NJSZT

MŰSZAKI ÉS TERMÉSZETTUDOMÁNYI
EGYESÜLETEK SZÖVETSÉGE
NEUMANN JÁNOS SZÁMÍTÓGÉPTUDOMÁNYI TÁRSASÁG

PROGRAMOZÁSI RENDSZEREK '81

I-II. kötet

SZEGED
1981. DECEMBER 2–4.

Neumann János Számítógéptudományi Társaság

PROGRAMOZÁSI RENDSZEREK '81

Konferencia előadásai

I. Kötet

Szeged
1981. december 2–4.

Szerkesztette: Dávid Sándor

Készült az OVK nyomdaüzemében 250 példányban
Felelős kiadó: Dr. Nemes Ferenc
Munkaszám: 811112

E L Ő S Z Ó

Kötetünk a Programozási Rendszerek'81 Konferenciára benyujtott 64 előadás közül elfogadott 46 előadás szövegét tartalmazza.

Aki ebből a kötetből kisérelné megitélni a programozók munkáját, bizonyára nem kapna teljes képet. Amennyire örvendetes, hogy két-három kutatási/fejlesztési téma /igy az ADA-projekt, a sejtprocesszor szoftverjének kutatása/ itt egységesen, a teljes kép lerajzolására törekedve jelenik meg ebben a kötetben - ugyanannyira elgondolkoztató, hogy az alkalmazási programokról beszámoló cikkek száma jelentőségükhöz képest még mindig kevés.

Bizunk abban, hogy a szoftver készitése szervezeti kötöttségeinek megszünése, oldása a jövőben a konferencia-sorozatunkra benyujtott előadások spektrumát szélesiteni fogja és ez választ fog adni arra a tartalmi kérdésre, hogy ezen formai oldás által felpezsdülő mozgalom a szoftver-bütykölés intézményesitése, vagy az alkalmazások igényeihez jobban igazodó szoftver készitésének intézményesitése alternativái közül melyik felé konvergál.

Ebben az értelemben uj korszak előtt állunk. Az utóbbi tiz év a szoftver-technológia jegyében telt el - komoly eredményeket mutathatunk fel. Az alkalmazási rendszerek technológiájába ezek közül átvehetünk és átfogalmazhatunk eredményeket, és uj kérdéseket kell feltennünk.

Ehhez szervezeti, társulati, tartalmi feltételek adottak.
Csak rajtunk mulik, hogy ezekkel a lehetőségekkel hogyan tudunk élni.

Dávid Gábor

TARTALOMJEGYZÉK

I. kötet

II. kötet

Almási József

A CHANGE NYELV IMPLEMENTÁLÁSA

A CHANGE nyelv kvázi párhuzamos folyamatok modellezésére alkalmas, önkiterjesztő, univerzális programozási nyelv. A CHANGE nyelvet FORTRAN nyelven implementáltuk. Az elkészült CHANGE programrendszer a CDC 3300-as, az IBM 3031-es és az R-40-es számitógépeken müködik. Dolgozatunkban először röviden leirjuk a CHANGE nyelv fő jellemzőit. A továbbiakban ismertetjük, az implementálás során használt néhány, fontosabb belső adatszerkezetet /valamint használatuk indokait/. Végül a nyelv néhány alkalmazását, és a továbbfejlesztési terveket ismertetjük.

Kulcsszavak: adat kiterjesztés, utasitás kiterjesztés, program generálás, kvázi párhuzamos processzálás, dinamikus adatstruktura

A CHANGE nyelvet Legendi Tamás definiálta [LEG 1] . Az implementálásnál nem törekedtünk a teljességre, a [LEG 1] -ben definiált utasitáskészletnek azt a részhalmazát implementáltuk, amely legjellemzőbb a CHANGE nyelvre. Ezen utasitáskészlet részletes leirása [LEG 3] -ban található.

1. A CHANGE nyelv rövid, általános jellemzése

A CHANGE nyelv processzor orientált, önkiterjesztő, univerzális programozási nyelv [LEG 2] .

A nyelv - utasitás és tipuskiterjesztő utasitásai segitségével alkalmas célorientált nyelvek definiálására és processzoraik létrehozására, valamint jól strukturált programok irására;

- lehetővé teszi a CHANGE programok futás közbeni /ön/módositását, generálását és azok ujrafordítás nélküli végrehajtását;

- a végrehajtási módot előiró utasitásai segitségével tetszőleges /fa-szerü/ processzorstruktura hozható létre /ezek a virtuális processzorok kvázi-párhuzamosan hajtják végre az adott CHANGE programot/,
- lehetővé teszi a program aktuális állapotának forrásnyelvi szintre való visszahozását, a kiterjesztő, a könyvtárkezelő, valamint speciális utasitásainak segitségével más nyelv/ek/re való forditását,
- nyomkövető utasitásai segitik a programok gyors, biztonságos belövését.

Minden CHANGE nyelven irt programot tekinthetünk olyan programnak, amelyet minden időpontban véges számu párhuzamos müködésü /virtuális/ processzor hajt végre. A processzorok alá- és mellérendelt viszonyban állhatnak egymással. A processzorstruktura a feladat jellegétől függően a végrehajtás alatt dinamikusan változhat. Minden processzornak /a futás során változtatható/ saját utasitáskészlete, könyvtára, programja lehet.

2. A CHANGE programok felépitése

Egy CHANGE_program_ /alap, vagy kiterjesztett/ CHANGE utasitások sorozatából áll. A CHANGE_utasitások alapszavakból és adathivatkozásokból állnak. A CHANGE utasitásokban lévő adathivatkozásokat paramétereknek nevezzük. Egy paraméter lehet: konstans, vagy változó, vagy változó és indexkifejezés.

3. A CHANGE programok végrehajtása

Egy program végrehajtását az 1. sorszámu processzor kezdi /utasitásszámlálójának és utasitásszámláló módositójának tartalma + 1, mellérendeltje önmaga, alá- és fölérendeltje nincs/.

A továbbiakban a programot tetszőleges számu processzor hajtja végre /saját sorszámmal, ütemezéssel, utasitáskészlettel, könyvtárral és programmal/. A processzorok /kvázi/ párhuzamosan müködnek. Müködésük csak a CHANGE nyelv által biztositott /végrehajtási módot előiró/ utasitások hatásán keresztül függhet egymástól.
A program végrehajtási egysége a végrehajtási lépés, amelynek során minden, aktuálisan müködő processzor egy /annak utasitásszámlálója által kijelölt/ utasitása hajtódik végre.
Ha nincs aktuálisan müködő processzor, a program végrehajtása befejeződik.
A CHANGE programok végrehajtása során, zárt utasitáskiterjesztéssel - a zárt kiterjesztett utasitások végrehajtásakor - egymástól független processzorstrukturákat hozhatunk létre /amelyek a fentiekben leirt módon müködnek/. A zárt kiterjesztett utasitások /szemantikájuktól függetlenül/ az adott utasitást végrehajtó processzor 1 végrehajtási lépésében hajtódnak végre.

4. A CHANGE nyelv utasitáskészletében lévő főbb utasitáscsoportok

4.1 Dinamikus deklarációs utasitások

Ezek az alap és kiterjesztett tipusu tömbök méretét dinamikusan előiró utasitások. A deklarációs utasitások végrehajtható utasitások. A CHANGE programban tetszőleges helyen szerepelhetnek, akárhányszor végrehajthatók.

4.2 Értékadó utasitások

Ebbe az utasitáscsoportba tartoznak az egyszerü értékadó, az aritmetikai müveleteket, és a belső függvények értékét kiszámitó utasitások.

4.3 Vezérlésátadó utasitások

Ezek az utasitások a feltétlen, a feltételes, a végrehajtási mód megváltoztatásával elérhető vezérlés átadások, valamint a szubrutin hivás /tetszőleges mélységü és rekur-

ziv szubrutin hivás is megengedett/.

4.4 Utasitás_generáló_utasitások_

Automatikus programmódositó /CHANGE/ utasitások, amelye a program valamely utasitásának helyébe a program valamely utasitásának aktuális értékét generálják.

4.5 NULL_utasitások

Hely biztositására és processzorok szinkronizálására használható /üres/ utasitások.

4.6 Végrehajtási_módot_előiró_utasitások_

A processzorstrukturát létrehozó, módositó /processzor inditó, megállitó, processzorok bizonyos paramétereit lekérdező, megváltoztató/ utasitások.

4.7 Adatátviteli_utasitások

Ide tartoznak az input/output utasitások.

4.8 Kiterjesztő_utasitások_

Ide tartoznak az utasitás és tipus kiterjesztő utasitások.

Tipus kiterjesztéssel az alaptipusokból /INT, REAL, BOOL, CHAR, TEXT/ tetszőleges bonyolultságu kiterjesztett adattipus létrehozható.

Utasitás kiterjesztéssel a már definiált /alap és kiterjesztett/ utasitásokkal uj utasitáskészletet hozhatunk létre. A szintaktikus kiterjesztés forditás, mig az adott szintaktikához való adott szemantika hozzárendelése a végrehajtás során /dinamikusan/ történik. Ugyanahhoz a szintaktikához futás során több /más/ szemantika is tartozhat /akár egyidejüleg is/. Pl. különböző processzorok számára ugyanazon szintaktikáju utasitáshoz más és más szemantika tartozhat.

4.9 Könyvtárkezelő_utasitások

Futás közben a processzorhoz rendelt utasitáskészletet, könyvtárat módositják, változtatják meg.

4.10 Nyomkövető utasitások

A program-végrehajtás ellenőrzését teszik lehetővé, segitik a programok tesztelését.

4.11 Speciális utasitások

Ezek az /OPEN, LIST, TRANSLATE/ utasitások a CHANGE program forrásnyelvi szintre, illetve más /pl. FORTRAN/ programnyelvre való forditását végzik el.

5. A termelő-fogyasztó probléma egy lehetséges megoldása CHANGE nyelven [SIM 1]

Hely hiányában csak a CHANGE nyelv három jellegzetességét /tipus, utasitás kiterjesztés, párhuzamos processzálás/ tudjuk bemutatni egy egyszerü példán keresztül.

A probléma felvetése: Adott egy ciklikus puffer, amelybe egy processzor folyamatosan tölti az adatokat, egy másik folyamatosan olvassa belőle az adatokat. A feladat megoldásához szemafor tipusu változókat definiálunk. A SEMAPHORE tipus kiterjesztett tipus. Definiálunk három szemafor utasitást /nyilt kiterjesztett utasitásként/, amelyek elvégzik a szemafor tipusu változók kezelését.
A feladatot két lépésben oldottuk meg. Az első /a., program/ definiálja a szemafor tipust, valamint a szemafor tipusu változók kezelését végző utasitásokat. A második /b., program/ felhasználva az első által definiált utasitáskészletet, megoldja az I/O processzorok szinkronizálását a ciklikus puffer töltése, olvasása közben.

```
a., program
C SZEMAFOR TIPUS ES A
C SZEMAFOR TIPUSU VALTOZOKAT KEZELŐ UTASITASOK
C DEFINIALASA
      EXTEND TYPE SEMAPHORE, INT
      TEXT O,P
      $OPEN$ TO O
      $PERMANENT$ TO P
```

```
      EXTEND O,P, SEMANTICS AT 5,$V SEMAPHORE:V $
      EXTEND O,P, SEMANTICS AT 1,$P SEMAPHORE:V $
      EXTEND O,P, SEMANTICS AT 7,$ SEMAPHORE:V=INT:E$
      EXTEND O,P, SEMANTICS AT 10,$ INCREMENT INT:V MOD INT:E$
      LIBRARY TO FILE 50
      REWIND 49
      REWIND 50
      STOP
 4    I==SEMAPHORE(1),INT(1)
 9    J==INT(1)
 13   K==INT(2)
 1    SEMANTICS BODY 2 - 3, PARAMETERS 4 - 4
 2    IF(I .LE. 0 ) 2,3
 3    I=I - 1
 5    SEMANTICS BODY 6 - 6, PARAMETERS 4 - 4
 6    I=I + 1
 7    SEMANTICS BODY 8 - 8, PARAMETERS 4 - 9
 8    I=J
 10   SEMANTICS BODY 11 - 12, PARAMETERS 9 - 13
 11   IF(J .LT. K ) 12, 14
 14   J=0
 12   J=J + 1
      FINIS
```

b., program

```
      LIBRARY FROM FILE 50
      SEMAPHORE S0,S1,S2,S3
      INT BUFFER(10),B,C,I,J,K,L
      B=1
      C=1
      S0=-1
      S1=1
      S2=10
      S3=0
      SUBPROCESSOR 2,2
      SUBPROCESSOR 3,3
```

```
      P(SO)
      STOP
C 2-ES PROCESSZOR PROGRAMJA
 2    LIBRARY FROM PROCESSOR 1
      DO 4 I=1,3O,1
      READ(6O,$(I4)$)K
      P(S2)
      P(S1)
      BUFFER(B)=K
      INCREMENT B MOD 1O
      V(S1)
      V(S3)
 4    REPEAT
      V(SO)
      STOP
C 3-AS PROCESSZOR PROGRAMJA
 3    LIBRARY FROM PROCESSOR 1
      DO 5 J=1,3O,1
      P(S3)
      P(S1)
      L=BUFFER(C)
      INCREMENT C MOD 1O
      V(S1)
      V(S2)
      WRITE(61,$(1X,I4)$)L
 5    REPEAT
      V(SO)
      STOP
      FINIS
```

6. A CHANGE nyelv implementálása

A CHANGE nyelvet FORTRAN nyelven implementáltuk. Az implementálásnál messzemenően figyelembe vettük a több gépre való átvihetőséget. /Ez is indokolja a FORTRAN bázis nyelv választását./ A programrendszer készitésekor gépfüggő jellemzőket

kizárólag akkor használtunk, ha az elkerülhetetlen volt. A gépfüggő jellemzőket igyekeztünk központositva kezelni, hogy más gépre való átvitelkor könnyen lecserélhetők legyenek. Az IBM 3031-en, az R-40-en és a CDC 3300-as gépen müködik a CHANGE rendszer.

A CHANGE rendszer kb. 8000 FORTRAN utasitás. Kb. másfélszer ennyi CHANGE nyelvü teszt is készült az átadáshoz /KOV 1/. Ezen kivül elkészült a CHANGE rendszer és a tesztprogramok teljes dokumentációja is [ALM 1 és KOV 1].

7. A CHANGE rendszer felépitése

Az implementációt három tagu programrendszer alkotja.

1./ LIBGEN: A CHANGE nyelv utasitásainak szintaktikus leirását transzformálja belső formába, létrehozva az alapkönyvtárat.

2./ COMP: A fordító az alap, vagy a kiterjesztett könyvtár alapján a forrásnyelvü programot belső kódra forditja.

3./ MULTI: A belső kódu programokat értelmezi és végrehajtja.

8. A CHANGE rendszer adatszerkezetei [ALM 1, ALM 2]

A CHANGE rendszer alap adatstrukturája dinamikus rendszert alkot. A CHANGE rendszerhez az operativ memóriában egy nagyméretü statikus tömb tartozik. Ebben a tömbben helyezkednek el a dinamikus rendszer elemei a dinamikus tömbök.

