetőleg eltérő tartalmi és formai elemekről, részint kiemeljük azokat a mozzanatokat, amelyek rámutatnak a formulavezérlésü számi-

tógép szükségességére. A fejezetet a formulavezérlésü számitógépekre vonatkozó elképzelések és realizációk fejlődéstörténetének vázlatos áttekintése zárja be.

A 3. fejezet lényegében egy komplex javaslat /és annak indokolása/ a KALMÁR-féle formulavezérlésü számitógép korszerü alakjának belső nyelvére. A nyelv szintaxisát a Függelék tartalmazza; a szintaxishoz részletesebb kommentárok, a legszükségesebb hardware-realizációs utasitásokat is beleértve, a 3. fejezetben megtalálhatók. Ez a belső nyelv természetesen a realizálás, vagy akár az azt megelőző, már meglevő számitógépen történő szimulálás utján való alkalmazás során szerzendő tapasztalatok, valamint ujabb elgondolások figyelembevételével még módosulhat.

1. A gépi utasitásrendszerek fejlődése

A formulavezérlésü számitógépek eszméjének kialakulását részben megelőzte a gépi utasitásrendszerek és programozási rendszerek, valamint a magasabb szintü programozási nyelvek hosszas fejlődése, részben vele párhuzamosan történt ez a fejlődés. Abból a célból, hogy kellően megvilágithassuk: miért szükséges a belső gépi nyelv szintjének lényeges megemelése, röviden fel kell vázolnunk a megelőző fejlődésben érvényesülő rendkivül változatos tendenciákat, valamint azokat az ellentmondásokat, amelyek főleg a gépek univerzális felhasználhatósága és hatékonysága között merültek fel. Az elemzést a gépi utasitásrendszerek és programozási rendszerek fejlődésével kezdjük.

Kiindulópont gyanánt a tudománytörténeti jelentőségü Neumann-Burks-Goldstine jelentést választjuk /Neumann, 1946/, amely lefektette a mai értelemben vett számitógépek fejlesztésének alapjait és hosszu időre meghatározta fejlődésüket. A jelentés igen hatékony szerepet játszott

a software fejlődés elindulásában is.

1.1. A tárolt program elve

A régebbi berendezéseknél /pl. a lyukkártyás gépeknél/ az információfeldolgozás menetét /programját/ teljesen különállóan kezelték a feldolgozandó adatoktól. A programok a feldolgozás folyamán állandóak maradtak. Ez a körülmény igen nehézkessé tette az algoritmusok gépre vitelét és erősen korlátozta körüket is. Nem volt lehe-

tőség pl. arra, hogy az algoritmusokban kisebb eltérésekkel sokszor ismétlődő részeket a gép -- csekély változtatások eszközlésével -- ugyanazon programrész alapján hajtson végre.

A tárolt program elvének alkalmazása megoldotta ezt a problémát, mivel ugyanolyan müveletvégzési lehetőségeket biztositott a program elemi részein, mint a feldolgozandó adatokon. A tárolt program segitségével tul lehetett lépni az un. direkt programozáson és az utasitásmódositás-nélküli ciklikus müveletvégrehajtáson. Az utasitások módosithatósága azután lehetővé tette bonyolultabb ciklusok leirását is. Lényegesen tágabbá váltak a lehetőségek, mivel az utasitásoknak mostmár nemcsak a cim-, hanem a müveleti részét is változtatni lehetett. Az utasitásokon végezhető müveletek /a gyorsan fejlődő automatikus programozási eszközökkel együtt/, lehetővé tették a programozás megkönnyitését, majd pedig a feladatok és a rendszer kezelésének automatizálását is /az operációs rendszerekkel/.

1.2. A gépi utasitások formai fejlődése

Az egy-akkumulátoros szóorientált gépek utasitásrendszere fix hosszuságu, tároló-referenciás utasitásokból állt. Az utasitásokban megadták a müvelet kódját, és egy, két, vagy három cimet.

Az egy-cimes utasitásokban a müvelet egyik operanduszát az akkumulátor-regiszter tartalma, a másikat az /esetleg módositott/ cim tartalma szolgáltatja. Ha a mü-

velet eredményét az akkumulátoron kivül a memóriában is tárolni kellett, ezt, vagy operandusznak az akkumulátorba töltését is, rendszerint külön utasitással lehetett elérni.

A két-cimes utasitásrendszerekben a müveletet vagy az akkumulátor és az egyik cim tartalma, vagy a két cim tartalma között lehetett elvégezni. A müvelet eredménye az utasitásban megadott valamelyik cimre, vagy az akkumulátorba volt beirható.

Az 1+1 cimes utasitások végrehajtása ugyanugy történt, mint az egycimeseké, a második cimmező a soron következő utasitás cimének megadására szolgált. A szekvenciális végrehajtástól való eltérést az operativ dobmemória használata miatt, a következő utasitás végrehajtásához szükséges elérési idő csökkenése végett volt szükség.

A három-cimes utasitás-rendszerekben a müveletet vagy az akkumulátor és az egyik cim tartalma, vagy két cim tartalma között lehetett elvégezni. A müvelet eredménye az akkumulátoron kivül az egyik cimre is bekerülhetett.

A 2+1 cimes utasitások végrehajtása a két-cimeseké-

hez hasonló; a harmadik cimmező a soronkövetkező utasitás cimét adta meg. Céljuk hasonló volt az 1+1 cimes utasitásokéhoz.

A korszerübb, harmadik generációs gépeknél az utasitásoknak mind formai, mind tartalmi szempontból rugalmasabbakká kellett válniok. Emiatt az egy-egy gép utasitás-

rendszerének jellemzésére használatos régebbi osztályozás -- főleg merevsége miatt -- már egyre kevésbé felelt meg. Figyelembe kellett venni ugyanis egyrészt, hogy az akkumulátorok, más néven regiszterek, megtöbbszöröződtek, szerepük merevsége is feloldódott /pl. ugyanaz az akkumulátor, a müveleti kódtól függően, egyszer operandusz, máskor index-regiszterként került felhasználásra/, másrészt, hogy a mező- majd byte-szervezésü gépeknél az utasitások /hasonlóan az adatokhoz/ változó hosszuságuakká váltak; a hosszt rendszerint a müveleti kódhoz kapcsolt néhány bit segitségével adták meg. Az ujabb osztályozásban a leggyakrabban használatos utasitástipusok a következők:

RR tipus: két regiszter tartalma közti müvelet előirása; az eredmény rendszerint az egyik regiszterben helyeződik el /un. regiszter-referenciás utasitás/;

RM tipus: egy regiszter vagy egy regiszter-csoport és egy főtároló-cim tartalma /szó, byte, byte-csoport/ közt végez müveletet; az eredményt, amennyiben egy regiszter és egy főtárolócim tartal-

ma közötti müveletről van szó, a regiszterben helyezi el;

RI tipus: egy un. I-mező közvetlen operanduszt tartalmaz az utasitás részeként; az utasitás ezzel, valamint egy regiszter tartalmával végez müveletet, az eredményt rendszerint ugyanabban a regiszter-

ben helyezi el;

MI tipus: egy közvetlen operandusz és egy főtároló-cim tartalma /szó, byte, byte-csoport/ közt végez müveletet; az eredményt rendszerint a főtároló-cimre helyezi el;

MM tipus: két főtároló-cim tartalma közt végez müveletet; az eredmény rendszerint az egyik operandusz helyére kerül.

Ennél az osztályozásnál elvileg nem lehetséges tároló-referenciás és regiszter-referenciás utasitásokat megkülönöbztetni; az utasitásokban egyes mezők, a müveleti kódtól függően, tartalmazhatnak tároló-referenciás cimeket, regiszter-referenciás bitcsoportokat vagy közvetlen operanduszokat. Az utasitásokban szereplő főtároló-cim/ek/ jelölése mellett a változó utasitás-hosszuságu /pl. byte-szervezésü/ gépeken szükség van az operandusz/ok/ hosszának jelölésére is. Ez történhet implicit módon, de szükség esetén lehetséges az operanduszok hosszának explicit jelölése is. Ilyenkor az utasitásban külön mező /az un. L-mező/ szolgál a hossz-specifikáció számára, esetleg minden főtároló-cimhez külön-külön is.

Vannak olyan utasitások, amelyek operanduszaikat speciális regiszterek tartalma által kijelölt főtároló helyekről vagy a müveleti kód és esetleg külön e célra szolgáló mező által specifikált regiszterekből veszik. Ilyenkor az utasitásban sem a főtároló-cim, sem a regiszter jelölésére nincs szükség, ezért ezeket cim-nélküli utasitásoknak ne-

vezzük. Az ilyen utasitások tehát kizárólag a müveleti kódból és esetleg azt kiegészitő mezőből állnak. A cim-nélküli utasitások jelentősége az ujabb főtároló-tipusok megjelenésével növekedett. Példaképpen megemlitjük az un. verem-memória /last-in-first-out, LIFO/ hatását az utasitások formájára. A verembe töltő és a verem "tetején" lévő /utoljára betöltött/ szót törlő, vagy onnan valamely regiszterbe áttöltő utasitásokon kivüli valamenynyi tényleges müveletet a verem tetején levő egy vagy két szón végezzük el, a müvelet eredménye pedig valamelyik operandusz helyére kerül.

1.3. A gépi utasitások tartalmi fejlődése

Az utasitásokat, az általuk végrehajtott tevékenység alapján a következő három fő csoportra oszthatjuk:

- információ-mozgató,
- transzformációs és
- vezérlés-átadó utasitások.

Az információ-mozgató utasitások regiszterek vagy főtároló-cimek tartalmát helyezik át változtatás nélkül, esetleg előjelváltással, vagy abszolutérték képzésével,

más regiszterekbe vagy főtároló-cimekre. Általában az utasitás végrehajtása után az áthelyezett információ rendszerint két azonos példányban található meg a számitógépben, de előfordulhat két információ felcserélése is. Az információ-mozgató utasitások legtöbbször RR-, RN-, vagy MM-tipusuak. Az input/output utasitások elvileg szintén

ide tartoznak, hiszen a főtároló és a perifériák között mozgatják az információt; ezeket azonban -- speciális tulajdonságaik miatt -- külön tárgyaljuk.

Az információ-mozgató utasitások legrégebben alkalmazott reprezentánsai a betöltő /LOAD/ és tároló /STORE/ utasitások. Az időosztásos rendszerek kialakulása általánosan szükségessé tette a csoportos betöltő és tároló utasitások kifejlesztését -- amelyek már egyes régebbi gépeken is megvoltak, csoportos müveleti utasitásokkal együtt -- mivel valamely tárolt programról egy másikra való áttéréskor meg kell őrizni bizonyos regiszterek tartalmát, hogy a későbbi visszatéréskor helyreállithatók legyenek. A csoportos utasitások használata megkiméli a programozót a megfelelő ciklusok irásától is.

Ugyancsak az időosztásos rendszerek szükségletei indokolják a hosszabb főtároló-területek tartalmának áthelyezésére szolgáló utasitások /MOVE/ kialakitását is. Például a bázisregiszteres cimzés /ld. később/ alkalmazása esetén ahhoz, hogy a szabadon maradt, illetőleg szabaddá vált főtároló-területek összefüggő egységgé összekapcsolhatók legyenek, teljes program- vagy adatterületek áthelyezésére van szükség. Ugyancsak jól használhatók a MOVE-jellegü utasitások a buffer-memóriák és

a programterületek közti információ-cserére is.

A transzformációs utasitások /ezek regiszterek vagy főtároló-cimek tartalma közt aritmetikai vagy más müveleteket végeznek/ képezik a gépi utasitásrendszerek legna-

gyobb részét. Hagyományosan ebbe a csoportba sorolják az aritmetikai és logikai utasitásokat, valamint a léptető /SHIFT/ utasitásokat. A gépi utasitásrendszerek tartalmi fejlődése elsősorban a transzformációs utasitások körének bővülésén mérhető le.

Az első számitógépek csupán fixpontos, általában bináris /néha binárisan kódolt decimális/ aritmetikával rendelkeztek /az oktális és a hexadecimális ábrázolásmódot a binárishoz számitjuk, hiszen attól csak a hármas vagy négyes bitcsoportok jelölésében tér el/. Az előjeles számábrázolás direkt, kiegészitő vagy forditott kódrendszerü volt. A lebegőpontos aritmetikai müveleteket eleinte software-interpreterek, majd csapdázott utasitások végezték. Csak később került sor a lebegőpontos müveletek hardware- illetve mikroprogram-realizációjára.

A számitógépes adatfeldolgozás térhóditása jelentősen befolyásolta a gépi utasitásrendszerek fejlődését. Bekerültek a standard utasitáskészletbe a decimális aritmetikai, a konverziós és a szerkesztő utasitások.

A decimális számok és általában a változó szóhosszuságot kivánó adatok kezelésének alapvetően két vál-

zatát alkalmazták. Az egyikben az utasitások a szó kezdő, vagy végző byte-jét cimezik meg és az utasitás L-mezeje explicit módon adja meg a számjegyek, karakterek, vagy byte-ok számát. A másikban az utasitások nem tartalmaznak hossz-specifikációt, hanem a szám, általában a

változó hosszuságu szó végét egy speciális végjel-karaktert vagy jelzőbitet tartalmazó byte jelöli. /Harmadik változatként tekinthető a software uton megvalósitott többszavas aritmetika, illetőleg nem-aritmetikai adatfeldolgozás; ezzel azért nem foglalkozunk, mert általában nem járt a gépi utasitásrendszer fejlődésével./ Mindkét változatban byte-onként egy karakter, esetleg egy vagy két decimális számjegy vagy előjel tárolható és numerikus alkalmazások esetén általában a programozóra hárul a tizedespont helyének megjegyzése.

Minthogy most már egy gép utasitásrendszere fixpontos, lebegőpontos és decimális aritmetikai utasitásokat is tartalmaz, tehát konverzió nemcsak input/output alkalmával fordul elő /amikor régebben software uton végezték/, ezért szükség van a számrendszerek közti konverziós utasitásokra. Igen gyakran beépitették a gépbe a fixpontos és a decimális számok közti mindkét irányu konverziót, mig a fixpontos és lebegőpontos számok közti konverzióra csak segitő utasitások vannak /normalizálás, egyenlő kitevőkre történő denormalizálás, léptetés, kitevő-müveletek/. Ezekhez járulnak még a külső ábrázolási formához alkalmazkodó szerkesztő utasitások, illetőleg a külön-

bözö kódrendszerek közti konverziós utasitások.

A logikai utasitások regiszterek, illetve főtárolócimek tartalma között bitenkénti logikai müveleteket végeznek. Ezeknek csupán mennyiségi jellegü fejlődése figyelhető meg, amennyiben mind több és több itéletkalkulus-

beli müveletet végző utasitás kerül az utasitásraendszerekbe, bár elvileg elképhelhető és nincs kizárva, hogy a jövőben szerepet kap a többértékü itéletkalkulus is, amikor egy-egy logikai értéket több szomszédos bit ábrázol. /Előfordul, hogy a "logikai" utasitások közé sorolják az összes byte- vagy byte-sorozat-müveletet, tehát pl. a MOVE-jellegü, a szerkesztő stb. utasitásokat is, valamint az olyan /"logikai"/ léptető utasitásokat is, amelyekben az egyébként előjelbitként használt bit és a többiekkel azonos szereppel vesz részt a léptetésben, mig az "aritmetikai" léptető utasitások esetén az előjelbit változatlan marad, esetleg jobbra léptetéskor baloldalt minden lépésben megismétlődik /az abszolut érték "magával huzza" az előjelbitet.//

Itt jegyezzük meg, hogy egyes utasitások több regisztert összekapcsolva is használhatnak akkumulátorként. Gyakori például, hogy lebegőpontos akkumulátorként két egymás utáninak sorszámozott fixpontos regiszter szerepel. Bár ez csökkentheti a hardware-költségeket, viszont azzal a hátránnyal jár, hogy a programozónak körültekintőbben kell kezelnie a regisztereket.

A vezérlés-átadó utasitások közé a feltétlen és felté-

teles ugró utasitások tartoznak. Ujabban a feltétlen ugró utasitást /ez pótolható egy olyan feltételes ugró utasitással, amelynek a feltétele biztosan teljesül/ az un. szubrutin-hivó utasitással helyettesitik: ez amellett, hogy egy adott cimre adja át a vezérlést, valamely regiszterben vagy főtároló-cimen elhelyezi a felhivás helyének cimét is. En-

nek birtokában a szubrutin egyszerüen kezelheti a felhivás után elhelyezett paramétereket, illetve adhatja visza a vezérlést a főprogramnak.

Korábban a vezérlésátadás feltételeként az /egyetlen/ akkumulátor tartalmának előjele vagy 0-val való egyenlősége, esetleg az előző müvelet eredményének tulcsordulása volt használható. Ma inkább azt az utat járják, hogy minden utasitás végrehajtásához bizonyos állapot-bitek értékének beállitása /esetleg változatlanul hagyása, vagy ellenkezőjére forditása/ is hozzátartozik. Ilyen állapotbitek lehetnek a 0-jel, az előjel, a tulcsordulás-jel, a kerekités-jel, stb. Ekkor egyetlen feltételes vezérlésátadó utasitás van, amelynek un. "maszk-bitjei" /legtöbbször közvetlen operanduszként az utasitásba beirva/ egyenként vagy kombinálva jelölik a feltételt. Bizonyos sorrendben a hardware összehasonlitja az állapot-biteket a maszk megfelelő bitjeivel, és ha valahol egyezést talált /esetleg ugy, hogy a megegyező bitek értéke 1/, akkor a feltételt teljesültnek tekinti, különben nem. Ez a rendszer egyszerüsége és könnyü mikroprogramozásos megvalósithatósága mellett azt az előnyt is nyujtja, hogy a maszk-bitek kombinációival a számunkra szükséges feltételt egy utasitáson belül előirhatjuk, mig korábban ehhez vagy az uta-

sitásrendszerben sok különböző feltételes utasitásra, vagy több egymás utáni utasitásra volt szükség.

A feltételes vezérlésátadó utasitások egy másik csoportja a ciklikus programozást hivatott segiteni. Az un. ciklus-vége utasitások növelnek /vagy gyakrabban csökken-

tenek/ egy számlálót /ez általában olyan regiszter, amely az utasitásokban index-regiszterként is használható/, ellenőrzik azt, hogy a számláló elért-e egy másik regiszterben vagy főtároló-cimen tárolt végértéket /illetőleg azt, hogy 0-e/ és ennek megfelelően vagy az utasitásban adott cimre, vagy a következő utasitásra adják át a vezérlést. A ciklus-vége utasitások leggyakrabban RM-tipusuak.

Lényegében a SUPERVISOR-hivó utasitások is a feltétlen vezérlésátadó utasitások közé tartoznak: hatásukra a program végrehajtása megszakad: tulajdonképpen egy szubrutin-hivó utasitás hajtódik végre, amely elinditja a SUPERVISOR egy szubrutinját. A harmadik generációs számitógépeken ez azzal is együttjár, hogy az un. program-állapotból egy un. SUPERVISOR állapotba kerül a gép. Ebben olyan /un. privilégizált/ utasitások is használhatók, amelyek a programozó számára egyébként tiltottak. Ilyen megszakitó és SUPERVISOR-hivó utasitások általában az input/output-tal kapcsolatos tevékenységeket végző utasitások /a program végét jelző, vagy a programhibát jelző utasitásokat nem kell külön emliteni, hiszen a program végén az eredmények kivitele, a hibaelemzés végén a hibalista kinyomtatása kö-

vetkezik/, de aktivizálhatunk igy állandóan a memóriában /esetleg a könytárként használt külső tárolóban/ levő közhasznu szubrutinokat is, amivel /felhasználói szempontból/ a gép utasitásrendszerét tudjuk bőviteni.

1.4. Cimzési rendszerek és eljárások

A programozási módszerekben elért haladás mind nyomatékosabban vetette fel a memória-cimzési eljárások fejlesztésének igényét, amit tovább sürgetett a programozási munka megkönnyitésében megtett következő lépés: a szubrutinok használata. Ezek a programrészletek több helyen felhasználhatók a programokban, vagy ugy, hogy minden felhasználás helyén többé-kevésbé változatlan formában beiktatják őket /nyitott szubrutinok/, vagy ugy, hogy csupán egy helyen kerülnek beépitésre, de vezérlés-átadással olymódon aktivizálhatók, hogy -- a paraméterek átadásának és átvételének valamely módját alkalmazva -- végrehajtás után a vezérlést visszaadják az őket aktivizáló programnak /zárt szubrutinok/.

A fentiek alapján jól látható a cimek kezelésének fontossága, hiszen mind a ciklikus programozásnál, mind a szubrutinok alkalmazásánál egyes kijelölt tevékenységek azonosak maradnak a különböző végrehajtások során. Ez azt jelenti, hogy az utasitáskódok nem változnak, de az operanduszok, tehát a nekik megfelelő cimrészek, rendre mások és mások lesznek, ami kényelmesen megvalósit-

ható az indexeléssel. Az indexelésnek az a lényege, hogy az utasitásban megadott /un. névleges vagy relativ/ cimet egy ugyancsak az utasitásban specifikált indexregiszter tartalmával /amit a relativ cimzés bázisának, vagy

röviden báziscimnek is szokás nevezni/ módositják /amikor kell/, és ezt a módositott /un. tényleges vagy abszolut/ cimet használják fel a müvelet elvégzésekor. Az indexregiszterek száma gépenként jelentősen eltér; esetleg egy névleges cimet több indexregiszter tartalmával is lehet módositani. /Az indexregiszter elnevezés akkor használatos, ha rendszerint valamely tömbelem indexét helyezzük el benne; más esetben inkább a bázisregiszter elnevezés a szokásos./

A hardware által indexeléssel elvégzett müvelet természetesen indexelési lehetőségekkel nem rendelkező gépeken is végrehajtható /programozási eszközök segitségével/, mivel a tárolt program lehetővé teszi bármely adott utasitás módositását. Nyilvánvaló, hogy a softwaremegoldás nehézkessége és lassusága volt a tevékenység hardware-esitésének kiváltó oka. Az indexregiszterek alkalmazása nagy előrelépést jelentett mind a ciklikus programozásban, mind a szubrutinkezelésben. Többek között megoldotta a tömbök /vektorok, mátrixok stb./ kezelésével kapcsolatos ciklikus programozási problémákat. A software további fejlődése azonban olyan ujabb igényeket támasztott, amelyek már tulmutattak az indexelési technikán.

Indexelésre felhasználhatók az indirekt cimzési eljárások is, amelyekkel részletesebben foglalkozunk.

Az indirekt cimzési eljárások kialakulásának az

volt az oka, hogy az egyre bővülő kapacitásu operativ tároló direkt cimzése az utasitásban szereplő cimzéssel mind inkább lehetetlenné vált. Az indirekt cimzési eljárásokban szereplő cim tartalma nem az operanduszt, hanem annak cimét szolgáltatja, és mivel a szavak hossza -- sokszor lényegesen -- meghaladja a cimrészek hosszát, igy jóval nagyobb fizikai cimtartomány vált átfoghatóvá. Az indirekt cimzés többszintü is lehet /azaz a cimzési eljárásban szereplő cim tartalmaként kapott cim ujabb tartalma ujabb cimet ad meg stb., mig végül többszörös iterációval kapjuk meg magát az operanduszt/.

Az indexregiszteres és indirekt cimzés egy utasitáson belüli használatához a gépi utasitásokban egy ujabb, un. cimzés mód-mezőt kellett kialakitani, amelyben a cimzési mód irható le. Ilyen cimzési módok lehetnek:

- direkt operandusz használata /a "cimmezőben" nem cim áll, hanem maga az operandusz/,
- direkt cimzés /a cimmezőben a tényleges cim szerepel/,
- direkt relativ cimzés /amit fent indexelés néven irtunk le; indexregiszterként az utasitásszámláló, vagy egy akkumulátor is szerepelhet, az előbbi esetben önrelativ cimzésről beszélünk/.

- indirekt cimzés /egy vagy több szintü/,
- indirekt relativ cimzés /ilyenkor elő- és/vagy utó-indexelés is lehetséges, azaz indexelés az

indirekt cimzés előtt és/vagy utána/.

- akkumulátoron át történő cimzés /az abszolut cimet egy vagy több előző utasitás egy akkumulátor-regiszterben alakitja ki és ennek tartalma adja az operandusz-cimet/.

Az indirekt cimzésnek ma, különösen rövid szóhosszu gépeknél van jelentősége, mivel igy nagyobb főtárolókapacitás esetén is külön segédregiszterek /pl. bázisregiszterek/ nélkül tudunk hivatkozni bármely főtárolócimre. Ilyen eljárás például az "indirekt lapcimzés". A lapcimzés lényege az, hogy a főtárolót egyenlő számu szót tartalmazó részekre, un. lapokra osztjuk. A tényleges cim magasabb helyértékü bitjei azon lap sorszámát /"lapcimet"/ határozzák meg, amelyen az operanduszt tartalmazó rekesz található, alacsonyabb helyértékü bitjei pedig e rekesz sorszámát a lapon belül /a "lapon belüli cimet"/. Az indirekt lapcimzésnél direkt módon, a cimzési mód-mező tartalmától függően, csak a memóriatartomány legelejére /a "0-dik lapra"/, vagy az utasitás cimének a környezetére, /amely esetben a lapcim az utasitás lapcime/ vagy pedig az előzőleg végrehajtott utasitásban használt operandusz-cim környezetére /amely esetben a lapcim ugyanaz, mint az előzőleg végrehajtott utasitás

esetén volt,/ hivatkozhatunk, de indirekt módon, pl. a "O-dik lap" valamely rekeszén keresztül a teljes operativ memória bármelyik cimét elérhetjük. Ebben az esetben a "O-dik lap" kérdéses rekesze a bázisregiszter szerepét játssza, amelynek a tartalmát utóindexeléssel még módo-

sithatjuk.

A szubrutinkönyvtárak kialakulása és az automatikus programozási rendszerek fejlődése, valamint az időosztás elterjedése megkövetelte, hogy egy adott program ne legyen szigoruan a memória valamely helyéhez kötve, hanem különböző helyeken lehessen felhasználni. Ez a lehetőség a forrásprogramok szintjén lényegében biztositva van, hiszen a memória-hozzárendelés rögzitését csak a forditóprogramok végzik el. A programok irásakor tehát /eltekintve a gépi nyelven, abszolut cimekkel irt programoktól/, biztositva van a változók, cimkék /utasitások/ helyének logikai kezelése, azaz ekkor még nincsenek hozzárendelve valamely memóriarekeszhez. Természetesen e programok végrehajtásakor már meg kell lennie az emlitett hozzárendelésnek, mivel a müveletvégzés fizikai elemeken /azok tartalmán/ történik. A feldolgozás folyamán tehát valamikor a logikai cimeket le kell képezni a fizikai cimekre.

Az első forditóprogramoknál ez a leképezés a forditás elvégzésekor történt meg, s az eredmény abszolut cimes gépi kódu program volt. Természetesen a forditóprogramoknak a szubrutin-könyvtárban szereplő programokat relativ /tehát logikai/ cimes programokként kellett kezelniök, mivel azok más-más főprogramban más-más hely-

re kerültek /bár egyes rendszerekben bizonyos szabványos szubrutinok a központi tároló előre kijelölt részein helyezkednek el/.

A következő fejlődési fok az volt, hogy a fordítást két lépésben végezték el. Az első lépésben egy, a gépi nyelvhez közelálló nyelvü, de még logikai cimeket használó tárgyprogram az eredmény, melyet a háttér-memóriában /az un. modulkönyvtárban/ helyeznek el. Ez lehetővé teszi a kész tárgyprogramok beépitését más programokba. A második lépésben valósul meg a modulok összefüzése /linking/, valamint betöltése a főtárolóba /loading/, és végbemegy a fizikai cimekre való leképezés. Ennél a technikánál tehát a fordítás első lépésében még nem szükséges rögziteni az elhelyezést, mert erre a betöltési idő során van lehetőség.