A dinamikus tömbök szerkezete:

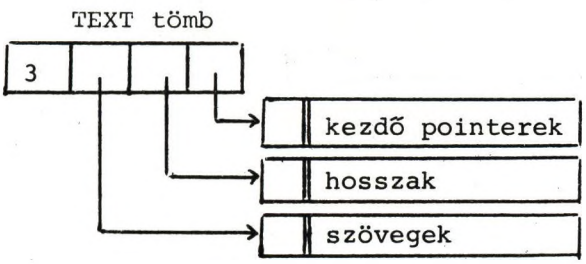
tömbfej	1.tömbelem	2.tömbelem	...	n.tömbelem

A tömbfej tartalmazza az összes tömbelem által lefoglalt terület hosszát. A tömbelemek szerekezete a tömb tipusától /valamint a gépi reprezentációtól/ függ.
A forditás, végrehajtás során a rendszer számára szükséges

táblázatokat, tömböket, valamint a felhasználói tömböket dinamikus tömbként ábrázoljuk.
A felhasználói tömbök dinamikus tömbként való ábrázolását a dinamikus deklarációs utasitások /ld. 4.1/, valamint az automatikus dimenzionálás utasitás szinten történő vezérelhetősége indokolja. A dinamikus deklarációs utasitások használatával elérheti a felhasználó, hogy a programjában definiált tömbök a program futásától függően mindig optimálisan foglalják le a rendelkezésre álló tároló területet.
A rendszer tömböket azért helyeztük el dinamikus tömbökben, mivel igy az egyes tömbök konkrét mérete nem befolyásolja egy adott tevékenység, funkció /pl. kiterjesztés/ végrehajtásának számát, milyenségét. Kizárólag a rendelkezésre álló memória /amely tárolja a dinamikus rendszert/ az egyetlen korlátozó tényező. A rendszer tömbök fenti ábrázolása lehetővé tette, hogy a rendszer és a felhasználói tömböket ugyanaz a rutin készlet egységesen kezelje. Az egységes kezelés csökkenti a teljes programméretet, valamint segiti az egyik gépről másikra való átvitelt.

9. A CHANGE nyelv alap adattipusainak belső ábrázolása

Az egész, valós, logikai és karakter tipusu tömbök egy-egy dinamikus tömbben vannak elhelyezve /lásd 8./.
Egy szöveg tipusu tömb elhelyezésére négy dinamikus tömb szolgál:



A szöveg tipusu tömb fenti ábrázolása lehetővé teszi, hogy:
a., az egyes szöveg tömb elemek változó hosszuságuak lehetnek

b., az egyes szövegek folytonos tárterületen elhelyezhetők

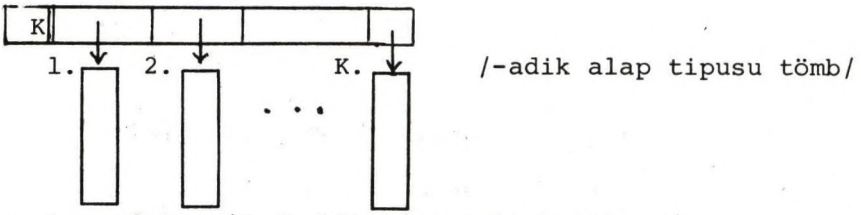
c., az egyes szöveg tömb elemek elérése, megváltoztatása egyszerüvé válik.

A fenti ábrázolásmód maximálisan támogatja a CHANGE nyelv szövegkezelő utasitásainak végrehajtását pl.: egy tömbelem hosszának lekérdezése csak a megfelelő hosszat tároló tömbelem kiolvasását jelenti.

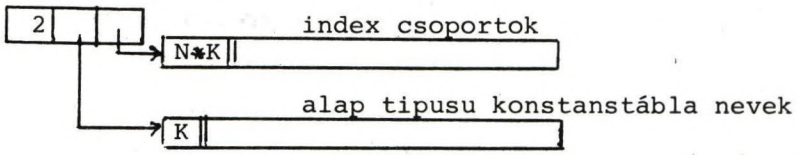
10. Kiterjesztett tipusu tömbök

A felhasználó a tipuskiterjesztő utasitás segitségével az alap tipusokból és a már definiált kiterjesztett tipusokból tetszőleges bonyolultságu kiterjesztett adattipust definiálhat. Minden kiterjesztett adattipust /a felhasználó által definiált módon/ alaptipusokból épithetünk fel.
Ha egy kiterjesztett adattipus K alap tipusból épül fel, akkor a neki megfelelő

- felhasználóitömb:



- konstanstábla: /N db kiterjesztett konstans/



A kiterjesztett tipusu konstansok ábrázolását helytakarékossági okokból visszavezettük az alap tipusu konstansok ábrázolására.
A kiterjesztett tipusu felhasználói tömbök fenti ábrázolását a következők indokolják:

a., kiterjesztett utasitások végrehajtásakor a formális-aktuális paraméter cserénél egyszerübb a kiterjesztett tipusu paraméterek átadása

b., a kiterjesztett tipusu változók tényleges kezelését a felhasználónak alap tipusu tömbök kezelésére kell visszavezetnie

c., a b.,-ben leirt felbontásban az egyes részparaméterek egymástól függetlenül kezelhetők.

11. A CHANGE programok belső ábrázolása

A CHANGE fordító elvégzi a CHANGE programok szintaktikus elemzését és a programnak megfelelő /magas szintü/ belső kódot generál, amit az interpreter végrehajt.
A program generálás és a magas szintü nyomkövetés szükségessé teszi, hogy a leforditott programban az eredeti, forrásnyelvi utasitásoknak megfelelő egységek hozzáférhetők legyenek. A magas szintü belső kódot indokolja még a forrásnyelvi szintre való visszahozhatóság /LIST utasitás/, valamint a más nyelvekre való fordithatóság /TRANSLATE utasitás/.
A programok belső ábrázolása az utasitások szintaktikus egységeinek megfelelően történik. A belső adatstruktura, amely leir egy CHANGE programot, tömöritett: minden utasitás, illetve paraméter leirása pontosan egyszer tárolódik. Ez az ábrázolás megkönnyiti a szekvenciális végrehajtási mód feloldását, gyorsitja a nyelv tipikus utasitásainak végrehajtását, másrészt memória nyereséget tesz lehetővé. /A CHANGE programok belső kódját dinamikus tömbökben tároljuk./

12. A processzorstruktura megvalósitása

A CHANGE végrehajtó minden CHANGE programhoz az abban utasitásokkal leirt/generált processzorstrukturát rendeli /dinamikusan/. A processzorok utasitáskészlete, programja, müködése függhet a CHANGE program végrehajtásától. A végrehajtó biztositja a processzorok kvázi párhuzamos müködését, a processzorok szinkronizálását azonban a felhasználóra bizza.
A szinkronizálást tipus, utasitás kiterjesztéssel viszonylag egyszerüen megoldhatja a felhasználó /lá. példaprogramot az 5.-ban/.

Mivel a processzorstruktura a program müködésének függvényében dinamikusan alakul, változik, kézenfekvő volt, hogy a strukturát leiró adatokat /a processzorleiró mezőket/ dinamikus tömbben helyezzük el. Egy processzorleiró mező tartalmazza a processzor müködéséhez szükséges /a müködést pillanatnyilag jellemző/ összes információt /pl.: a processzor sorszámát, utasitásszámlálójának tartalmát, a többi processzorhoz való viszonyát leiró adatokat/.
A futás során létrejövő /fa-szerü/ processzorstrukturát kétirányu láncok irják le. Ez az ábrázolás meggyorsitja a processzorokon végzett müveletek végrehajtását, valamint maximálisan támogatja a processzorokkal kapcsolatos CHANGE utasitások végrehajtását /pl.: alá-, vagy mellérendelt processzorok sorszámának lekérdezését/.
A CHANGE végrehajtó a kvázi párhuzamos müködést ugy biztositja, hogy a végrehajtási lépést két részre bontja. Az első részben végrehajtódik az összes olyan utasitás, amely a végrehajtási módot, a processzor jellemzőket és a végrehajtandó programot nem változtatja meg /pl.: ilyenek az értékadó utasitások/, az összes többi utasitás végrehajtásának /pl.: CHANGE utasitás, utasitásszámláló módositás stb./ csak az előkészitése történik meg. A második részben a végrehajtási lépés lezárása történik; itt hajtódnak végre ténylegesen azok az utasitások, amelyeket az első részben nem lehetett végrehajtani.

13. A kiterjesztések végrehajtása

A CHANGE nyelv kiterjesztő utasitáskészletének segitségével uj utasitáskészletet állithatunk elő az alap utasitásokból /utasitás kiterjesztés/, valamint uj adattipusokat definiálhatunk az alap adattipusokból /tipus kiterjesztés/. A kiterjesztés során bármely előzőleg, kiterjesztéssel létrehozott objektumra is hivatkozhatunk. A kiterjesztés speciális, két fázisra bomlik. A szintaktikus kiterjesztés a forditás során /statikusan/, a szemantikus kiterjesztés a végrehajtás során /dinamikusan/ történik.

A fordító a kiterjesztő utasitás feldolgozása során uj szintaktikus egységekkel bőviti /a dinamikus tömbökben elhelyezett/ szintaktikus könyvtárat. A szintaktikus kiterjesztés után a fordító egységesen kezeli, forditja az alap és a kiterjesztett utasitásokat. /Ezt a szintaktikus könyvtár felépitése és a program belső kódu ábrázolása teszi lehetővé./
A szemantikus kiterjesztés a végrehajtás során akkor történik meg, amikor a kiterjesztő utasitás végrehajtódik. Mivel a kiterjesztés paraméterezhető, és a kiterjesztő utasitás akárhányszor végrehajtható, lehetőség van arra, hogy a program végrehajtása során ugyanahhoz a szintaktikához több szemantikát is hozzárendeljünk. /Ugyanezt a hatást elérhetjük a könyvtárkezelő utasitások segitségével is, ha ugyanazt az utasitást más és más szemantikus könyvtár alapján hajtjuk végre./
A kiterjesztő utasitás végrehajtásakor a végrehajtó az aktuális szemantikát felveszi az utasitást végrehajtó processzor /dinamikus tömbökben elhelyezett/ szemantikus könyvtárába.
A fentiekben leirt kiterjesztő mechanizmus viszonylag egyszerüen megvalósitható és rugalmas eszközt ad a felhasználó kezébe célorientált nyelvek definiálásához, jól strukturált programok irásához.

14. A CHANGE nyelv néhány alkalmazása

Az implementálással párhuzamosan megkezdtük az alkalmazói programok irását is. Készültek:

- célorientált nyelveket definiáló, alkalmazó CHANGE programok pl.: dialógus nyelv [LEG 4]
  listakezelő nyelv [KOM 1]
  NC nyelv [ALM 3]
  mikroszoftver generáló nyelv [ALM 4]
  oktató programok /CAI/;
- a CHANGE nyelv párhuzamos processzálását kihasználó programok pl.: K-fa kereső
  sejtszerü müködés szimulálása
  osztott differenciák kiszámitása
  rekurziv egyenletrendszer megoldó program.

15. A CHANGE rendszer továbbfejlesztésével kapcsolatos tervek

A CHANGE rendszer tesztelése és az alkalmazói programok futtatása során szerzett tapasztalatok alapján jelenleg is folyik a programrendszer továbbfejlesztése, gyorsitása.
Jelenleg kifejlesztés alatt áll egy olyan CHANGE nyelvü tesztrendszer, amellyel a fordító, végrehajtó bizonyos tevékenységekre forditott gépidő igényét szeretnénk kimérni. Ezen tesztrendszer futtatása során szerzett tapasztalatok alapján szeretnénk még hatékonyabbá, gyorsabbá tenni a CHANGE programrendszert.
A továbbiakban kifejlesztésre kerül egy olyan programrendszer, amely a CHANGE rendszer egyik géprőI másikra való átvitelét automatizálja.

Abstract

There is a self-extensible programming language named CHANGE, which can be used for modelling of quasi-parallel processes. The implementation of the CHANGE was made in FORTRAN language. The system is now running on the CDC-3300, IBM 3031 and R-40 computers. In the first part of the lecture the main properties of the CHANGE language are introduced. After that the representations of some important data structures used in the course of implementation will be found. Finally some application examples and the design of developement of the system are presented.

Keywords: data extension, statement extension, self-modifying programs, quasi-parallel processing, dynamic representation of data structure

Irodalomjegyzék:

ALM 1 Almási J.: A CHANGE rendszer karbantartói dokumentációja MTA Automataelméleti TKCS, Szeged, 1980

ALM 2 Almási J.: A CHANGE processzor implementálása és CHANGE programozás
Diploma munka, JATE, 1981

ALM 3 Almási J.: A CHANGE nyelv alkalmazása NC nyelvek definiálására
Mérés és Automatikához benyujtott cikk

ALM 4 Almási J., Legendi T., Szajbély Gy., Szekeres Sz., Tóth K.: Mikroszoftver generálása felhasználói kiterjesztésekkel létrehozott célnyelveken irt és tesztelt programokból a CHANGE processzor felhasználásával
Programozási Konferencia '81-re benyujtott előadás

KOM 1 Komoróczki E.: A CHANGE programozási nyelv alkalmazása Diploma munka, JATE, 1981

KOV 1 Kovács I., Szajbély Gy., Rózsás A.: A CHANGE tesztrendszer
Dokumentáció, Szeged, 1980

LEG 1 Legendi T.: A CHANGE nyelv/multiprocesszor
SZTAKI Tanulmányok 1973/7
Egyetemi doktori értekezés, JATE, 1975

LEG 2 Legendi T.: A CHANGE nyelv szemlélete, implementálása, és alkalmazása
Kézirat

LEG 3 Legendi T., Almási J., Karvaly G., Vas Z.:
CHANGE felhasználói ismertetők CDC 3300 felhasználói ismertetők
Budapest, 1980

LEG 4 Legendi T., Szajbély Gy.: Dialógus programok készitése a CHANGE programozási nyelv segitségével
Programozási Konferencia '81-re benyujtott előadás

SIM 1 Simon E.: Language design objectives and the CHANGE system
közlésre elfogadva a CL& CL folyóiratnál

Almási József
MTA Automataelméleti Tanszéki Kutató Csoport
6720 Szeged, Somogyi u. 7.

Almási József–Legendi Tamás–Szajbély György–Szekeres Szilveszter–Tóth Károly

MIKROSZOFTVER GENERÁLÁSA FELHASZNÁLÓI KITERJESZTÉSEKKEL LÉTREHOZOTT CÉLNYELVEKEN ÍRT ÉS TESZTELT PROGRAMOKBÓL A CHANGE PROCESSZOR FELHASZNÁLÁSÁVAL[1]

A VOLÁN 10. sz. Vállalatnál kifejlesztésre került egy mikrogép - a TENV -, amelyet adatgyüjtési és adatlekérdezési céleszközként alkalmaznak. A céleszköz programozása direkt módon csak a gépi kódhoz közel álló SLANG nyelven történhet, amely hosszadalmas, nehézkes munkát igényel. Ezen probléma megoldására kerestünk egy olyan, a SLANG-nél magasabb szintü programozási nyelvet, amelyen a feladatok kényelmesen programozhatók, tesztelhetők, valamint a TENV-en is futtathatóvá tehetők. Céljaink megvalósitásához a CHANGE nyelvet választottuk.

A CHANGE nyelven megirt és letesztelt programokat egy közbülső nyelvi szintre, a SLAGH nyelvre forditottuk. /A SLAGH nyelv utasitásainak szemantikáját SLANG nyelven irt rutinok hivási sorozata adja meg./

KULCSSZAVAK: CHANGE, kiterjeszthető nyelv, mikrogép, mikroszoftver generálás

---

[1] Ebben a cikkben ismertetett szoftver készitési eszköz fejlesztése a VOLÁN 10. sz. Vállalat és a JATE Számitástudományi Tanszék között létrejött 237/1980. számu szerződés keretében történt.

1. A TENV számitógép adottságai, hatékony programozásának lehetősége

A VOLÁN 10. sz. Vállalat Számitástechnikai Főosztályán 1979/80-ban kifejlesztésre került egy mikroprocesszor bázisu számitógép tipus, a TENV, amely utasitáskészlet szinten kompatibilis a PDP-8, ill. TPA/i számitógéppel. Ezek fő felhasználási célja az, hogy különböző kihelyezett pontokon intelligens adatgyüjtőként, ill. adat felvevőként üzemeljen, valamint egyszerübb adatfeldolgozásokat elvégezzen szakképzett operátor nélkül. A kihelyezett céleszköz csak mágneskazettás perifériákkal és esetenként konzol display-vel van ellátva. Igy ezen operációs rendszer müködtetése nem lehetséges.

A fentiek alapján a céleszköz programozása csak a gépi kódhoz közel álló SLANG nyelven történhet. Ezen a nyelven a programozás lassu és nehézkes. Célunk az volt, hogy olyan lehetőséget teremtsünk, amely ezt a problémát megoldja.

1.1 A TENV számitógép adottságai:

A TENV CMOS és TTL bázisu, a processzor INTERSIL 6100-as tipusu, a memória 16 k-szor 1 bites RAM IC-kből épül fel. A főtár maximális kiépitettsége 32 K szó, amely 8 db 4K szavas fieldből van felépitve. Egy field 32 db 128 szavas lapból, egy szó 12 bitből áll. A hardware adottságokból és az operációs rendszer hiányából az alábbi problémák következnek:

1.1.1 A field-váltás nem automatikus, ezt szoftverrel kell biztositani.

1.1.2 Egy memória lapról csak a kurrens és a zérus lap érhető el direkt cimzéssel, a többiek csak indirekten cimezhetők.

1.1.3 A gép csak egészen elemi aritmetikával rendelkezik.

1.1.4 Nincsenek nyomkövetési lehetőségek.

1.2 A hardver-szoftver adottságokból adódó problémák megoldásának vázlata:

Az eddigiekben felvetett problémák megoldására kerestünk egy olyan, a SLANG-nél magasabb szintü programozási nyelvet, amelyen a felmerülő feladatokat kényelmesen programozhatjuk és tesztelhetjük, valamint ugyanakkor ezeket a letesztelt programokat a TENV számitógépen is futtathatóvá tehetjük. A lehetőségek figyelembevételével a CHANGE nyelvet választottuk céljaink eléréséhez.

1.3 A CHANGE nyelv választásának okai:

- A CHANGE nyelv /jelölése: CH/ utasitáskészletéből könnyen kiválaszthatók az adatfeldolgozó rendszerhez szükséges utasitások. Az igy kiválasztott utasitásokból létrejött nyelv /a továbbiakban: CH-i nyelv/ viszonylag alacsony szintü, igy elég könnyen fordítható a TENV számára értelmezhető programmá.
- A CHANGE nyelv EXTEND kiterjesztő utasitásának segitségével a CH-i nyelvből mindig a pillanatnyi feladathoz leginkább alkalmazkodó célnyelvet hozhatja létre a felhasználó. Ezen a célnyelven irt program a CHANGE processzorral rendelkező gépen futtatható és tesztelhető.
- A CHANGE nyelv rendelkezik a megfelelő nyomkövetési lehetőségekkel.

1.4 A SLAGH közbenső nyelv bevezetése:

Mivel a SLANG nyelvi szint alacsony, a céleszközök müködéséhez pedig jól körülhatárolható funkciókra van szükség, ezeket SLANG nyelvü szubrutinok formájában megvalósitottuk /SLAGH futtató rendszer/. Ezeknek a rutinoknak a paraméterezett hivásából álló programokat egy közbenső nyelvi szinten, a SLAGH nyelven irott programoknak tekintjük. A SLAGH programok a SLANG fordítóval fordithatók, a generált tárgyprogramot a TENV számitógép a SLAGH futtató rendszer segitségével tudja végrehajtani. Az eddigiek szerint megoldandó feladatok a következők:

- A SLAGH nyelv definiálása
- A SLAGH futtató-vezérlő rendszer elkészitése
- a CHANGE nyelv megfelelő részhalmazainak kiválasztása
- A CHANGE-SLAGH forditási mechanizmus megvalósitása.

2. A SLAGH programozási nyelv

2.1 A SLAGH nyelv áttekintése:

2.1.1 A SLAGH utasitásai, általános alakja:

LABEL, OPC; PAR1; PAR2;... PARi;

LABEL: utasitás cimke, opcionális

OPC: utasitáskód

PARi: az utasitás i-edik paramétere

A paraméter lehet:

TOMB: tömbnév

VALT: változó név

INDEX:

62n1: field hivatkozás, n értéke adja a field sorszámát

PER : periféria név

CIM : cimke vagy relativ cimhivatkozás

$n_1n_2n_3$: oktális szám

Az utasitás hossza i+1 gépiszó

2.1.2 Adatstrukturák:

- változó vagy konstans
- egydimenziós tömb

2.1.3 File-ok:

A SLAGH nyelvben soros elérésü file-ok definiálhatók és kezelhetők. Minden file vagy csak input vagy csak output file lehet.

2.2 Főbb utasitás tipusok:

2.2.1 Értékadó utasitások

Pl.: - változó töltése az akkumulátorba

GET; VALT;

- tömbelem hozzáadása az akkumulátorhoz

ADDX; TOMB; INDEX;

- karakter letárolása szövegbe akkumulátorból
  PUTCH; TOMB; INDEX; VALT;

2.2.2 Vezérlésátadó utasitások

Pl.: - Feltétlen ugrás:
  JUMP; 62nl; CIM;
- Ugrás tömbelem előjele szerint
  IFX; TOMB; INDEX; $n_1n_2n_3$; CIM1;CIM2; CIM3;
- Szubrutin hivás
  CALL; 62nl; CIM;

2.2.3 Adatátviteli utasitások:

Ez az utasitáscsoport lehetőséget nyujt mind a bináris, mind a karakteres adatátvitelhez.

Pl.: - Szám beolvasása
  RDNUM; PER; hossz; VALT;
  hossz: oktális szám = a beolvasandó jegyek maximális számával
- Formátumos kiirás
  PRINT; PER; VALT; TOMB1; TOMB2; INDEX;
- File lezárás
  ENDFILE; PER;

2.2.4 Speciális utasitások

Pl.: - üres utasitás
  FOP;
- Gépi nyelvü szubrutin hivása
  SLANG; 62n2; CIM;

2.3 A SLAGH nyelvü programok forditása és végrehajtása

2.3.1 A forditás:

A 2.1.1-ben megadott SLAGH nyelvü utasitások szintaktikus definiciója megegyezik a SLANG nyelvü utasitások szintaktikájával. Ez lehetővé teszi, hogy a SLAGH nyelvü programokat a SLANG forditóval forditsuk.

2.3.2 A végrehajtás:

Minden SLAGH utasitás funkcióját megvalósitó SLANG utasitás sorozat egy-egy rutinként a Ø-ás fieldben van tárolva. A SLAGH utasitás végrehajtásakor a vezérlő-futtató

rendszer /amely szintén a Ø-ás fieldben van/ kiértékeli az utasitáskódot és az annak megfelelő rutinra adja a vezérlést. A rutin végrehajtása után ugrás történik a következő SLAGH utasitásra. Tehát a futtató rendszer lényegében mint egy interpreter müködik.

3. Alapvető és felhasználó által létrehozható nyelvi szintek

3.1 A CHANGE nyelv /CH/

A CHANGE nyelv utasitáskészletének részletes ismertetésére itt hely hiányában nem térünk ki, az az /1/ felhasználói kézikönyvben megtalálható. Egy, a nyelvnek a fordítás és alkalmazás szempontjából lényeges tulajdonságát kiemeljük. Ez a nyelvnek a kiterjeszthetősége.

3.1.1 Az utasitáskiterjesztő utasitás - EXTEND utasitás. Ezen utasitással uj utasitáskészletet állithatunk elő az alaputasitásokból, valamint a már korábban kiterjesztett utasitásokból. Ilyen módon lehetőség van arra, hogy processzoroknak[2] saját utasitáskészletet /könyvtárat/ definiáljunk, és lehetséges az is, hogy különböző processzorok azonos szintaktikáju utasitásaihoz más-más szemantikát rendeljünk. /Ezt erősen ki fogjuk használni a forditáskor/. A kiterjesztett utasitások végrehajtása lényegesen különböző attól függően, hogy a definiálás OPEN vagy CLOSE paraméterrel történik. OPEN paraméter esetén a kiterjesztett utasitás helyébe bemásolódik a test, hasonlóan a nyilt makróhiváshoz. Ha a definició CLOSE megadásával történik, végrehajtáskor a zárt makro hivásnak megfelelően a kiterjesztett utasitás változatlan marad. Az utasitások "kinyitása" az OPEN utasitással is megvalósitható.

---

[2] Processzoron itt azt a CHANGE processzort értjük, amelyik az adott programot az adott pillanatban végrehajtja.

3.2 A CH-i nyelv

Általánosan megfogalmazva: a CHANGE nyelv azon utasitásai, amelyek lehetővé teszik az adatfeldolgozási célfeladatok megoldását és ugyanakkor a SLAGH-ra történő fordításukkor minimális problémákat kell megoldani.
A CH-i nyelv utasitáscsoportjai:

- deklarációs utasitások
- értékadó utasitások
- vezérlésátadó utasitások
- adatátviteli utasitások
- NULL utasitás

3.2 A CH-s nyelv

A SLAGH nyelvnek vannak olyan utasitásai, amelyek hatását a CHANGE nyelvben szimulálni kell. /Pl.: a speciális display kezelő utasitások/. Ezen utasitásokat zárt kiterjesztésként definiáltuk a CHANGE nyelvben /jelölése: CH-s/ ugy, hogy a kivánt hatás szimulációja az utasitástestben történik.

3.4 A CH-is+ nyelv

A felhasználó a programozási munka jelentős megkönnyitésére nyilt kiterjesztésekkel egy, a céljainak leginkább megfelelő utasitáskészletet /nyelvet/ definiálhat. A kiterjesztések testei a CH-i, valamint a CH-s nyelvek utasitásaiból épülhetnek fel. Az ilyen módon definiált /a továbbiakban CH-is+/ nyelven irt program a CHANGE rendszer alatt futtatható, tesztelhető és a SLAGH nyelvre forditható a 4. pontban leirt módon. A kiterjesztésekkel elérhető szint szemléltetésére bemutatunk egy CH-is+ nyelven irt lekérdező rendszer rövid programrészletét.

```
    :
    A TORZSSZÁM BEOLVASASA
    $ A DOLGOZO MUNKAHELYE $
    $ SZEGED$ AKKOR 51
    $ VIDEK $ AKKOR 52
    $ ISMERETLEN $ AKKOR 53
    VEGE
51  $ KEREM TEGYE BE A T1/SZ KAZETTAT $
    :
```

4. A CHANGE-is+ SLAGH forditás

A programozás és forditás menetét az 1. ábra tartalmazza, részleteit a fejezet alpontjai fejtik ki.

4.1 A PASS Ø:

A felhasználó CH-is+-ban megirt és belőtt programja a forditás inputja. Ez a program tartalmazhat felhasználói kiterjesztéseket is. Ebben a menetben ezen felhasználói kiterjesztések helyébe a megfelelő testek bemásolása történik meg, melyet a már emlitett OPEN utasitás végrehajtásával érünk el. /A CH-s utasitások természetesen változatlanok maradnak, mivel azok zárt kiterjesztések/.

4.2 A PASS 1:

Ezt a menetet egy módositott utasitáskönyvtárral rendelkező processzor hajtja végre.

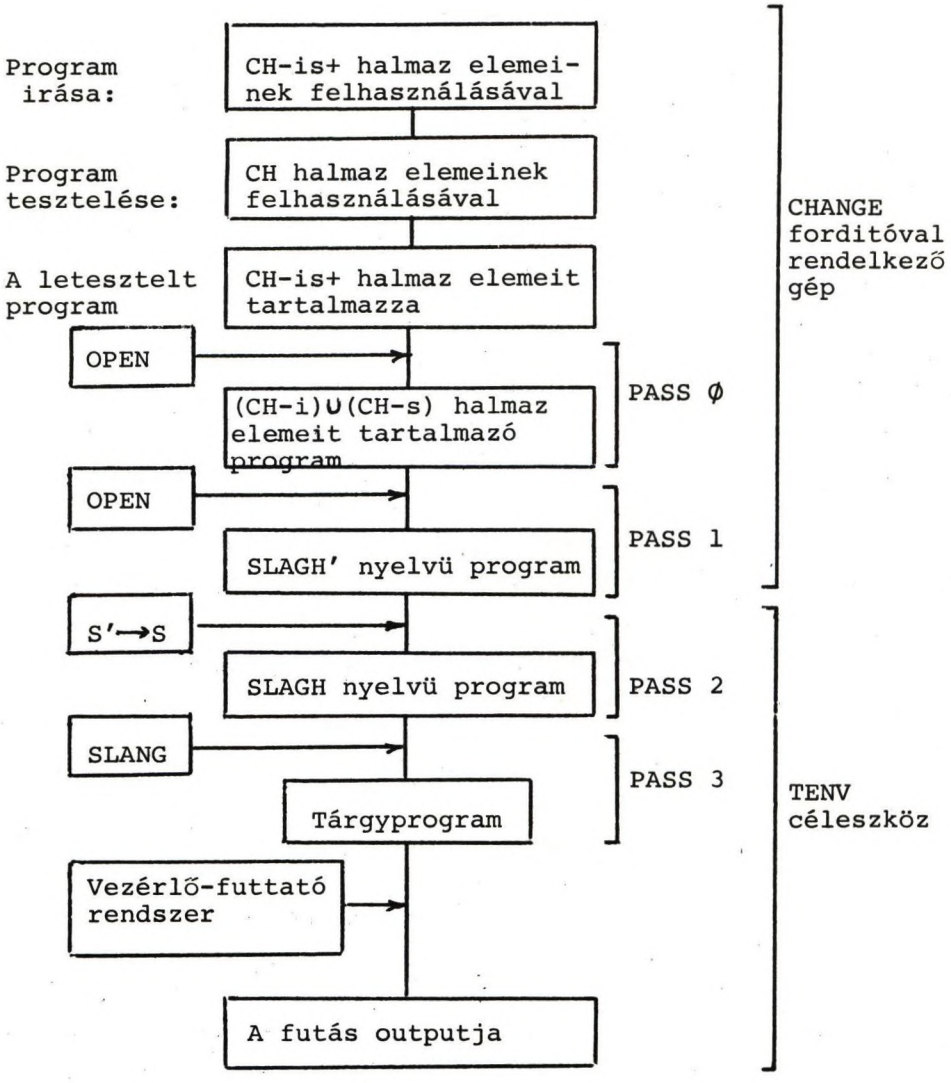
4.2.1 A végrehajtó processzor könyvtárában megadott utasitások:

A SLAGH nyelv teljes utasitáskészletének szintaktikus definiciója üres szemantikával.

A CH-i utasitáskészlet az eddigi szintaktikával, de a deklarációs utasitások kivételével uj szemantikával.

4.2.1.1 A CH-i utasitáskészlet uj szemantikája:

Egy-egy CH-i utasitáshoz rendelt szemantika egy SLAGH utasitásból, vagy utasitás sorozatból áll, amely az adott CHANGE utasitás funkcióját valósitja meg SLAGH


Program írása:
CH-is+ halmaz elemeinek felhasználásával
Program tesztelése:
CH halmaz elemeinek felhasználásával
A letesztelt program
CH-is+ halmaz elemeit tartalmazza
OPEN
(CH-i)∪(CH-s) halmaz elemeit tartalmazó program
PASS ∅
OPEN
SLAGH' nyelvü program
PASS 1
CHANGE fordítóval rendelkező gép
S'→S
SLAGH nyelvü program
PASS 2
SLANG
Tárgyprogram
PASS 3
TENV céleszköz
Vezérlő-futtató rendszer
A futás outputja

1. ábra

nyelven. A hozzárendelés nyilt EXTEND utasitás alkalmazásával történik.

```
Pl.: a CHANGE értékadás:        INT;V=INT;E
     a paraméter átadó rész:    W1==INT(1)
                                W2==INT(2)
     az utasitás test:          GET;W2;
     /SLAGH nyelven/            PUT; W1;
```

4.2.2 A PASS 1 végrehajtása:

A 4.2.1-ben ismertetett könyvtárral rendelkező processzor végrehajtja az OPEN utasitást. Ennek hatására a CH-i utasitások helyébe a megfelelő SLAGH utasitások másolódnak. /A programban szereplő CH-s utasitások nem változnak/. Az igy nyert utasitásokból felépülő programnyelv a SLAGH'.