A régebbi fordítási módokra jellemző, hogy a fizikai cimek egy végrehajtás előtti fázisban kerülnek rögzitésre, az ujabb módszerek esetén azonban csak közvetlenül a végrehajtás előtt. Ennek az az előnye, hogy a lefordított programmodulok a memória különböző helyein is felhasználhatók, áthelyezhetők /relocation/, azonban a fentiekben a vázolt módszernél csak statikusan /static relocation/. Ez a mamória-hozzárendelési forma elsősorban az időosztás nélküli rendszereket jellemzi, bár segitségével időosztásos rendszerek is készithetők, ám csekély praktikus értékkel.

Az összefüzés és a betöltés idején /software-techni-

kával/ történő memóriahozzárendelés előrelépést jelentett. Segitségével megoldhatóvá vált a programok szegmentálása. Ez utóbbi esetben a tárgyprogram különböző modulokból épül fel, s ezek mindegyike önálló cimtartománnyal rendelkezik, ami — a forrásprogram szintjén -- kétdimenziós logikai

cimzési rendszert eredményez.

Mig a fentebb vázolt előrelépés software jellegü volt és a fordítóprogramokkal, valamint a tárgyprogramok célszerü kezelésével kapcsolatban vetődött fel, addig a dinamikus áthelyezés /dynamic relocation/, amelyet az időosztásos kezelőrendszerek megjelenése helyezett előtérbe, /nagyrészt/ hardware eszközök segitségével volt megoldható. A különböző hardware eszközök egyuttal különböző szinteket is képviselnek, ahol a magasabb szinteken általában felhasználják az alacsonyabbakat.

Megkülönböztetünk:

- báziscimzéses,
- lapcimzéses és
- szegmentum-cimzéses rendszereket.

Báziscimzéses rendszer alkalmazása esetén az összeállitott program lényegében egy statikus betöltési eljáráson megy keresztül, de végrehajtásakor az összes tároló-referenciás cimekhez hozzaadódik a bázisregiszter tartalma, és az igy kapott cim kerül felhasználásra /miközben természetesen emellett más indexelési lehetőségek is megmaradnak/. Igy a program áthelyezése végrehajtás közben /dinamikusan/ is megtörténhet, mivel ez nem von maga után más változtatást, mint a bázisregiszter tartalmának megváltoztatását. Ezzel a technikával a program számára

az operativ tároló tetszőleges szabad helyén folytonos fizikai cimtartomány jelölhető ki, melyen a program egyet-

len egységként helyezkedik el. Egyes rendszerekben egy-egy program számára több bázisregisztert is biztositanak, ekkor pl. nemcsak a program, hanem az adatok számára is külön folytonos területek jelölhetők ki. A futtatásra kész program cimei ilymódon logikai cimekké alakulnak át, a fizikai cimekre való leképezés csak a végrehajtás során megy végbe.

A báziscimzéses rendszer alkalmazása azonban az emlitett előnyök mellett néhány hátránnyal is jár. A feldolgozások során több felhasználatlan memóriaterület képződik, mivel nem várható, hogy a felszabaduló összefüggő helyek teljesen kitölthetők lesznek ujonnan belépő programrészekkel. Amennyiben ezeket is fel akarjuk használni, állandóan gondoskodni kell a programok fizikai áthelyezéséről. A logikai cimtartomány terjedelme is korlátozott: nem lehet nagyobb a fizikai cimtartományánál, mivel a logikai cimtartománynak a központi tároló folytonos szelete felel meg /Watson, 1970/. Ezek a fogyatékosságok természetesen csak a további igények szempontjából tekinthetők azoknak, egyébként a báziscimzéses rendszer alkalmazása jelentős előrelépés volt.

A báziscimzéses rendszer alkalmazásával lehetőség nyilik olyan eljárás-programok, rutinok készitésére, amelyeket szimultán használhatnak különböző programok. A

rollaszrajas -- amennyiben a rutinok megfelelő feltételeket teljesítenek -- meglehetősen bonyolult lehet, pl. mielőtt valamely rutin az egyik program számára befejezné a tevékenységét, valamilyen megszakítás folytán egy

másik program is beléphet ugyanazon rutinba /pure procedure, reentrant code /Barron, 1971/, reentrant program /Watson, 1970//.

A lapcimzéses /paging/ rendszereknél, /mint már említettük,/ a memória fizikai lapokra van osztva, a programok pedig ezekkel azonos méretü logikai lapokra hivatkoznak. A programban szereplő cim lényegében két részből áll /a lapcimből és a lapon belüli cimből/, ami azonban a felhasználó számára egyetlen egységet képez. A végrehajtás során a logikai cimben szereplő logikai lapcimhez ugynevezett laptábla segitségével történik meg a fizikai lap /blokk/ hozzárendelése, majd ezen belül a megjelölt cimü tárolórekesz kiválasztása. A laptábla egy programra vonatkozóan a logikai lapcimeknek megfelelő blokkok sorszámait /a "fizikai lapcimeket"/ tartalmazza. Kezdetben egyetlen laptáblán kellett osztozkodniok a különböző programoknak, később általánossá vált külön laptáblák hozzárendelése az egyes programokhoz, ahol a laptábla kiválasztása egy laptábla-bázisregiszter segitségével történik meg.

A lapcimzés alkalmazásával lehetővé vált a szabad területek logikai összefüzése /a laptáblák megfelelő módon való kitöltésével/, miközben a programokat nem kellett menetközben fizikailag áthelyezni. A lapcimzés bizonyos kiegészitéssel lehetővé tette, hogy a programok csak

azt a memóriaterületet foglalják le, amelyet egy adott időben ténylegesen használnak, és ezzel együtt biztosi-

totta a logikai cimtartomány kiterjesztését a fizikai cimtartományt meghaladó mértékben /un. virtuális memóriakezelés/. Ez a fogás leegyszerüsitette, sőt esetenként megszüntette az átfedési /overlay/ technikával kapcsolatos nehézségeket. Az emlitett technikával ugyanis ugy oldódik meg a virtuális memóriakezelés, hogy a program különböző részei felváltva helyezkednek el ugyanazon operativ memóriaterületen. Nincs azonban lehetőség arra, hogy közvetlenül hivatkozzunk az operativ tárolóban nem tartózkodó részekre, és csak software eszközök segitségével kezdeményezhető és valósitható meg a programrészek cseréje /roll-in-roll-out/. Az un. demand paging rendszerekben a laptábla egy jelzéssel egészül ki, amely azt mutatja, hogy az adott lap az operativ tárolóban tartózkodik-e vagy sem. Amennyiben a hozzáforduláskor a hivatkozott lap nincs az operativ tárolóban, bekerül egy éppen nem használt lap helyére, miközben ez a tevékenység, illetve ennek eredménye, átvezetődik a laptáblákon.

A báziscimzéses rendszerrel kapcsolatban felvetett két probléma megoldása során a forrásnyelvi szinten kétdimenziós cimzésnek a betöltés után a lapcimzéses rendszernél is egydimenziós logikai cimzés felel meg. Ez azonban hátrányos abban az esetben, ha az összeállitott program egyes részei a végrehajtás során változó terjedelmüek, ilyenkor ugyanis számukra a maximális helyet

kell biztositani az egydimenziós cimskálán.

Ezt a hátrányt megszünteti, továbbá kedvezőbb lehetőségeket biztosit e jelenleg legfejlettebb cimzési eljárás, a szegmentum-cimzés /Watson, 1970/. Ilyenkor a programot szegmentumokra bonthatjuk, és minden szegmentumban elölről kezdhetjük a lapszámozást. Egy logikai cim alakja ilyenkor a következő: szegmentumszám, szegmentumon belüli logikai lapcim, lapon belüli /relativ/ cim. Minden programnak saját szegmentumtáblája, és minden szegmentumnak saját laptáblája van. Az éppen futó program szegmentumtáblájának a kezdőcimét a szegmentumtábla-bázisregiszter tartalmazza. Az átcimzési folyamat a következő: először az indexelés zajlik le, amely azonban nem érinti a szegmentumszámot. Ezután a szegmentumtábla-bázisregiszter tartalmához hozzáadódik a szegmentumszám, az igy kapott szegmentumtábla-cimről kiolvasásra kerül egy laptábla báziscime. Az ebből az indexelt logikai lapcim alapján kiolvasható fizikai lapcimmel konkatenálódik az indexelt lapon belüli cim, és ez adja végül az abszolut fizikai cimet, amelyhez a hozzáfordulás történik.

Mind a lapcimzés, mind a szegmentumcimzés lényegesen összetettebb a korábbiaknál, s ez azzal a következménnyel jár, hogy lelassul a cimkiszámitás. Ha ugyanis a szegmentumtáblák és a laptáblák az operativ memóriában helyezkednek el, akkor pl. egyetlen operandusz

kiolvasásához háromszor kell a memóriához fordulni. E-

zért ajánlatos lenne olyan asszociativ /tartalom-cimzésü/ táblát alkalmazni, amelynek egyik oszlopában a szegmentumszámok és a logikai lapcimek, másik oszlopában a nekik megfelelő fizikai lapszámok vannak.

Ilyenkor a cimzésnél egy szegmentumszám-bázisregiszter tartalmához adódik hozzá a /logikai, relativ/ szegmentumszám, kialakitva ezáltal egy abszolut szegmentumszámot, mig az abszolut logikai lapcim indexeléssel alakul ki. Ha igy kapott szegmentumszám-logikai lapcim kombináció megtalálható a tartalom-cimzésü tábla első oszlopában, a hozzátartozó sorból kiolvasásra kerül a fizikai lapcim, amellyel konkatenálódik az indexelt lapon belüli cim. Mivel az asszociativ tábla viszonylag gyors lehet, ez a cimzési eljárás nem növeli lényegesen az operandusz hozzáférési idejét. Minthogy pedig a szegmentumok cimtartománya külön-külön változtatható, ez a rendszer amellett, hogy megőrzi a lapcimzés minden előnyét, lehetőséget biztosit a programrészek, vagy adatstrukturák terjedelmének dinamikus bővitésére, vagy szükitésére.

Igen rugalmas tárolórendszert kapnánk abban az esetben, ha az asszociativ tárolókat gyors és nagykapacitásu főtárolóként is sikerülne alkalmazni. A közeljövőben azonban még nincs komolyabb kilátás asszociativ főtárolókra.

A cimzési rendszerek fejlődése is amellett tanuskodik, hogy a software igények jobb kielégitésére az

ujabb hardware-megoldások, a software-ben addig is többé-kevésbé meglévő eljárások "hardware-esitése" utján jöttek létre.

1.5. Az I/O utasitások és tevékenységek fejlődése

A számitógépes információfeldolgozás leglassubb eszközei kezdettől fogva az input/output /I/O/ berendezések. A központi egységek fejlődése során az azokhoz viszonyitott sebességi arány még tovább romlott. Tekintettel arra, hogy a feldolgozások nagy hányadánál jelentős részt képeznek az I/O tevékenységek, a korábbi I/O kezelés javitására uj elveket kellett kidolgozni.

A kezdeti időszakot a perifériák közvetlen, egyedi kezelése jellemezte. Az utasitások elemi, vagy perifférián lévő inputanyag által meghatározott, pl. tömbbeviteli tevékenységeket inditottak el, miközben a központi processzornak várakoznia kellett. Nyilvánvaló, hogy a központi egység müködésének meggyorsulásával egyre nagyobb probléma lett ez a várakozás.

A megoldásra irányuló erőfeszitések egyik eredménye az I/O csatornák bevezetése volt. Az I/O csatorna egy hardware eszköz, melynek segitségével a tevékenység elinditása után a központi processzortól függetlenül oldható meg az I/O eszköz/ök/ vezérlése. Megkülönböztetünk programozott és autonom /multiplexor vagy szelektor/ csatornákat. Az előzők esetében csak elemi I/O tevékenységek végezhetők párhuzamosan és autonóm módon.

ez utóbbiak esetében viszont összetettebb tevékenységet leiró /csatorna-/ programok is végrehajthatók a központi számitási tevékenységgel párhuzamosan.

A fejlődés ezen a területen olyan utasitások kidolgozásával indult el, melyek képesek elemi átviteli tevékenységek elinditására, majd egy későbbi időpontban a csatorna lekérdezésére annak megállapitása végett, hogy befejeződött-e már az átvitel. Ezzel elvileg lehetővé vált a feldolgozási folyamat olyan szervezése, amelyben az I/O tevékenység teljes egészében átlapoltan folyik a központi processzor munkájával. A bevezetett ujitások azonban a gyakorlatban lényegesen bonyolultabbá tették az I/O tevékenység programozását, mivel a programozónak figyelmet kellett forditania a pufferezésre, vagy egyes területek blokkolására, feloldására és az autonóm csatornákkal való szinkronizálásra. Ez ujabb nehézségeket támasztott, amiket I/O rendszernek /Input-Output Control System: IOCS/ nevezett szubrutincsomagokkal igyekeztek megoldani.

Az I/O tevékenységek hatékonyabb kezelésére kialakult másik irányzat: a lassu I/O müveletek off-line elvégzése. Ennél valamilyen /hardware/ eszköz felhasználásával valósitják meg az átvitelt a lassu müködésü periféria és mágnesszalag /később mágnestárcsa/ között, függetlenül a központi egység munkájától. A programok végrehajtása során a hozzájuk tartozó I/O müveletek csak ehhez az eszközhöz fordulnak. Az előzőekben leirt esz-

közök felhasználásával készültek el az első operációs rendszerek kötegelt feldolgozásra /Share Operating System, az IBM 709-re/.

A következő lényeges előrelépés a megszakitórendszerek megjelenése volt. A korábbiakban már utaltunk arra, hogy milyen nehézségekbe ütközött a program futásának és az I/O csatornák müködésének szinkronizálása. Amennyiben az utasitások természetes sorrendjétől csak vezérlésátadó utasitásokkal térhetünk el, rendkivül nehézkesen lehet összehangolni a futást külső eseményekkel /pl. egy I/O tevékenység befejeződésével/. Az összehangolásra kezdetben kényszerüségből software eszközöket alkalmaztak. Magasabb szintü megoldást a megszakitórendszerek alkalmazása jelentett, természetesen ujabb software eszközök felhasználásával. A megszakitórendszernek az a lényege, hogy bizonyos külső események bekövetkezésekor /az eseménnyel kapcsolatban álló eszközökről érkező megszakitójel hatására/ a végrehajtási sorrend eltér a program által meghatározott utasitássorrendtől, és a vezérlés olyan programnak adódik át, amely értelmezi és feldolgozza a megszakitójelet, majd miután elvégzi az ehhez szükséges tevékenységeket, visszaadja a vezérlést vagy a megszakitott programnak, vagy valamely más, időközben futásra késszé vált, valamilyen prioritási elv értelmében "sürgősebb" programnak. Megszakitáskor a későbbi folytatáshoz szükséges regiszterek tar-

talma tárolódik, hogy a vezérlés visszavételekor az eredeti program zavartalanul müködhessen tovább. Természetesen most csak egy egyszerüsitett sémán mutattuk be azt a hardware adta lehetőséget, amelyet az un. felügyelő /supervisor/ programoknak kell kihasználniok. A kezdeti alkalmazásoknál a transzfer tevékenység egy-egy fázisának a befejeződésével megjelenő megszakitójel hatására a /programozott/ csatornát kezelő szubrutinnak adódott át a vezérlés, majd az I/O tevékenység soron következő átviteli fázisának a beinditása után ismét visszza a megszakitott programnak. Ezt az elvet eleinte csak egy programon belül alkalmazták, később azonban az időosztásos rendszerek alapjává vált. Az autonom /csatorna-programok vezérlése alatt müködő/ csatornák kialakitásával a csatornát kezelő szubrutin szerepét a csatorna-program vette át, és megszakitójel csak ennek a csatorna-programnak a befejeződésekor keletkezett.

A megszakitó rendszerekhez kifejlesztett I/O rendszereknél szükségtelen, hogy a programozó közvetlenül forduljon az I/O eszközökhöz, mivel ezek kezelését rendszerprogramok végzik. Sőt, az esetleges zavarok elkerülése érdekében, egyenesen célszerü volt megtiltani a felhasználók számára a hozzáfordulást. E probléma megoldását szolgálja az, hogy a központi egységben különböző állapotokat lehet kialakitani, amelyek közül nem mindegyikben hajtható végre az összes utasitás. Egyes utasitások privilégizáltak bizonyos állapotok számára. Ezen

utasitások körébe tartoznak elsősorban az I/O utasitások /START I/O: inditsd el az I/O tevékenységet; HALT I/O: függeszd fel az I/O tevékenységet; TEST I/O: kérdezd meg a legutoljára végrehajtott I/O tevékenységre vonatkozó perifériális eszköz és eszközvezérlő egység állapotjellemzőit; TEST CH: kérdezd meg a csatorna és a csatornaprogram állapotjellemzőit/, továbbá egyes kifejezetten rendszerfeladatok végzésére szolgáló utasitások /pl. a program-állapotszó megváltoztatására, a memóriavédelem megszervezésére, az óraregiszter tartalmának megváltoztatására szolgáló utasitások/.

A különleges processzor-állapot bevezetésével ugynevezett csapdázott utasitásokká váltak a korábban megállást okozó, valamint a nem definiált kódu utasitások. Ezek, ha a processzor normális üzemmódjában alkalmazzák őket, szintén megszakitásokat okoznak. Egyes gépeknél a program adott szakaszára csapdázottnak nyilvánithatunk normális utasitáskódokat is. Ilyenkor használva őket, normális végrehajtásuk helyett szintén megszakitás jön létre, és az ennek nyomán beinduló rutin helyettesiti a normális müködést. Ugyancsak csapdázott utasitásként viselkednek -- de most már programtól és processzor-állapottól függetlenül -- a szorzás, osztás vagy pl. a lebegőpontos utasitások, ha a hozzájuk tartozó opcionális hardware-t a processzor nem tartalmazza. Megemlitjük, hogy csapdázott utasitásokat először, még a hagyományos szó-orientált gépekre irt értelmező /interpretive/ szub-

rutinok használtak, amennyiben az értelmező szubrutin hatása alatt végzett feldolgozás során csapdázásra került minden olyan müveleti kóddal rendelkező utasitás, amelyet a normális feldolgozásban használt módtól eltérően kellett értelmezni /akár értelmezve volt a normális feldolgozásra, akár nem/. Ezek természetesen megszakitás helyett az /uj/ értelmezésüket megszabó rész-szubrutinnak való vezérlésátadást okoztak.

Meg kell emlitenünk a normális üzemmódban is végrehajtható, szubrutin-hivás-szerüen müködő un. definiálható utasitásokat. Ezek tulajdonképpen hardware eszközök segitségével kialakitott értelmező szubrutinokként müködnek. A normális szubrutin-hivó utasitásokhoz képest a különbség itt annyi, hogy a cimmezőben nem a szubrutin kezdőcimére hivatkozunk, hanem egy operandusz /vagy paraméter/ cimére. Az utasitás hatására vezérlésátadás történik a müveleti kód által meghatározott fix cimre; az utasitást definiáló rutinnak itt kell kezdődnie. Az operandusz cimének a megjegyzése és az operandusz előhivása hardware uton történik. Ezért a definiálható utasitással megvalósuló szubrutinhivás esetén a paraméter-átadás egyszerübb, mint a szokásos zárt szubrutinok hivásánál.

A csatornák és a megszakitási rendszerek adtak lehetőséget arra, hogy a felhasználók logikai szinten kezelhessék a perifériákat. A kezdeti I/O rendszerekben csupán azonos tipusu perifériák helyettesithették egy-

mást, tehát a program irásakor rögziteni kellett az eszköztipust, a konkrét fizikai eszközt azonban már nem. A teljesen eszköz-független I/O kezelésnél csupán az adatfile-okra vonatkozó információkat kell tartalmaznia a forrásprogramnak, a file-t tartalmazó I/O eszköz logikai specifikálása csak a futás előtt történik meg /a felhasználó adja meg!/, a fizikai hozzárendelésről azonban már maga a rendszer gondoskodik. A teljesen eszköz--független kezelésre módot ad az is, ha a hozzáfordulás valamely programhoz és tárolóterülethez történik, amely a hivatkozott eszköznek megfelelő formában közli az adatokat az aktivizálandó programmal /Freeman, 1968/.

1.6. Az időosztásos rendszerek kialakulása és fejlődése

Időosztásról akkor beszélünk, ha a gép valamely egységét /a központi egységet/ több egymástól független program használja időben váltakozva.

Az autonóm csatornák és a megszakitási rendszerek képezték a hardware alapot /a cimzési rendszerek és perifériák fejlődése mellett/ az időosztásos rendszerek kialakulásához. Megjegyezzük, hogy az időosztás elsősorban software fogalom, mivel a rendszerprogramoktól függ, milyen módon használjuk ki a hardware-lehetőségeket.

Az időosztás történetileg első formája a multiprogramozás volt, amely azt a lehetőséget használta fel, hogy az egyes programokhoz tartozó I/O tevékenységek során más programok futtathatók a központi egységben. Ez-

zel párhuzamositani lehet több program futását. Gondoskodni kell természetesen arról, hogy a programok futását megfelelő operációs rendszer koordinálja, és hogy /ahol csak lehetséges/ a tevékenységek átlapoltan menjenek végbe. Ezt az elgondolást először a Ferranti Orion gépen valósitották meg részlegesen, teljes egészében pedig a Ferranti Atlason /Barron, 1971/, /Kilburn, 1961/. Az időosztásos rendszerekben követelmény, hogy amint a központi egység felszabadul, azonnal valamely egyértelmüen meghatározott program foglalja le. Multiprogramozás esetén ez a kiválasztás prioritási számok összehasonlitása alapján történik.

A prioritási rendszeren alapuló feldolgozási módok számos előnyt biztositanak a gép kapacitásának jobb kihasználásához. Felhasználói oldalról azonban lényeges kifogásokhoz is vezettek, amelyek közül a legfontosabb az, hogy kedvezőtlen esetben hosszu ideig kell várakozni a program futására. Ezek a hiányosságok vetették fel azt a gondolatot, hogy magának az időnek a mulását is jelentős faktorként kellene figyelembe venni. Ez megoldható olymódon, hogy hardware-eszközzel /timer/ gondoskodunk arról, hogy meghatározott időintervallum elteltével megszakitójel érkezzék. A programok legmegfelelőbb ütemezését lehet elérni, ha e megszakitjel felhasználásával rögzitett időszakonként más-más program kapja meg a vezérlést, miközben megtarthatók az időosz-

tás előző formájánál használatos megszakitási formák is. Az igy kapott módszert időszeletelésnek nevezzük, tekintettel arra, hogy a legkarakterisztikusabb megszakitójelet a timer adja.

Ujabb számitógépeken már nemcsak a perifériás eszközök szerepelnek többszörözötten, hanem a központi feldolgozó egységek is. Ezáltal a feldolgozási tevékenység ujabb részei válnak párhuzamosithatókká. Azokban az esetekben, mikor csak az aritmetikai-logikai egység fordul elő többszörösen, csupán a memória megfelelő előkészitését kell elvégezni, mivel mindegyik egység ugyanazokat a müveleteket képes végrehajtani. Ha mindegyik processzor egyenértékü /szimmetrikus processzorok/, a munkák szervezésétől függ a felhasználási mód. Megfelelő kezelőrendszert feltételezve a gép ugyanazon munka különböző részeit hajthatja végre egyidejüleg, meggyorsitva ezzel a munka átfutását. Egyes modern programozási nyelvekben lehetőség van a programok elágaztatására és egyesitésére, ami ugyanazon programon belüli párhuzamos feldolgozást is lehetővé tesz. Mindkét esetben ugyanaz a probléma jelentkezik: nehéz biztositani a processzorok egyenletes terhelését. Ennek megoldását szolgálják az olyan rendszerek, melyekben több kezelőrendszer dolgozik egy magasabbszintü kezelőrendszer felügyelete alatt.

x x x

2. A gépi nyelvek, valamint a magasabbszintü programozási nyelvek és kapcsolatuk elemzése

2.1. Gépi nyelvek és "programozhatóság"

A mai univerzális számitógépeknek a Turing-géptől /TURING 1960/ eltérő sajátosságai nem eredményeznek semmiféle "általánosság-növekedést": ma is pontosan olyan algoritmusok hajthatók végre számitógépeken, amelyeket a Turing-gép is képes elvégezni. Mik indokolják tehát, hogy a számitógépek strukturája mind bonyolultabbá válik? Bár nem tagadható a korszerü, fejlődő technikai szinvonal befolyása sem, mégis azt válaszolhatjuk, hogy elsősorban a software-igények. Ezek részben felhasználói oldalról jelentkeznek, részben a számitógép -- mint információfeldolgozó rendszer -- egészének vezérléséből adódnak.

Az első számitógépeket numerikus eljárások végrehajtására készitették: ennek következtében gépi kódu utasitásrendszerükre az aritmetikai és logikai utasitások voltak jellemzők. Aritmetikai egységeiket a számokon végzendő müveletek követelményeinek elsőrendü figyelembevételével tervezték. A numerikus eljárások nagyrészt ciklikus programokban realizálhatók, ami egyrészt a vezérlésátadó utasitások, másrészt az indexregiszterek megjelenéséhez vezetett.

Az uj, nem-numerikus felhasználási területek nagy

mértékü térhóditása további hardware-fejlesztéseket követelt. A fejlesztések a gépi utasitásrendszerek bővülésében is tükröződtek: az adatfeldolgozási feladatok jobb ellátására decimális aritmetikai és az input/outputot megkönnyitő szerkesztő utasitások, szövegfeldolgozási célokra a string-kezelő utasitások stb. jelennek meg.

A technikai szinvonal fejlődése eközben fokozatosan elvezetett az időosztásos rendszerek kifejlesztéséhez. Ezzel azonban uj vezérlési feladatok hárultak a gépre /software oldalról tekintve az operációs rendszerekre/: bonyolultabb cimzési eljárások, autonom input/output csatornák, táblázatok, várakozási sorok stb. kezelése válik szükségessé. Ezek egy része hardware uton oldható meg; emellett a hardware-fejlődés jelentős segitséget nyujt a software számára is. Igy alakultak ki a bázisregiszterek, a lap- és a szegmentumcimzési eljárások, a megszakitási rendszerek, csatorna- és periféria-lekérdező utasitások stb., amelyekkel az előző fejezetben részletesen foglalkoztunk.

Talán a legerősebb, illetőleg a legállandóbb felhasználói igény a minél egyszerübb programozhatóság, amit csak nagyon kis mértékben elégitett ki a gépi utasitásrendszerek fejlődése. A gépi kódu utasitások már formájukat tekintve is igen messze állnak a különböző felhasználási területek hagyományos irásmódjától. Numerikus jellegü feladatokban pl. hagyományosan algebrai kifejezéseket használunk, müveleti jelekkel és zárójelekkel, ezzel szemben a gépi kódu utasitások inkább a függvény-irásmódra /vagy a

prefixumos lengyel jelölésre/ emlékeztetnek /ahol a függvény-név szerepét a müveleti kód, az argumentumok szerepét pedig a cimek játsszák/. Még lényegesebb eltérés a gépi kód és a hagyományos irásmód között az, hogy a gépi kódban irt program az egyes müveleti jeleknek megfelelő utasitásokra bontja fel a hagyományos irásmódban több müveleti jelet és esetleg zárójeleket is tartalmazó kifejezés értékének kiszámitását. Ezen eltérések miatt a gépi kódban való programirás már az 1950-es évek második felétől kezdődően többé-kevésbé magasabb szintü algoritmikus programozási nyelveket használ az egyes felhasználási területek jellegzetes algoritmusainak megfogalmazására. Az algoritmikus nyelveken felirt programokat vagy kompilátorok transzformálják a számitógép belső nyelvére, vagy interpretátorok hajtják végre, közvetlenül értelmezve a forrásnyelvi szöveget. /Természetesen a két módszer kombinálható is./

Ezzel azonban egy uj, igen jelentős "felhasználási terület" alakult ki. Az algoritmikus nyelveken felirt programok szintaktikus elemzése, a szöveg-transzoformációk a gépi utasitások és programok szintézise, stb. olyan összetett müveletek, amelyek speciális adatelemek és adatstrukturák kezelését teszik szükségessé. Habár egyes becslések szerint a gépi idő kétharmad részét compilálásra és programpróbára használják fel /Anderson, 1961/, mégis a hardware nagyon kevés segitséget nyujt az ilyen jellegü software-munkához.

Az algoritmikus nyelvek implementációja /a kompilá-

ció vagy az interpretáció/ két szempontból is problematikus. Az egyik azzal az ellentmonndással függ össze, amely az algoritmikus nyelvek adatelemei és a gépi utasitások operanduszaiként használható adatelemek között fennáll. Ez adódhat abból, hogy bár valamely adattipus egy adott programozási nyelvben elemi jellegünek számit, a gépi nyelven mégis összetett adatstrukturaként kell kezelni /pl. string implementációja egy szószervezésü számitógépen/. A kompilátorok által elvégzendő teendők jelentős része éppen az ilyen ellentmondások feloldásával kapcsolatos. A másik szempont, amelyből tekintve nehézséget okoz az implementáció, a forrásprogram szintaktikus elemzéséhez illetőleg a gépi program szintéziséhez kapcsolódik. Ez a nehézség főleg az algoritmikus nyelvek és a gépi nyelvek részben már emlitett formai különbségeinek rovására irható.

A kompiláció abból a szempontból sem szerencsés megoldás, hogy az elkészült program próbája nem forrásnyelvi szinten, hanem többnyire gépi-kód szinten végezhető el. Emiatt a felhasználóknak -- legalábbis bizonyos mértékig -- tisztában kell lenniük a gép utasitásrendszerével és strukturájával ahhoz, hogy a számitógép hibajelzéseit helyesen tudják értelmezni. Az interpretáció ebből a szempontból hasznosabb, itt ugyanis végrehajtás során rendelkezésre áll a forrásnyelvi szöveg, igy a hibajelzések forrásnyelvi szinten adhatók. Jelentős hátránya viszont a kompilációval szemben a lassuság.

2.2 A gépi utasitásszint növelésének lehetőségei

Azt a felhasználói igényt, hogy a számitógép könnyen programozható legyen, részben hardware-fejlesztéssel, részben software-eszközökkel igyekeznek kielégiteni. Az is kézenfekvő, hogy a software bizonyos elemei "hardware-esithetők", azaz időről-időre /technikai és gazdaságossági okok befolyásától függően/ a softwareben jelentkező egyes uj feladatok ellátását a hardware átveheti. Ebben a folyamatban igen jelentős fejlemény volt a mikroprogramozható gépek megjelenése abból a szempontból, hogy a gépi utasitásrendszert /elég nagy mikroprogramtárat feltételezve/ tulajdonképpen tetszőleges irányban továbbfejleszthetővé és bővithetővé tette /GREEN, 1966/.

Egy számitógépet mikroprogramozottnak nevezünk, ha gépi utasitásait /vagy azok egy részét/ olyan szubrutin hivásával hajtja végre, amely egy erre szolgáló, rendszerint read-only tárolóban, az un. mikroprogramtárban van elhelyezve, és olyan a gépi kódnál egyszerübb utasitásokból van felépitve, amelyeket a hardware közvetlenül képes interpretálni.

A hardware által interpretálható /azaz a mikroprogramokban használható/ utasitások rendszere az elemi müveleti szint. Ennél magasabb szintet képvisel az utasitásszint: a gépi utasitások rendszere. A számitógép, mint hardware-egység szempontjából, a gépi utasitásrend-

szer egy virtuális számitógép utasitásrendszere. Ebből a szempontból tehát a mikroprogramtárban elhelyezett szubrutinok egy másik számitógépet szimulálnak. Abban az esetben, ha mikroprogramtár segitségével egy ténylegesen létező számitógépet szimulálunk egy másikon, emulációról beszélünk, ami a software-szimulációnál lényegesen gyorsabb és emiatt tágabb feladatkörben alkalmazható módszer.

Mivel mikroprogramozott számitógépek esetében ahhoz, hogy a számitógép univerzális legyen /azaz, hogy a Turing-gép általánossági szintjét elérje/, az szükséges, hogy már elemi müveleti szinten is univerzális legyen, elvileg bármilyen utasitásszint emulálható. Természetesen a mikroprogram-tár kapacitása e tekintetben korlátot jelent. Kézenfekvő tehát, ha arra gondolunk, hogy egy magasabb szintü programozási nyelvet, mint egy virtuális számitógép belső nyelvét emuláljuk. Egy ilyen számitógép utasitásszinten ugy viselkedik, mintha belső nyelve ez a magasabb szintü programozási nyelv lenne. Természetesen minden mikroprogram hardware-ben is realizálható, azaz megépithető olyan számitógép is, amelynek már elemi müveleti szintje is egy magasabb szintü programozási nyelvvel egyezik meg. A továbbiakban mindkét változatról, mint formulavezérlésü számitógépről beszélünk. /Az elnevezést az algoritmikus nyelvekben általában használatos algebrai és logikai kifejezések, formulák indokolják./

Megemlitjük még az "egyszerü programozhatóság" felhasználói igény kielégitésének két, nem formulavezérlésü számitógépekkel történő példáját, amelyek sok tekintetben éppolyan hatékonyak, mint az. Mindkettő végső soron egyfelől gépi utasitásszinten biztosit egyszerübb programozhatóságot /a változatos adatstrukturák gépi utasitásszintre hozásával és fejlett memóriaszervezéssel/, másfelől messzemenően figyelembe veszi a kompilátorok igényeit /azáltal, hogy assembly-szintü nyelveik "formula-nyelvek"/.

A HP/3000 /HP/3000, 1972/ számitógépnél LIFO memóriastruktura alkalmazásával értek el magas szervezettséget.

Mind a program-, mind az adat-mező/k/ LIFO memóriák és a fizikai memória bármely szegmentumában elhelyezkedhetnek. Egy-egy szegmentumot a hozzátartozó mutató-/pointer/ csoport jellemez: a mutatók a LIFO memória alsó és felső határát, az utoljára betöltött szó cimét, stb. tartalmazzák. Az utasitások a müveleti kód mellett jelzik, hogy a cimrész mely mutatóra nézve relativ. Ily módon elérhető a mutatókban tárolt cimek "környezete". Ehhez /a lényegében fejlett báziscimzéshez/ társul az indirekt cimzési lehetőség, amely most már a szegmentum bármely részéhez hozzáférést biztosit.

Az utasitások végrehajtása ~~részint~~ a mutatók tartalmától függően automatikusan változik /ld. cimnélküli utasitások/, részint speciális utasitások szolgálnak a ki-

ván módositások elvégzésére.

Ha összehasonlitjuk az SPL/3000 assembly nyelvet /SPL/3000, 1972/ más gépek assembly nyelveivel, kétségtelenül megállapitható, hogy a szóbanforgó számitógép tervezésekor a hardware és software szempontok egyaránt szerepet kaptak oly módon, hogy következményként a korábbiaknál egyszerübben valósithatók meg az időosztásos rendszerek, könnyebben irhatók kompilátorok, stb.

A másik példában /KILBURN, 1969/ elsősorban az utasitások cimrészének strukturáltságával érik el a jobb programozhatóságot. Ebben az esetben a hagyományos báziscimzés mellett olyan cimzési mód is használható, amely az utasitás által kezelt adatstrukturáktól függően szegmentálja a cimrész bitjeit. A számitógép ezeket a szegmentumokat, mint az adott struktura egyik elemének eléréséhez szükséges paramétereket kezeli és segitségükkel számitja ki az elem memóriabeli cimét. Példaképpen: a vektorelemre hivatkozó utasitás cimrésze a vektor első elemének cimét, az index alsó és felső határát és a vektorelemek tipusát tartalmazza /az indexet előzőleg az aritmetikai egység meghatározott regiszterébe kell elhelyezni/.

További előnyöket biztosit ebben az esetben is a virtuális cimzési rendszer, valamint a LIFO memória az eljárások szervezésére stb.

2.3. Néhány speciális programozási nyelv formai és tartalmi tulajdonságainak elemzése

A formulavezérlésü gép nyelvére teendő javaslataink további előkészitéseként most néhány olyan általánosan elterjedt programozási nyelv formai és tartalmi tulajdonságait vizsgáljuk, amelyeket elsősorban speciális információfeldolgozási területeken alkalmaznak. Bár ezek közül is egyeseket univerzális nyelvnek terveztek /mint pl. az ALGOL 60-at/, a későbbiek során azonban kiderült, hogy használatuk csak meghatározott korlátok között célszerü. Másrészt valamennyi speciális célu nyelv létrehozásánál is bizonyos foku univerzalitásra törekedtek. Nehéz lenne tehát egyértelmüen határvonalat huzni a speciális célu és az univerzális programozási nyelvek közé. Vizsgálataink céljából mégis helyesnek tünik néhány alkalmazási területet kiemelni, és az ezek problémáinak megoldására használt, illetve tervezett nyelveket speciális célu programozási nyelveknek tekinteni. /Az univerzális célu nyelvekkel külön fejezetben foglalkozunk./

A kiemelt alkalmazási területek és a nekik megfelelő nyelvek a következők:

- numerikus problémák /ALGOL 60, FORTRAN/
- adatfeldolgozási feladatok /COBOL, RPG/
- string-manipulációs feladatok /COMIT, SNOBOL/
- lista-kezelési problémák /LISP, SPRINT/
- folyamatirányitás /INDAC/

- rendszerprogramozás /ELISS, EULER, MARY, CDL/

Először áttekintjük az egyes területekhez tartozó nyelvek általános tulajdonságait, majd nyelvi elemek szerinti csoportositásban megvizsgáljuk, hogy

- milyen operanduszok /egyszerü és összetett változók, konstansok/ fordulnak bennük elő,
- milyen a változók és azonositók kezelése /tipus-hozzárendelés, hatáskör, paraméterátadás szubrutinoknak, tárhozzárendelés/,
- milyen vezérlési formákat alkalmaznak /operátor-hierarchia, blokk-szerkezet, utasitások tipusai, szubrutinformák, I/O utasitások/,
- milyen programszöveg-szerkesztési, nyomkövetési lehetőségek találhatók bennük.

E vizsgálat célja az, hogy összefogó képet adjunk a jelenleg alkalmazott elemekről és formai megoldásokról.

2.3.1. Numerikus problémák megoldására szolgáló nyelvek

Numerikus problémák megoldásában a legelterjedtebb nyelveknek az ALGOL 60 és a FORTRAN tekinthető. Közülük a FORTRAN rendelkezik régebbi multtal, bár a ma általánosan használt FORTRAN IV lényegesen különbözik az 1957-ben alkalmazott első FORTRAN rendszertől. Az ALGOL 60 -- első publikálása óta -- lényegében változatlanul maradt. Megjegyezzük, hogy a két nyelv közötti lényeges különbségek ellenére az ALGOL 60-nak jelentős befolyása volt a FORTRAN fejlődésére.

Az ALGOL 60 programok adott szimbólumokból /az ALGOL 60 alapjeleiből/ meghatározott szabályok /szintaxis/ szerint felépitett véges hosszuságu sorozatok. Az ALGOL alapjelei: betük, számjegyek, logikai értékek és elhatároló jelek. Az elhatároló jelek közé egyrészt az aritmetikai és logikai müveletek, valamint a relációk jeleit sorolják, másrészt egyéb kisegitő jeleket /ezek mindegyike grafikailag is általában egyetlen jel/, végül un. összetett alapjeleket /a hivatkozási nyelvben aláhuzva/, amelyek grafikailag több elemből állnak.

A számitási algoritmusokban az alapjelek segitségével a megszokott formában irhatók fel a matematikában használatos aritmetikai és logikai kifejezések. A kifejezések konstansokat /számokat, logikai értékeket/, változókat és függvényeket tartalmaznak. A kifejezések értékeit a változókhoz értékadó utasitásokkal rendelhetjük hozzá. Az értékadó utasitások mellett az algoritmusok leirásához vezérlésátadó, ciklusszervező, valamint eljárásutasitások állnak rendelkezésre. Mind az utasitások, mind a kifejezések lehetnek feltételesek és feltétlenek. Utasitások sorozataként -- utasitászárójelek alkalmazásával -- összetett utasitásokat képezhetünk, amelyek ugyanolyan módon kezelhetők, mint az alaputasitások. Az emlitett utasitászárójelek szolgálnak az összetett utasitásokhoz hasonló blokkok kijelölésére is, amelyeket ugyancsak az alaputasitásokhoz hasonlóan kezelhetünk. Ezek azonban egy deklarációs részt is tartalmaznak. A dekla-

rációs rész arra szolgál, hogy meghatározzuk a változók kezelésére használt azonositók tipusát, a tömbök dimenziószámát és indexhatárait, továbbá a kapcsolók és eljárások jelentését. /Ez utóbbiak jelölésére is azonositókat használunk./ Az eljárások deklarációiban azt az algoritmust irjuk le, amelyet a megfelelő eljárásutasitás előfordulásakor kell végrehajtani /természetesen a definició bizonyos kiegészitő információkat is tartalmaz/.

A blokk -- illetőleg az, hogy éppugy kezelhető, mint az alaputasitások -- ad lehetőséget az ALGOL 60 sajátos blokkstrukturájának kialakitására. A programban a blokkok mind egymás mellett, mind egymásbaskatulyázott formában előfordulhatnak. Minden blokk -- deklarációi révén -- uj jelölésszintet vezet be a változókra, a kapcsolókra és az eljárásokra. A deklarált változók csak abban a blokkban érvényesek, amelyben deklarálva vannak. A blokkon kivül nem lehet rájuk hivatkozni, ezért ezeket a szóban forgó blokkra nézve lokális változóknak nevezzük. Tekintettel arra, hogy valamennyi ALGOL 60 program egy blokk /elfajult esetben összetett utasitás/, a programra nézve minden változó lokálisnak tekintendő. Az olyan változókat, amelyek deklarációja valamely külső blokkban található, de amelyeket egy belső blokkban is felhasználunk, e belső blokkra nézve globális változóknak nevezzük. Valamely globális és lokális változó azonositójának megegyezése esetén az utóbbi érvényességi körén belül a globális változó hozzáférhetet-

len. Minden utasitás ellátható egy vagy több cimkével. Hasonlóan a változókhoz, a cimkékkel kapcsolatban is beszélünk hatáskörről. A cimkék nem esnek deklarációs kötelezettség alá, a hozzájuk tartozó blokkra nézve azonban lokálisak.

Az ALGOL 60 egy további lényeges jellegzetessége az eljárás- és függvényeljárás fogalma. Az eljárás lehetőséget ad uj utasitások bevezetésére azáltal, hogy az eljárásdeklarációban megadott algoritmust az eljárás nevét tartalmazó egyetlen utasitás /az eljárásutasitás/ képviseli a programban. Az eljárásutasitás olyan /aktuális/ paramétereket is tartalmaz az eljárás neve mellett, amelyekkel /esetleg értékeikkel/ a deklarációban leirt algoritmus formális paramétereit kell helyettesiteni végrehajtása előtt. A paraméter átadásának módja szerint megkülönböztetünk név- és értékszerinti paraméterátadást. A függvényeljárások valamilyen függvény értékét definiálják a deklarációban megadott eljárás segitségével. A függvényértékekre kifejezésekben a szokásos formáju függvényjelöléssel hivatkozhatunk. Ez formailag és a paraméterek kezelése tekintetében megegyezik az eljárásutasitással.

Az ALGOL 60 nyelv egyik komoly hiányosságaként azt emlithetjük, hogy nincsenek benne fejlett, egységesitett I/O utasitások.

A FORTRAN és az ALGOL 60 között számos hasonlóság mellett lényeges különbségeket is találunk. Hasonlóság

van pl. az operanduszok tipusaiban és a kifejezések felépitésében. Az ALGOL 60-nal ellentétben azonban a FORTRAN-ban nem használhatjuk a feltételes kifejezési formát és az utasitásokban is csak a vezérlésátadó utasitások tartalmazhatnak feltételeket. Az indexkorlátok, az indexkifejezések és a ciklusutasitás sem érik el az ALGOL 60-beli megfelelőik általánossági szintjét. A FORTRAN-ban viszont megengedett a skaláris változók implicit deklarációja, ami azt jelenti, hogy -- hacsak külön információkat nem adunk -- bizonyos betüvel kezdődő azonositókat a fordító automatikusan egész, illetve valós tipusuaknak tekint. Ezzel szemben a kifejezések feldolgozásában hiányzik az automatikus tipuskonverzió.

Tekintettel arra, hogy a FORTRAN programok nem blokkszerkezetüek, hiányzik a lokális és globális változók fogalma. A FORTRAN-ban a program részekre /szegmentumokra/ tagolódik. Minden szegmentum főprogramból, valamint szubrutinokból /szubprogramból/ áll. A szegmentumok változói közötti kapcsolatot a közös mezőbe /COMMON/ deklarált változók teremtik meg. A közös mező felosztásának -- amennyiben a változók egyeztetése a cél -- kompatibilisnek kell lennie az érintett szegmentumokban, és erről a programok iróinak kell gondoskodniok. A változóterület felhasználását gazdaságosabbá, tömbökre való felosztását pedig az egyes programrészek számára célszerübbé tehetjük az ekvivalencia-utasitással, amelynek hatására egyugyanazon programszegmentum különböző változóival hivatkozhatunk ugyanarra a memóriaterületre, különféle célokra használhatjuk ugyanazt a tárolórészt, és

-- végül -- különböző nevekkel hivatkozhatunk ugyanarra az operanduszra.

Az egyes szegmentumok külön-külön forditódnak le, és közöttük a vezérlésátadási kapcsolatot a futás során speciális szubrutin hivásával /CALL LINK/ adjuk meg.

A FORTRAN-beli programozás megkönnyitésének igen fontos eszközei az ALGOL 60-nál tárgyalt eljárások és függvényeljárások megfelelői: a szubrutinok /SUBROUTINE/ és a függvények /FUNCTION/. Ezek nem tartalmazhatnak belső szubrutin-, vagy függvénydefiniciót. A szubrutinok önálló utasitásként aktivizálhatók. A függvények egy függvényértéket határoznak meg és kifejezésekben használhatók. Paramétereknek a főprogram és a szubrutinok közötti átadása részint az aktuális, illetve a formális paraméterlista segitségével, részint pedig a közös /COMMON/ változóterület felhasználásával történhet. A szubrutinok és a függvények megadhatók közvetlenül azokban a szegmentumokban, amelyekben felhasználjuk őket, de lehetőség van arra is, hogy az általánosan használt szubrutinokat és függvényeket rendszerkönyvtárban helyezzük el. Ha igy járunk el, az ujabb programokban nem kell azokat ismételten definiálni.

Végül fontos jellemzője a FORTRAN-rendszernek a fejlett, szabványos I/O kezelés. Az I/O utasitásokkal szerkesztési, file-definiciós és kezelési feladatok oldhatók meg /logikai file-okra/.

2.3.2. Adatfeldolgozási feladatok megoldására szolgáló nyelvek

Az adatfeldolgozásra orientált nyelvek /elsősorban a COBOL/ egyik jellemző tulajdonsága a programok sajátos felépitése. A forrásnyelvi programok általában a következő részekből tevődnek össze:

- azonositó rész,
- környezetleiró rész /amely meghatározza, hogy a célprogram futtatása milyen gépkonfiguráción történik/,
- adatleiró rész /amely megadja az I/O file-ok logikai strukturáját/ és
- eljárás-rész.

Az egyes részek felépitése hierarchikus; fejezetekből vagy paragrafusokból állnak, a paragrafusok mondatokból, a mondatok pedig -- szabályoknak eleget tevő -- alapjel-sorozatokból. Tul nagy memóriaigényü programok esetében lehetőség van az eljárási rész szegmentálására. A szegmentum: fejezetek összefüzött sorozata. Vannak olyan szegmentumok, amelyek a futtatás során állandóan a központi memóriában tartózkodnak, mások csak végrehajtásuk idején kerülnek be a belső tárolóba.

A használatos operandusztipusok a többnyire fixpontos decimális, binárius, logikai és szövegszerü skalármennyiségek. Az azonos tipusu, számrendszerü, ábrázolásmódu és pontosságu skalárok tömböket alkothatnak. Egy másik összetett változó a struktura /record/, amely ska-

lárokból, tömbökből és strukturákból épül fel hierarchikusan.

Az I/O müveletek többnyire file-okat kezelnek /teljesen általános értelemben/. Mind a háttértárolókon, mind az egyéb perifériákon definiálhatunk file-okat.

Tekintettel arra, hogy bonyolult aritmetikai kifejezések az adatfeldolgozási feladatokban nem szerepelnek, az itt megengedett aritmetikai kifejezések általában alacsony szintüek. A COBOL jelölésformája is jelentősen eltér az aritmetikai kifejezésekben szokásostól.

A COBOL-ban megengedett a szubrutinok használata; a paraméter-átadás az adatleiró részben található LINKAGE SECTION segitségével valósul meg. A COBOL-rendszerhez programkönyvtár is kapcsolódik, amelynek elemei valamennyi program számára hozzáférhetők /hivatkozhatók/. A könyvtári programok beépitése a fordítás fázisában történik.

2.3.3. String-manipulációs feladatok megoldását célzó nyelvek

A string-manipulációs nyelveket -- noha a stringek a listák speciális eseteinek tekinthetők -- külön tárgyaljuk, mivel a végrehajtandó müveletek miatt megkülönböztetett figyelmet érdemelnek.

Stringen elemi stringek olyan sorozatát értjük,

melyben valamilyen konkatenációs jel /pl. szóköz, vesszó/ választja el egymástól az elemi stringeket. Elemi stringnek azokat az alfanumerikus jelsorozatokat nevezzük, amelyekben a konkatenáció jele nem fordul elő.

Mivel a string-manipulációs nyelvekben csak ez az egyféle tipusu operandusz fordul elő, nincs szükség deklarációra a programokban. Az implementációkban a stringeket általában lista-strukturában ábrázolják, az elemi stringeket atomnak tekintve.

A string-manipulációs nyelvekben általában a következő elemi müveletek találhatók:

- elemi string helyettesitése stringgel /az elemi stringre kontextus-feltétel is előirható/,
- rész-string törlése, átrendezése,
- string beszurás kontextus-feltétellel megadott helyre,
- rész-string duplikációja,
- ismeretlen rész-string keresése kontextusban,
- szótár generálása /strukturált elemekből álló táblázat előállitása/, keresés szótárban.

A program szabályok sorozata. Minden szabály az un. munkamezőn /ennek tartalma egy string/ végez valamilyen -- egy elemi müveletből álló, vagy több ilyenből összetevő - müveletet, melynek eredménye: a munkamező megváltozott tartalma. A munkamezőben tárolt stringet, vagy annak egy részét azonositóval is megjelölhetjük és a későbbi szabályokban ezzel hivatkozhatunk rá. A szabályok feltételes müveleteket jelölnek. A müvelet lehet eredmény-

telen is, pl. abban az esetben, ha a helyettesitendő elemi string nem fordul elő a munkamezőben tárolt stringben. Ezért lényeges, hogy minden szabályhoz -- eredményes, illetve eredménytelen teljesités esetére is -- egyértelmüen megadjuk a következő végrehajtandó szabályt. A szabályoknak általában cimkéjük és vezérlő részük van; a vezérlő részben az eredménytelen teljesités után végrehajtandó szabály cimkéjét adjuk meg és egy jelzést arra vonatkozóan, hogy ugyanez, vagy a sorrendben következő szabály hajtandó-e végre eredményesség esetén. Igy bármely szabály végrehajtását ciklizálni lehet.

A string-manipulációs nyelvekben a szubrutinok kissé nehézkesen kezelhetők, de nincs elvi akadálya az algoritmikus nyelvekben szokásos módszerek használatának.

2.3.4. Listakezelő nyelvek

A listakezelő nyelvek jellemző adatstrukturája a lista. Ezek a nyelvek igen erősen különböznek egymástól a kezelt listák általánossága és a nyelvi szint tekintetében. Minél magasabb szintü a nyelv, annál nagyobb szabadságot biztosit a programozó számára a listák reprezentációja és a belső tárolásra vonatkozó információk megadhatósága terén. Ezen a területen a LISP 1.5-et kell a legfontosabb nyelvnek tekintenünk.

A kezelt listáknak lehetnek részlistáik és ezek is

további részlistákból állhatnak, és igy tovább. A lista elemi egysége az atom, amely lehet numerikus és nemnumerikus. A nem-numerikus atomon olyan betükből és számokból álló jelsorozatot értünk, mely betüvel kezdődik.. /Egyes reprezentációkban minden olyan jelsorozatot atomnak tekintenek, amely nem értelmezhető számként./ A számok ábrázolása /általában megengedve decimális és oktális egész számokat/ szintén listastrukturában történik. /A nagyobb gépeken implementált rendszerekben lehetőség van lebegőpontos számok használatára is./

A listák kezeléséhez olyan müveleteket biztositanak, amelyek segitségével elvégezhető a listák generálása és elemekre való bontása. Valamennyi emlitett tipusu müvelet három alapmüveletből állitható össze. A CONS müvelet a hozzátartozó két operanduszból állit elő listát, amelynek bal komponense az első, jobb komponense pedig a második operandusz lesz. A CAR egyoperanduszu müvelettel a listák első komponensét kaphatjuk meg. Ez a művelet atomokra nincs definiálva. A CDR -- szintén egyoperanduszu művelet -- a listának azt a részét adja eredményül, amely az első komponens leválasztása után visszamarad. Ez a művelet sincs definiálva atomokra.

A feltételes kifejezések felirásához lista-relációk állnak rendelkezésre. Az implementációkban általában előforduló alaprelációk /az EQUAL és az ATOM/ az argumentumként szereplő kifejezések identitását, illetve azt fejezik ki, hogy a vizsgált elem atom-e, vagy logikai érték-e /T=igaz, NIL=hamis; NIL az üres listát jelöli/. A predi-

kátumfüggvények körében megkülönböztetünk aritmetikai jellegüeket, illetve listákra vonatkozóakat.

Az általános célu listanyelveknél megtalálhatunk aritmetikai éa logikai utasitásokat vagy függvényeket /beleértve a rekurziv formákat is/, valamint eljárásdeklaráló, -hivó és egyéb -- a programok felirását elősegitő -- utasitásokat /pl. feltételes és vezérlésátadó utasitásokat/.

A LISP megenged egészen általános lista-strukturákat, pl. cirkuláris listákat is. /Megjegyezzük, hogy ezeknek az általános strukturáknak nincs minden esetben megfelelő külső ábrázolási formájuk./ A lista-strukturák szerkezete bizonyos müveletekkel megváltoztatható. Ezek azonban óvatosan kezelendők, mivel helytelen használatuk végtelen keresésekhez, cirkuláris listák nyomtatásához stb. vezethet. Az un. tulajdonséglistákkal kapcsolatban is használatosak speciális operátorok /függvények/.

Megemlitjük, hogy a LISP-ben van lehetőség makrók definiálására és használatára is.

2.3.5. Folyamatirányitási feladatok megoldására szolgáló nyelvek

A folyamatirányitási feladatok megoldására szolgáló nyelvek közül elsősorban az INDAC-ot tartottuk szem előtt. Ennek általános jellemzője, hogy erősen FORTRAN-szerü, ezért a számitási algoritmusok leirására vonatkozó

részeivel külön nem is foglalkozunk.