4.3 A PASS 2:

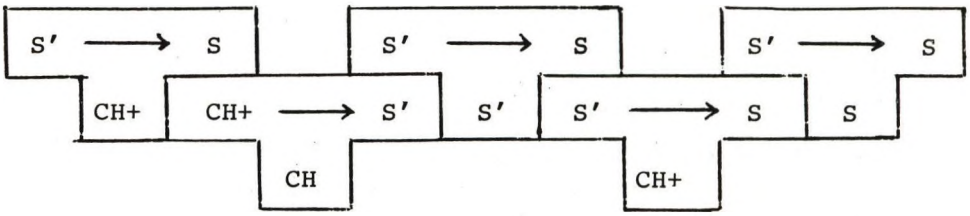
A SLAGH'-SLAGH fordító /S'→ S/ feladatai:

- a deklarációkban szereplő tömböknek és változóknak a megfelelő tárterületek biztositása
- a konstansok kigyüjtése és tárolása
- a fieldek kiosztása és az azokra való hivatkozások kitöltése
- cimkék konvertálása

4.3.1 Az S'→ S

Ezen feladatokat elvégző fordító megirása rutin feladatnak tekinthető. Megvalósitásának ismertetésével azért foglalkozunk, mert egy érdekes BOOTSTRAP megoldáshoz vezetett. A SLAGH'-ig való fordítás helyileg egyértelmüen a CHANGE fordítóval rendelkező géphez volt kötve. A PASS 2 fordítási menet végrehajtása az interpretálás miatt lassu, ezért felmerül, hogy ezt a menetet már jó lenne a TENV-en végezni. Az S'→ S megirása viszont SLANG nyelven hosszadalmas, fáradságos munkát igényelne. Igy a következő megoldás adódott: a fordító megirása és belövése CH-is+ nyelven történt, majd a

fordítóra alkalmazva a PASS 0 és PASS 1 meneteket előáll a SLAGH' nyelvü változat. Az igy kapott programot a CH-is+-ban irt fordítónak inputként odaadva nyerjük az S'⟶S fordítóprogramot SLAGH nyelven, amely már a TENV célgépen is üzemeltethető. /2. ábra/



Jelölések:
CH+: CH-is+ nyelv
S' : SLAGH' nyelv
S : SLAGH nyelv
CH : CHANGE nyelv

2. ábra

4.4 PASS 3:

A SLAGH nyelvü program leforditása SLANG fordítóval, amelynek eredménye a már futtatható tárgyprogram.

5. A kisérleti futások tapasztalatai

Jelen dolgozat irásának időpontjában a rendszer fejlesztésének utolsó fázisánál tartunk, a tesztelések és kisérleti futtatások időszakában. Igy a fejlesztés eredményének összegzése még nem aktuális, de az eddigi tapasztalatok alapján az alábbi megállapitásokat tehetjük:

- megfelelően kiválasztott CH-is+ célnyelven a programok irásának és belövésének ideje jelentősen lerövidült.
- A generált célprogram mérete és futási ideje nem tér el nagymértékben az optimálistól.
- Az S'⟶ S fordító müködése CH-is+ nyelven lassu, a SLAGH'-SLAGH forditást feltétlenül a céleszközön kell végezni.

A rendszer igazi főpróbája most következik: a VOLÁN 10. sz. Vállalat Munkaügyi Főosztályán elhelyezésre kerülő "Munkaügyi-nyilvántartási céleszköz" programcsomagjának elkészítése.

Abstract:

At the company VOLÁN 10. a microcomputer - the TENV - has been developed and applied as a dedicated tool for data collection and query. This specialized microcomputer can be programmed in the SLANG language that is near to the machine-code, which requires a tedious work. To solve this problem we looked after a higher level programming language, that helps the tasks to be conveniently programmed, tested as well as made runable on TENV e.g. cross-compiled too. To realize our aim we have chosen the CHANGE language. The tested programs written in the CHANGE language were translated into a middle level language SLAGH.
The semantics of commands of the SLAGH language is given by a calling sequence of rutins written in language SLANG.
The translation is relatively simple by the utilization of the extensibility feature of the CHANGE language.
The object of our paper is to show the different language-levels and the translation process.

Irodalomjegyzék:

1. Almási József: A CH-i és CH-s nyelvek specifikációja /kézirat, 1981./
2. Legendi Tamás, Vas Zoltán, Almási József, Karvaly Gellért: CHANGE /MTA SZTAKI CDC 3300 Felhasználói ismertetők. 1980./
3. Legendi Tamás: A CHANGE nyelv/multiprocesszor /MTA SZTAKI Tanulmányok 7/1973./

4. Legendi Tamás - Szajbély György: Dialógusprogramok készítése a CHANGE programozási nyelv kiterjesztésével /kézirat, 1980./
5. Tóth Károly: A SLAGH nyelv specifikációja /kézirat, 1981./

Almási József, Legendi Tamás: MTA Automataelméleti Kutató Csoport, 6720 Szeged, Somogyi u. 7.
Szajbély György, Szekeres Szilveszter, Tóth Károly: VOLÁN 10. sz. Vállalat, 6724 Szeged, Bakay N. u. 48.

Arató András–Burány Katalin–Sarkadi-Nagy István–Telbisz Ferenc

INTELLIGENS TERMINÁLOK A CEDRUS RENDSZERBEN

A KFKI-ban üzemelő R-40 számitógépre alapozott CEDRUS nevü interaktiv terminál rendszert továbbfejlesztettük. A továbbfejlesztett rendszerben TPA-1140 kisszámitógépek intelligens végberendezésként használhatók. A file átviteli szolgáltatás párbeszédes formában vehető igénybe az intelligens terminálokról. Az adatátvitel a "front-end" számitógép és a távoli kisszámitógépek között soros aszinkron vonalon történik keret /frame/ strukturált byte orientált transzparens protokoll használatával.

Kulcsszavak

adatátvitel, fileátvitel, intelligens terminál, protokoll, számitógép hálózat

1. Bevezetés

A KFKI-ban üzembeállitott ESzR nagygépre /EC-1040/ alapozva kifejlesztettük a CEDRUS hálózatot, amelyik a kiépités első lépcsőjében csak interaktiv módon használható képernyős terminálokat tartalmazott [1,2,3].Az interaktiv terminálokról igénybe vehető szolgáltatások a következők: szövegszerkesztés /beleértve a dokumentáció készitését is/, "job"-ok átadása a kötegelt feldolgozás számára, illetve a lefutott "job"-ok eredményeinek, listáinak lekérése.

A nagygép szolgáltatásainak igénybevétele a kisgép felhasználóknak is számos előnyt jelentett; a nagyobb volumenü és megbizhatóan üzemeltetett, archivált háttértárakat

keresztfordítókat, stb. Igy az eredetileg is nagygépre orientált felhasználók mellett megjelentek a kisgép felhasználók is, és hamarosan felmerült a közvetlen kapcsolat igénye is, mivel ez a külső adathordozóknál jóval egyszerübb, kényelmesebb. Ezért a CEDRUS rendszer kiépitésének második fázisában lehetőséget nyujtottunk ahhoz is, hogy a TPA-1140-es számitógépeket intelligens terminálként hozzá lehessen kapcsolni a CEDRUS rendszerhez. Az intelligens terminálok mint önálló kisgépek müködnek, de szükség esetén igénybe vehetik az R-40 szolgáltatásait két területen:

- File-ok vihetők át az R-40 és a kisgépek háttértárai között, illetve a kisgép felhasználható "batch" terminálként.
- A kisgép kompatibilis termináljai használhatók CEDRUS terminál üzemmódban is.

Hasonló rendeltetésü rendszert szinte minden nagyobb kutató intézetben kifejlesztettek, igy a Daresbury Magfizikai Laboratóriumban /Anglia/, a CERN-ben /Genf, Svájc/, a jülichi Magfizikai intézetben /NSzK/, stb.

2. File átvitel az ESzR gép és a kisszámitógépek között

A file átvitelt az ESzR gép és a kisszámitógépek között a "File Manager" rendszer valósitja meg. A rendszer használata a kisgépekből történhet segédprogramokkal, de történhet közvetlenül felhasználói programból is. A felhasználó parancsait párbeszédes formában adhatja meg a 1140-ban az FML segédprogramnak. Meghatározza az átvitel irányát, a file formátumát /ASCII vagy bináris/, a file-specifikációt a TPA-1140-ban és az R-40-ben.
Ezenkivül kötegelt terminál üzemmód is lehetséges.
A File Manager támogatja mind szekvenciális, mind particionált file-ok kezelését a nagygépben.

Lehetőség van az FLM segédprogram használatára és az interfész rutinok felhasználói programból való hivására is. Az interfész rutinok az elemi I/O funkciókat valósitják meg és a kommunikációs bufferek összeállitását végzik a nagygép számára. Az R-40 File Manager számára az alábbi parancsok adhatók ki:

MSTART - hatására az adott felhasználó szempontjából alaphelyzetbe hozza a File Manager táblázatokat

MOPEN - megnyit egy adott számlaszámon lévő, adott nevü file-t a kivánt módon

MCLOSE - lezár egy megnyitott file-t a megadott törlési opcióval

MGET - FM kiolvassa az előzőleg már bemenetre megnyitott file következő logikai rekordját

MPUT - FM az előzőleg már kimenetre megnyitott file-ba beir a felhasználói bufferből egy rekordot

MFIND - egy particionált bemenő file adott particiójának elejére pozicionál

MADD - egy particionált kimenő file-hoz hozzáad egy uj particiót

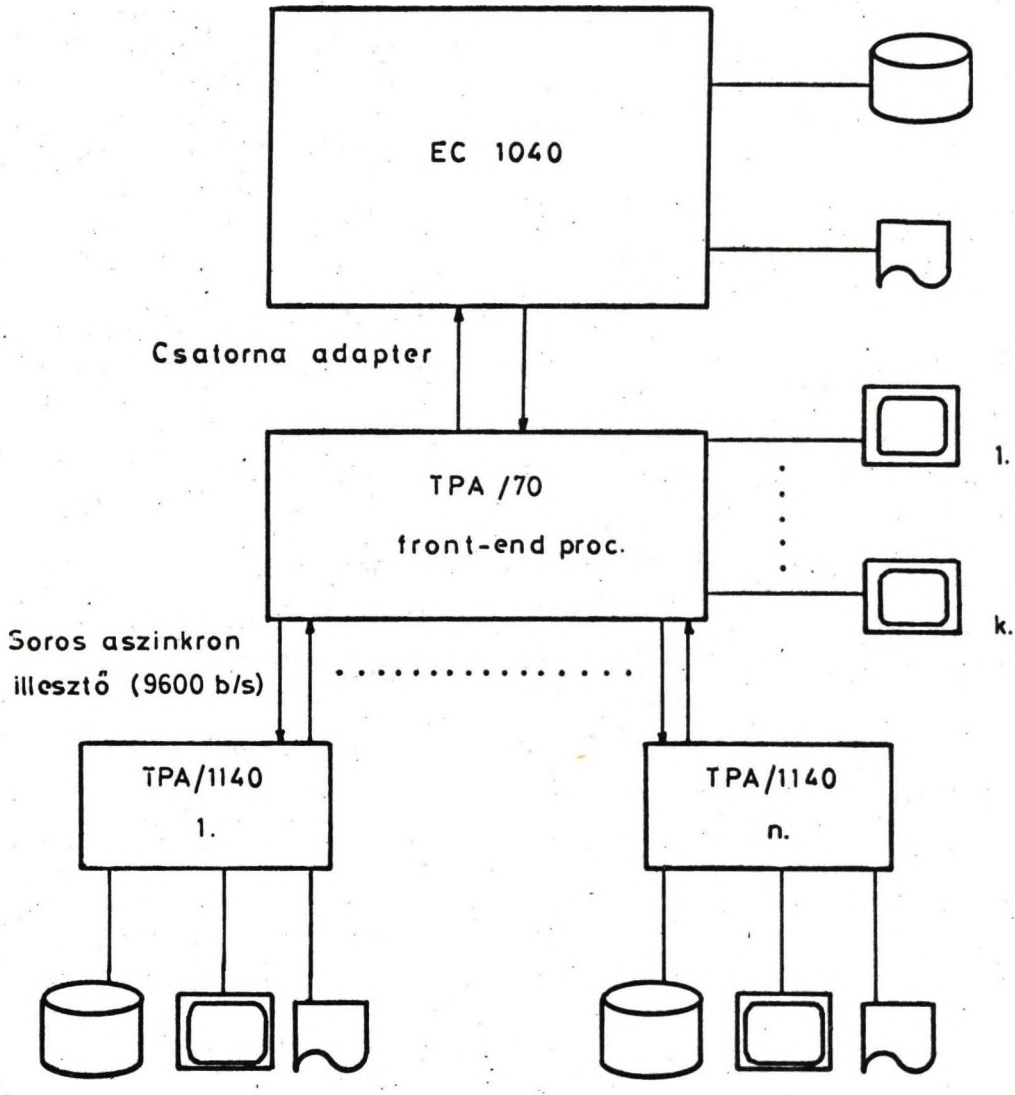
MDELETE - egy particionált kimenő file-ból töröl egy particiót

MREAD - kiolvas egy bemenetre megnyitott file-ból egy blokkot /512 byte/

MWRITE - egy kimenetre megnyitott file-ba beir egy blokkot

3. Az átviteli hardver paraméterei

Az adatátvitel soros aszinkron módon folyik a front-end processzorként müködő TPA-70 és a TPA-1140 között (1. ábra). A hardver teljes-duplex összeköttetést tesz lehetővé, az átviteli sebesség 9600 bit/sec. Az össze-


EC 1040
Csatorna adapter
TPA /70
front-end proc.
1.
k.
Soros aszinkron
illesztő (9600 b/s)
TPA/1140
1.
TPA/1140
n.

1. ábra

köttetést aszinkron GDN vonaladaptereken keresztül valósitottuk meg. A gépek közötti távolság max. 2 km. A TPA-70 és az R-40 között ESzR csatornaadapter müködik, amely kapcsolatot teremt az ESzR csatorna és a TPA-70 busza között.

4. Adatátviteli protokoll

Az adatátvitelhez saját protokollt dolgoztunk ki. Az eljárásban megtartottuk a BSC szabvány [4] alapvető tulajdonságait, de eltértünk a szabványtól. A protokoll karakter orientált, transzparens átvitelt biztosit. Az átvihető adatblokk maximális mérete 540 byte, ebből 28 byte vezérlő információ, 512 byte a hasznos adathossz. Ez megfelel a TPA-1140 diszk file blokkméretének.

Az adó először ENQ küldésével jelzi adási kérelmét. A vevőnek erre nyugtázással kell válaszolnia. Pozitiv nyugtázás esetén az adó elküldi az adatblokkot. A vevő, ha helyesen fogadta a blokkot, akkor pozitiv, ha BCC hibát észlel, akkor negativ nyugtát küld. Ha a vevő az ENQ fogadásakor még nem rendelkezik fogadó bufferrel, akkor 'küldj később' sorozattal jelzi ezt. Ha majd lesz buffere, akkor 'felébreszti' az adót.

Negativ nyugtázáskor az adó maximum háromszor ismétel. Ha adott időn belül nem jön válasz a szekvencia valamely pontját, akkor szintén ismétlésre kerül sor. Ez lehet ENQ illetve adatblokk ismétlés. A protokoll szerint a TPA-1140 'master', a TPA-70 pedig 'slave', verseny-helyzet esetén tehát a 1140 győz, ő küldhet.

beszurt DLE karakter

D L E	S T X	adat	D L E	D L E	D	E	X	B C 1	B C 2

2. ábra

A vevő bufferébe csak az adatok kerülnek be, a kezdő- és záró sorozat, a transzparenciát biztositó beszurt DLE és a BCC karakterek nem (2. ábra).
A BBC számitás az adó oldalon az adatokra és az ETX karakterre történik. A vevőnél ugyanezekre és a vett BCC karakterekre, az eredménynek nullát kell adnia. A BCC számitásnál a CCITT polinomot használjuk.