Az INDAC fontos további lehetőségeket tartalmaz időütemezésre, ipari interface készülékek specifikálására és külső megszakitási kérelmek kiszolgálására. A folyamatot mindig egy algoritmus irányitja, amely végezhet adatgyüjtést, operációkat folyamatváltozókon, információkat küldhet a folyamatirányitó készülékekhez stb.

A szegmentált struktura lehetőséget biztosit ahhoz, hogy nagyterjedelmü algoritmusok esetén minden funkcióra külön szegmentumot hozzunk létre. Ezeket az időütemező rész felhasználásával oly módon lehet végrehajtani, mintha a teljes program állandóan az operativ memóriában tartózkodna, ami természetesen gyors és nagykapacitásu háttértárolók alkalmazását teszi szükségessé.

Az INDAC-programok szerkezete nem hasonlit a FORTRAN--programokéhoz, amelyeket egyetlen egységként hajtunk végre, hanem -- a folyamatirányitás követelményeinek megfelelően -- inkább különböző feladatok, programegységek gyüjteményének tekinthető, amelyeket szegmentáló utasitások, az adott részre vonatkozó információk választanak el egymástól.

A programok első részét a feladatspecifikáció képezi. Az ebben megadott információk a feladat globális paramétereit definiálják, amelyek a rendszerszintü programtevékenységhez szükségesek. A feladat első hivásakor ezek

a központi tárba kerülnek és a program teljes futása idején ott is maradnak /védett állapotban/. A megadható információk specifikálják:

- a logikai egységeket, csatornákat, vezérlési módokat és az egyes eszközökhöz tartozó szubrutinokat,
- a megszakitó eszközöket,
- a rendszerszintü adattároló mezőket,
- azokat az üzeneteket, cimeket, vagy egyéb információkat, melyeket a programban lévő, különböző szegmentumok kérésére ki kell nyomtatni, vagy más formában megjeleniteni és
- a rendszer közös adatkiviteli formátumát.

A feladatspecifikáló részt egy vagy több fázis /PHASE/ szegmentum követi. Ennek bevezetését olyan utasitáscsoport alkotja, amely a fázisban végrehajtandó müveletek menetrendjét specifikálja. A további részek a fázisban lévő feladatelemek /SNAP/ vagy más szegmentumok használatát időzitő paramétereket /aktivitási utasitások/ és az időzités nélküli végrehajtást előiró akciólistát tartalmazzák. Ezeket a részeket egy vagy több feladatelem követi, amelyekben az aktuális müveleti utasitások vannak. A fázis csak akkor tartózkodik a központi tárban, ha éppen végrehajtás alatt áll. Amennyiben kilépés után hivatkozunk ujra rá, ez a fázis teljes inicializálását, betöltését eredményezi.

A feladatelemek a bennük leirt tevékenység elvégzéséhez szükséges lokális paramétereket megadó specifikációt és a végrehajtandó utasitásokat tartalmazzák.

A fázisokat a feladat bármely fázisából hivható szubrutinok szegmentuma követi. Ezeket és az INDAC könyvtárába beépitett szubrutinokat külső szubrutinoknak nevezzük. A külső szubrutinok mellett belsők is használhatók, amelyeket a SNAP-ek program- /PROCESS/ részében adunk meg. Ezek csak az őket tartalmazó feladatelemből hivhatók.

Az INDAC-programokban igen jelentős szerepet játszanak a müveletek menetrendjét meghatározó ütemező-, valamint az akciólistában szereplő utasitások. Az ütemező utasitásokban azt adjuk meg, hogy a bennük megnevezett fázisokat vagy feladatelemeket

- milyen időközönként vagy
- milyen késleltetés után vagy
- milyen időpontban és
- milyen prioritással

kivánjuk végrehajtani. Az akciólistában az egyes ütemező utasitásokhoz hozzárendelt időzitőket /timer/ indithatjuk, illetve megadhatjuk azon fázisok és feladatelemek prioritásait, amelyek végrehajtását timerektől függetlenül kérjük. Megjegyezzük, hogy ütemező utasitások a feladatelemek PROCESS részében is megadhatók és ugyancsak itt kérhető timerek leállitása.

Az eddigiekből jól látható, hogy az INDAC hatékonyan müködő futásszervező rendszert követel, amely lehetővé teszi a programok optimális végrehajtását. A rendszernek biztositani kell

- a feladatelemek idő és prioritás szerinti mú-

feladatelemek idő és prioritás szerinti végrehajtását,

- a dinamikus tárkezelést,
- az I/O pufferek kezelését és
- a kezelővel való üzenetváltást.

A rendszerprogramozási nyelvekkel önállóan nem foglalkozunk részletesebben, minthogy összefoglaló jellemzésükben lényegében ugyanazokkal az elemekkel találkozunk, amelyeket a programozási nyelvek általános, belső szempontok szerinti elemzésénél veszünk figyelembe. Az olyan uj elemekkel, amelyeket az univerzális nyelvekből vettek át a rendszerprogramozási nyelvekbe, a 2.4. fejezetben foglalkozunk.

2.3.6. A programozási nyelvekben általánosan használt elemek és formák összefoglaló elemzése

A következőkben áttekintjük a programozási nyelvekben általánosan használatos elemeket és formákat, miközben kisérletet teszünk egy rendszeres és viszonylag teljes leirásra.

I. A programozási nyelvek egyik alapvető jellemzője a kezelhető operanduszok tipusa és strukturája. Az előforduló tipusok a következők:

- egész
- valós
- komplex
- logikai
- karakter
- referencia.
- cimke
- eljárás
- esemény /pl. szemafor/
- szó /értelmezés nélkül/

Az egyes nyelvekben előforduló változótípusokhoz általában a megfelelő konstans formát is megtaláljuk.

A fenti egyszerű operanduszokból a következő összetett operanduszokat képzelhetjük:

- tömbök
- listák
- recordok
- fák
- stringek
- stack-ek /LIFO-k/
- sorok /queue/
- halmazok /set; strukturátlan/.

Az összetett operanduszokkal kapcsolatban megjegyezzük, hogy nem felel meg valamennyi egyszerű operandusznak mindenféle összetett. Egyes esetekben lehetőség van arra, hogy az összetett operanduszok képzését iteráljuk, azaz belőlük ujabb, esetleg bonyolultabb strukturákat épitsünk fel.

A magasabb szintü rendszerprogramozási nyelveknél ma már elterjedt a további, definiálható tipusok és struktúrák használata.

II. A változókat és a nyelv elemeinek megnevezésére szolgáló eszközöket a következőkkel jellemezhetjük:

a/ Tipus hozzárendelés. Ebből a szempontból megkülönböztethetünk nyelveket, amelyeknél

- a változóknaknak nincs tipusa; minden változóhoz hozzárendelhető minden literáltipus, maga a belső ábrázolási forma dönti el a változóhoz tartozó érték tipusát;
- ignorálódik a tipus; minden változóhoz hozzárendelhető minden literáltipus, de a tipusin-

formáció elvész;

- egyértelmüen kötődik valamilyen tipus az azonositókhoz /pl. ALGOL 60/.

b/ Hatáskör. Ebben a vonatkozásban találunk megoldásokat, amelyekben a programok egységei

- csak lokális változók kezelését végzik;
- tartalmazhatnak lokális, vagy nagyobb egységben definiált változókat is;
- hivatkozhatnak egymás közös /common/ változóira.

c/ Paraméterátadás. A szubrutinoknak való paraméterátadásnál két aspektust kell megvizsgálnunk. Az egyik: milyen tipusu paraméterek adhatók át; a másik: milyen a paraméterátadás módja. Az első szerint azzal jellemezhetünk egy nyelvet, hogy felsoroljuk az átadható paramétertipusokat. A második, tehát az átadás módja, lehet

- érték-szerinti,
- referencia-szerinti /pl. a FORTRAN/
- eredmény-szerinti és
- név-szerinti.

d/ Memóriakijelölés. Ez történhet

- a fordítás során statikusan,
- stack /LIFO/ jelleggel dinamikusan és
- szabad-lista felhasználásával dinamikusan.

A felsoroltakon kivül természetesen még egyéb vizsgálati szempontok is felmerülhetnek és a vázolt alternatívák sem meritik ki az összes lehetőséget /előfordulnak pl. kom-

binált esetek is/, de ezek látszottak a leglényegesebbek

nek és a legelterjedtebbeknek.

III. A következőkben az utasitások és a vezérlési formák általános jellemzését adjuk.

Valamennyi nyelvre jellemző, hogy milyen utasitáscsoportositási formákat tesz lehetővé. Utasitáscsoportositási forma pl. az ALGOL 60 összetett utasitása és blokkja, az INDAC SNAP-je stb. Több helyen találkozunk a lineáris prógram-szerkezettel, de magasabb szintünek a hierarchikus felépitést tekintjük.

A kifejezésekben a teljes vagy részleges operátor-hierarchia terjedt el, a köznapi matematikai használtnak megfelelően. A fejlettebb változatokban megtalálhatók a feltételes kifejezések is.

A vezérlésátadó utasitások feltétlen alakja majdnem mindenütt megtalálható, a kiszámitott vezérlésátadás és a kapcsoló is számos helyen előfordul. A feltételes utasitások közül a vezérlésátadók tekinthetők általánosan elterjedteknek; a magasabb szintet itt az képviseli, ha lehetőség van bármely utasitást függővé tenni feltételtől. Gyakran előforduló formák az if-then és az if-then-else. A ciklusutasitásoknál ismét két tipust szükséges megkülönböztetnünk: az until-t és a while-t.

A szubrutinhasználatban maguk a nyelvek viszonylag ritkán engednek meg makroutasitásokat, elterjedtebb az a megoldás, hogy a korszerübb rendszerekben makrofeldolgozó részt illesztenek a már meglévő kompilátorok elé. A paraméter-listával hivható szubrutinformát /az eljárásokat/ általánosan

használják. Ezeket vagy utasitásformában /tetszőleges be- és kimenő paraméterrel/, vagy függvényformában /a kifejezésekben/ alkalmazzák. Az utóbbi esetben a szubrutin egyetlen kimenő értéket szolgáltat: a függvény értékét.

Az I/O utasitások pl. az ALGOL 60-ban nem szerepelnek a rögzitett utasitások között, de ez ma már komoly hiányosságnak számit. Könnyebben használhatók azok a nyelvek, amelyek általános I/O utasitásokkal és formátum-szerkesztési lehetőségekkel rendelkeznek. A legáltalánosabbak a logikai file-kezelést biztositó I/O utasitások.

IV. A programok leirásában általánosan elterjedt bizonyos vezérlő és egyéb alapjelek alkalmazása, elsősorban angol nyelvü szavak vagy rövidités alakjában. A programok összeállitásának és próbájának megkönnyitésére számos implementációtól függő lehetőség áll rendelkezésre. Tekintettel arra, hogy a magasabb szintü nyelvek ritkán teszik lehetővé a gép valamennyi lehetőségét és állapotát kihasználó programok irását, ezért a különböző implementációk -- nem-standard formában -- megengedik alacsonyabb /pl. assembly/ nyelven irt blokkok beágyazását.

2.4. Az általános célu nyelvek jellemzése

A speciális célu nyelveknek az az alapvető sajátossága, hogy csak egy-egy meghatározott területen tekinthetők a programozás hatékony eszközének és emiatt a megoldandó problémától függően más és más programozási rendszer használata válik szükségessé, felvetette az általános célu programozási nyelvek kérdését. Előbb emlitettük, hogy már az ALGOL 60-at is ilyen céllal készitették. De mivel

az univerzalitás tekintetében nem váltotta be a hozzá füzött reményeket, ujabb általános célu nyelvek kidolgo-

zására került sor.

Két nagy felhasználási terület /a numerikus alkalmazások és az adatfeldolgozás/ igényeinek kielégitésére tervezték a PL/1-et, amelyben egyrészt az ALGOL 60, a FORTRAN és a COBOL előnyös tulajdonságait igyekeztek egyesiteni, másrészt olyan uj elemeket is felvettek, amelyek sokkal több lehetőséget biztositanak a kiinduló nyelveknél az adatok és a file-ok leirására. Ha a programozó a lehetséges változatok egyikét sem jelöli ki, akkor a forditóprogram egy megfelelő alapértelmezés szerint végzi el a forditást. Ez az alapértelmezés a PL/1 egyik legfontosabb jellemzője.

A nyelv megalkotói azt tartották szem előtt, hogy a programozó számára a lehető legnagyobb szabadságot biztositsák egyrészt a /nyelvi/ kifejezhetőség, másrészt a gépi eszközökhöz és az operációs rendszerhez való hozzáférhetőség szempontjából.

Ebben a vonatkozásban az volt a cél, hogy ne kelljen alacsonyabb szintü nyelvet használni semmilyen probléma megfogalmazásánál.

A PL/1 programok strukturáját az jellemzi, hogy az utasitásokból nagyobb egységeket /blokkokat/ képezhetünk. A PL/1-ben használt blokk-fogalom megfelel az ALGOL 60-ban használt blokk-fogalomnak. A programok egy, vagy több blokkból épülnek fel. Meg van engedve mind az egymásba ágyazott,

mind pedig az egymás melletti, független blokkokból álló

programszerkezet. A blokkok kettős szerepet játszanak: részint egyértelmüen definiálják a változók és egyéb elemek /cimkék, eljárások/ jelölésére használandó nevek érvényességi körét /lokálisak arra a blokkra nézve, amelyben deklarálva vannak/, részint lehetővé teszik, hogy a memória-hozzárendelést a változókhoz a futtatás során a blokkokba való belépéskor végezzük el és a blokkokból való kilépéskor a lekötött memória-területet felszabaditsuk.

Az eljárások, amelyek mind utasitással aktivizálható, mind pedig függvényformában megtalálhatók, speciális blokkokat alkotnak. Maga a PL/1-program is eljárásként irandó fel.

Megállapitható tehát, hogy a PL/1 átvette az ALGOL 60 két legfontosabb jellemzőjét: a blokk-strukturát és az eljárást. Egyuttal rendelkezik azzal a /FORTRAN-ban is megtalálható/ előnnyel, hogy egyaránt alkalmazhatók benne a programban és könyvárban definiált eljárások.

A PL/1-beli operanduszok tipusait két nagy csoportra oszthatjuk. Az egyikhez a probléma-megoldásban használt adattipusok /valós, képzetes, fixpontos, egész, karakter--string, bit-string tipusok/ tartoznak, a másikhoz pedig a program vezérlésével kapcsolatos adattipusok /label, task, esemény, pointer, terület, cella/. A különböző adattipusoknak a kifejezésekben való egyszerü kezelhetőségét automatikus konverzió biztositja.

Az egyszerü változókból tömböket és strukturákat épithetünk fel. A strukturák hierarchikus felépitésü ösz-

szetett változók, melyek változókat és tömböket tartalmaznak. A struktura elemei nem szükségképpen azonos tulaj-

donságuak; lehetnek strukturák is. A tömbök elemeire indexes változókkal, a strukturákéra a megnevezésükkel hivatkozhatunk. Mind a tömbök, mind a strukturák teljes értékü változóként használhatók a megfelelő utasitásokban és kifejezésekben. A PL/1-ben megoldható a listák kezelése is, a pointerváltozók és a programozó által vezérelt dinamikus memóriaterület változói segitségével. A változók köre tehát lényegesen bővebb, mint az alapul vett nyelvek bármelyikében.

Megtalálható mind a /teljes futási időre szóló/ statikus, mind pedig a dinamikus memória-hozzárendelés. A dinamikus hozzárendelésnél azonban nemcsak az automatikus /blokkstruktura által vezérelt/ kezelést engedi meg a nyelv, hanem a programozónak is lehetőséget ad arra, hogy tetszés szerint irányitsa /controlled dynamic storage alocation/.

Az I/O utasitások közel tetszőleges szintü I/O müveletek kijelölését teszik lehetővé, az egyszerü átviteli és konverziós müveletektől az összetett file-kezelési tevékenységekig.

A felsoroltakon tul a PL/1-ben figyelembe vették még a párhuzamos /több processzoros/ feldolgozás igényeit is. Ezzel elvileg alkalmassá válik real-time folyamatvezérlési müveletek elvégzésére is.

A főbb jellemzők alapján megállapitható, hogy a PL/1 -- a három kiválasztott nyelv szintézise terén -- elérte célját.

Áttérünk az APL főbb jellemzőinek ismertetésére. Ennél más volt a célkitüzés mint a PL/1-nél, ezért természetes, hogy éles különbségek vannak köztük. Az APL segitségével egyaránt könnyen kezelhetők összetett nagy programok és kis, elemi számitások. Egyszerü matematikai jelölésmódot használ; ennek ellenére nem csupán numerikus feladatok megoldására alkalmas. A konverzációs üzemmód -- amelyben használható -- biztositja univerzális jellegét.

Az APL rendszereknél az utasitások két nagy csoportja található: rendszert vezérlő és tulajdonképpeni APL utasitások. Az APL utasitások két üzemmódban /definiciós és végrehajtási üzemmódban/ használhatók. Az előbbi uj APL függvények /eljárások és függvényeljárások/ bevezetését teszi lehetővé, amelyek a későbbi utasitásokban felhasználhatók. A függvényeknek lehetnek lokális változóik is. A specifikáló /értékadó/ utasitások sorrendjétől el lehet térni elágazó utasitások segitségével. Végrehajtási üzemmódban a programozó által közölt utasitások azonnal végrehajtódnak. A kifejezéseknek -- amelyek a szokásos matematikai müveleteken kivül összetettebb müveleteket jelölő operátorokat is tartalmazhatnak -- kiértékelése a matematikai kifejezésekben alkalmazott bal-asszociációtól eltérően, jobb-asszociáció alapján történik, operátorhierarchia nélkül. A müveleti sorrendet zárójelentéssel meg lehet vál-

toztatni. A változókból tömbök épithetők fel, amelyeket egyes utasitásokban egyszerü müveleti komponensekként kezelhetünk. Megtalálhatók az APL-ben a szokásos aritmeti-

kai és logikai müveletek is.

Az APL-nek speciális jelkészlete van, amely lehetővé teszi, hogy a nyelv összetettebb müveleteit /pl. a mátrixmüveleteket/ is egy-egy jellel jelöljük.

Igen hasznosak a függvénymódositást végző müveletek. A definiált függvényjelet ezek segitségével javithatjuk, vagy módosithatjuk. Speciális nyomkövető függvény könnyiti meg a programok kipróbálását.

Az APL igen hatékonynak bizonyult az interaktiv programfejlesztés, a programkipróbálás és a futtatás területén, de sem a programstruktura, sem a kezelt operanduszok terén nem jelent lényeges előrelépést.

A legáltalánosabb nyelvnek ma az ALGOL 68-at tekinthetjük. A cél egy olyan nyelv konstruálása volt, amelynek segitségével könnyen leirhatók algoritmusok és emellett a legkülönbözőbb számitógépeken biztositható a leirt algoritmusok végrehajtása. A programrészek és eljárások önállóan fordithatók, anélkül, hogy ez a hatékonyság rovására menne.

Az ALGOL 68 elemzésénél célszerü a benne előforduló elemeket más nyelvek /elsősorban az ALGOL 60/ megfelelő elemeivel összehasonlitani, noha ezek nem egyszerü átvétellel kerültek bele. Az ALGOL 68 ugyanis nem valamely korábbi nyelv kiterjesztése, hanem az információ-feldolgozás általános fogalmain alapuló uj nyelv. Ez természetesen módosulást jelent a régebben is használt fogalmaknál és egy sor

uj fogalmat is eredményez.

Az ALGOL 60-ban használt tipus-fogalom helyett /mely korlátozott számu lehetőséget tartalmazott/ bevezették a

tipus általánositásaként tekinthető mód fogalmát. Az alap-módok között tetszőleges hosszuságu valós és egész mód is szerepel. A mód-deklaráció segitségével a már meglévő módokból ujabbakat definiálhatunk, sőt, bizonyos megszoritásokkal rekurziv mód-deklaráció is meg van engedve, amely teljesen uj módhoz vezet. Hasonlóan definiálhatunk strukturákat is, a módok egyesitése /union/ pedig lehetővé teszi, hogy olyan változókat deklarálhassunk, amelyek több különböző módu értékeket felvehetnek. Az ALGOL 68 tartalmaz összetett változókat. A "név" fogalmának bevezetésével olyan módhoz jutunk, amelynek segitségével valamilyen értékre lehet hivatkozni. A hivatkozott érték ismét lehet "név"; ezzel tehát az indirekt cimzés előnyeit el lehet érni. Az eljárás általánositása a rutin, amely egyébként biztositja az összes korábbi lehetőségeket is. A transzput /I/O/ tevékenységgel kapcsolatos szerkesztési feladatokat a forma fogalmának segitségével oldhatjuk meg, tullépve ezzel az ALGOL 60 egyik lényeges hiányosságán.

Az operátorok körének kiterjeszthetőségére a prioritás- és a müvelet-deklarációk adnak lehetőséget. Az ALGOL 60 tipus-, tömb-, kapcsoló- és eljárásdeklarációinak általánositása a változó-deklaráció és az azonosság-deklaráció, amelynek segitségével pl. bármilyen módu konstansok is deklarálhatók és amely egy hatékony paramétermechanizmus alapját képezi.

A memóriahozzárendelésben megtaláljuk az automatikus dinamikus tárkiosztás mellett a programozó által vezérelt hozzárendelés lehetőségét is.

A korabbiaknoz kepest uj elem a parhuzamos feldol-
gozás lehetőségének biztositása az un. szemafor-eljárások segitségével. A szabványos függvények között pedig olyanokat is találunk, amelyekkel az implementáció és a konfiguráció bizonyos tulajdonságai meghatározhatók, valamint az I/O tevékenységek vezérelhetők.

A "zárt clause" magában foglalja az ALGOL 60-ból ismert blokk, összetett utasitás, zárójelezett kifejezés fogalmakat, a "hozzárendelés" /assignation/ pedig az értékadó utasitás fogalmát. Az indexezéssel nemcsak elemeket jelölhetünk ki, hanem résztömböket is, amelyeknek speciális esete a tömb "épitőeleme". A zárójelezéshez hasonlóan az "eset clause" /illetőleg a "feltételes clause"/ a feltételes kifejezés és utasitás általánositása. A ciklusutasitásnál tágabb az "ismétlő utasitás".

Három általános célu nyelvet tekintettünk át. Mindegyik megalkotásánál más és más szempontokat vettek figyelembe. Ennek megfelelően lényeges formai és tartalmi különbségek találhatók közöttük, amelyekre a megfelelő helyeken rámutattunk. Az APL-nél a speciális felhasználási üzemmódot, az ALGOL 68-at a korábbiakat is tartalmazó igen általános fogalmakat kell ismételten hangsulyoznunk.

2.5. A formulavezérlésü számitógépek történetének áttekintése

Jelenlegi munkánk számára a következő definiciót vezetjük be: a formulavezérlésü számitógép belső nyelve egy véges ábécé feletti ugynevezett string-nyelv, amelyet a kérdéses számitógép ugy hajt végre, hogy a program minden egyes alapjele /dinamikus sorrendben/ bekerül a /hagyományos értelemben vett/ utasitásregiszterbe.

A használatos magasabb szintü programozási nyelvek nyilvánvalóan string-nyelvek, definicíónk tehát nem zárja ki pl. a kifejezések forditott lengyel jelölésén alapuló nyelveket sem. Kizárja viszont a hagyományos gépi kódot akkor, ha itt az utasitásoknak csupán a müveleti-kód része kerül az utasitásregiszterbe, egyéb részei pedig a vezérlő egység más regisztereibe jutnak.

A formulavezérlésü számitógépek gondolata az 1950-es évek végén merült fel a számitógépek fejlődéstörténetében. Egymás után születnek formulavezérlésü számitógép-tervek, sőt 1963 óta több terv megvalósitására is sor került: Bauer és Samelson /Bauer, 1957, 1960/ verem-memóriát javasolt aritmetikai és logikai kifejezések értékének közvetlen kiszámitására; javaslatuk jelentősége azonban inkább annak felismerésében állt, hogy a verem-memória -- a hardware és software-megoldások közötti analógiánál fogva -- lehetőséget adott egymenetü /szekvenciális/ kompilációra. Kalmár /Kalmár, 1959, 1960/ egy teljes magas-

szintü programozási nyelvhez, mint belső nyelvhez tervez a hagyományostól eltérő kifejezés-feldolgozó és aritmetikai egységet és változókezelést; Pawlak /Pawlak, 1960, 1961/ formulával megadott függvények kezeléséhez konstruál cimnélküli számitógépet; Anderson /Anderson, 1961/ lé-

nyegében hagyományos aritmetikai-logikai müvel, de ujszerü memóriaszervezéssel tervez ALGOL 60-szerü belső nyelvvel rendelkező számitógépet; Gluskov /Gluskov, 1965, 1971/ és munkatársai 1963-tól kezdik meg a MIR formulavezérlésü számitógép-sorozat megvalósitását; számos, speciális programozási nyelven irt programokat közvetlenül végrehajtó számitógépet terveztek és valósitottak meg /Melbourn, 1965/ /Weber, 1967/ /Bashkow, 1967/ /Hauck, 1968/ /Abrams, 1970/ /Chesley, 1971/ /Ercoli, 1966/.

A következőkben összefoglaljuk az alkalmazott módszerek főbb jellemzőit anélkül, hogy az egyes formulavezérlésü számitógépeket teljes részletességgel ismertetnénk.

A megvalósitott formulavezérlésü számitógépek egy jelentős részében a mikroprogramozási technikát alkalmazzák. Előfordul, hogy már meglévő, mikroprogramozható számitógép mikroprogram-tárát bővitik ki és töltik fel valamely magasabbszintü programozási nyelv interpretátorával. Ezek általában egy "one-pass-load-and-go" kompilátornak felelnek meg: a végrehajtandó programot beolvasás közben /vagy a memóriában elhelyezett program egyszeri átfutása során/ transzformálják egy egszerübb strukturáju program-

má /azonositó-táblázatok összeállitása, szintaktikai elemzés végrehajtása, sztatikus memóriafelosztás elvégzése, összetett alapjelek felismerése stb./, amelyet

ezután közvetlenül hajtanak végre. Mind a transzformáció, mind pedig a végrehajtás a mikroprogramok feladata.

Még abban az esetben is, ha a számitógépet magasabb szintü programozási nyelvvel, mint belső nyelvvel tervezik, jellemző a hagyományos processzor megtartása: a két operandusz számára két regiszter szolgál, amelyek feltöltése, illetve a müvelet eredményének tárolása cim-regiszterek segitségével valósul meg. Jellemző továbbá, hogy a prccesszor feladatává válik a különböző tipusu operanduszok tipus-kezelése illetve konvertálása: a vezérlőegység, tipusra való tekintet nélkül, adja át az operanduszokat a processzornak /ezáltal a szintaktikus elemzés egyszerüsödik/, az operanduszok pedig un. "tag"-et is tartalmaznak az információs bitek mellett, amelyek a processzort tájékoztatják arról, hogy az operandusz integer, real, long real stb. Ezzel egyidejüleg megfigyelhető a tipusok bővülése: a hagyományos /pl. szám--jellegü/ operanduszok mellett a processzor kontrollinformációkat is kezel, amelyek összetett adatstrukturák elemeire /pl. indexes változókra/ utaló szavak. A kontrollszavak és a hozzájuk rendelt adatelemek azonositása szintén a processzor feladatai közé tartozik.

A formulavezérlésü számitógépek memóriája több rész-

re tagozódik, amelyek egyrészt szerepükben, másrészt a vezérlőegységhez való viszonyukban különböznek egymástól. Legtöbbször azonban, az aktuális programtól függően, egy homogén jellegü memóriát osztanak fel

megfelelő részekre.

Strukturájában a program-terület a legegyszerübb: a program alapjeleit tartalmazza vektorkomponensként. Az aktuális feldolgozandó jel cimét az utasitásszámláló regiszter tartalmazza, s ez a tartalom általában egyesével növekszik. Ettől csak a vezérlésátadó és ciklusutasitások feldolgozása során tér el a gép, mégpedig ugy, hogy előre történő /még fel nem dolgozott cimkéhez tartozó/ vezérlésátadás céljára van egy un. kereső-mód, amelyben végigvizsgálja a program alapjeleit, keresve a kivánt cimkét, a jeleket azonban /a közben talált cimkék feldolgozásán kivül/ nem hajtja végre. Nincs szükség kereső üzemmódra, ha a gép végrehajtás megkezdése előtt teljes cimketáblázatot előállit, ekkor ugyanis a táblázatból ki tudja olvasni a cimke programterületbeli cimét.

A másik lényeges memória-terület az azonosito-táblázat. Ennek segitségével asszociálható a programban használt változónevekhez az érték-memória azon mezőjének cime, amely a változó értékét tartalmazza. Összetett adatstrukturák /tömbök, stringek, listák/ neveihez a táblázat annak a kontroll-szónak a cimét rendeli hozzá, amely a struktura-elemek cimének kiszámitásához szükséges, és amely általában a deklaráció végrehajtásakor alakul ki.

/Vektor esetében pl. a kontroll-szó a vektor első elemének érték-memóriabeli cimét, az index alsó és felső határát és a vektor-elemek hosszát tartalmazza./

Ha a változó-értékek nem tartalmaznak "tag"-et,

azaz tipus-jelölést, akkor a változótáblázat az azonositók tipusát is közli /kivéve, ha az azonositó implicit módon egyuttal a tipusra is utal, vagy ha a nyelv csak egyfajta tipust ismer/.

Használnak még segédmemóriaként LIFO memóriákat is. Ezek elsősorban a kifejezések feldolgozása során szükségesek; az utasitásregiszterbe már bekerült, de azonnal végre nem hajtható jelek tárolására szolgál az operátor- illetve a operandusz-stack a hozzájuk tartozó mutatókkal. Gyakori megoldás, hogy az aritmetikai regiszterek szerepét az operandusz-stack két utoljára feltöltött mezője játssza. Általánosan elterjedt a számitógép állapot-stackjének használata: ez tartalmazza pl. a dinamikus blokk--mélységről az információkat, segit a ciklusok, rekurziv eljáráshivások multiprogramozásos kezelésében, stb. Ezek a LIFO-memóriák részben a processzor, részben a vezérlőegység részeinek tekinthetők, bár legtöbbször a főmemóriához tartoznak.

3. Javaslatok a KALMÁR-féle formulavezérlésü számitógép korszerü alakjának megtervezésére, különös tekintettel a belső nyelvre

3.1. Bevezető megjegyzések

Az alábbiakban a KALMÁR-féle formulavezérlésü számitógép /KALMÁR, 1959; lásd még KALMÁR, 1965/ egy korszerü alakjának /lásd AZ ELEKTRONIKUS, DIGITÁLIS SZÁMITÓGÉPEK EDDIGI FEJLÖDÉSE ÉS A VÁRHATÓ FEJLÖDÉS FŐ IRÁNYAI, 1972/ megvalósitására vonatkozó javaslatokat ismertetünk, különös tekintettel a gép belső nyelvének megtervezésére. Az itt következő elgondolásokban -- mint ahogy eddig követett tárgyalásmódunknak is megfelel -- elsőrendüen a software-követelményeket vettük figyelembe. Nem mellőzhetünk azonban néhány utalást a hardware-re, mint a software megvalósitásának eszközére sem.

A kérdéses formulavezérlésü számitógépről tartott első előadás idejében, 1959-ben, még nagyon általános volt, legalábbis nálunk, a gép belső nyelvén /gépi kódban/ történő programirás. Bár ekkor már ismeretes volt néhány magasabb szintü programozási nyelv is /a LJAPUNOV-féle operációs programozási módszer nyelvén és a FORTRAN régi változatán kivül már az ALGOL első változata is, amelyet később ALGOL 58 néven említettek/ és ezekre támaszkodva több helyen megindult forditóprogramok irása is, abban az időben egy kompilátor megirását nagyon nehéz, 10-20, sőt gyakran több szakember együttes munkáját legalább egy éven át igénybe vevő feladatnak tekintették. Ezért akkor a formulavezérlésü számitógép megvalósitásának célja az lett volna, hogy ha már egyszer a gép belső nyelvén

való programozás az általános, legyen ez a belső nyelv minél magasabb szintü és igy a gép forditóprogram nélkül müködni tudjon egy magasabb szintü programozási nyelven megirt program alapján. Ennek megfelelően az akkori tervek egynyelvü gépnek képzelték el a formulavezérlésü számitógépet. Ma azonban már,

mint az előzőkben is említettük, általánossá vált a magasabb szintü algoritmikus nyelveken történő programirás, és ma már különböző alkalmazásokra nagyon sok ilyen algoritmikus nyelvet fejlesztettek ki. Emellett különböző software és hardware segédeszközök /pl. a beépitett stack, a rendszerprogramiró nyelvek kifejlődése és implementálása/ alakultak ki, emiatt ma már nem számit nehéz feladatnak egy kompilátor megirása. Ennélfogva a korszerü számitógépeknek többnyelvüeknek kell lenniök, amelyekben az alkalmazások szempontjából legfontosabb algoritmikus nyelvek implementálva vannak. Ennélfogva egy korszerü formulavezérlésü számitógép megvalósitásának csak az lehet a célja, hogy még tovább megkönnyitse a kompilátor-irást -- ennélfogva a kompilátor-irás automatizálását is rendszerprogramiró nyelvek felhasználásával -- arra támaszkodva, hogy a gép belső nyelve sokkal közelebb áll az algoritmikus forrásnyelvekhez, mint a szokásos számitógépek gépi kódja.

Egy formulavezérlésü számitógép müszaki megvalósitása természetesen lényegesen nagyobb ráforditásokat kiván, mint egy szokásos számitógépé. Igy többek között nagyszámu regisztert igényel a kifejezések, vagy a ciklusutasitások feldolgozása. Ma azonban viszonylag olcsón be lehet szerezni gyorsmüködésü LSI-regisztereket és rendelkezésünkre áll a korszerü mikroprogramozási technika. Ezért ma könnyebben megvalósitha-

tónak látszik egy korszerü formulavezérlésü számitógép, mint az 1960-as években egy az akkori szinvonalnak megfelelő formulavezérlésü számitógép.

Azt talán említeni sem kell, hogy egy korszerü formulavezérlésü számitógépnek megfelelő, többszintü megszakitási

rendszerrel és erre támaszkodva kidolgozott operációs rend-

szerrel is rendelkeznie kell.

Ami egy korszerü formulavezérlésü számitógép alkalmazásait illeti, e tekintetben épp a nagyobb ráforditások miatt minél nagyobb mértékü univerzalitásra kell törekedni. Ez azt jelenti, hogy egy korszerü formulavezérlésü számitógépnek alkalmasnak kell lennie nemcsak tudományos-müszaki és adatfeldolgozási-gazdasági feladatok megoldására, hanem pl. folyamatirányitási, szöveg-, lista-, fa- és általánosabban gráfkezelési, rendszerprogramozási stb. feladatok megoldására is. Ez természetesen a sokféle alkalmazási területnek megfelelően sokféle, azonban egymással bizonyos mértékben közlekedő és közös részeket is tartalmazó mikroprogram-tárat igényel, hiszen különböző alkalmazási területeken egyrészt különböző adatstrukturákon, különböző müveleteket kell a gépnek végeznie, másrészt vannak olyan müveletek /pl. az aritmetikai alapmüveletek, a döntésekhez szükséges feltételek logikai értékének kiszámitása stb./, amelyek a különböző alkalmazási területeken egyaránt előfordulnak. Természetesen az egyik mikroprogramtárról a másikra való áttérésnek minél egyszerübbnek kell lennie, tehát nem a mikroprogramtár megváltoztatása, hanem a kész mikroprogramtárak közötti átkapcsolás utján kell azt végrehajtani.

Az emlitett szempontok, valamint az a célkitüzés, hogy a kompilátorirás megkönnyitése végett a gép belső nyelvének egyaránt közel kell állnia valamennyi használatos algoritmikus nyelvhez, szinte egyértelmüen előirja, hogy a korszerü formulavezérlésü számitógép belső nyelvének azt kell tartal-

maznia, ami a különböző algoritmikus nyelvekben közös, ez pedig elsősorban a kifejezések és formulák használata, e fogalmakat a matematikában és alkalmazásaiban az évezredek során kialakult legáltalánosabb értelemben véve. Közismert, hogy a matematika a maga formulanyelvét nemcsak numerikus számitási célokra, hanem a legkülönbözőbb, akár nagy absztraktságu elméleti célokra is eredményesen tudja alkalmazni, különösen amióta a matematikai logikai tipuselmélet utat mutatott az időszakosan /pl. a halmazelméletben/ felmerült logikai nehézségek kiküszöbölésére. Előrelátható tehát, hogy egy ilyen általános értelemben vett formulanyelv a legkülönbözőbb alkalmazások területén is megállja a helyét, hiszen minden alkalmazásban konstans és változó adatokból müveletek és függvények segitségével felépitett /feltétlen és/vagy feltételes/ kifejezésekkel leirható tevékenységekről van szó, ha a konstans, a változó, a müvelet és a függvény fogalmát a legáltalánosabb értelemben értjük. A sokféle tipus pedig nemcsak rugalmasságot biztosit a formulanyelvnek, hanem alkalmas a sokféle adatstruktura tükrözésére is.

Természetesen felmerülhet ezzel kapcsolatban két aggály, amelyeket előre el kell oszlatni. Vajon a sokféle adatstruktúrán végzett sokféle müvelet nem kiván-e a formula-irásmódban ijesztően sokféle müveleti jelet? És vajon lehet-e olyan mü-

szaki strukturát adni egyetlen számitógépnek, amellyel a sokféle müveleti jeleket tartalmazó, sokféleképpen értendő kifejezéseket egyaránt képes feldolgozni?

Az első kérdésre azt lehet válaszolni, hogy egyáltalán nem szükséges, hogy minden müvelet más müveleti jelet kapjon,

hiszen ugyanazon müveleti jel különböző műveletet jelölhet aszerint, hogy egy vagy két operandusza van-e és aszerint, hogy ezek az operanduszok milyen tipusuak. Pl: a + jel numerikus számitásokban jelölheti a két operandusz összeadását /egyébként más-más algoritmus szerint annak megfelelően, hogy rögzitett vesszős egész vagy lebegő vesszős valós operanduszokon végezzük-e ezt a müveletet/, de jelölheti pl. digitális technikai alkalmazásokban a logikai összeadást /diszjunkciót/, amit a mérnök szivesen jelöl + jellel, amikoris az operanduszai logikai értékek /bár a matematikai logikának külön müveleti jele is van a diszjunkcióra/; string-operanduszok esetén lehet ugyanakkor a + jel a konkatenáció jele is, stb. Hiszen a matematikus is szinte a matematikának minden ágában mást-mást nevez pl. összeadásnak és szorzásnak és jelöl az aritmetikai összeadáshoz és szorzáshoz hasonló módon. A lényeges csak az, hogy az operanduszok tipusa adott mikroprogramtár esetén egyértelmüen meghatározza a müvelet jelentését /és ezzel a müvelet eredményének tipusát is/. Emellett arra is hivatkozhatunk, hogy az APL nyelvet alkalmazó programozók rövid idő alatt megszokták a 21 különbö müveleti jel alkalmazását /ezenkivül 13 egytagu müveleti jelét is, amelyek csak részben esnek egybe a kéttagu müveleti jelekkel/. Meg arra is, hogy a kereskedésben kapható APL bemenő egységek /alfanumerikus megjelenitők/ lehetővé teszik e jelek kombinálását is és ha még ez sem volna elég, azonositóval

is lehet müveleti jelet jelölni /mint ahogy függvény jelölésére már régóta használ a matematikus olyan azonositókat, mint pl. sin, cos, ln stb./, hiszen adott tipusu operanduszokon végzett adott tipusu eredményt szolgáltató müvelet maga is egy bizonyos /az operanduszok és az eredmény tipusa által meghatározott/ tipusu fogalom és természetes, hogy azonositókkal

jelölhetünk mindenféle tipusu konstanst vagy változót.

A második kérdésre pedig azt válaszolhatjuk, hogy éppen a KALMÁR-féle formulavezérlésü számitógépnek a régebbi elgondolások szerint is fontos szerepet játszó kifejezésfeldolgozó egysége az alkalmas eszköz mindenféle kifejezés szerkezetének elemzésére, a benne szereplő müveletek jelentésétől függetlenül. /A kifejezésfeldolgozó egység természetesen nem végzi el a müveleteket, csak előkésziti a müveletek elvégzését az aritmetikai egység -- a korszerü számitógépek esetén a mikroprogramtár -- által; ilyen szempontból inkább a vezérlőegység részének kellene tekinteni, mint az aritmetikai egység részének./ Ez annak köszönhető, hogy regiszternégyes-hierarchiás szerkezete a szokásos, zárójeleket is alkalmazó kifejezések szerkezetét tükrözi /a bal-operandusz, müveleti jel, jobb-operancusz sorrenddel és az egyes zárójel-mélységeknek megfelelő, egy-egy regiszternégyest tartalmazó "emeletek" között a kifejezésben előforduló zárójeleknek megfelelő összeköttetések létesitésének lehetőségével, amelyek révén a sorrendben negyedikként -- az aritmetikai egységtől, illetőleg a mikroprogramtártól -- tartalmat átvevő eredményregiszter tartalma eggyel alacsonyabb "emeleten" operanduszként /az előzőleg létesitett összeköttetésnek megfelelően bal- vagy jobb-operanduszként/ kerülhet a megfelelő regiszterbe, vagy gazdaságosabb müszaki megvalósi-

tás esetén az eggyel magasabb "emelet" eredményregisztere helyett egyenesen oda az aritmetikai egységből, illetőleg a mikroprogramtárból/. Ez a körülmény éppen a KALMÁR-féle terv alapján megvalósitandó formulavezérlésü számitógépet teszi alkalmassá arra, hogy sokféle alkalmazási területen egyaránt megállja a helyét.

Nem akarjuk elhallgatni, hogy az emlitett kifejezésfeldolgozó egység leghivebben a teljes zárójelezés esetén -- amikor minden, ujabb müvelet operanduszaként felhasználandó /kijelölt/ müveleti eredményt zárójelbe kell tenni -- tükrözi a kifejezések szerkezetét. Azonban minden nehézség nélkül felhasználható annál a prioritás-nélküli jelölésrendszernél is, amely a bal-asszociáción alapul, amelynél tehát a müveleteket szekvenciálisan balról-jobbra végezzük el és csak az e sorrendtől való eltérés kivánságát kell zárójellel kijelölni. Ezt a jelölésrendszert semmivel sem nehezebb megszokni, mint az APL ehhez képest pontosan forditott /jobb-asszociáción alapuló/ jelölésrendszerét, sőt, könnyebb megszokni, hiszen a bal-asszociáció jobban megfelel az iskolától kezdve megszokott sorrendnek. /Pl. a - b + c minden matematikus és a matematika minden alkalmazója számára a - b és c összegét jelenti, ezzel szemben az APL-ben a és b + c különbségét./ Egyébként azonban annak sincs akadálya, hogy más-más mikroprogramtár segitségével, előzetes erre vonatkozó jelzés adása után, akár a bal-asszociáción alapuló jelölésrendszert /pl. az ehhez már hozzászokott fejlett programozók számára, valamint a kompilátorok tárgynyelve gyanánt/, akár a szokásos, a müveletek prioritás-sorrendjén alapuló jelölésrendszert /pl. az ezt előnyben részesitő, a gép belső nyel-

vén programozni kivánó programozók számára/ lehessen alkalmazni. Az ilyen, "jelölésrendszer-kiválasztó" mikroprogramtárak szerepe természetesen más, mint a fentemlitett, az aritmetikai egység szerepét átvevő mikroprogramtáraknak. Ez utóbbiakat "alkalmazási terület-kiválasztó" mikroprogramtáraknak nevezhetjük, mert az alkalmazási területtől /is/ függ/het/, hogy melyik müveleti jellel jelölt müveletet hogyan kell végrehajtani.

Egyidejüleg egy-egy mikroprogramtárnak kell bekapcsolt állapotban lennie mindkét fajtából ahhoz, hogy a gép müködése egyértelmüen meg legyen határozva; az átkapcsolás más mikroprogramtár/ak/ra a program e célra szolgáló /eljárás-/utasitása végrehajtásaként történhetik, célszerüen megszakitás alkalmazása nélkül, hogy a gép operációs rendszere minél egyszerübb legyen. Ilyen utasitás adhatja azt a jelzést, hogy a következő utasitások már másik jelölésrendszerben értendők, pl. mert azokat más jelölésrendszert használó programozó irta, ill. hogy bennük a müveleti jelek mást jelentenek pl. mert folyamatirányitási programon belül numerikus számitás kezdődik. /Jelölésrendszeren itt pl. bal-asszociáción, vagy a müveletek valamely adott prioritás-sorrendjén alapuló jelölésrendszert értjük; eljárás-utasitáson a későbbiekben megadandó szintaxis értelmében vett parancs tipusu függvénykifejezéseket értünk./

A KALMÁR-féle formulavezérlésü számitógép korszerü alakjának belső nyelve gyanánt ennélfogva egy nagyon általános értelemben vett, elvileg akárhány tipust megengedő formulanyelvet javasolunk, amelynek szintaxisa független a számitógép alkalmazási területétől, de amely a különböző alkalmazási területek számára különböző szemantikával látható el. /A szeman-

tika, nevezetesen az egyes müveletek és függvények jelentése, valamint a kifejezésekben szereplő müveletek végrehajtásának sorrendje, az éppen bekapcsolt állapotban levő mikroprogramtáraktól függ, a szintaxist a gép többi részének szerkezete tükrözi./

A gép több mikroprogramtáras rendszere alapját képezheti a gép egy olyan továbbfejlesztésének, amelyben multiprocesz-

szor üzemmódban, valamint párhuzamos üzemmódban is felhasználható.

3.2. A szintaxisban használt jelölésmódok

A formulavezérlésü számitógép fentieknek megfelelő belső nyelvének szintaxisát a Függelék tartalmazza. A szemantika kidolgozása később fog megtörténni az egyes alkalmazási területek igényeinek megfelelően.

A szintaxis leirásában az ALGOL 68 algoritmikus nyelvnek -- amely az első olyan nyelv volt a programozási nyelvek között, amely elvileg akárhány tipust, ALGOL 68 terminológiával módot, megengedett -- szintaktikus leirásában használt jelölésmódot alkalmazzuk. Ez a jelölésmód a nyelv absztrakt szintaxisának leirását két szintre /metaszintaxis és tulajdonképpeni szintaxis/ bontja; a harmadik szint volna a konkrét reprezentációhoz szükséges szabályok, un. reprezentációs szabályok megadása. Ez utóbbiak ugyanazon nyelv esetén is függhetnek annak a gépnek a jelkészletétől, amelyen a nyelvet implementálni szándékozunk. A reprezentációs szabályoktól függetlenül ugy lehet a nyelv szintaxisát megadni, hogy a nyelv mindenegyes alapjele helyett annak nevét használjuk, pl. a + jel helyett mindig azt irjuk, hogy plusz; ezek az alapjel-nevek éppen olyan szintaktikus fo-

galmai a nyelvnek, mint a többiek, azzal a különbséggel, hogy pl. a program fogalmának több, valójában végtelen sok terminális kifejtése van /végtelen sok program van/, ezzel szemben csak egy plusz jel van. /Az ALGOL 68 jelentés megengedi, hogy egy és ugyanazon alapjel-névnek többféle reprezentációja is lehessen egyidejüleg, vagyis a programozó akár ugyanabban a programban is különféleképpen irhassa azt; ez azonban a formula-

vezérlésü számitógép belső nyelve szempontjából csak felesleges bonyodalmat jelentene./ A terminális ábécé szerepét tehát azok az un. terminális fogalmak játsszák, amelyekre meg kell adni reprezentációs szabályt, mig azoknak a fogalmaknak összessége, amelyekre van a tulajdonképpen szintaxisban legalább egy produkciós szabály -- ezeket produktiv fogalmaknak nevezzük --, játssza a nemterminális ábécé szerepét. Előfordulhat olyan nemproduktiv fogalom is, amelyre nincs megadva reprezentációs szabály; ilyennek fellépte egy produktiv fogalom iterált kifejtése során a produkciós szabályok alkalmazásával "zsákutcát" jelent, amelynek mentén nem lehet terminális kifejtésig jutni, épp ezért az ilyen fogalmakat zsákutca-fogalmaknak nevezzük. /Az ALGOL 68 ezeket nem is tekinti fogalmaknak, hanem protofogalmaknak nevezi a produktiv és a terminális fogalmakkal együtt./

A formulavezérlésü számitógépnek, annak megfelelően, hogy elvileg akárhány tipust megenged, elvileg akárhány produktiv fogalma lehet, hasonlóan az ALGOL 68 nyelvhez. Ezekre tehát végtelen sok produkciós szabályt kellene megadni. Ez természetesen lehetetlen. Az igy keletkező ellentmondás áthidalására szolgál a metaszintaxis, amely egy másik nyelvnek, az un. metanyelvnek szintaxisa. A metanyelv kizárólag arra szolgál, hogy véges számu un. előszabály alapján /az ALGOL 68 leirása ezeket hiperszabá-

lyoknak nevezi/ generálja a nyelv tulajdonképpeni szintaxisának végtelen sok produkciós szabályát. A metanyelvnek már csak véges számu produktiv fogalma van, ezeket metafogalmaknak nevezzük, és ezekre véges számu produkciós szabálya, az un. metaszabályok. /Az ALGOL 68 jelentés a metanyelvvel kapcsolatban metaprodukcióról beszél produkció helyett./ A metafogalmak összessége játssza a metanyelv nemterminális ábécéjének szerepét;

terminális abécéjének szerepét viszont bizonyos véges számú fo-
galom játssza, amelyek konkatenációjaként a tulajdonképpeni
nyelv /a nem-metanyelv/ minden fogalma megkapható.

A metafogalmakat csupa nagybetüvel irt szavak képviselik; ezeket akkor is egybeirjuk, ha olyan összetett szóról van szó, amelyet nem szokás egybeirni. Ez lehetővé teszi a szóköznek alkalmazását konkatenáció jeleként, anélkül, hogy mint az ALGOL 60 jelentésben a fogalmakat, csucsos zárójelbe kellene zárnunk a metafogalmakat. Attól is eltekintünk, hogy -- amint ezt az ALGOL 68 jelentés erősen hangsulyozza -- elvi okokból különbséget kell tennünk azon nagy /és később kis/ betük között, amelyeket a metafogalmak /illetőleg később a fogalmak/ összeállitására használunk /ezeket az ALGOL 68 jelentés nagy, illetőleg kis szintaktikus jeleknek nevezi/, és a jelen tanulmányban /illetőleg az ALGOL 68 jelentésben/ használt nagy és kis betük között, valamint természetesen az alapjelek között /amelyeket a reprezentációs szabályok adnak meg/ szereplő nagy és kis betük között is.

A fogalmakat /a produktiv, a terminális és a zsákutca-fogalmakat/ csupa kisbetüvel, egy vagy több szóba irt szövegek képviselik. Az egybe- vagy különirás tekintetében ezeknél már igyekeztünk a helyesirási szokásokat követni. Elvileg lehetne a fogalmakat is mind egybeirni, igy azonban nagyon áttekinthe-

tetlen lenne a szintaxis és tetszőlegesen hosszu szavak is létrejönnének. A különirás elsősorban az áttekinthetőséget hivatott szolgálni. A fogalom részei /szavai/ közötti szóközöket azonban nem tekintjük szignifikánsaknak, vagyis két olyan fogalmat, amelyek ugyanazokból a betükből ugyanolyan sorrendben jönnek létre, akkor is azonosnak tekintünk, ha az egyikben máshol vannak a szóközök, mint a másikban. Ezt a tényt ki is fogjuk használni.

Viszont a különirás megengedése /és az a körülmény, hogy a

fogalmakat sem akarjuk csucsos zárójelek vagy más hasonló jelek közé zárni/ szükségessé teszi külön /a szóköztől különböző/ konkatenációs jel használatát. Ilyen jel gyanánt a vesszőt /,/ használjuk, amelyet ezért elvileg meg kell különböztetnünk a vessző-nek nevezett fogalomtól, akkor is, ha ennek majd a két sorral feljebb zárójelbe tett irásjel lesz a reprezentánsa. /Ugyanez vonatkozik a pontosvesszőre, a kettőspontra és a pontra is./ Tehát pl. index kifejezés ugyanazt a fogalmat jelöli, mint indexkifejezés, viszont index, kifejezés már két különböző fogalom, az index és a kifejezés /egy-egy speciális esetnek/ konkatenációját jelöli. A konkatenációs vesszők természetesen a kész programban már nem szerepelnek, a reprezentációs szabályok alkalmazásával egyidejüleg kerülnek elhagyásra /és egyuttal végrehajtásra/. A metaszabályokban nem szerepelnek konkatenációs vesszők, csak a tulajdonképpeni szintaxis produkciós szabályaiban, amelyeket röviden szabályoknak nevezünk, valamint az előszabályokban, amelyekből a szabályok keletkeznek. Az a véges számu fogalom, amelyek összessége játssza a metanyelv terminális ábécéjének szerepét, mind egybeirt, csupa kisbetüből álló szó.

Az ALGOL 60 jelentésben használt ::= jel helyett, amely ott a produkciós szabályok bal- és jobboldalának elválasztására

szolgál, a metaszabályokban a :: jelet, a szabályokban és az előszabályokban a : jelet használjuk, hasonlóan, mint az ALGOL 68-ról szóló revideált jelentésben. Az ALGOL 60 jelentésben a produkciós szabályok jobboldalán álló alternativák elválasztására használt | jel helyett mind a metaszabályokban, mind az előszabályokban a pontosvesszőt /;/ használjuk. A szabályokban

már nem szerepel ez a jel. Mind a metaszbályok, mind az elő-szabályok és a szabályok végét pont /./ jelzi.

Eszerint a metaszabályokban a :: jel baloldalán egy metafogalom áll, jobboldalán pedig egy vagy több alternativa /az ALGOL 68 jelentés ezeket metaalternativáknak nevezi/, amennyiben több, ezek ;-vel vannak elválasztva. Az egyes alternatívák metafogalmak és egyszavas fogalmak szóközökkel elválasztott sorozatai. Ezeket az alternatívákat a kérdéses metaszabály baloldalán álló metafogalom közvetlen kifejezéseinek nevezzük. Belőlük általában ugy kapjuk e metafogalom kifejtéseit, hogy valamely már megkapott kifejtésben, amennyiben még szerepel benne metafogalom, ennek helyébe valamely közvetlen kifejtését tesszük. Ha ezt az eljárást addig ismételjük, amig olyan kifejtésekhez nem jutunk, amelyekben már nem szerepel metafogalom, akkor megkapjuk a kiindulásul választott metafogalom terminális kifejtéseit, más néven speciális eseteit. Ezek mindegyike tehát egy-egy fogalom /egy vagy több szóba irva/.

Az előszabályokban a : jel baloldalán is, jobboldalán is egy-egy metafogalmakból és egybeirt fogalmakból álló, a baloldalán szóközökkel, a jobboldalán szóközökkel, pontosvesszőkkel /;/ és/vagy vesszőkkel /,/ elválasztott sorozat áll. A jobboldalt ugy is lehet tekinteni, mint egy vagy több alternativa sorozatát, amennyiben ez több alternativából áll, ;-kkel elválasztva, ahol minden alternativa egy vagy több tagból áll, amennyiben egynél többől, vesszőkkel /,/ elválasztva és egy-egy tag metafogalmakból és/vagy fogalmakból álló sorozat, szóközökkel elválasztva.