5. Implementációs kérdések

5.1 File Manager program a TPA-1140-ben

A file átvitel a TPA-1140-ben az FML segédprogrammal a BLDRV kommunikációs driveren keresztül történik DOS-RV operációs rendszer alatt. A fizikai protokollt a driverben valósitottuk meg PL-11 nyelven. Az átviteli kérelmeket a drivernek QIO direktivák kiadásával adjuk meg. A drivertől kérhető a vonal inicializálása (IO.INL funkció), logikai blokk olvasása (IO.RLB), logikai blokk irása (IO.WLB), valamint vonal állapot lekérdezése (IO.GST). A kérelem sikerességéről az FLM az I/O státusz-szóból értesül. Sikertelen átvitel esetén a státusz-szó tartalmazza a hibakódot. A driver a DOS-RV rendszerben betölthető, felhasználó által irott driverként van generálva.

Az FLM segédprogram feladatai:

- felhasználói parancsok értelmezése
- file kezelés megvalósitása
- R40 számára szükséges buffer összeállitása
- átviteli kérelmek kiadása a BLDRV-nek
- hibakezelés, szinkronizáció biztositása

5.2. File Manager program a TPA-70-ben

A TPA-70 feladata a kapcsolatteremtés a TPA-1140-ek és az R40 között. Ehhez a 1140-ek illetve az R40 felől

érkező adatok fogadását, továbbitását kellett megvalósitani. Az adatcsere a 1140-ek és a TPA-70 között a már leirt kommunikációs protokoll TPA-70 MINOR-COM operációs rendszer alatt müködő változatának segitségével történik. Az R40-nek a File Manager a LINKTASK-on keresztül tart kapcsolatot két cimpáron. Az egyiken a rövid (GET/PUT, READ/WRITE, KEY operációk), a másikon a hosszu végrehajtási idejü (OPEN/CLOSE) parancsok mennek.
A TPA-70 feladata, hogy a 1140-ek felől érkező utasitásokat szétválassza, és a megfelelő cimre továbbitsa. Az R40 felől érkező választ annak a 1140-nek kell adnia, amelyiktől a kérés érkezett.

5.3 File Manager program az R40-ben

A File Manager program az R40-ben az OS operációs rendszer alatt müködő CEDRUS rendszer részeként került implementálásra. Végrehajtható parancsként az elemi I/O funkciónak tekinthető OPEN/CLOSE, GET/PUT illetve READ/WRITE utasitásokat hajtja végre, a front-end processzor felőli beérkezés sorrendjében. Egyidejüleg több file átvitelt képes folytatni, de a helyes szekvencia betartásáról (irás/olvasás csak megnyitás után, irás csak irásra megnyitott file-be történhet, stb.) a kisszámitógépben futó felhasználói illetve segédprogramoknak kell gondoskodniuk. Ezt az R40-ben implementált File Maganer hibajelzésekkel támogatja.

6. CEDRUS terminál távoli gépről

Intelligens terminálokat a CEDRUS rendszerhez a TCP /Terminal Communication Program/ segitségével is hozzá lehet kapcsolni. A program két változatban készült el: kisebb konfigurációk számára a FOBOS, nagyobb konfigurációk számára a DOS-RV operációs rendszer alá.

Mindkettő a TPA-1140 vagy a vele kompatibilis MSzR gépeken fut.

A TCP program lehetővé teszi a TPA-1140 display-e számára, hogy mint CEDRUS terminál müködjön. A display kompatibilitásától függően az ernyőszerkesztés lehetőségei is kihasználhatók.

A TCP program további előnye, hogy segitségével az intelligens terminál háttértáráról file-ok küldhetők a CEDRUS munkafile-jába, ahonnan SAVE parancs segitségével állandó file hozható létre az ESzR gép diszkjein. A CEDRUS LIST utasitásával pedig a munkafile-ból az MSzR gép háttértárolójára vihetünk szöveges file-okat. Az átvitelt kivánság szerint a display-n is követhetjük.

A Terminal Communication Programot interaktiv üzemmódban kezelhetjük, egykarakteres utasitásokkal vezérelhető és rendelkezik saját HELP funkcióval is.

A program hardver igénye minimális. Bármilyen aszinkron illesztő, vagy multiplexer felhasználható az összeköttetésre. Hátránya az FLM-mel szemben, hogy az átvitel csak kisebb sebességen valósitható meg.

7. Összefoglalás

Az interaktiv terminálokat kiszolgáló CEDRUS rendszert tovább fejlesztettük ugy, hogy TPA-1140-es számitógépek is hozzákapcsolhatók a rendszerhez intelligens terminálként. Ily módon, tekintettel a TPA-1140 MSzR kompatibilitására, egy rendszeren belül közvetlen adatkapcsolat létesithető ESzR gép és MSzR gépek között.

Abstract

The interactive terminal system-called CEDRUS-working in the Central Research Institute for Physics and based on the EC-1040 computer was further developed. In this enhancement TPA-1140 small computers can be used as intelligent terminals.

A file transfer service between the EC-1040 computer and the small computers is available at the intelligent terminals. This service can be used in a dialogue manner. Data transfer between the front-end processor and the small computers is on serial asynchronous lines by using a frame structured byte oriented transparent protocol.

Alternatively, the remote small computer can be used as an interactive terminal in the CEDRUS system with the help of the TCP /Terminal Control Program/ running in the remote small computer.

Irodalomjegyzék

[1] A.Arató, I.Sarkadi Nagy, F.Telbisz: A Local Network for the Support of Software Development COMNET'77, 1977.

[2] Arató A., Sarkadi Nagy I., Telbisz F.: Feladatmegosztás ESzR gép és front-end processzor között a CEDRUS terminálhálózatban. Programozási Rendszerek'78.

[3] Arató A., Sarkadi Nagy I., Telbisz F.: A CEDRUS interaktiv terminálrendszer mérése és szimulálása, Neumann Kongresszus Szeged, 1979.

[4] General Information-Binary Synchronous Communications, Form GA27-3004-1, IBM

Szerzők neve és cime:

Arató András, Burány Katalin, Sarkadi Nagy István,
Telbisz Ferenc
MTA Központi Fizikai Kutató Intézet
Számitástechnikai Főosztály
1525. Budapest, Konkoly Thege u. 29-33. Pf. 49.

Aszalós János

AIR: ANSWER INFORMÁCIÓS RENDSZER

Az AIR az ANSWER operációs rendszer egyik eleme. Célja, hogy a fejlesztés során felhasznált vagy előállitott információkat összegyüjtse, tárolja, karbantartsa, a megfelelő időben a kivánt személyekhez automatikusan eljuttassa. A cikk vázolja a feladatok végrehajtásához szükséges formalizmust és azokat a szoftver komponenseket, melyek a rendszer müködését biztositják. A rendszer központi összetevői az önálló tudással és intelligenciával biró, önállóan müködő robotok, melyek az adatbázis integritását és a programozás módszertanában előirt szabályok betartását ellenőrzik.

Kulcsszavak: információs rendszer, információs háló, módszertan, formalizmus.

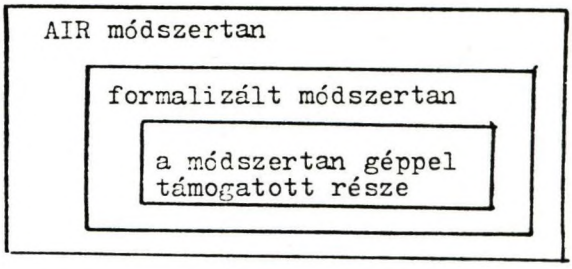
Az ANSWER operációs rendszer tervezésére és legfontosabb részrendszereinek megvalósitására a KSH-OSZI megrendelésére, az SZKCP CF-31 célfeladat keretén belül került sor. /Az ANSWER - szemben a legtöbb ismert operációs rendszerrel - nem a programok futtatására, hanem elsősorban a kifejlesztésére szolgáló eszközökkel van felszerelve. Lásd. Na 78 -ban/.

Egyik lényeges részrendszer az információs rendszer /AIR/. Ez a tanulmány áttekintést ad az AIR céljáról, alapelveiről, a formális specifikáció módjáról és legfontosabb komponenseiről.

Az AIR nem teljesen gépesitett rendszer. Lényege nem is a gépi programok halmaza, hanem az a programozási módszer és fegyelem, melyet /gépi eszközökkel, előirásokkal, javaslatokkal/ támogat, illetve kényszerit. A módszertannak csak egy része formalizált, de mindenképpen valamilyen előirás-

és definicióhalmazon alapul.

Az AIR alapszerkezetét az alábbi diagram mutatja:



A tanulmányban csak a legbelső blokkról, az AIR programrendszeréről adunk áttekintést, s az "AIR" kifejezést is ebben a szükitett értelemben használjuk.

1. Az AIR célja

Az AIR célja, hogy
- a szoftver előállitás bürokratikus terheit csökkentse;
- gépi eszközöket biztositson különböző tipusú információk automatikus, félig automatikus vagy a felhasználó által irányitott rögzitésére;
- az összegyüjtött információkat különböző szintü és szempontú részletezéssel, a rendeltetési helyükre szétossza;
- bizonyos előjelzéseket, figyelmeztető jelzéseket adjon az adatbázis állapotától függően;
- segitséget nyújtson a vezetői döntések előkészitésében, tárolásában és a következmények nyilvántartásában;
- a vezetők, programozók egymásközti kommunikációját segitse és dokumentálja;
- a programdokumentációk létrehozását, javitását és kultúrált kinyomtatását gépi eszközökkel támogassa;
- bizonyos alapvető módszertani előirások betartását ellenőrizze.

2. Alapelvek

Az AIR kifejlesztésénél az alábbi elvi döntésekre támaszkodtunk.

1/ A vezetésnek és a szoftver előállitásnak több szintje van /gazdasági vezetés, szakmai vezetés, stb./. Az AIR-nek több szintü, s az egyes szinteken is érvényesülő több szempontú felhasználói igényeknek kell eleget tennie.

2/ Az AIR-t bizonyos fokú intelligenciával kell felruházni. Ennek érdekében a külvilág bizonyos objektumainak modelljeit kell beépiteni a rendszerbe, s a külvilág rátartozó eseményeit, tényeit is tárolnia kell. A kidolgozott modellek között a "manager", "vezető programozó", "programozó", "könyvtáros", "szervezet", "dokumentum", a programozás módszertanilag kötött fázisai stb. szerepelnek. A rendszer intelligenciája ezen modellek által irányitott következtetési eljárásokban nyilvánul meg.

3/ Az AIR-nek a módszertani előirások, és a fegyelmezett programozás biztositására megfelelő kényszeritő erővel és vétó-joggal kell rendelkeznie. Ezek a funkciók részben az előbb emlitett modellek, részben a vezetők közvetlen előirásai szerint működnek.

4/ Az AIR-t az intelligenciája segitségével fel kell ruházni olyan képességekkel, hogy bizonyos területeken a felhasználó tanácsadója, tanitója vagy irányitója lehessen.

5/ Az AIR beépitett tudása nem lehet teljesen merev. Uj eljárások, új modellek, új adatok beépitésével lehetőséget kell biztositani ezen tudás módositására és kiegészitésére.

3. Az AIR formális leirása

Ezen pont keretén belül az AIR legfelsőbb absztrakciós szintjéről adunk /röviditett/ leirást.

Az AIR-t egy négyesnek foghatjuk fel:

AIR = $\langle U, I, \Sigma, \Omega \rangle$, ahol

U az AIR univerzuma,

I az univerzum elemein müködő, általában mellékhatásokkal rendelkező információs eljárások /röviden információk/ halmaza ,

Σ az univerzum lehetséges állapotainak halmaza, /állapottere/

Ω az állapothalmazra vonatkozó előirások, megkötések halmaza.

A fenti komponenseket részletesebben is ismertetjük.

3.1 Az AIR univerzuma

Az univerzum formálisan egy négyes:

$U = \langle O, A, R, T \rangle$, ahol

$O = o_1, o_2, \ldots, o_m$ objektumok véges halmaza,

$A = a_1, a_2, \ldots, a_n$ tulajdonságok /attributumok/ véges halmaza,

$R = r_1, r_2, \ldots, r_o$ relációk véges halmaza,

$T = S \cup \mathbb{N}$, ahol $S = s_1, s_2, s_3, s_4$ és $\mathbb{N}$ a természetes számok halmaza.

Az univerzum interpretációján egy leképezést értünk a /szoftver gyártásra korlátozott/ való világ elemei és a fenti formális halmazok elemei között. Az interpretáció végrehajtása céljából igen sok programtervet, dokumentumot, szerződést, stb. tanulmányoztunk át a legfontosabbnak tünő elemek kiválogatására és rendszerezésére. Ezeket standard objektumoknak, standard tulajdonságoknak stb. nevezzük, lehetőséget adva az interpretáció felhasználó részéről történő kiterjesztésére.

A standard objektumokat öt kategóriába /tipusba/ osztottuk: személyek, projektek, események, eszközök, elméletek.

Az egyes kategóriákhoz egy-egy logikai file tartozik; személyi file, projekt-file stb. Az egyes kategóriák fastruktúrát mutatnak. A személyek például három fő ágra /altipusra/ oszlanak: élő személyek, jogi személyek és robotok. A fastruktúra leveleinek egyedi személyek, vállalatok felelnek meg.

A tulajdonságok is fastruktúrába szervezhetők, a következő főágakkal:

mennyiség, minőség, modalitás.

A tulajdonságokat relációk segitségével kapcsolhatjuk az objektumokhoz.

A relációk halmazt alkotnak. A relációk objektumokból és tulajdonságokból álló n-esek. Néhány példa különböző relációfajtákra /a paraméterek elhagyásával/:

fia, apja, leszármazottja, őse, tulajdona, jogosult vmre, hozzárendelt vmhez, beosztása, stb.

Ahogyan az objektumok esetében, úgy a tulajdonságokkal és a relációkkal kapcsolatban is beszélünk standard és felhasználói jellegről.

S a pontatlanul definiált időmeghatározások halmaza:

s_1: "a múltban", s_2: "a jövőben", s_3: "a közelmúltban", s_4: "a közeljövőben".

$\mathbb{N}$ az univerzum órája. Az 1-hez célszerüen az az időpont társitható, amikor a rendszert az adott intézménynél használatba vették. Az időegység megválasztása szabad döntés kérdése.

3.2 Az információk

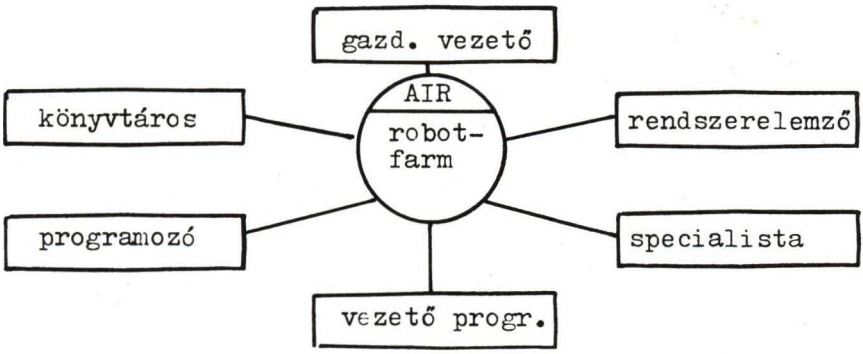
Az AIR-ben az információk speciális reláció-hetesek:
$i_j = <t, f, c, rl, felt, sz, k>$, ahol
$i_j \in I$,
t: az információ tipusa,
f: a feladó,
c: a cimzett,
rl: az átadandó relációk listája,
felt: logikai kifejezés,
sz: szempont, mely az információk szürésére szolgál,
k: kulcsszó-lista.