Előszabályból ugy jönnek létre szabályok, hogy mindenekelőtt kiválasztunk egyet a jobboldalon álló alternativák kö-

zül, egyedül ezt hagyjuk meg a jobboldalon, a baloldalt pedig változatlanul hagyjuk. Majd az igy keletkező ; nélküli előszabályból /un. egyszerü előszabályból/ ugy készitünk szabályt, hogy mindenegyes benne előforduló metafogalom helyébe valamelyik speciális esetét tesszük, ugyanazon metafogalom helyébe, ha többször is előfordul, akár a : ugyanazon az oldalán, akár különböző oldalain, mindig ugyanazt a speciális esetét. Amennyiben azonban egy metafogalom végén áll egy vele egybeirt számjegy, akkor e metafogalom különböző előfordulásait az egyszerü előszabályban akkor és csakis akkor tekintjük ugyanazon metafogalom előfordulásainak, ha mindegyiknek ugyanaz a számjegy áll a végén /vagy egyiknek sem áll a végén számjegy/.

Az igy keletkező szabályok baloldalán egy fogalom, jobboldalán egy vagy több fogalom áll, amennyiben több, konkatenációs ,-vel elválasztva. A jobboldalt a baloldal közvetlen kifejtésének nevezzük. Többi kifejtéseit ugy kapjuk meg, hogy már egy megkapott kifejtésében valamely produktiv fogalmat valamely közvetlen kifejtésével pótolunk. /A produktiv fogalmakra már természetesen nincs olyan megkötés, hogy ugyanazon produktiv fogalmat, ha több helyen szerepel a jobboldalon, ugyanazzal a közvetlen kifejtésével kellene pótolni; a metafogalmakra sincs ilyen kikötés, ha a metaszabályok jobboldalán szerepelnek. Az a fenti kikötés, amely szerint ugyanazon metafogalom helyébe egy egy-

szerü előszabályon belül mindenütt ugyanazt a speciális esetét kell helyettesiteni, csak az egyszerü előszabályokban szereplő metafogalmak esetére vonatkozik./ Ha ezt a lépést addig ismételjük, amig a jobboldalon már egy produktiv fogalom sem szerepel, hanem csupa terminális fogalom, akkor a szabály baloldalának terminális kifejtéséhez jutunk. Ha a kifejtés során a jobboldalon zsákutca-fogalom jelenik meg /legalább egy, az esetleges produktiv és/vagy terminális fogalmaktól ,-vel elválasztva/, akkor felesleges tovább folytatnunk a produktiv fo-

galmak emlitett pótlását közvetlen kifejtésekkel, hiszen mindenképpen zsákutcába jutunk.

Azt, hogy egy nem-produktiv fogalom terminális-e, vagy zsákutca-fogalom, csak a reprezentációs szabályok ismeretében lehet majd eldönteni. Minthogy mi nem adunk meg a függelékben reprezentációs szabályokat, hiszen ezek attól is függnek, hogy milyen jelkészlete lesz a formulavezérlésü számitógéphez használandó adatelőkészitő berendezéseknek, egyelőre az is függőben marad, hogy melyek a zsákutca-fogalmak /az ALGOL 68-ban a terminális fogalmakat a symbol végződés különbözteti meg a zsákutca-fogalmaktól, mi azonban ilyen megkülönböztető jelet nem alkalmazunk/. A legtöbb nem-produktiv fogalom, amely egyáltalán előfordul valamely szabály jobboldalán, terminális fogalom lesz; de az pl. bizonyos, hogy valós operanduszu valós értékü müvelet lánc eleme zsákutca-fogalom, mert valós operanduszu valós értékü müvelet olyan fogalom, amely ugyan a TIPUS metafogalomnak speciális esete, de az ALAPTIPUS metafogalomnak nem. Avégett, hogy az olvasó kapjon némi tájékoztatást arról, hogy egyes fogalmak terminális fogalmak lesznek, a metaszabályokban és az előszabályokban szereplő terminális fogalmakat a Függelék szövegében /nem a cimekben!/ aláhuzással jelöljük

meg minden helyen, ahol előfordulnak; ezeken kivül természetesen egyéb /a kifejtés során megjelenő/ terminális fogalmak is lesznek. Szaggatott aláhuzás viszont a Függelék szövegében vagy olyan fogalmat jelöl első előfordulásakor, amelyre még meg kell adni egy olyan előszabályt, amelyben a : után minden alternativa terminális fogalom /nevezetesen fel kell még sorolni az összes szabványos müveleti jelet és az összes fenntartott azonositót/, vagy olyan, metafogalom és fogalom konkatenációjával előállt tagot vagy alternativát jelöl, amelyből a benne

javai előállit tagot vagy alternativat jelöl, amelyből a benne szereplő metafogalomnak bármely speciális esetével való helyettesitése utján vagy mindig terminális fogalom lesz /pl. FAJTA kezdőzárójel-ből a karakter kezdőzárójel, atom kezdőzárójel, tömb kezdőzárójel stb. terminális fogalmak/, vagy pedig olyan fogalom lesz, amely vagy zsákutca-fogalom, vagy olyan fogalom, amelyre szintén még meg kell adni olyan előszabályt, amelyben a : után minden alternativa terminális fogalom. Ez utóbbi egyedül TIPUS ábécé betüje esetén fordul elő: egyes tipusokhoz egyáltalában nem tartozik speciális ábécé, pl. ha cimke ilyen tipus, akkor cimke ábécé betüje zsákutca-fogalom; más tipusok esetén pedig még fel kell sorolni a hozzájuk tartozó speciális ábécé betüit. Néhány fontosabb tipushoz ugyanis célszerü bevezetni egy-egy speciális ábécét. Ennek betüit /egybetüs azonositóként/ deklaráció nélkül lehet használni a kérdéses tipusu skaláris változók gyanánt. Ezt az, a formulavezérlésü számitógép realizációjában követhető eljárás indokolja, hogy minden egyes ilyen betü az általa jelölt változó mindenkori aktuális értékét tartalmazó /célszerüen szupergyors hozzáférésü/ memóriahelyre utal. Ennélfogva nincs szükség arra, hogy az ilyen egybetüs azonositókat, a többi azonositókhoz hasonlóan, olyan asszociációs táblázatban helyezzük el, amely hozzárendelné tipusát és az értékét tartalmazó /vagy bonyolult tipus esetén arra utaló/ memória-cimet. Ugyanazt a speciális ábécét az egyes alkalmazási terület-kiválasztó mikroprogramtárak más-más tipushoz rendelhetik hozzá. Speciális ábécé betüivel jelölhetjük ezeken kivül a "program-vektort" és egyes olyan regiszterek tartalmát is, amelyek esetében célszerü, hogy tartalmukat a programban közvetlenül felhasználhassuk és/vagy a program segitségével megváltoztathassuk.

Metafogalmak: általános konst-

/Előzetesen lásd a függelék Metafogalmak, definíciós konstrukciók c. részét./

A metafogalmak közül a TIPUS a legfontosabb, ennek végtelen sok fogalom speciális esete, ezek a formulavezérlésü számitógép belső nyelvében elvileg szerepelhető tipusok. Ezek az alaptipusokból /az ALAPTIPUS metafogalom speciális eseteiből/ kiindulva, részben müvelet- és függvény-tipusok ismételt képzésével jönnek létre, ahol az ilymódon képezett uj tipusokat az operandusz/ok/nak, illetőleg az argumentum/ok/nak, valamint az értéknek /vagyis a müvelet eredményének, illetőleg a függvény értékének/ tipusa határozzák meg /ezek a paraméteres tipusok/; részben összetételek /vagyis az ÖSSZETÉTEL metafogalom speciális esetei/, ezenkivül még a lista is tipus. Az alaptipusok részben szövegszerüek, ilyenek a valós, egész, parancs és cimke tipusok, részben nem, ilyenek a karakter és a logikai tipusok. Ennek a megkülönböztetésnek az a szerepe, hogy szövegszerü tipusokhoz tartozó jelölésekből /amelyek ilyen tipusok egy-egy speciális esetét jelölik/ vessző, mint elválasztójel segitségével képezünk majd láncokat, mig karakterlánc és logikai lánc esetén nem kell vesszőt alkalmazni.

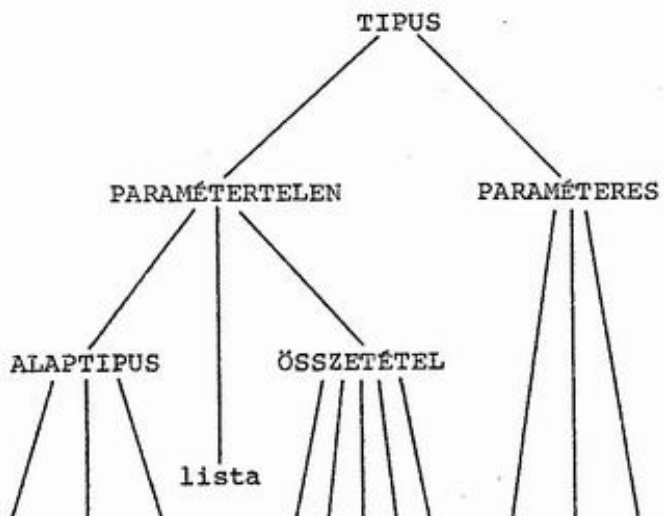
A parancs tipus felvételét a tipusok közé az indokolja, hogy ez lehetővé teszi, hogy a program végrehajtása során meg-

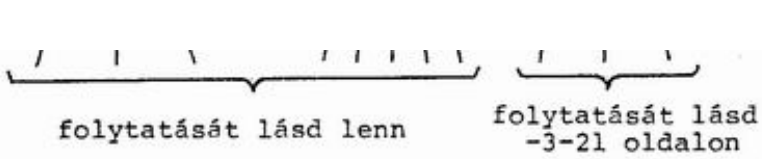
változtassa önmagát; ez a lehetőség a szokásos számitógépek gépi kódjában megvan, indokolt tehát az a kivánság, hogy formulavezérlésü számitógép belső nyelvében is meglegyen /KALMÁR, 1969/.

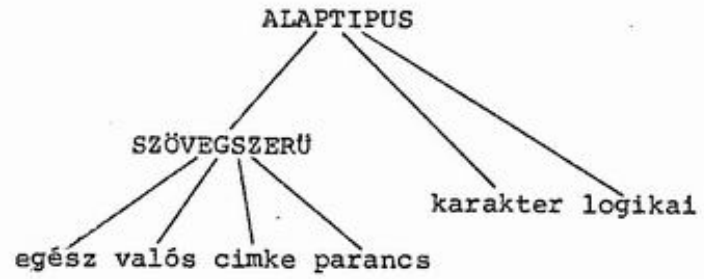
Összetételeknek a tömböket, szerkezeteket, láncokat /string/, vermeket /LIFO stack/ és a /FIFO/ sorokat tekintjük. Tömb tipusát dimenziószáma, numerikusan megadott indexkorlátpárjai és elemeinek /közös/ tipusa határozza meg. /Tovább vitatandó kérdésnek tartjuk, hogy hozzátartozzanak-e a

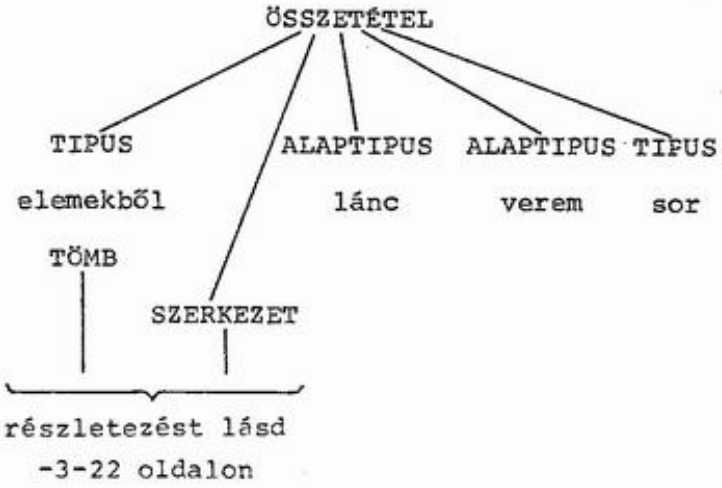
korlátpárok a tömb tipusának meghatározásához./ Szerkezet vegyes tipusu elemekből is készithető; egy-egy elemét annak megnevezéseként szereplő azonositó segitségével lehet kiválasztani. Szerkezet tipusát ezek az azonositók és az általuk megnevezett elemek tipusa határozza meg. Láncot és vermet csak valamely alaptipushoz tartozó elemekből lehet készitení. Sor elemei ismét tetszőleges tipusuak lehetnek. A sorok elsősorban a gyors memória és olyan háttértárolók közötti információcsere automatikus végrehajtásának eszközeként szolgálnak, amelyekre irás, illetőleg amelyekről olvasás nem kiván információ-konverziót /pl. mágnes-szalagok, mágnes-lemezek/. A FIFO-sorokat input/output bufferként használjuk. A sor aktuális eleme tulajdonképpen egy szekvenciálisan kezelt file aktuális rekordja. A rekord-váltás müvelete /GET, PUT/ egytagu, sor értékü müveletként használható a programban.

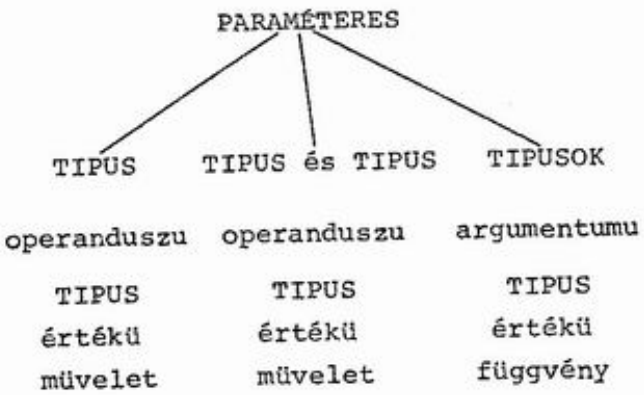
A FOGALOM metafogalomnak nyilván minden fogalom speciális esete /annak következtében, hogy a szóköz, mint mondtuk, nem számit szignifikánsnak/. Ennélfogva bármilyen fogalom speciális eseteiből készithetünk szavakat /elválasztójel nélkül irt /véges/ sorozatokat/, felsorolásokat /vesszővel elválasztott sorozatokat/ és szerelvényeket /pontosvesszővel elválasztott

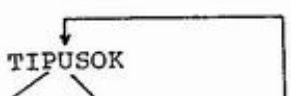


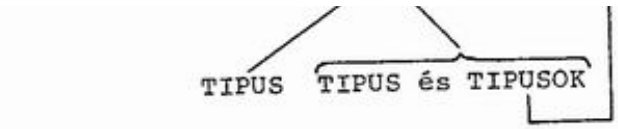

folytatását lásd lenn
folytatását lásd
-3-21 oldalon

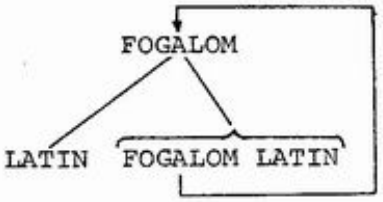

ALAPTIPUS
SZÖVEGSZERÜ
karakter logikai
egész valós cimke parancs

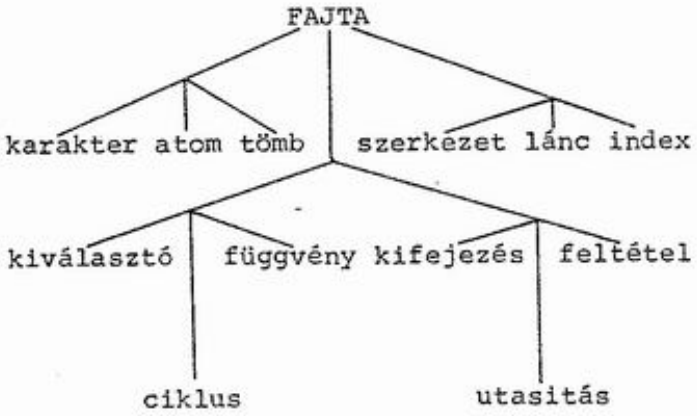

ÖSSZETÉTEL
TIPUS
ALAPTIPUS
ALAPTIPUS
TIPUS
elemekből
lánc
verem
sor
TÖMB
SZERKEZET
részletezést lásd
-3-22 oldalon


PARAMÉTERES
TIPUS
TIPUS és TIPUS
TIPUSOK
operanduszu
operanduszu
argumentumu
TIPUS
értékü
müvelet
TIPUS
értékü
müvelet
TIPUS
értékü
függvény

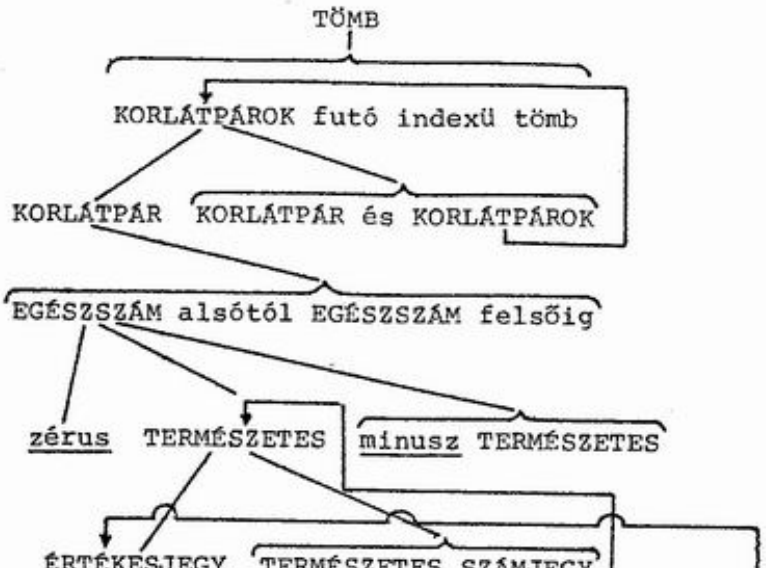

TIPUSOK

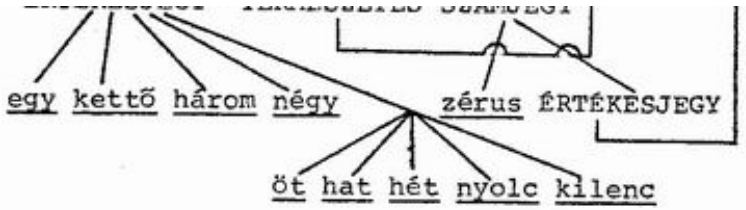


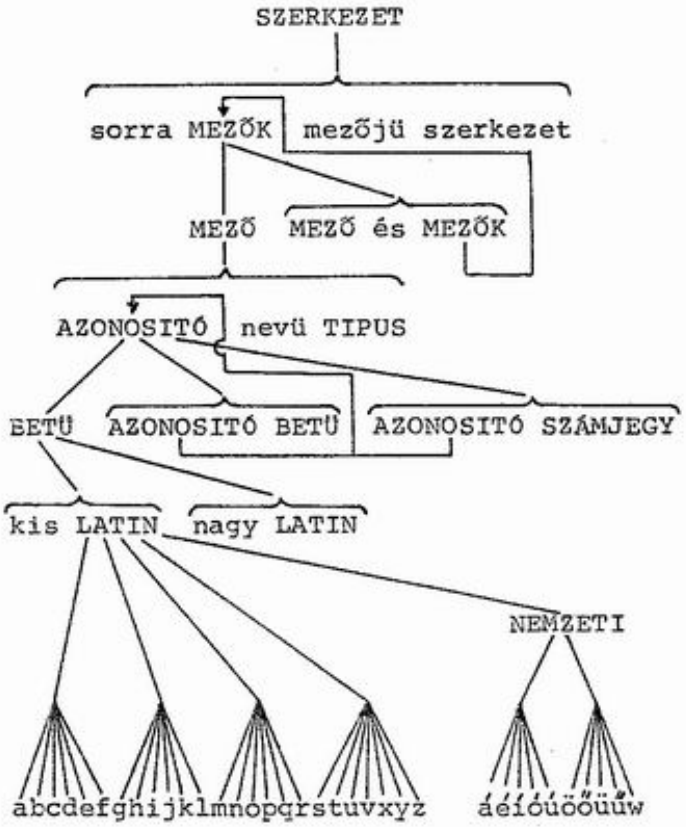




/emlékeztető: TIPUS elemekből/







3.3. Konstansok, változók

/Előzetesen lásd a Függelék hasonló cimü részét./

Minden tipushoz tartoznak a kérdéses tipusu konstansok is, változók is. A konstansok vagy jelölések /pl., a szokásos reprezentációt feltételezve -- mint az alábbiakban is --, 3,1415926535 rögzitett vesszős valós jelölés/, vagy definicióval bevezetendő /definiált/ azonositók /pl. pi ugyancsak valós konstans, definiciója a def real pi = 3,1415926535 valós definiciós egyenlet lehet/

valós definíciós egyenlet lehet/.

A valós jelölések a szokásosak; a tizedesvessző előtt sem, után sem kell kiirni a 0-t. Az egész jelölések ugyancsak a szokásosak; negativ egész számra nincs jelölés, mert pl. -1 már a - jellel, mint egytagu müvelet jelével képezett kifejezés.

Cimke-jelölés a cimke azonositója. /Itt természetesen az azonositó fogalomnak és nem az AZONOSITÓ metafogalomnak kell szerepelnie, különben csak a cimke szóval kezdődő azonositók lennének cimke-jelölések./

Parancs-jelölések a szokásos alaputasitások, hozzájuk véve a ciklusutasitásokat és az összetett utasitásokat is, de elhagyva az eljárás-utasitásokat, amelyek szerepét a parancs tipusu függvénykifejezések veszik át. /A parancsfogalmat -- tipusként, valamint a parancs jelölés és a parancs kifejezés fogalom részeként -- azért használjuk az utasitás helyett, mert az maga is fontos, más szerepet játszó fogalom./

A karakter-jelölések a szokásos módon /pl. egyszerü idézőjelek közé tétellel, ha azokat választjuk a karakter-

zárójelek reprezentásaiként/ keletkeznek. A lánc-zárójeleket célszerü megkülönböztetni a karakter-zárójelektől, hogy az egyelemü karakter-lánc jelölését meg lehessen különböztetni egyetlen karakterének jelölésétől.

A logikai jelölések természetesen a szokásosak. Listajelölésként a lista-atomokat vezetjük be, amelyeket /degenerált/ listáknak tekintünk. Atomokat bármely alaptipushoz tartozó jelölésből, továbbá bármely tipushoz tartozó definiált azonositóból vagy skaláris változóból képezhetünk atom-záró-

jelpárba zárással.

Egydimenziós tömb /tetszőleges tipusu elemekből álló vektor/ jelölése: elemeinek jelölése, vesszőkkel elválasztva, növekvő index szerint felsorolva, tömb-zárójelek közé zárva. Többdimenziós tömb jelölését nem különböztetjük meg olyan egydimenziós tömb jelölésétől, amelynek elemei eggyel kevesebb dimenziós tömbök. Pl., ha a tömb-zárójeleket kerek zárójelekkel reprezentáljuk, ((1,2,3), (4,5,6)) egyaránt jelöli azt a vektort, amelynek komponensei sorra az (1,2,3) és a (4,5,6) vektorok, és az $\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix}$ mátrixot. /A mátrixot tehát sorvektorokból álló oszlopvektornak tekintjük. Különböző hosszuságu sorvektorok összefogása egy /pl. háromszögü/ mátrixba abba az általános szabályba ütközik, hogy tömb elemeinek mind azonos tipusunak kell lenniük./

Szerkezet jelölését ugy adjuk meg, hogy szerkezet zárójeleken belül felsoroljuk, vesszőkkel elválasztva, elemeiket, mindegyik elé kettősponttal odairva azt az azonositót, amellyel hivatkozunk rá. /Pl., amennyiben a szerkezet-zárójelek is kerek zárójelek, a lánc-zárójelek pedig kettős macskakörmök, akkor

(név: "KisJános", anyja neve: "NagyMária", kor: 24, rendfokozat: "tizedes") egy katona jelölése lehet, ha sorra név nevü karakterlánc és anyja neve nevü karakterlánc és kor nevü egész és rendfokozat nevü karakterlánc mezőjü szerkezet tipusunak tekintjük./

Lánc jelölését ugy adjuk meg, hogy lánc-zárójelpárba zárva felsoroljuk elemeinek jelölését, ha szövegszerüek, akkor vesszővel elválasztva, különben elválasztójel nélkül, azonban karakterlánc esetén elhagyva elemei jelöléséből a karakter-zárójeleket /ugy, hogy csak maguk a karakterjelek maradnak meg a lánc-záró-

jelek között/. Bármely alaptipus esetén meg van engedve az üres lánc is; ez esetben a kérdéses alaptipushoz tartozó ürjel van lánc-zárójelpárba zárva.

Verem-, illetőleg sor-jelölésként egyedül az üres vermet ill. sort vezetjük be.

Szabványos müveleti jelekként, amelyek egy- és kéttagu müveletek jelöléseként használhatók, szóbajöhet minden, az APL megjelenitők billentyüzetén meglevő, a betüktől különböző jel; ugyanaz a jel az operanduszok számától /egy, vagy kettő/ és tipusától, valamint az aktuális alkalmazási terület-kiválasztó mikroprogram-tártól függően más-más müveletek jeleként is.

/Rögzitett/ függvények jelölésére fenntartott azonositókat /pl. sin, cos stb./ használunk. Azt, hogy melyek az ilyenek, további előszabályban kell majd rögziteni, mihelyt eldöntjük, hogy a különböző alkalmazási területeken milyen rögzitett függvényekre lesz szükség. A rögzitett függvények közé tartoznak a szabványos eljárások is, mint parancs értékü függvények. Ugyanaz a jelölés alkalmazható az argumentumok /aktuális paraméterek/ tipusától függően más-más függvények jelölésére is. A

fenntartott azonositók deklarálása vagy definiálása, vagy cimkeként, vagy formális paraméterként való használata megszakitást okoz, a megfelelő hibajelzéssel; ugyanigy megszakitást okoz definiált azonositók deklarálása, ujradefiniálása, vagy cimkeként vagy formális paraméterként való használata is. Paraméteres tipusu azonositó, vagy szabványos müveleti jel esetén nem számit hibás /ujra-/ deklarálásnak vagy definiálásnak a más tipusu operandusz/ok/, illetőleg argumentum/ok/ /formális paraméter/ek// esetére szóló deklaráció, illetőleg definició.

Az adott tipushoz tartozó változók nemcsak /általános/

skaláris változók /a kérdéses tipushoz deklarációval, vagy definicióval hozzárendelt azonositók, müvelet vagy függvény definiciós egyenletében szereplő specifikációval hozzárendelt formális paraméterek, ill. az aktuális alkalmazási terület-kiválasztó mikroprogram-tár által hozzárendelt speciális ábécé elemei/, továbbá az adott tipusu elemekből álló tömbök elemei /indexes változók/ lehetnek, hanem olyan szerkezet-elemek is, amelyek tipusa az adott tipus, ezenkivül az adott tipusu elemekből álló sor aktuális eleme is, valamint, amennyiben az adott tipus alaptipus, az ilyen tipus elemekből álló lánc vagy verem eleme is. /Sor-elem esetén mindig csak az aktuális elemhez, vagyis a sorbaállitott és a sorból még fel nem használt elemek közül az elsőhöz lehet hozzáférni, mégpedig a sor azonositójára való hivatkozással; verem esetén azonban nemcsak a verem tetején levő elemre lehet hivatkozni, mégpedig a verem azonositójával, amikoris ez az elem /ha kifejezésben felhasználjuk/ kilép a veremből, hanem lejjebb levő elemekre is, a verem tetejétől /amelynek a 0 index felel meg/ számitott index megadásával indexzáró-

jelbe tett egész kifejezés segitségével, ami viszont nem jelenti a felhasznált elem kilépését a veremből /vagyis a veremmutató átállitását/. Pl., ha v valós tipusu elemekből álló verem, akkor a $v + v \Rightarrow v$ értékadó utasitás hatása a következő: a gép összeadja a verembe utoljára betöltött két valós számot, törli ezt a két elemet és a verem tetejére a betöltés sorrendjében utolsó előtti szám helyére beirja az összegüket / $\Rightarrow$ a tárolás jelének reprezentációja/. Ezt az összeadást a $v[0] + v[1] \Rightarrow v[1]$ értékadó utasitással is el lehet érni, azonban ekkor nem $v[1]$ lesz, hanem $v[0]$ marad a verem tetején. Viszont a $v[0] + v[0] \Rightarrow v[0]$ értékadó utasitás hatására a verembe utoljá-

ra betöltött szám megkétszerezodik; tehat annak ellenere, hogy v[0] magában véve ugyanazt a veremelemet jelenti, mint v, nem mindegy, hogy v-t irunk-e, vagy v[0]-t.

A sor és a verem kivételével a többi összetétel tipusu azonositó nem az összetétel valamely elemét, hanem az egész összetételt jelölő "skaláris változó"-ként szerepel a kifejezésekben. Igy nincs akadálya annak, hogy pl. az A és B /négyzetes/ mátrixok mátrix-szorzatát jelöljük A × B-vel /ha az aktuális alkalmazási terület-kiválasztó mikroprogramtár a × jelhez a mátrix-szorzás müveletét rendeli/. Viszont mátrix, és általában tömb elemére ugy hivatkozunk, hogy a tömb azonositója után index-zárójelbe olyan egész kifejezéseket irunk, amelyek aktuális értékei adják a tömb-elem indexeit, szerkezet elemére pedig ugy, hogy a szerkezet-elem kiválasztására szolgáló azonositó után /amely a szerkezet tipusával adva van/ kiválasztó-zárójelpárba irjuk a szerkezet azonositóját. Pl. M[i, j], mint szokás, az M mátrix i-edik sorának j-edik elemét jelöli, kor(k) + 1 ⇒ kor(k) pedig a k katona /katonai időszámitás szerinti/ korának átkeltezését /updating/ jelentő értékadó utasi-

tás /minden év január 1-én/. Ha viszont a k hadnagy előlép főhadnaggyá, ezt a "fő" + rendfokozat(k) ⇒ rendfokozat(k) értékadó utasitás vezeti keresztül a katonai nyilvántartásban /feltéve, hogy + karakterlánc tipusu operanduszok esetén a konkatenáció jele/.

Megemlitjük még, hogy a //LISP 1.5/-höz hasonló értelemben vett/ listák elemei is, hasonlóan a szerkezetekéihez, tetszőleges tipusuak lehetnek. Közismert, hagyományos alkalmazásaikon kivül megemlithetjük a többszintü rekordok képzésének lehetőségét, amely az adatfeldolgozás céljaira teszi alkalmasabbá a formulavezérlésü számitógépet /az ugyanazon szintü rekord-

mezőket egy cella car részében helyezzük el, mig az eggyel alacsonyabb szintet ugyanazon cella cdr részében/. Egy másik alkalmazás a táblázatok definiálásának, illetőleg kezelésének lehetősége. A táblázati elemek argumentum-, illetőleg tulajdonság-elemeit tipusonként egy-egy vektorban helyezhetjük el; e vektorokat pedig listaként kapcsolhatjuk össze. Egy harmadik alkalmazás: a rendszerprogramozás szempontjából igen hasznos adatstrukturákat épithetünk fel listaként, ha számitásba vesszük pl. több verem összekapcsolhatóságának a következményeit. Végül megemlitjük, hogy a lista az a nyelvi elem, amely dinamikusan változó adatstrukturát képvisel: az értékadó utasitások és a listakifejezések biztosithatják a végrehajtás közbeni adatstruktura-módositást, a deklaráció által specifikált formából kiindulva.

3.4. Kifejezések

/Előzetesen lásd a Függelék hasonló cimü részét./

A kifejezések értéke -- mint már emlitettük -- akármilyen tipusu lehet. Felépitésük konstansokból, változókból, függvények

nevéből -- mindegyik után téve függvény-zárójelben, vesszőkkel elválasztva, a hozzátartozó aktuális paramétereket -- keletkező függvénykifejezésekből, kifejezés-zárójelbe zárt, előzőleg képzett kifejezésekből, valamint feltételes kifejezésekből /mint elsődleges kifejezésekből/ egy- és kéttagu müveletek segitségével történik. A kifejezés értékének tipusa konstans és változó esetén ennek tipusa; függvénykifejezés esetén a függvény értékeinek /a függvény tipusában, argumentumainak tipusával együtt, megadott/ tipusa /az egyes aktuális paraméterek esetén, amelyek maguk is kifejezések, értékük tipusának meg kell egyeznie a megfelelő argumentumnak a függvény tipusában megadott ti-

pusával/; zárojelbe tétel nem változtat a kifejezés értékének tipusán; feltételes kifejezés értékének tipusa azon két /a feltétel teljesülése, ill. nem teljesülése esetén kiszámitandó/ kifejezés értékének tipusa /ezek értéke tipusának meg kell egyeznie egymással, a feltételt pedig olyan kifejezéssel kell megadni, amely logikai tipusu értékeket vehet fel/. Egy-, vagy kéttagu müvelet alkalmazásával keletkező kifejezés értékének tipusa a müvelet tipusában van megadva /az operandusz/ok/ tipusával együtt/; annak a/z egy, vagy két/ kifejezés értéke tipusának, amely/ek/ből a kérdéses kifejezés müvelet alkalmazásával épül fel, meg kell egyeznie az operandusz/ok/ számára megadott tipus/ok/kal.

Müveletek, illetőleg függvények gyanánt olyan konstansok és változók jönnek tekintetbe, amelyek tipusa TIPUS1 operanduszu TIPUS értékü müvelet vagy TIPUS1 és TIPUS2 operanduszu TIPUS értékü müvelet, illetőleg TIPUSOK argumentumu TIPUS értékü függvény alaku, ahol TIPUS, TIPUS1 és TIPUS2 helyébe egymástól függetlenül tetszőleges tipusok, TIPUSOK helyébe pedig tetszésszerinti számu tipus teendő, és-sel elválasztva. A kons-

tans /vagyis rögzitett/ müveletek jele mindig valamely szabványos /pl. az APL-jelkészletben szereplő/ müveleti jel, vagy, amennyiben definiált müveletről van szó, annak jele lehet ismét szabványos müveleti jel, vagy azonositó. A konstans függvények jele /ezen ismét rögzitett függvényt értve, nem olyan függvényt, amelynek minden helyen ugyanaz az értéke/, amennyiben szabványos függvényről van szó, fenntartott azonositó, amennyiben pedig definiált függvényről, akkor ennek definiciójában bevezetett azonositó.

A munka jelen szakaszában nem kivánjuk rögziteni, hogy melyek legyenek a szabványos müveleti jelek, valamint a szabványos függvények jelölésére fenntartott azonositók, és hogy me-

lyik melyik müveletet, illetőleg melyik függvényt jelölje, az operanduszok, illetőleg argumentumok számától és tipusától, valamint az aktuális alkalmazási terület-kiválasztó mikroprogramtártól függően, hiszen ez a reprezentáció és a szemantika kérdése. A szabványos müveletek között mindenesetre szerepelniük kell az aritmetikai alapmüveleteknek; a +, -, × jelek egyuttal értelmezhetők karakterekre is ugy, hogy kódjaikon végezzük el őket a byte-hosszuságra /amit célszerü 8-nak választani/, mint modulusra nézve. Szerepelniük kell továbbá az aritmetikai alaprelációknak /<, >, ≠, =, ≦, ≧/ mint logikai értékü müveleteknek, ismét nemcsak számokon, mint operanduszokon, hanem pl. karaktereken is értelmezve. Ez utóbbi esetben kétféle jelentés is szóba jöhet: vagy kódjaikra vonatkoztatjuk a relációt, vagy az ábécében elfoglalt helyükre. Az utóbbi esetben célszerü karakterláncokon is értelmezni e relációkat, a lexikografikus rendezés alapján. Persze a két jelentés nem zárja ki egymást, legfeljebb más müveleti jelet használunk az egyik, mint a másik esetben. Az

itéletkalkulus müveleteit is célszerü logikai értékeken kivül logikai érték-láncokon és, bit-reprezentációjukra vonatkoztatva, karaktereken, esetleg karakterláncokon is értelmezni, mint szabványos müveleteket. A leggyakoribb egytagu müveleteken kivül /mind például az előjel megforditása, a logikai érték negációja/ érdemes számos olyan egytagu szabványos müveletet bevezetni, amelyeket általában egyargumentumu függvényeknek irnak; ilyenek a valós számok egész-része, tört-része, előjele, abszolut értéke, logaritmusa, a trigonometrikus függvények és inverzeik, továbbá karakter kiegészitő kódja, sorszáma az ábécében, ezenkivül valamely tipusnak más tipusra való konvertálásának müvelete,

például karakter numerikus kódja, logikai érték aritmetikai reprezentációja 1 és 0 segitségével, egész szám vagy rögzitett, vagy lebegő vesszősen ábrázolt megfelelője, konverzió rögzitett vesszős alakról lebegő vesszősre és viszont, a számrendszer váltáshoz szükséges konverzió stb.

Fontos előrelépés a szokásos számitógépek utasitásrendszeréhez képest, hogy a formulavezérlésü számitógépben az összetételek /vektorok, mátrixok és többdimenziós tömbök, szerkezetek, láncok, vermek, FIFO sorok/, valamint a listák között végzett müveletek ugyanolyan sullyal szerepelhetnek az utasitásrendszerben, mint a hagyományos müveletek. Bár ilyen irányu törekvések már korábban is felmerültek /GLUSKOV, 1965/, de ezek elsősorban a mátrix-müveletekre korlátoződtak. A lineáris algebrához szükséges vektor- és mátrix-müveletek bevezetését /például skaláris szorzás, vektoriális szorzás, mátrix transzponálása, inverze, mátrix sorának, oszlopának képzése, mátrix-szorzás, négyzetes mátrix determinánsának kiszámitása, mátrix rangszámának meghatározása és természetesen mátrixok és vektorok összeadása is,

továbbá általában tömb résztömbjének képzése stb./ mi is javasoljuk, de egyben a polinom-müveleteket, az egyenletrendszerek megoldását stb. is, mint szabványos müveleteket.

A szerkezet-müveletek közül elsősorban a komplex aritmetikában, a formula-manipuláció terén, valamint az adatnyilvántartások kezelése során előnyösen felhasználható müveletek jönnek tekintetbe. Megjegyezzük, hogy a jelenlegi adatfeldolgozási célokra orientált nyelvek, bár használják a szerkezet fogalmát, nem annyira a szerkezeteken, mint inkább elemeiken végzett müveletekkel dolgoznak, a rendezés, válogatás, listázás kivételével.

A láncokon definiált szabványos müveletek hivatottak

mindazoknak a tevékenységeknek a végrehajtására, amelyeket jelenleg csak a string-manipulációs nyelvekben biztositanak. Hangsulyozzuk azonban, hogy láncokat nemcsak karakterekből, hanem bármilyen alaptipushoz tartozó elemekből fel lehet épiteni. Célszerünek látszik szabványos lánc-müveletként bevezetni a következőket: rész-lánc kiválasztása kontextus-feltétel alapján, valamint helyettesitése lánccal; lánc átrendezése valamilyen értelemben; rész-lánc törlése, megkettőzése; lánc keresése szótárban; láncok konkatenációja; lánc kezdő- és végszeleteinek leválasztása. Igen lényeges az, hogy láncokon is értelmezzünk logikai értékü alapmüveleteket, amelyek közül a következők látszanak legfontosabbaknak: annak eldöntése logikai érték formájában, hogy valamely lánc előfordul-e részláncként egy láncban, egyelemü-e valamely lánc, meghalad-e valamely adott hosszuságot.

Verem-müveletek gyanánt a verem elemein végzett müveleteken kivül, amelyekben csak formálisan szerepel a verem azonositója, szóbajöhetnek szabványos müveletekként a következők: a verem elemei számának meghatározása /ez lényegében a mutató értékéhez való hozzáférést jelenti/; verem elemeinek szétválogatása valamilyen szempont alapján /pl. aszerint, hogy egy a verem elemein értelmezett logikai értékü müvelet melyik logikai értéket szolgáltatja/; két verem egyesitése valamilyen módon meghatározott sorrendben; verem kiüritése; valamint egy müvelet iterált elvégzése a verem tetején lévő elemekkel mindaddig, amig a verem ki nem ürül.

A FIFO sorokon végzett müveletek közül a következők felvétele jöhet számitásba a szabványos müveletek közé: a sor kiüritése; előrelépés adott számu lépéssel, vagy addig, amig

adott tulajdonságu elem nem jön sorra; visszalépés adott számu lépéssel; két sor összefésülése; sor rendezése valamilyen szempontból. Figyelembe jöhetnek még tömegkiszolgálási feladatok megoldására előnyösen felhasználható müveletek, elsősorban olyanok, amelyek operációs rendszer-feladatok megoldását megkönnyitik.

A lista-müveletek közül elsősorban a LISP 1.5 nyelv car, cdr, cons alapmüveletei jönnek számitásba, mint szabványos müveletek. Az ebben a nyelvben definiált egyéb müveletek nagy részét megvalósitják a már emlitett struktura-, lánc-, aritmetikai és logikai müveletek. A LISP 1.5 nyelvben szereplő atomok szerepét, mint emlitettük, a formulavezérlésü számológép esetén tetszőleges tipusu definiált azonositók és skaláris változók, valamint alaptipushoz tartozó jelölések veszik át /atomzárójelpárba zárva/.

Ennyiféle szabványos müvelet bevezetése után nyilvánvaló, hogy a formulavezérlésü számológép belső nyelvében nagymérték-

ben csökken a szabványos függvények jelentősége. A nyelv emiatt "algebraibb" jellegü lesz, mint a szokásos magasrendü programozási nyelvek. Mindamellett a szabványos müveletek fenti felsorolásában szereplő esetekhez hasonló olyan esetekben, amikor kettőnél több adatból célszerü kiszámitani valamilyen eredményt, szóbajöhet e célra szabványos függvények bevezetése.

Az viszont már -- mind a müveletekre, mind a függvényekre nézve -- külön eldöntendő kérdés, hogy a szabványos müveletek és függvények megvalósitásában mekkora szerepet juttatunk hardware-nek, mekkorát a firmware-nek és mekkorát a softwarenek. Erre a kérdésre még a következő fejezetben visszatérünk.

3.5. Utasitások

/Előzetesen lásd a Függelék hasonló cimü részét./

A formulavezérlésü számitógép javasolt belső nyelve tartalmazza az algoritmikus nyelvek szokásos utasitásait. Formájukat igyekeztünk a hardware-implementáció szempontjainak figyelembevételével egyszerüsiteni.

Az értékadó utasitásban, a bal-asszociáció következetes alkalmazásának megfelelően, a jobboldalra került az a változó, amelynek értéket adunk.

A vezérlésátadó utasitásban az ugrás jele után szereplő cimke-kifejezés legalább olyan általános /valójában általánosabb/, mint az ALGOL 60-beli helymegjelölő kifejezés. A kapcsolók szerepét olyan, 1 alsó index-korlátu vektorok játsszák, amelynek elemei cimke tipusuak. Az ALGOL 60 kapcsoló-deklarációjának szerepét ilyen vektor definiciója játssza. Cimkék deklarációja felesleges; minden olyan azonositó, amely utasitás

előtt áll, attól kettősponttal elválasztva, implicite cimkévé deklarálódik. Ha azt akarjuk, hogy a nyelv szintaxisa kizárja az explicit cimke-deklarációt, nem kell mást tennünk, mint zsákutca-fogalommá tenni a cimke deklarálója fogalmat azáltal, hogy nem adunk rá reprezentációs szabályt.

A ciklusutasitásban szerepelhet lépés, függetlenül attól, hogy az ellenőrzés végérték, vagy végfeltétel alapján történik-e. Ez több lehetőséget biztosit, mint ami az ALGOL 60 nyelvben megvan, azonban nem jelent megszoritást, hiszen megengedjük azt is, hogy a lépés-rész üres legyen. Lépésnek azonban csak akkor van értelme, ha a + jellel jelölt müvelet definiálva van olyan tipusu operanduszokra, amilyen tipusu a ciklusváltozó. Végértékes ellenőrzés esetén az egyenlőtlenség-jelekkel jelölt re-

lációknak /logikai értékü müveleteknek/ definiálva kell lenniük ilyen tipusu operanduszokra. Maga az a körülmény, hogy a ciklusváltozó tipusa tetszőleges lehet, ugyancsak lényegesen több lehetőséget nyit meg, mint amik a jelenleg használatos programozási nyelvekben megvalósithatók a ciklusszervezés terén. E lehetőségek szöveg-elemek iterativ feldolgozásában és ennélfogva többek között a rendszerprogramozásban is nagyon jól kihasználhatók. Ugyancsak megnöveli a felhasználási lehetőségek számát az, hogy mind a végérték, mind a végfeltétel alapján történő ellenőrzés esetén van lehetőség sztatikus és dimanikus ciklusszervezésre is, a kétféle ciklusdinamika-jelnek megfelelően. Sztatikus szervezés esetén, ha az ellenőrzés arra az eredményre vezetett, hogy a ciklusmagot ismételten végre kell hajtani, a ciklusmag elejére adódik át a vezérlés; dinamikus szervezés esetén pedig az értékadás jele /legyen-jel/ utáni kifejezés elejére /tehát ennek a kifejezésnek értéke, hasonlóan mint

az ALGOL 60 while ciklusa esetén, ujból kiszámitódik, azonban, ha a lépés-rész nem üres, akkor a lépés-kifejezés értéke is, továbbá az until ciklushoz hasonló esetben a végérték is/.

Az összetett utasitásról nem kivánunk semmit mondani. Meg kell azonban emlitenünk az ALGOL 60 nyelv eljárás-utasitásának megfelelő szubrutin-hivást, amely parancs tipusu függvénykifejezés formájában irt utasitás. Ha a függvénykifejezés tipusa nem parancs, akkor az inkább az ALGOL 60 nyelv függvényeljárásának megfelelő szubrutin hivásának tekinthető. Magát a szubrutint, tetszőleges számu és tipusu argumentummal és tetszőleges tipusu értékekkel rendelkező függvény, vagypedig egy vagy két, tetszőleges tipusu argumentummal és értékkel

rendelkező müvelet definiciója generálja a formulavezérlésü számitógép belső nyelvén. Célszerü azonban ezt a szubrutint a géppel lefordittatni a gép belső nyelvének azon részнyelvére, amelyben kifejezésben csak egyetlen egytagu vagy kéttagu müvelet jele szerepelhet; ezt nevezhetjük a formulavezérlésü számitógép assembly-nyelvének. Ez esetben ugyanis el lehet kerülni a kifejezészárójel-mélység nehezen ellenőrizhető emelkedését. Viszont ez a fordités a munkaváltozók számának emelését kivánja. E célra célszerü ismét speciális ábécék alkalmazása, ugy, hogy ilyen ábécé betüje hozzáférést biztositson a megfelelő, szupergyors memóriában elhelyezett munkarekeszhez.

Maga a szubrutin szabványos függvény vagy müvelet esetén egy csak irásra használható tárolóban /ROM/ elhelyezett rendszerkönyvtárba kerül, hacsak nincs a szabványos függvény vagy müvelet már valamely mikroprogrammal megvalósitva. Egyelőre nem tudunk sokat mondani a mikroprogramozás nyelvéről. Célsze-

rü volna azonban, ha sikerülne ezt is, mint a formulavezérlésü számitógép assembly nyelve résznyelveként kijelölni. Erre reményt nyujt a sokféle tipus, a konverziós szabványos müveletek stb. Persze az e célra felhasználható, pl. minden mást logikai értékek láncára konvertáló müveletek szubrutinjait a cipőhuzó /bootstrap/ elvének megfelelően "még-mikróbb" nyelven kell megfogalmazni és egy alap-mikroprogramtárban elhelyezni.

Annak, hogy a formulavezérlésü számitógép különböző szintü nyelvei résznyelv -- nyelv viszonyban legyenek egymással, az volna a legfőbb előnye, hogy a ROM jellegü mikroprogram-tárolóban vagy a rendszerkönyvtárban elhelyezett szubrutinokat át lehet tölteni a gép operativ memóriájába és igy azokat módositani lehet, persze nem a ROM tárolóban. Célszerü lenne lehetővé tenni az igy kialakitott

[illegible] szubrutin betöltését egy RMM tároló /read mostly memory/ még üres részébe, amely rész attól kezdve csak kiolvasásra használható. Igy lehetővé válnék a gép alkalmazási területeinek bővitése ujabb mikroprogram-tárak kialakitása utján.

Természetesen a szabványos függvények, esetleg müveletek között kell lenniük azoknak a szubrutinoknak is, amelyek mint legegyszerübb fizikai input/output utasitások felhasználhatók a logikai input/output utasitások hatására kialakitandó csatornaprogramokban. Maga a csatornaprogram-összeállitó program mikroprogram-tárban helyezhető el. Az emlitett legegyszerübb fizikai input/output utasitások az ismert START, HALT, TEST I/O és TEST CH jellegüek, a logikai input/output utasitások pedig részben a FORTRAN-jellegü, külső és belső ábrázolási forma közötti konverzióval járó READ- és WRITE-szerü utasitások, részben a konverziót nem igénylő, a FIFO sorokkal kapcsolatban már

emlitett GET- és PUT-szerü utasitások. Paraméterként a READ és WRITE jellegü utasitásoknak a fenntartott logikai periféria azonositót karakter-lánc alakban, a formátum specifikációját karakter-láncokból álló vektor alakjában, a kiirandó vagy beolvasandó változók együttesét pedig szerkezet, alakjában kell tartalmazniuk. A GET és a PUT jellegü utasitások egytagu müveleteknek tekinthetők, amelyeknek operandusza FIFO sor, értéke pedig sor-elem tipusu.

3.6. Deklarációk, definiciók, program

/Előzetesen lásd a Függelék hasonló cimü részét./

A formulavezérlésü számitógép belső nyelvében éles különbséget teszünk deklaráció és definició között. A deklaráció egy vagy több azonositót adott tipusu index[illegible]

egy több dimenziós tömb típusú index nélkül, tehát a nyelv fogalmaiban kifejezve, skaláris változóvá deklarál. Ezzel szemben a definició nemcsak a konstansként vagy skaláris változóként használandó azonositó/k/ /közös/ tipusát, hanem értékét /jelentését/ is megadja, amely változó esetében természetesen csak kezdőértékét jelenti. Ilyen értelemben tehát az ALGOL 60 tipus- és tömb-deklarációja deklaráció, kapcsoló- és eljárás-deklarációja azonban definició.

A deklaráció áll egy, a kérdéses tipust meghatározó deklarálóból és a deklarálandó azonositók ettől is, egymástól is vesszővel elválasztott felsorolásából.

A definició a formulavezérlésü számitógép belső nyelvén egy definiciójellel kezdődik, utána jön a tipust meghatározó deklaráló, majd a definiciós egyenletek felsorolása.

Amennyiben paramétertelen tipusról, tehát alaptipusról, lista-tipusról, vagy összetételről /tömb, szerkezet, lánc, verem,

sor/ van szó, minden egyes definiált azonositóhoz /skaláris változóhoz/ egy-egy olyan definiciós egyenlet tartozik, amelyben az egyenlőség-jel, illetőleg az értékadás jele /legyen-jel/ baloldalán a kérdéses definiált azonositó, illetőleg skaláris változó áll, jobboldalán pedig az értékét /illetőleg kezdőértékét/ megadó kifejezés.

Amennyiben azonban egy vagy kéttagu müvelet vagy akárhány argumentumu függvény definiciós egyenletéről van szó, akkor az egyenlőség-jel baloldalán a kérdéses müvelet operandusza/i/ helyén, illetőleg a függvény argumentuma/i/ helyén formális paraméterek vannak, a jobboldalán pedig, utasitás-zárójelek közé zárva, egy vagy több utasitás, amennyiben több, pontosvesszőkkel elválasztva, majd a végén, tőlük ugyancsak pontosvesszővel elválasztva, egy olyan tipusu kifejezés, amilyen tipusu értéket

ad a definiálnadó művelet, illetőleg függvény. A formális paraméterek egy-egy azonositóból állnak, amelyeknek különbözőknek kell lenniük, mindegyik elé irva a megfelelő operandusz, illetőleg argumentum kivánt tipusát. /Ez az ALGOL 60-beli specifikációnak felel meg./ A egyenlőség-jel, illetőleg a legyen-jel jobboldalán szereplő utasitásokban és kifejezésben e formális paraméterek azonositói éppugy szerepelhetnek, mint ugyanolyan tipusu skaláris változók. A formális paraméterek a definiciós egyenletekre nézve lokális változók.

A definiciós egyenlet tulajdonképpen az a szubrutin, amelynek hivása a definiált azonositóju művelet alkalmazásával létrejövő kifejezés, illetőleg a definiált azonositóju függvény segitségével létrejövő függvénykifejezés által megtörténik. Az aktuális paraméterek átadása e szubrutinnak mindig név szerint tör-

ténik. Ez természetesen nem optimális, ha az aktuális paraméter bonyolult kifejezés és a megfelelő formális paraméter sokszor előfordul a definiciós egyenlet jobboldalán. Ezen csak ugy lehetne segiteni, ha az aktuális paraméterek érték szerinti átadását is megengednők, amikoris a formális paraméter előtt tipusán kivül azt a tényt is jelezni kellene, hogy az aktuális paraméter értékét fogja átvenni.

Maga a program deklarációk, definiciók és utasitások pontosvesszővel elválasztott sorozata. Előre történő /még nem szerepelt, tehát a gép által a deklarációk, definiciók és az utasitások cimkéi alapján összeállitott, asszociativ tárolóban elhelyezett azonositótáblázatban elő nem forduló cimkére való/ vezérlésátadás esetén különleges "kereső" üzemmódban történik a cimke megkeresése és egyuttal a közben előforduló /a vezérlésát-

adással átugrott/ deklarációk, definiciók és cimkék feldolgozása, igy ezek az átugrás ellenére sem maradnak figyelmen kivül. Hátra történő vezérlésátadás esetén viszont az átugrott deklarációk, definiciók és a cimkéknek megfelelő program-helyek ujra-feldolgozása akkor történik meg, amikor ujra sor kerül rájuk.

Eszerint a formulavezérlésü számitógép belső nyelve nem blokk-szerkezetü, mint pl. az ALGOL 60 nyelv. Blokk-szerkezetü belső nyelv ugyan megkönnyitené a blokk-szerkezetü forrásnyelvek kompilálását, de a nem ilyen szerkezetü nyelvek implementálásában nem jelentene előnyt. Ezért a blokk-szerkezet mellőzését javasoljuk azzal, hogy az emiatt felmerülő problémák megoldását hagyjuk a kompilátor irójára, hiszen ezek a szokásos /nem-formulavezérlésü számitógép esetén alkalmazott/ módszerekkel biztosan megoldhatók.