Az információ, tipusát tekintve, dokumentum, üzenet, levél, kérdés, válasz, ellenőrzés, vétó, kényszeritás, figyelmeztetés vagy hibaüzenet.

A feladó és a cimzett az ún. információs háló /i-háló/ csomópontjait alkotják. A háló élei az információs csatornáknak felelnek meg. Megjegyezzük, hogy a feladó és a cimzett nemcsak élő személy lehet, hanem jogi személy vagy robot is, sőt több csomópontból álló együttes is.

Az információ csakis akkor aktiválódik, ha a feltétel igaz. /A feltétel lehet eleve igaz - True - is./ Ebben az esetben a rendszer az átadandó relációkat az sz szempont szerint a cimzett "testére szabja". Ennek módjára most nem térünk ki. Az sz és k lehet üres is.

Minden programozási feladatnak /projektnek/, mely a rendszerbe bekerül, egy ún. "információs háló" felel meg, mely teljes kiépitettségében a következő alakú: /l. [Ba72] /



A hálót a felhasználó /pl. a vezető programozó/ állitja elő megfelelő utasitások segitségével. A háló konkrét esetben lehet hiányos is, illetve bizonyos tipusú csomópontokból /pl. programozó/ több is lehet. A csomópontokhoz a felhasználó konkrét személyneveket rendel.

Látható, hogy a szükitett értelemben vett AIR-ben a csomópontok között minden kapcsolat a gépen keresztül történik. A robotok az információ átalakitó és összegyüjtő szerepét játszák. /Részletesebben 1. a 4.3 pontban./

3.3 Az állapottér és a megkötések

Minden t /$t \in \mathbb{N}$ / időegységben a rendszer egy σ_t állapotban van:

$\sigma_t = \langle D_t, I_t, P_t \rangle$, ahol

$\sigma_t \in \Sigma$

D_t az AIR adatbázisának állapota,

I_t a t pillanatban aktiv eljárások halmaza,

P_t a t időpillanatban az AIR-et használók és a robotok halmaza.

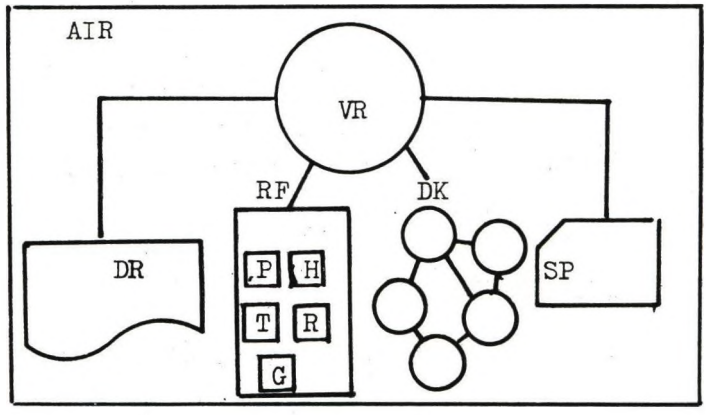
A megkötések biztositják, hogy

- a file-ok minden időpillanatban "jól formáltak" legyenek;

- az egyes file-okat csak az arra jogosultak módosithassák, illetve olvashassák;
- a párhuzamos hozzáférési igények kielégitése se holtpont-állapotra, se a jólformáltsági szabályok megsértésére ne vezessen.

4. Az AIR architektúrája

Az AIR négy részből áll. A részrendszerek a következők:
- vezérlő rendszer /VR/
- dokumentációs rendszer /DR/
- robot-farm /RF/
- döntéskezelés /DK/
- statisztikai programcsomag /SP/.



Az alábbiakban röviden bemutatjuk az egyes alrendszereket.

4.1 A vezérlő rendszer /VR/

A VR feladata a parancselemzés, az egyes processzek időbeli ütemezése, a jogosultsági megkötések ellenőrzése, és bizonyos default-értékek beállitása. /Hozzá tartozik a felhasználóval történő társalgás kultúrált lebonyolitása is, de ennek a részrendszernek a kivitelezését el kellett halasztanunk./

4.2 A dokumentációs rendszer /DR/

Az AIR egyik legfontosabb feladata a programdokumentálás gépesitése. A DR legfontosabb tulajdonságai:
- bizonyos tipusú dokumentációk vázát eleve tartalmazza;
- az egyes dokumentációk fastruktúrájúak /könyv, kötet, fejezet, bekezdés, stb./;
- a dokumentációk elemeihez természetes fogalomkészlet segitségével lehet hozzáférni;
- bizonyos szövegrészeket "minősiteni" lehet /pl. gépkezelő, gazdasági vezető stb. érdeklődésére számot tartó fejezet, bekezdés/, s később a minősitések alapján lehet újabb dokumentumokat előállitani;
- cimkézési lehetőség; szövegmakrók; mintaillesztés;
- újrainditási lehetőség.[+1]

4.3 A robot-farm/RF/

Az AIR terveiben öt robot szerepel:
a Postamester /P/, a Hivatalnok /H/, a Tanácsadó /T/, a Rendőrfelügyelő /R/ és a Gondolkodó /G/.

Mostanáig csak a Gondolkodó készült el az ELTE Numerikus Tanszékének munkatársai segitségével, a tervezettnél egyenlőre kisebb intelligenciaszinten.

A Postamester az információs hálózat csomópontjai, azaz - gyakorlatilag - ugyanazon projektben dolgozó munkatársak között teremt ellenőrizhető és jól dokumentált kapcsolatot, a "folyósói döntések" kiküszöbölésére. Különböző tipusú leveleket, üzeneteket kezel, pl. egyszerü, poste-restante, válaszlevél, körlevél stb.

[+1] A DR CDL2 nyelven készült el, kb. 5o ooo forrássor, 35o Kbyte terjedelemben IBM 37o OS-VS-re.

A Hivatalnok a sok helyen alkalmazott szoftver-titkárnő, diszpécser, vagy könyvtáros egyesitett szerepét tölti be. Feladatához tartozik pl. a határidők figyelemmel kisérése, különböző felhivások, üzenetek postázása a rendelkezésére álló szöveg-makrók segitségével, az erőforrások /gép, lyukasztókapacitás, gépidőbeosztás stb./ nyilvántartása, az események naplózása az event-file-ban.

A Tanácsadó a javasolt módszertanban és az ANSWER használatában tájékozott. Ez kezeli a HELP rendszert is. Ismeri a szakirodalmat témák és szerzők szerinti csoportositásban, tehát bibliográfiai adatokat is kezel. Ez az első "emberi" lény, mellyel a járatlan programozó találkozik, amikor az AIR-t használni kezdi.

A Rendőrfelügyelő a módszertani megkötések betartásáért felelős. A megkötések vagy általános érvényüek /pl. az inspekció lezárása előtt senki sem kódolhat/ vagy egyediek /pl. Kovács, ha belépsz a rendszerbe, azonnal cseréld ki az x modul nevét y-ra/.

A Gondolkodónak eleve beépitett tudása nincs. A felhasználótól függ, hogy milyen tudást közöl vele. A Gondolkodó tulajdonképpen az MPROLOG rendszer, melyet az AIR-ben is használunk; intelligenciája tehát az MPROLOG intelligenciájával ekvivalens. Jelenleg a /szabványositott/ CDL2 modulközi kapcsolatok vizsgálatára képes.

4.4 A döntéskezelés

A döntéseket összefüggő, ún. döntési hálók csomópontjaiként kezeljük. A döntés egy reláció, mely a problémát, a választható alternativákat, a kiválasztott alternativát, indoklást stb. tartalmazza.

A háló élei az egyes döntések közötti kapcsolat jellegét fejezik ki: pl. "következmény", "előfeltétel" stb.

A háló kezeléséhez különböző eljárások tartoznak, melyeket részben a felhasználó indit, részben automatikusan indulnak valamilyen felhasználói eljárás nyomán.

4.5 A statisztikai programcsomag

Ez a programcsomag PERT hálókat és kantdiagramokat kezelő-előállitó programokat, továbbá bizonyos hatékonyság-elemző és szimulációs programokat fog tartalmazni a programfejlesztés irányitásához szükséges határidő táblázatok, kiértékelések, stb. elkészitésére.

Befejezés

Az AIR koncepciójának kialakitását Langefors nevéhez füződő informatikai iskolában kialakult alapelemekre támaszkodva végeztük /l. pl. [Bu71] /. Figyelembe vettünk számos más magyar és külföldi rendszert is. /l. [IN74], [Me73] /. A rendszer részletesebb dokumentációja az [As77], [As78], [As81] és az [ANSWER77] kötetekben található.

Abstract

The AIR is an element of the ANSWER operating system which is designed to support the development of large programs. The purpose of the AIR is to collect, store and manage the information which are used or produced during the development process, and to distribute them automaticly in the proper time, to the proper persons. The article presents the outlines of the necessary formalism and the software-components realizing the main system functions. There are some basic components, called robots, with some built-in knowledge and reasoning power, which ensure the integrity of the data base and enforce the rules of the intended programming methodology.

Irodalomjegyzék

ANSWER77 Az ANSWER rendszer általános leirása.
D 2, SZÁMKI, Budapest. 1977.

As77 Aszalós J.: ANSWER információs rendszerének leirása.
D 6. SZÁMKI, Budapest. 1977.

As78 Aszalós J.: ANSWER információs rendszerének dokumentáció-kezelő programcsomagja.
D 18. SZÁMKI, Budapest, 1978.

As81 Aszalós J.: ANSWER dokumentációkezelő programjának felhasználói kézikönyve.
SZÁMKI, Budapest. 1981.

Ba72 Baker, F.T.: Chief Programmer Team: Management of Production Programming.
IBM Sys. Journal, Vol. 11, No. 1, /1972/
pp. 56-73.

Bu71 Bubenko, J., Langefors, B. and Solvberg, A.: Computer Aided Information Systems Analysis and Design. Studentliteratur Lund, /1971/.

In74 Management Information Systems. Infotech State of the Art Reports. 1974.

Me73 Metzger, P.W.: Managing a Programming Project. Englewood Cliffs, 1973. Prentice Hall.

Na78 Programozási Rendszerek '78 Konferencia Szeged

Szerző neve és cime:

Aszalós János
Számitógépalkalmazási Kutató Intézet
1356 Budapest, Pf. 227.

Asztalos Domonkos–Koltai Tamás–Krekó Béla

AMETIST: EGY METAINFORMÁCIÓS RENDSZER

A relációs adatszemlélet, a szemantikus hálók és az algebrai programspecifikálás eredményein alapuló olyan formalizmus kerül ismertetésre, amely egy információs rendszer bármely elemére alkalmazható. A tervezett rendszerben automatikus tételbizonyitó eljárások támogatják a modellezői, tervezői és programozói tevékenységet.

Kulcsszavak: automatikus programozás, fogalmi séma, konceptualizálás, mesterséges intelligencia, metainformációs rendszer.

1. Bevezetés

Egy-egy számitóközpont hatókörét, alkalmazói környezetét vizsgálva két szélsőséges tipus fedezhető fel. Az első: jól szervezett, egyértelmüen definiált rendszerek vezérlését, irányitását, müködtetését igénylő - egyik képviselőjéről talán "vállalati"-nak nevezhető - környezet; szemben a sokféle, egymással tartalmilag nem összefüggő területen, szervezettségében alacsonyabb, módszereiben viszonyt kifinomultabb szinten dolgozó - "egyetemi"-nek mondható - környezettel.

Ezek legfőbb különbségét abban látjuk, hogy mig az egyikben a modellezendő világdarabokat emberi céltudatosság szervezi rendszerré, elsőrendü szempontként érvényesitve annak müködtethetőségét, irányithatóságát, stb., addig a másiknál a valóság modellezése ugy folyik, hogy a modellezés érdekében számottevő beavatkozásra nincs lehetőség.

E kétféleség jól kimutatható azokban az elvárásokban, módszerekben, szolgáltatásokban, melyek napjaink software-iparának termékeit zömükben jellemzik.

Bár a tisztának mondható tipusok adják az alkalmazók tömegét, egyre nő a mindkettő vonásait elegyitő alkalmazási környezetek száma és fontossága is. Ezeknél ugyanugy megtalálhatók az egyedi elképzelések mint egy kutatóintézetben, de az egyes felhasználók itt már nem függetlenek egymástól - például ugyanazokat az adatokat használják. Találunk itt is müködtetendő, irányitandó világdarabokat, mint egy vállalatnál, de nem találjuk meg az ahhoz szervezett információs rendszert, találunk viszont - többnyire egymással versengve vagy ellentmondásban - részterületekről szóló elképzeléseket, önálló kis modelleket, szorosabb-lazább kapcsolatban müködő információs rendszereket, melyek "minél közösebb nevezőre" hozása egyetlen alkalmazónak sem igazi célja, de az egész környezetnek mégis egyik legfontosabb érdeke.

Részben a számitástechnika eszközeinek polarizáltsága, részben a problémák sulyához mérten korlátozott teljesitőképessége miatt, az ilyen környezetekben jónéhány informatikai igény kielégitetlen marad.

Ezek közül csak egyet emelünk ki: hogyan lehet összeegyeztetni, egyesiteni különböző célu és szemléletü modelleket /inf.rendsz./ egyetlen konzekvens modellé /információs rendszerré/ és mit jelent ez a már létező adatok értelmezésének szempontjából?

E kérdések megválaszolásánál hangsulyt kapnak a legkülönbözőbb területekről ismerős olyan módszerek, elvek, mint a pontos fogalmi definiciók [1], a nem-procedurális feladatleiró módszerek [2], az adat és jelentése közötti különbségtétel [3], stb. Ezeknek a jegyében készülnek a "tudás-bázis rendszer"-eknek /knowledge-base system/[4] vagy "szakértő rendszer"-eknek /expert system/ [5] nevezett sofware-termékek is, mely elnevezések kezdenek ugyanugy kategóriákká válni, ahogy például az "adatbázis kezelő rendszer" elnevezés azzá vált.

A fenti rendszerek létesitésekor figyelembe vett elveket is hasznositva alakitottuk ki egy metainformációs rendszer koncepcióját, amelynek realizálása több éves kutató, fejlesztő

munkát igényel. A tanulmány a koncepció lényegét és egy azon belül alkalmazott formalizmus fontosabb tulajdonságait kívánja ismertetni a tervezet szintjén.

Mindezen törekvések mélyén a megoldandó alapprobléma azoknak az elemeknek a formalizálása /géppel kezelhetősége, adatositása/, melyek a mai elfogadott gyakorlat szerint "csupán" dokumentációkban, konvenciókban vagy közmegegyezésekben vannak jelen, jó esetben élőnyelven megfogalmazva, de gyakran még abban sem. Vagyis olyan adatokat, információkat kell találni vagy konstruálni, melyek adatokról, konvenciókról, közmegegyezésekről szólnak és ugy kell tudni ezeket kezelni, hogy ezzel elősegitsük egy alkalmazói környezet informatikai tevékenységét. Ez indokolja a metainformációs rendszer elnevezést.

2. Az AMETIST alapjai

Az AMETIST /A Metainformation System/ az Országos Tervhivatal - mely tipikus képviselője az előbbiekben vázolt "kevert" alkalmazási környezeteknek - tervezett egységes számitástechnikai rendszerének központi magja, s igy általános célkitüzései mellett megfogalmazásában, hangsulyaiban tükröződnek a közgazdasági szemléletmód, az országos méretek és a tervezési tevékenység sajátosságai is.
Legfontosabb vonásait jól mutatják azok a feladatkörök, melyeket ellát:

- Adatszótár funkciók.
  Tárolt és elképzelt adatok, adatkapcsolatok rögzitése, kellő pontosságu dokumentálása. Az adatok elérhetősége, feldolgozásokkal, rendszerekkel való kapcsolatuk. Mindez könnyen lekérdezhető, csoportositható, változtatható formában.
- Modellezés támogatása
  a rendszertervezés, a konceptualizálás, a matematikai modellalkotás minden fázisában. A modellekből történő

építkezés elősegitése. Egy modell ellentmondástalanságának igazolása, a formális leirás és a szándék egyeztetése. A modellek fejlesztésének, részletezésének, specializálásának segitése, stb.

- Programozás támogatása
  A feladatspecifikáció nem-algoritmikus, azaz fogalmi lehetősége, a feladatspecifikációból automatikus vagy majdnem automatikus módon programspecifikáció előállitása. Bizonyos fajta programspecifikációk automatikus programozása. A programozás általában mint ember-gép együttmüködése. A job-control program automatikus előállitása. Igen magas szintü nyelvek használata, stb.

- Üzemeltetés támogatása.
  Automatikus naplózás, számlázás. Ujrainditások, archiválások, biztonsági másolatok készitése.
  Az adatok keletkezésének, elhalásának történeti nyomonkövetése, stb.

Mindezt egy nyilvántartás - a továbbiakban metaadatbázis-, az ezen müködő meghatározott célu és interaktiv logikai következtetö programok: problémaorientált feladatspecifikáló és megoldó eszközök, valamint egy ezeket kiszolgáló, önállóan is müködő lekérdező és karbantartó programrendszer és a felhasználóval kapcsolatot tartó vezérlö program valósitja meg /1. ábra/.

A MDB alapeleme az u.n. modul. Ez a világ tetszőleges darabjának, vonatkozásának akármilyen részletezettségü fogalmi leirását tartalmazza. Fogalmi leiráson két dolgot értünk. Egyrészt valahány /primitiv/ fogalomnak a bevezetését, amiből - egy általános érvényünek tekintett fogalomalgebra konstrukciós szabályain keresztül végtelen sok /további/ fogalomnak a modulon belüli értelmezettsége is következik /ld. 1.példa/.

Másrészt a fogalmak közötti valahány összefüggés kimondását, amit a szóbanforgó világdarab egyfajta axiomatizálásának tekinthetünk, s melyek logikai következményeként általában végtelen sok további összefüggés vezethető le /ld.2/a példa/.

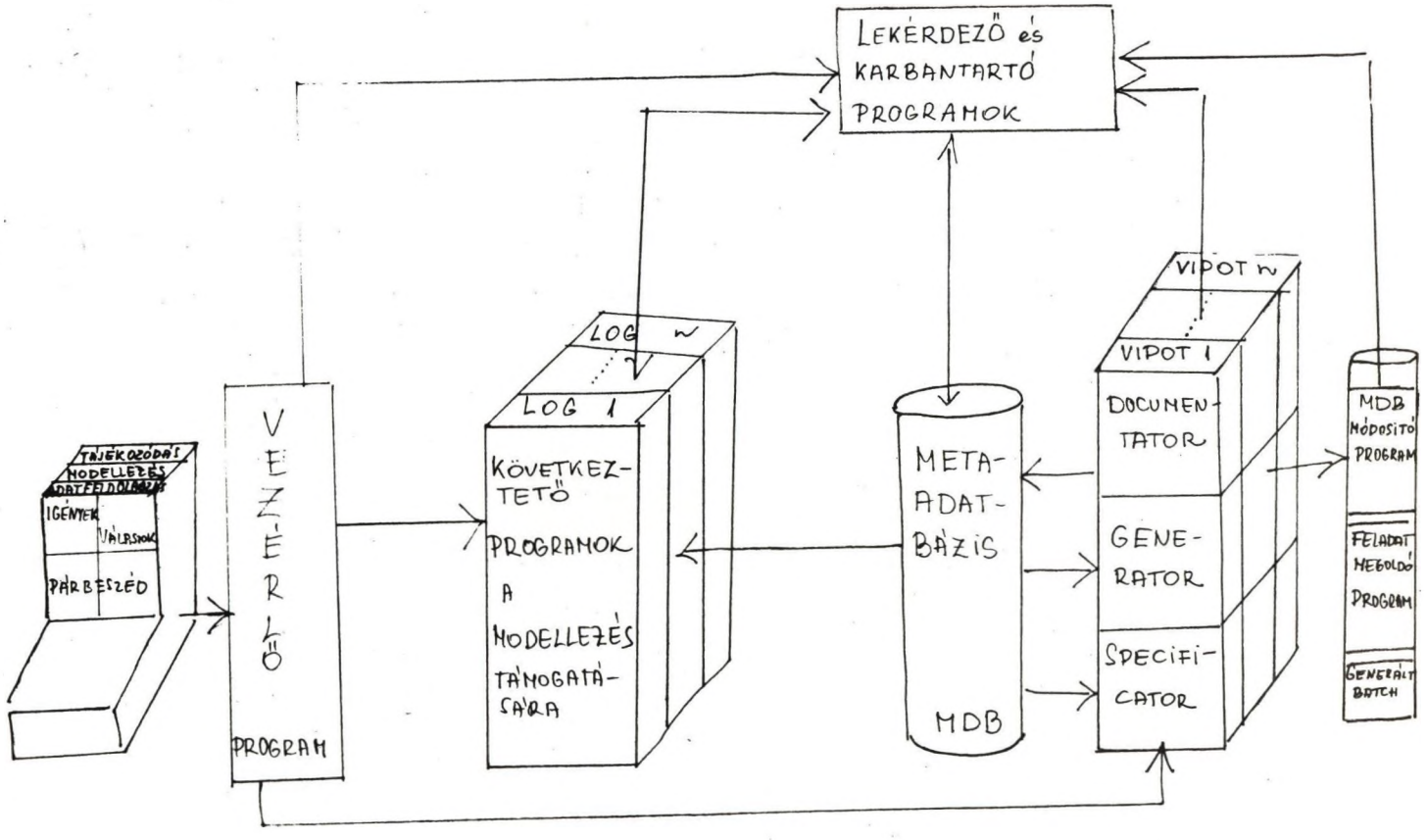
A modulokban értelmezhető fogalmak egy része, terjedelmüket tekintve adatszerü, azaz gépen létezhet, más részük nem. Az adatszerü fogalmak terjedelmét ismertnek nyilvánithatjuk, ha megadjuk azt az algoritmust /eljárást/, melynek végrehajtása előállitja azt.

Egy modul ismert terjedelmü fogalmaiból, a fogalomalgebra müveleteinek és a fogalmi összefüggéseknek a logikai következményeként más fogalmak terjedelme is előállitható. Ezt hivjuk adatfeldolgozásnak. Az ehhez vezető logikai következtetés sorozat az adatfeldolgozó program specifikációja. Annak vizsgálata, hogy ismert terjedelmü fogalmakból egyáltalán milyen más fogalmak terjedelme származik logikai következményként, a rendszertervezés egyik fontos mozzanata. Mivel mindhárom lépés megfogalmazásunkban logikai következtetések sorozata, a mesterséges intelligencia eszközeinek alkalmazásával elvben teljesen mechanizálható. /Ez nem jelenti tetszőleges probléma automatikus megoldását/ De ha ez gyakorlati okokból nem is lehetséges /célszerü/, az emlitett tevékenységek, a tisztán emberi helyett, ember-gép együttmüködésévé válhatnak. Ehhez adják a keretet az AMETIST rendszer VIPOT-jai /Very Intelligent Problem Oriented Tool/. Ezek a mesterséges intelligencia "vak" tételbizonyitó rendszerei helyett, egy-egy problémakörhöz speciálisan igazodó, gyakran direkt matematikai módszereket alkalmazó eszköztárat jelentenek. Programozási eszköztárat, amely az ember intuicióját és tudását anélkül viszi bele a feladatmegoldásba, hogy a programozás mechanikus, érdektelen részének terheit vállalnia kellene. Ez elsősorban egy problémaorientált formalizmussal érhető el, mely az adott problémakör fogalmi leirását, és a feladatot értelmező fogalmi és terjedelmi

leirást illetve az ezeknek megfelelő modulokat épiti össze, egyetlen konzisztens modullá. A VIPOT-oknak az a komponense, mely párbeszédes üzemmódban, az adott formalizmust használva eljuttatja a felhasználót az adott problémakörben mozgó feladat megfogalmazásához, az u.n. SPECIFICATOR /ld. 2/b példa/. A VIPOT-okba beépitett speciális következtető rendszer ebből a megfogalmazásból készit automatikusan vagy további párbeszéd során programot illetve a futtatásához szükséges job control utasitásokat. /GENERATOR/.
A VIPOT-ok DOCUMENTATOR komponense részben az előkészitett, futtatható job-bal kapcsolatban rak megfelelő leirást a metaadatbázisba, részben a futtatandó job-ba épit bele egy az operációs rendszerből végrehajtással kapcsolatos információkat átvevő, és azokat a metaadatbázisba beillesztő programot.

A lekérdező programok egy dokumentációvisszakereső rendszerhez hasonlóan müködnek. A visszakeresés adott tulajdonságu modulokra irányul, tulajdonságnak tekintve a modulokhoz rendelt fix attributumokat - például készitő, téma, stb. - valamint magukat a modulban előforduló fogalmakat, mint a tartalomra utaló kulcsszókat.

A karbantartó programok alacsonyabb szinten egyszerü szövegszerkesztőként müködnek, mig a legmagasabb szinten a modellépitkezés müveleteit hajtják végre.
A vezérlő program tranzakció processorként értékeli ki a rendszerhez fordulásokat - prioritás, jogosultság, stb. szempontokból - és intézkedik a megfelelő komponensek aktivizálásáról.



Az AMETIST komponensei

/1. ábra/

3. A modulok felirásának nyelve: CSDL-OT /1. példa/

A CSDL-OT nyelv fejlesztésének, tervezésének kezdeti fázisának eredményeit ismertetjük. Jelenlegi felfogásunkban egy modul egy fogalmi sémaként jelenithető meg, ezért a nyelv neve: CSDL-OT /Comceptual Schema Definition Language-Országos Tervhivatal/. Az OT kiegészitès arra utal, hogy figyelembe vettük a nyelv tervezésekor a tervhivatali alkalmazások által támasztott követelményeket /ld. pl: beszámolás és aggregálás jelenségeit/.
A nyelv központi adattipusa a fogalom. Megkülönböztetünk primér és származtatott fogalmakat. Az utóbbiak a nyelv következtetési szabályaival vannak szoros kapcsolatban.
A fogalmak között meghatározott tipusu kapcsolatok lehetnek. A tipusok száma korlátozott. A fogalmak és a közöttük fennálló kapcsolatok leirására szolgálnak a sémakifejezések, amelyeknek két formája megengedett: a/fogalomnév, b/ kapcsolattipus (fogalomnév-1, fogalomnév-2). A fogalmi séma sémalezáró delimiterek közötti sémakifejezések sorozatából áll.
A fogalmi séma megnevezhető. A fogalmi séma alakja:

[sémanév] schema sémakifejezések sorozata endshema

A fogalmi sémát majd ki kell egésziteni megszoritó, környezeti és kiértékelő állitásokkal. Ezekről itt nem lesz szó.
A fogalmi séma definiciójából látható, hogy könnyen ábrázolható irányitott gráfként, ahol a fogalomneveknek cimkézett szögpontok, a kapcsolattipusoknak cimkézett élek felelnek meg. Fontos megszoritás a nyelvben, hogy két fogalomnév között egy adott irányitással maximum egy kapcsolat állhat fenn.

3.1. A megengedett kapcsolattipusok

1., s-kapcsolat: s (A,B).
Az A fogalom részét képezi a B fogalomnak, azaz A minden előfordulása egyben B-nek is előfordulása. Az s-kapcsolat teszi lehetővé fogalomhierarchiák kijelölését. Az s-kapcsolatok egy parciális rendezési relációt határoznak meg a fogalmi séma fogalmain.

2., f-kapcsolat: f(A,B),
Az A és B fogalmak között függvénykapcsolat áll fenn: A minden előfordulásához hozzá van rendelve B-nek egy és csak egy előfordulása.

3., m-kapcsolat: m (A,B).
Az A és B fogalmak között fennálló mérték tulajdonságu függvénykapcsolat.

4., i-kapcsolat: i(A, B) .
Az A és B fogalmak között fennálló intenziv mérték tulajdonságu függvénykapcsolat.

5., a-kapcsolat: a (A,B).
Az aggregálás függvénye. B előfordulásai A előfordulások halmazai.

6., syn-kapcsolat: syn (A,B).
Az A fogalom minden előfordulása B-nek is előfordulása és forditva is. Röviden: A és B szinonimák.

Megjegyzések: I., Az s- és f- kapcsolatokkal felirható fogalmi sémák ekvivalensek az SDLA-ban [6] felirható logikai sémákkal. II., Az m- és i- kapcsolatok bevezetését a közgazdasági alkalmazás indokolja.

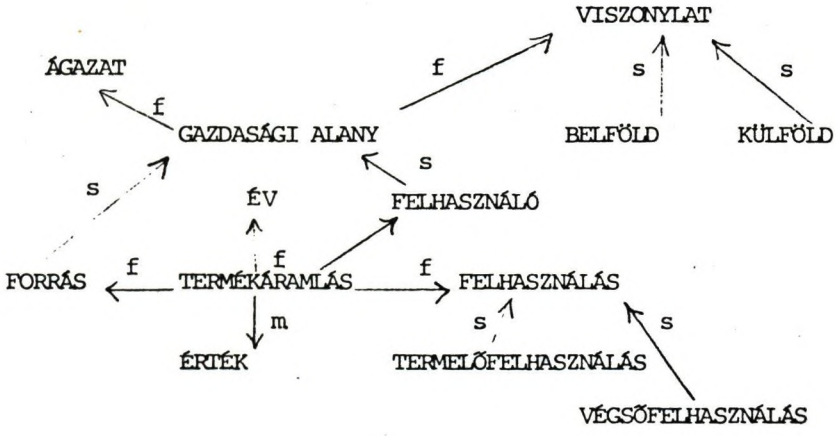
1. Példa. A termékáramlásokkal kapcsolatos fogalomkörnek megfelelő fogalmi séma

TERMÉKÁRAMLÁSOK VILÁGA séma

f/ TERMÉKÁRAMLÁS,FORRÁS/; f/TERMÉKÁRAMLÁS,FELHASZNÁLÓ/;
f/ TERMÉKÁRAMLÁS,ÉV/; f/TERMÉKÁRAMLÁS,FELHASZNÁLÁS/;
m/ TERMÉKÁRAMLÁS,ÉRTÉK/; s/FORRÁS,GAZDASÁGI ALANY/;
s/ FELHASZNÁLÓ,GAZDASÁGI ALANY/; f/GAZDASÁGI ALANY,ÁGAZAT/;
f/GAZDASÁGI ALANY,VISZONYLAT/;
s/ BELFÖLD,VISZONYLAT/; s/KÜLFÖLD,VISZONYLAT/;
s/ TERMELŐFELHASZNÁLÁS, FELHASZNÁLÁS/;
s/ VÉGSŐFELHASZNÁLÁS, FELHASZNÁLÁS/;
syn/KÜLFÖLD,VISZONYLAT-BELFÖLD/;
syn/VÉGSŐFELHASZNÁLÁS,FELHASZNÁLÁS-TERMELŐFELHASZNÁLÁS/

sémavége

A példa grafikus ábrázolása:



3.3. Következtetési szabályok

Egy adott fogalmi séma a következtetési szabályok használatával bővithető. A következtetési szabályok két csoportra oszthatók: egyszerü és fogalomgeneráló szabályok. Az előbbiek csak egy uj kapcsolatot /élt/, az utóbbiak uj kapcsolatot /él/ és fogalmat /szögpont/ vezetnek be a meglévő sémából kiindulva.