3.7. Befejező megjegyzések

A formulavezérlésü számitógép javasolt belső nyelve a Magyar Tudományos Akadémia Matematikai logikai és automataelméleti Tanszéki Kutató Csoportjának vezetője és tagjai, valamint a szegedi József Attila Tudományegyetem Kibernetikai Laboratóriumának és Számitástudományi Tanszékének a munkába bevont tudományos dolgozói közötti éles vita során alakult ki. A jelen tanulmányban és mellékletében leirt formája e vita során kialakult és az emlitettek által többé-kevésbé elfogadhatónak itélt véleményt tükrözi. Ez nem jelenti azt, hogy egy-egy ujabb nehézség, vagy előnyös lehetőség felmerülésekor nem lángolhat fel ismét a vita és nem alakulhat ki ettől eltérő vélemény. Nyilván figyelembe kell még vennünk azon intézmények dolgozóinak véleményét is, akiknek együttmüködő segitségére számitunk a formulaven-

zérlésü számitógép müszaki megvalósitásában.

E müszaki megvalósitását megelőzően természetesen szimulálnunk kell a formulavezérlésü számitógépet valamely müszakilag már megvalósitott számitógépen, hogy tapasztalatokat szerezzünk különböző alkalmazási területeken való felhasználhatóságára nézve. E tapasztalatok éppen arra valók, hogy előre nem látott nehézségek, vagy előnyös lehetőségek elszalasztása kiderüljenek és ennek alapján terveinket, szükség esetén a gép belső nyelvének szintaxisát is, módosithassuk a müszaki megvalósitás előtt. Természetesen az sincs kizárva, hogy a müszaki megvalósitás során további módositást kivánó nehézségek merülnek fel. Ennélfogva a formulavezérlésü számitógép belső nyelvének itt kifejtett tervezete semmiképpen sem tekinthető véglegesnek.

A szintaktikus leirásnak lényegében az ALGOL 68 jelentéstől átvett formája semmiképpen sem jelenti azt, hogy az ALGOL 68

nyelvet kivánjuk megtenni a formulavezérlésü számitógép belső nyelvének. Tisztában vagyunk az ALGOL 68 nyelv előnyeivel és hátrányaival is. Az előbbiek miatt sok mindent átvettünk az ALGOL 68-tól, de az utóbbiak miatt az ALGOL 68 lényeges tulajdonságainak átvételét mellőztük. Ezek között emlitem a már emlitett blokk-szerkezet mellett a név-jellegü tipusok mellőzését, mint legfontosabbat, amely ugyan egységessé tenné a paraméter-átadást anélkül, hogy olyankor is név szerinti átadást kivánna, amikor ez az optimalitás rovására megy, és biztositaná az indirekt cimzésben rejlő lehetőségek megtartását magasszintü programozási nyelven is, viszont a szintaxist annyira bonyolitaná, hogy az már veszélyeztetné a formulavezérlésü számitógépként való megvalósitás lehetőségét. Ugyancsak mellőztük a különböző szóhosszuságok lehetőségét az ALGOL 68-ban tükröző hosszu és rövid tipusokat. Véleményünk szerint a többszavas

aritmetikát is helyesebb software-uton megvalósitani, mint hardware-megvalósitásával megnehéziteni a formulavezérlés elve fontosabb lehetőségeinek realizálását.

Tisztában vagyunk azzal, hogy ezeknek és más itt nem emlitett, az ALGOL 68-ban meglevő lehetőségeknek mellőzése ellenére is bonyolult belső nyelvről van szó. A müszaki megvalósitás természetesen egy ilyen bonyolult nyelvnek mindenképpen csak egy résznyelvére vonatkozhat. Nem szeretnénk azonban, ha e résznyelv megválasztása veszélyeztetné a belső nyelvnek a tipusfogalom általánosabb kezelése folytán észlelhető, véleményünk szerint az ALGOL 68-nál magasabb foku eleganciáját, és ami ezzel jár, a gép rugalmasságát és igy a fentiekben sokszor emlitett előnyös felhasználhatóságát a legkülönbözőbb alkalmazási területeken.

FÜGGELÉK

Az FCCL-4 programozási nyelv szintaxisa

Metafogalmak; általános konstrukciók

a/ metaszintaktikus szabályok

TIPUS:: PARAMÉTERTELEN; PARAMÉTERES.

PARAMÉTERTELEN:: ALAPTIPUS; lista; ÖSSZETÉTEL.

ALAPTIPUS:: SZÖVEGSZERÜ; karakter; logikai.

SZÖVEGSZERÜ:: egész; valós; cimke; parancs.

ÖSSZETÉTEL:: TIPUS elemekből TÖMB; SZERKEZET; ALAPTIPUS lánc;
ALAPTIPUS verem; TIPUS sor.

PARAMÉTERES:: TIPUS operanduszu TIPUS értékü müvelet;
TIPUS és TIPUS operanduszu TIPUS értékü müvelet;
TIPUSOK argumentumu TIPUS értékü függvény.

TIPUSOK:: TIPUS; TIPUS és TIPUSOK.

TÖMB:: KORLÁTPÁROK futó indexü tömb.

KORLÁTPÁROK:: KORLÁTPÁR; KORLÁTPÁR és KORLÁTPÁROK.

KORLÁTPÁR:: EGÉSZSZÁM alsótól EGÉSZSZÁM felsőig.

EGÉSZSZÁM:: zérus; TERMÉSZETES; minusz TERMÉSZETES.

TERMÉSZETES:: ÉRTÉKESJEGY; TERMÉSZETES SZÁMJEGY.

ÉRTÉKESJEGY:: egy; kettő; három; négy; öt; hat; hét; nyolc; kilenc.

SZÁMJEGY:: zérus; ÉRTÉKESJEGY.

SZERKEZET:: sorra MEZŐK mezőjű szerkezet.

MEZŐK:: MEZŐ; MEZŐ és MEZŐK.

MEZŐ:: AZONOSITÓ nevü TIPUS.

AZONOSITÓ:: BETÜ; AZONOSITÓ BETÜ; AZONOSITÓ SZÁMJEGY.

BETÜ:: kis LATIN; nagy LATIN.

LATIN:: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q; r; s; t;
u; v; x; y; z; NEMZETI.

NEMZETI:: á; é; í; ó; ú; ö; õ; ü; ű; w.

FOGALOM:: LATIN; FOGALOM LATIN.

ELVÁLASZTÓ:: ÜRES; vessző; pontosvessző.

ÜRES:: .

FAJTA:: karakter; atom; tömb; szerkezet; lánc; index; kiválasztó;

függvény; kifejezés; feltétel; ciklus; utasitás.

NÉV:: konstans; változó.

b/ általános konstrukciók

ELVÁLASZTÓ segitségével elválasztott FOGALOM sorozat: FOGALOM;
ELVÁLASZTÓ segitségével elválasztott FOGALOM sorozat,
ELVÁLASZTÓ, FOGALOM.

FOGALOM szó: ÜRES segitségével elválasztott FOGALOM sorozat.

FOGALOM felsorolás: vessző segitségével elválasztott FOGALOM sorozat.

FOGALOM szerelvény: pontosvessző segitségével elválasztott FOGALOM sorozat.

FAJTA zárójelpárba zárt FOGALOM: FAJTA_kezdőzárójel, FOGALOM,
FAJTA_végzárójel.

Konstansok, változók

a/ konstansok, jelölések

TIPUS konstans: TIPUS jelölés; TIPUS definiált azonositó.

egész jelölés: zérus: TERMÉSZETES.

valós jelölés: rögzitett; lebegő.

rögzitett: egész rész, tizedes rész; egész rész, tizedes vesszõ; tizedes rész.

egész rész: egész jelölés.

tizedes rész: tizedes vesszõ, egész jelölés; tizedes vesszõ, zérus szó, TERMÉSZETES.

lebegõ: rögzitett, kitevõ rész; egész jelölés, kitevõ rész; kitevõ rész.

kitevõ rész: tizes alap, elõjel, egész jelölés.

elõjel: ÜRES; plusz; minusz.

cimke jelölés: cimke azonositó.

parancs jelölés: alaputasitás.

karakterjelölés: karakter zárójelpárba zárt karakterjel.

karakterjel: BETÜ; SZÁMJEGY; speciális karakterjel.

speciális karakterjel: TIPUS ábécé betüje; FAJTA kezdõzárójel; FAJTA végzárójel;
szabványos müveleti jel; ALAPTIPUS deklarálója;
lista deklarálója; irásjel; dollárjel; egyéb elválasztó.

irásjel: pont; vesszõ; kettõspont; pontosvesszõ; kérdõjel; felkiáltójel.

egyéb elválasztó: plusz; minusz; tizedes vesszõ; tizes alap; igaz; hamis;
választójel; tárolás jele; legyen; ugrás jele;
lépés jele; végérték jele; végfeltétel jele; sztatikus;
dinamikus; szerkezetdeklaráló; láncdeklaráló;
veremdeklaráló; sordeklaráló; egytagu müveletdeklaráló;
kéttagu müveletdeklaráló; függvénydeklaráló; input;
output; definiciójel; egyenlõ; ürjel.

ürjel: ALAPTIPUS ürjel; üres verem; üres sor.

logikai jelölés: igaz; hamis.

lista jelölés: atom zárójelpárba zárt atom jelölés.

atom jelölés: ALAPTIPUS jelölés; TIPUS definiált azonosító;
TIPUS skaláris változó.

TIPUS elemekből KORLÁTPÁR futó indexü tömb jelölés: tömb zárójelpárba zárt
TIPUS jelölés felsorolás.

TIPUS elemekből KORLÁTPÁR és KORLÁTPÁROK futó indexü tömb jelölés: tömb
zárójelpárba zárt TIPUS elemekből KORLÁTPÁROK futó indexü tömb
jelölés felsorolás.

sorra MEZŐK mezőjü szerkezet jelölés: szerkezet zárójelpárba zárt MEZŐK jelölés.

AZONOSITÓ nevü TIPUS jelölés: AZONOSITÓ, kettőspont, TIPUS jelölés.

MEZŐ és MEZŐK jelölés: MEZŐ jelölés, vesszö, MEZŐK jelölés.

ALAPTIPUS lánc jelölés: lánc zárójelpárba zárt ALAPTIPUS jelölés kapcsolat.

SZÖVEGSZERÜ jelölés kapcsolat: SZÖVEGSZERÜ ürjel; SZÖVEGSZERÜ jelölés
felsorolás.

karakter jelölés kapcsolat: karakter ürjel; karakterjel szó.

logikai jelölés kapcsolat: logikai ürjel; logikai jelölés szó.

ALAPTIPUS verem jelölés: üres verem.

TIPUS sor jelölés: üres sor.

TIPUSl operanduszu TIPUS értékü müvelet jelölés: szabványos müveleti jel.

TIPUS1 és TIPUS2 operanduszu TIPUS értékü müvelet jelölés: szabványos müveleti jel.

TIPUSOK argumentumu TIPUS értékü függvény jelölés: fenntartott azonositó.

b/ változók

TIPUS változó: TIPUS általános skaláris változó; TIPUS elemekből TÖMB eleme; SZERKEZET TIPUS eleme; TIPUS lánc eleme; TIPUS verem eleme; TIPUS sor eleme.

TIPUS általános skaláris változó: TIPUS skaláris változó; TIPUS formális paraméter azonositója.

TIPUS skaláris változó: TIPUS azonositó; TIPUS ábécé eleme.

TIPUS elemekből KORLÁTPÁROK futó indexü tömb eleme: TIPUS elemekből KORLÁTPÁROK futó indexü tömb skaláris változó, index zárójelpárba zárt KORLÁTPÁROK indexkifejezés.

KORLÁTPÁR indexkifejezés: egész kifejezés.

KORLÁTPÁR és KORLÁTPÁROK indexkifejezés: egész kifejezés, vesszö, KORLÁTPÁROK indexkifejezés.

SZERKEZET TIPUS eleme: SZERKEZET egyik TIPUS tipusu mezöjének azonositója, kiválasztó zárójelpárba zárt SZERKEZET skaláris változó.

sorra MEZÖK mezöjü szerkezet egyik TIPUS tipusu mezöjének azonositója: MEZÖK közül egyik TIPUS tipusunak azonositója.

AZONOSITÓ nevü TIPUS közül egyik TIPUS tipusunak azonositója: AZONOSITÓ.

MEZŐ és MEZŐK közül egyik TIPUS tipusunak azonositója: MEZŐ közül egyik TIPUS tipusunak azonositója; MEZŐK közül egyik TIPUS tipusunak azonositója.

ALAPTIPUS lánc eleme: ALAPTIPUS lánc skaláris változó, index zárójelpárba zárt egész kifejezés.

ALAPTIPUS verem eleme: ALAPTIPUS verem skaláris változó, verem indexkifejezés.

verem indexkifejezés: ÜRES; index zárójelpárba zárt egész kifejezés.

TIPUS sor eleme: TIPUS sor skaláris változó.

Kifejezések

TIPUS kifejezés: TIPUS konstans; TIPUS változó; TIPUS függvénykifejezés; TIPUS bezárt kifejezés; TIPUS feltételes kifejezés; TIPUS1 operanduszu TIPUS értékü müvelet NÉV, TIPUS1 kifejezés; TIPUS1 kifejezés, TIPUS1 és TIPUS2 operanduszu TIPUS értékü müvelet NÉV, TIPUS2 kifejezés.

TIPUS függvénykifejezés: TIPUSOK argumentumu TIPUS értékü függvény NÉV, függvény zárójelpárba zárt TIPUSOK aktuális paraméterrész.

TIPUS aktuális paraméterrész: TIPUS kifejezés.

TIPUS és TIPUSOK aktuális paraméterrész: TIPUS kifejezés, vesszö, TIPUSOK aktuális paraméterrész.

TIPUS bezárt kifejezés: kifejezés zárójelpárba zárt TIPUS kifejezés.

TIPUS feltételes kifejezés: feltétel zárójelpárba zárt TIPUS választás.

TIPUS választás: logikai kifejezés, választójel, TIPUS kifejezés,
választójel, TIPUS kifejezés.

Utasitások

alaputasitás: cimkétlen alaputasitás; cimke azonositó, kettőspont,
alaputasitás.

cimkétlen alaputasitás: üres utasitás; értékadó utasitás; vezérlésátadó
utasitás; ciklus utasitás; összetett utasitás.

cimke azonositó: AZONOSITÓ.

üres utasitás: ÜRES.

értékadó utasitás: TIPUS kifejezés, tárolás jele, TIPUS változó.

vezérlésátadó utasitás: ugrás jele, cimke kifejezés.

ciklus utasitás: ciklus zárójelpárba zárt ciklus belseje.

ciklus belseje: vezérlő rész, utasitás.

vezérlő rész: TIPUS változó, legyen, TIPUS kifejezés, TIPUS lépés rész,
TIPUS ellenőrző rész, ciklusdinamika.

utasitás: parancs kifejezés; cimke azonositó, kettőspont, utasitás.

TIPUS lépés rész: ÜRES; lépés jele, TIPUS kifejezés.

TIPUS ellenőrző rész: végérték jele, TIPUS kifejezés; végfeltétel jele, logikai kifejezés.

ciklusdinamika: sztatikus, dinamikus.

összetett utasitás: utasitás zárójelpárba zárt utasitás szerelvény.

Deklarációk; definiciók; program

a/ deklarációk

deklaráció: TIPUS deklaráció.

TIPUS deklaráció: TIPUS deklarálója, TIPUS azonositó felsorolás.

TIPUS elemekből KORLÁTPÁROK futó indexü tömb deklarálója: index zárójelpárba zárt KORLÁTPÁROK bontás, TIPUS deklarálója.

EGÉSZSZÁM1 alsótól EGÉSZSZÁM2 felsőig bontás: EGÉSZSZÁM1, kettőspont, EGÉSZSZÁM2.

KORLÁTPÁR és KORLÁTPÁROK bontás: KORLÁTPÁR bontás, vesszö, KORLÁTPÁROK bontás.

sorra MEZŐK mezőjü szerkezet deklarálója: szerkezetdeklaráló, szerkezet zárójelpárba zárt MEZŐK bontás.

AZONOSITÓ nevü TIPUS bontás: AZONOSITÓ, kettőspont, TIPUS deklarálója.

MEZŐ és MEZŐK bontás: MEZŐ bontás, vesszö, MEZŐK bontás.

ALAPTIPUS lánc deklarálója: láncdeklaráló, ALAPTIPUS deklarálója.

ALAPTIPUS verem deklarálója: veremdeklaráló, ALAPTIPUS deklarálója.

TIPUS sor deklarálója: sordeklaráló, TIPUS deklarálója, háttértároló azonositója, file azonositója, file iránya.

háttértároló azonositója: fenntartott azonositó.

file azonositója: AZONOSITÓ.

file iránya: input; output; logikai kifejezés.

TIPUS1 operanduszu TIPUS értékü müvelet deklarálója: egytagu müveletdeklaráló, TIPUS1 deklarálója, TIPUS deklarálója.

TIPUS1 és TIPUS2 operanduszu TIPUS értékü müvelet deklarálója: kéttagu müveletdeklaráló, TIPUS1 deklarálója, TIPUS deklarálója.

TIPUSOK argumentumu TIPUS értékü függvény deklarálója: TIPUSOK függvénydeklaráló, TIPUSOK deklarálói, TIPUS deklarálója.

TIPUS függvénydeklaráló: függvénydeklaráló, kis i.

TIPUS és TIPUSOK függvénydeklaráló: TIPUSOK függvénydeklaráló, kis i.

TIPUS deklarálói: TIPUS deklarálója.

TIPUS és TIPUSOK deklarálói: TIPUS deklarálója, TIPUSOK deklarálói.

TIPUS azonositó: AZONOSITÓ.

b/ definiciók

definició: TIPUS definició.

TIPUS definició: definiciójel, TIPUS deklarálója, TIPUS definiciós egyenlet felsorolás.

PARAMÉTERTELEN definiciós egyenlet: PARAMÉTERTELEN definiált azonosító, egyenlő, PARAMÉTERTELEN kifejezés; PARAMÉTERTELEN skaláris változó, legyen, PARAMÉTERTELEN kifejezés.

TIPUS1 operanduszu TIPUS értékü müvelet definiciós egyenlet: TIPUS1 operanduszu TIPUS értékü müvelet konstans, TIPUS1 formális paraméter, egyenlő, TIPUS eljárástörzs; TIPUS1 operanduszu TIPUS értékü müvelet skaláris változó, TIPUS1 formális paraméter, legyen, TIPUS eljárástörzs.

TIPUS1 és TIPUS2 operanduszu TIPUS értékü müvelet definiciós egyenlet: TIPUS1 formális paraméter, TIPUS1 és TIPUS2 operanduszu TIPUS értékü müvelet konstans, TIPUS2 formális paraméter, egyenlő, TIPUS eljárástörzs; TIPUS1 formális paraméter, TIPUS1 és TIPUS2 operanduszu TIPUS értékü müvelet skaláris változó, TIPUS2 formális paraméter, legyen, TIPUS eljárástörzs.

TIPUSOK argumentumu TIPUS értékü függvény definiciós egyenlet: TIPUSOK argumentumu TIPUS értékü függvény konstans, függvény zárójelpárba zárt TIPUSOK formális paraméterrész, egyenlő, TIPUS eljárástörzs; TIPUSOK argumentumu TIPUS értékü függvény skaláris változó, függvény zárójelpárba zárt TIPUSOK formális paraméterrész, legyen, TIPUS eljárástörzs.

TIPUS formális paraméter: TIPUS deklarálója, TIPUS formális paraméter azonositója.

TIPUS formális paraméterrész: TIPUS formális paraméter.

TIPUS és TIPUSOK formális paraméterrész: TIPUS formális paraméter, vesszö, TIPUSOK formális paraméterrész.

TIPUS formális paraméter azonositója: AZONOSITÓ.

TIPUS definiált azonositó: AZONOSITÓ.

TIPUS eljárástörzs: utasitás kezdözárójel, utasitás szerelvény, pontosvesszö, TIPUS kifejezés, utasitás végzárójel.

c/ program

programelem: deklaráció; definició; utasitás.

program: programelem szerelvény, pontosvesszö, utasitás.

ű

Irodalomjegyzék:

1. Neumann J.: A számológép és az agy. Gondolat. Bp. /1964/

2. A.W. Burks -- H.H. Goldstine -- J.von Neumann: Preliminary discussion of the logical design of an electronic computing instrument. Inst. for Advanced Study, Princeton. /1946/

3. D.W. Barron: Computer Operating Systems. Chapman and Hall Ltd, London. /1971/

4. R.W. Watson: Timesharing System Design Concepts. MacGraw-Hill, New York. /1970/

5. D.N. Freeman: Pseudo-devices in OS/360.Proc. of the 1968 Southeastern ACM Regional Conference.

6. A.M. Turing: On computable numbers, with an application to the Entscheidungsproblem, Proc. of London Math. Soc. /2/, 42 /1936-37/. Reprinted in: Annual Review in Automatic Programming. 1 /1960/

7. HP 3000 Computer System Reference Manual. Hewlett-Packard Company /Apr. 1972/.

8. SPL 3000 HP 3000 Systems programming language textbook. Hewlett-Packard Data Systems Development Division /May 1972/.

9. J.P. Anderson: A computer for direct execution of algorithmic languages. AFIPS-Conf. Proc. /1961/, pp. 184-193.

10. A.J. Melbourne: A small compiler for the direct processing of FORTRAN statemens. The Computer Journal, Vol. 8. Nr. 1 /1965/, pp. 24-27.

11. T.R. Bashkow -- A. Sasson -- A. Kronsfeld: A system design of a FORTRAN-machine. IEEE-Trans, Vol. BC-16. Nr. 4 /1967/, pp. 485-499.

12. E.A. Hauck -- B.A. Dent: Burroughs'B 6500/B 7500 stack mechanism. Spring Joint Computer Conf. 1968, AFIPS-Conf. Proc., Vol. 32, pp. 245-251.

13. V.M. Gluskov: Analitik. Kibernetika, No.3 /1971/, pp. 102-134.

14. H. Weber: A Microprogrammed Implementation of EULER on IBM System/360 Model 30. Comm. ACM, Vol. 10. No. 9 /1967/, pp. 549-558.

15. T. Kilburn -- J. Morris -- J. Rohl -- F. Summer: A System design proposal. Information processing 68 -- Amsterdam. /1969/, pp. 806-811.

16. J. Green: Microprogramming, emulators and programming languages. Comm. ACM, Vol. 9 /March 1966/, pp. 230-232.

17. P.S. Abrams: An APL machine. Ph.D. Th. Stanford U., Stanford, Calif. /1970/

18. G.D. Chesley -- W.R. Smith: The hardware-implemented high-level machine language for SYMBOL. Proc. AFIPS 1971 SJCC, Vol. 39, AFIPS Press, Montvale, N.J., pp. 563-573.

19. P. Ercoli: System and programming aspects of the computer

19. P. Ercoli: System and programming aspects of the [illegible] INAC. Calcolo 3 /1966/, pp. 441-470.

20. Az elektronikus, digitális számítógépek eddigi fejlődése és a várható fejlődés fő irányai /Tanulmány/, Szeged, /1972/.

21. K. Samelson -- F.L. Bauer: Sequential Formula translation. Comm. ACM, 3 /1960/, pp. 76-83.

22. Kalmár L.: A practical infinitistic computer. Infinitistic Methods, Proceedings of the Symposium on Foundations of Mathematics, Warsaw 1959 /Warszawa 1961/, pp. 347-362.

23. Kalmár L.: Über einen Rechenautomaten, der eine mathematische Sprache [illegible] versteht. ZAMM, 40 /1960/, pp. T 64-65.

24. Z. Pawlak: The organization of a digital computer and computable functions. Bull. Acad. Polon., Sci., Sér. Sci. Techn. 8 /1960/, pp. 41-51. Organization of the addressfree digital computer for calculating simple arithmetical expressions. Ugyanott, 8 /1960/, p. 685; Organization of address-free computer B-100. Ugyanott, 9 /1961/, pp. 229-234.

25. V.M. Gluskov: Vhodnüj jazik vücsiszlityelnoj masinü "MIR". Kibernetika, 1. /1965/.

26. L. Kalmár: O cifrovoj vücsiszlityelnoj masine sz programirovanyijem poszredsztvom formalnogo matematicseszkogo jazika. Kiberneticseszkij Szbornyik, Novaja Szerija 1. Moszkva /1965/, pp. 215-226.

27. L. Kalmár: Ist ALGOL wirklich eine algoritmische Sprache? Automatentheorie und formale Sprachen, Mannheim, Wien, Zürich. /1969/, pp. 305-315.

28. T. Kilburn et al.: The Manchester University Atlas Operating System. The Computer Journal, Vol. 4. No. 3 /1961/

29. A. van Wijngaarden /editor/: Report on the Algorithmic Language ALGOL 68. Mathematisch Centrum, Amsterdam. /1969/

30. Draft Revised Report on the Algorithmic Language ALGOL 68. Az IFIP WG 2.1 munkacsoportja Los Angelesben tartott ülésének anyaga. /1973/

Hibajegyzék a "JAVASLAT A FORMULAVEZÉRLÉSŰ SZÁMITÓGP BELSŐ

NYELVÉRE VONATKOZÓAN" c. tanulmányhoz

2. fejezet

3. old., alulról 9. sor végén helyesen: szöveg-transzformációk
alulról 5. sor végén helyesen: kompilá-

4. old., felülről 13. sorban az elemzéséhez szó után vessző kell

13. old., felülről 7. sorban az eljárás szó után nem kell kötőjel

33. old., alulról 1. sorban zárójelentéssel helyett: záró-jelezéssel

34. old., felülről 9. sorban függvényjelet helyett: függvé-nyeket

3. fejezet

4. old., felülről 2. sorban kompilátorirás helyett: kompi-látor-irás

5. old., alulról 10. sorban a hasonló szó elé betoldandó: és szorzáshoz

8. old., felülről 4. sorban mikroprogramtáraknak helyett: mikroprogramtáraké
alulról 12. sorban a jelentenek szó után vessző kell
alulról 7. sorban értünk helyett: értjük

9. old., alulról 11. sorba a leirásában szó után betoldandó: /WIJNGAARDEN, 1969/; alulról 3. sorban minden egyes két szóba irandó

13. old., felülről 3. sorba a jelentésben szó után betoldandó: /DRAFT REVISED REPORT ON THE ALGORITHMIC LANGUAGE ALGOL 68, 1973/
felülről 7. sorban helyesen: metaszabályok
felülről 15. sorban kifejezéseinek helyett: kifej-téseinek

14. old., felülről 10. sorban minden egyes két szóba irandó
felülről 12. sorban az ugyanazon szó után az az szó törlendő
alulról 8. sorban már egy helyesen egy már
alulról 6. sorban pótolnunk helyett: pótoljuk

15. old., felülről 2. sorban az egyszerü is aláhuzandó
felülről 10. sorban kifejtésekkel helyett: kifej-téseikkel

17. old., felülről 2. sorban utaló helyett: mutató

19. old., felülről 8. sorban a karakter szó után vesszo kell

23. old., felülről 11. sorban után helyett: utána

alulról 7. sor végén nem kell kötőjel

24. old., alulról 4. sorban elemeiket helyett: elemeinek jelölését .

29. old., felülről 13. sorban azon helyett: a benne a feltételen kivül szereplő

32. old., alulról 6. sorban a látszanak szó után névelő kell.

33. old., alulról 6. és 1. sorban számológép helyett: számitógép

37. old., felülről 1. sorban az is, mint szavak törlendők

38. old., felülről 5. sorban a szerkezet szó után nem kell vessző

alulról 7. sorban az ettől is, szavak /a vesszővel együtt/ és a sor végén az is szó is törlendők

39. old., felülről 9. sorban az egyenlőségjel szó után, vesszővel, betoldandó: illetőleg a legyen-jel

alulról 14. sorban helyesen: definiálandó

36. old. alulról 4. sorban . jára helyett : olvasása