A. Egyszerü szabályok:

1., Az s-kapcsolat tranzitiv tulajdonsága:

$s(A,B) \wedge s(B,C) \Rightarrow s(A,C)$; $\Rightarrow s(A,A)$.

2., A syn-kapcsolat ekvivalencia tulajdonságai:

$\Rightarrow syn(A,A)$; $syn(A,B) \Rightarrow syn(B,A)$; $syn(A,B) \wedge syn(B,C) \Rightarrow syn(A,C)$.

3., Az i-, m-, a-, és s-kapcsolatok az f-kapcsolat partikuláris esetei:

$i(A,B) \Rightarrow f(A,B)$; $m(A,B) \Rightarrow f(A,B)$; $s(A,B) \Rightarrow f(A,B)$; $a(A,B) \Rightarrow f(A,B)$.

B. Fogalomgeneráló szabályok:

1. Összetett függvénykapcsolat képzése:

a., $f(A,B) \wedge f(B,C) \Rightarrow f(A,B.C)$.

b., $s(A,B) \wedge f(B,C) \Rightarrow f(A,C)$

c., $s(A,B) \wedge m(B,C) \Rightarrow m(A,C)$.

d., $s(A,B) \wedge i(B,C) \Rightarrow i(A,C)$.

e., $\Rightarrow s(B.C,C)$.

2. A beszámlálás szabálya:

$f(A,B) \Rightarrow m(B, \#(A,B))$.

$\#(A,B)$ egy származtatott fogalom, és a B-beli előfordulásokhoz rendelt A-beli előfordulások számát jelöli.

3. Az aggregálás.

a., $f(A,B_1) \wedge f(A,B_2) \wedge \ldots \wedge f(A,B_n) \Rightarrow a(A,\{A:(B_1,\ldots,B_n)\})$.

$n \geq 1$, $B_1 \neq B_j$, ha $i \neq j$.

b., $\Rightarrow a(A,\{A\})$.

Az $\{A:(B_1,\ldots,B_n)\}$származtatott fogalom, előfordulásai az egyes $(b_1,\ldots,b_n)$, $b_i \in B_i$ előfordulásokhoz hozzárendelt A előfordulások összeségei. Előfordulások halmazából egy uj fogalom egy előfordulását képezzük: ez az aggregálás.

c., $f(A,B_1)\wedge\ldots\wedge f(A,B_n) = f(A,(B_1,\ldots,B_n))$.

$A(B_1,\ldots,B_n)$ származtatott fogalom fogalmak csoportositását jelöli. Előfordulásai a $B_1,\ldots,B_n$ fogalmak előfordulásaiból előállitható n-esek.

d., $f(A,B_1)\wedge\ldots\wedge f(A,B_n) = f((B_1,\ldots,B_n), \{A:(B_1,\ldots,B_n)\})$.

A többváltozós függvény implicit bevezetése.

4. A szükités.

$f(A,B_1)\wedge\ldots\wedge f(A,B_n)\wedge s(C_1,B_1)\wedge\ldots\wedge s(C_n,B_n)$

$\Rightarrow s([A:(B_1\supset C_1,\ldots B_n\supset C_n)], A)$

$n \geq 1$, $B_i \neq B_j$ ha $i \neq j$.

Az $[A:(B_1\supset C_1,\ldots,B_n\supset C_n)]$ származtatott fogalom, előfordulásai mindazon A előfordulások, amelykhez az $f(A,B_i)$ megfeleltetés során C_i-beli előfordulás tartozik $(i=1,\ldots,n)$.

5. Részfogalom - fogalom viszonyok.

a., $s(\underline{\text{EMPTY}}, A)$

Az előfordulás nélküli fogalom bármely fogalomnak részfogalma.

b., $s(A,B) \Rightarrow s(B-A,B)$.

B-A az A-nak B-re vonatkoztatott relativ kiegészitésének fogalma.

c., $s(A_1,B)\wedge\ldots\wedge s(A_n,B) \Rightarrow s(A_1\cap\ldots\cap A_n,B)\wedge s(A_1\cup\ldots\cup A_n,B)$.

A.4., Az aggregátumokra vonatkozó egyszerü szabályok irják le a függvénykapcsolatok átörökitésének jelenségét.

a., $a(A,\{A:(B_1,\ldots,B_n)\})\wedge m(A,B) = m(\{A:(B_1,\ldots,B_n)\},B)$.

b., $a(A,\{A:(B_1,\ldots,B_n)\})\wedge i(A,B) = i(\{A:(B_1,\ldots,B_n)\},B)$.

c., $a(A,\{A:(B_1,\ldots,B_n)\}) \Rightarrow f(\{A:(B_1,\ldots,B_n)\},B_i)$,

$i=1,\ldots,n$.

B.6. Az s-kapcsolat és syn-kapcsolat közötti összefüggések:

a., $s(A,B) \wedge s(B,A) \Rightarrow syn(A,B)$.

b., $s(A,B) \Rightarrow syn(A.B,A)$.

c., $s(A,B) \wedge *(C,X.A.B.Y) \Rightarrow *(C,X.A.Y)$,
$s(A,B) \wedge *(X.A.B.Y,C) \Rightarrow *(X.A.Y,C)$.

$*$ tetszőleges kapcsolattipust jelöl.

Bemutatjuk a példa kapcsán a következtetési szabályok alkalmazását. Legyen a feladat azon függvénykapcsolat meghatározása, amely az ÉV, a FORRÁS szerinti ÁGAZAT-ok, valamint a FELHASZNÁLÓ szerinti ÁGAZAT-ok bontásában megadja a TERMELŐ-FELHASZNÁLÁS-u TERMÉKÁRAMLÁS-ok összegzett ÉRTÉK-ét.

1. f(TERMÉKÁRAMLÁS,FELHASZNÁLÁS)
2. s(TERMELŐFELHASZNÁLÁS,FELHASZNÁLÁS)
3. s([TERMÉKÁRAMLÁS:(FELHASZNÁLÁS⊃TERMELŐFELHASZNÁLÁS)], TERMÉKÁRAMLÁS) B.4 /1,2/
4. syn([TERMÉKÁRAMLÁS:(FELHASZNÁLÁS⊃TERMELŐFELHASZNÁLÁS)], X) uj kapcsolat röviditéshez
5. s (X,TERMÉKÁRAMLÁS)
6. f (TERMÉKÁRAMLÁS,FELHASZNÁLÓ)
7. f (TERMÉKÁRAMLÁS,FORRÁS)
8. f (TERMÉKÁRAMLÁS,ÉV)
9. f (X,FELHASZNÁLÓ) B.1.b./5,6/
10. f (X,FORRÁS) B.1.b./5,7/
11. f (X,ÉV) B.1.b./5,8/
12. s (FORRÁS,GAZDASÁGI ALANY)
13. s (FELHASZNÁLÓ,GAZDASÁGI ALANY)
14. f (GAZDASÁGI ALANY, ÁGAZAT)
15. f (FORRÁS,ÁGAZAT) B.1.b. /12,14/
16. f (FELHASZNÁLÓ,ÁGAZAT) B.1.b. /13,14/
17. f (X,FORRÁS.ÁGAZAT) B.1.a. /10,15/

18. f (X,FELHASZNÁLÓ.ÁGAZAT) B.1.a./9,16/

19. f (X,(ÉV,FORRÁS.ÁGAZAT,FELHASZNÁLÓ.ÁGAZAT)) B.3.c./11,17,18/

20. f ((ÉV,FORRÁS.ÁGAZAT,FELHASZNÁLÓ.ÁGAZAT),
{X:(ÉV,FORRÁS.ÁGAZAT,FELHASZNÁLÓ.ÁGAZAT)}) B.3.d/11,17,18/

21. f(X,{X:(ÉV,FORRÁS.ÁGAZAT,FELHASZNÁLÓ.ÁGAZAT)}) B.3.a /11,17,18/

22. m (TERMÉKÁRAMLÁS,ÉRTÉK)

23. m (X,ÉRTÉK) B.1.c/5,22/

24. m({X:(ÉV,FORRÁS.ÁGAZAT,FELHASZNÁLÓ.ÁGAZAT)},ÉRTÉK) A.4.a /21,23/

25. f({X:(ÉV,FORRÁS.ÁGAZAT,FELHASZNÁLÓ.ÁGAZAT)},ÉRTÉK) A.3/24/

26. f((ÉV,FORRÁS.ÁGAZAT,FELHASZNÁLÓ.ÁGAZAT),
{X:(ÉV,FORRÁS.ÁGAZAT,FELHASZNÁLÓ.ÁGAZAT)}.ÉRTÉK). B.1.a/20,25/

Az egyes lépéseknél megadtuk az alkalmazott szabály és alkalmazott kapcsolatok sorszámait /pl. B.1.c./5,22//.

A fenti levezetés az 1. Táblázatban/ld.4.pont/ levő TF megfeleltetés jobboldalán álló függvénykapcsolat létezését bizonyitotta. Igy ez a függvénykapcsolat, ha a fogalmi séma kapcsolatai adottak, egyértelmüen adottnak tekinthető a termékáramlások világában.

A fogalmak között további olyan összefüggések létezhetnek, amelyek igazak a séma bármely modelljében /előfordulások megadása/. Pl.: bármely ember születési éve nagyobb bármely szülőjének születési événél. Az ehhez hasonló u.n. megszoritó állitások /constraint/ formalizmusát még nem dolgoztuk ki a CSDL-OT nyelvben. Éppen ezért a termékáramlások világára vonatkozó megszoritó állitásokat a példa további kidolgozása során az alkalmazott VIPOT nyelvén /megfelelő VIPOL/ adjuk meg.

4. A VIPOT-ok helye és szerepe egy példán bemutatva /2.a.példa/

Minden VIPOT egy-egy problémakörre orientált feladatspecifikációt és megoldást támogató eszközök együttese. Minden VIPOT-nak van egy formalizmusa, ez a megfelelő VIPOL nyelv. A CSDL-OT is egy

speciális VIPOL. A MDB elemei, a modulok, mindig valamely VIPOL nyelven kerülnek megfogalmazásra. Tegyük fel,hogy előző példánkhoz kapcsolódóan numerikus számitásokat kivánunk végezni a termékáramlások világában értelmezett adatokkal. Feltesszük, hogy erre a célra létezik egy VIPOT, amelynek neve legyen "Numerikus számitások" és ennek nyelve megengedi,hogy állitásainkat a megszokott matematikai jelölésekkel adhassuk meg.Többek között, lehetőség van többdimenziós tömbök kijelölésére, amelyeknél az indexek rendezett halmazokon futnak, mig a tömbérték numerikus változó.

Az 1. Táblázat adja meg az adott VIPOL-ban kijelölt tömbök indexei és tömbváltozói, valamint az 1. példában adott fogalmi séma származtatott függvénykapcsolatai közötti megfeleltetéseket /Á,R - index tartományok; FO, FH, stb. - tömbváltozók/.

1. Táblázat. Megfeleltetések a "Numerikus számitások" VIPOL és a CSDL-OT VIPOL deklarációjai között.

Á	→	ÁGAZAT
R	→	KÜLFÖLD
É	→	ÉV
FO	→	f((ÉV,FORRÁS.ÁGAZAT),{TERMÉKÁRAMLÁS:(ÉV,FORRÁS.ÁGAZAT)}. .ÉRTÉK)
FH	→	f((ÉV,FORRÁS.ÁGAZAT),{TERMÉKÁRAMLÁS:(ÉV,FORRÁS.ÁGAZAT)}. .ÉRTÉK)
TM	→	f((ÉV,FORRÁS.ÁGAZAT),{[TERMÉKÁRAMLÁS:(FORRÁS.VISZONYLAT⊃ BELFÖLD)]:(ÉV,FORRÁS.ÁGAZAT)} .ÉRTÉK)
IM	→	f((ÉV,FORRÁS.ÁGAZAT,FORRÁS.VISZONYLAT),{[TERMÉKÁRAMLÁS: :(FORRÁS.VISZONYLAT⊃KÜLFÖLD)]:(ÉV,FORRÁS.ÁGAZAT,FORRÁS.VISZONYLAT)}.ÉRTÉK)
TF	→	f((ÉV,FORRÁS.ÁGAZAT,FELHASZNÁLÓ.ÁGAZAT),{[TERMÉKÁRAMLÁS: :(FELHASZNÁLÁS ⊃ TERMELŐFELHASZNÁLÁS)]:(ÉV,FORRÁS.ÁGAZAT, FELHASZNÁLÓ.ÁGAZAT)} . ÉRTÉK)
VF	→	f((ÉV,FORRÁS.ÁGAZAT),{[TERMÉKÁRAMLÁS:(FELHASZNÁLÁS⊃VÉGSŐ-FELHASZNÁLÁS)]:(ÉV, FORRÁS. ÁGAZAT)} . ÉRTÉK)

Most már felirhatók a megszoritó állitások is, amelyek a termékáramlások világában érvényesek:

$$\left.\begin{array}{ll} (1)\ FO(é,á) = FH(é,á) & \\ (2)\ FO(é,á) = TM(é,á) + \sum_{r \in R} IM(é,á,r) & \\ (3)\ FH(é,á) = \sum_{á_2 \in Á} TF(é,á,á_2) + VF(é,á) & \text{minden } á,á_1 \in Á, \\ (4)\ FF(é,á_1,á) = TF(é,á_1,á) / TM(é,á) & é \in É\text{-re.} \end{array}\right\}$$

A megfeleltetések egy ezzel a VIPOT-tal megoldandó feladatnál egyrészt magyarázzák, definiálják a szóbanforgó halmaz és változó szimbólumokat, másrészt ellenőrzési lehetőséget nyujtanak a VIPOT-beli müveletek értelmességére, vagyis a feladatmegfogalmazás jóságára nézve.

Egy sémához tartozó feladathoz jutunk, ha a séma fogalmainak egy részéről azt állitva, hogy terjedelmük adott, valamely más sémabeli fogalom terjedelmének meghatározottságát kérdezzük.

További feladat lehet, annak meghatározása, hogy az ismert terjedelmekből a séma függvényeit /függőségeit/ felhasználva, hogyan állitható elő a kérdéses terjedelem.

5. <u>Feladatkijelölés és megoldás</u> /2.b. példa/

Adott $VF(é,á)$, $IM(é,á,r)$, $FF(é,á,á_1)$ minden $é \in É1; á,á_1 \in Á$-ra.
Kiszámitható-e $TF(75,á,á_1)$ és ha igen, hogyan?
A szóbanforgó VIPOT olyan következtetésekre képes, amelyek a következő elemi lépésekből állnak:

- egy indexes változó kifejezése egy egyenletből;
- behelyettesités.

Képes továbbá arra, hogy ilyen tipusu következtetésekkel a keresett indexes változók meghatározásának feladatát visszavesse más indexes változók értékének meghatározására.

Akkor áll le, ha a visszavezetésben adott változókig jut el, vagy ha olyan szituációba kerül, hogy egy változó értékének meghatározásához önmagát kellene felhasználnia. Ez utóbbi esetben felhasználói közbeavatkozást kér.
A levezetés menete /a zárójelbe tett számok a felhasznált egyenletekre utalnak/:

$$\left.\begin{aligned} TF(75,á,á_1) &= \underline{FF(75,á,\overset{*}{á_1}) * TM(75,á_1)} && /4/ \\ TM(75,á_1) &= FO(75,á_1) - \sum_{r \in R} IM(75,á_1,r) && /2/ \\ FO(75,á_1) &= \sum_{á_2 \in Á} TF(75,á_1,á_2) + \underline{VF(75,á_1)} && /1,3/ \end{aligned}\right\} á,á_1 \in Á,$$

Közbeavatkozást igénylő szituációba kerültünk: $TF(75,á,á)$-t önmagával kellene meghatározni. A felhasználó /a levezetés elemzése, ill. intuiciói alapján/ ugy dönt, hogy a VIPOT probálja meg $TM(75,á_1)$-et meghatározni. Ekkor a levezetés a következő lépéssel folytatható:

$$TF(75,á_1,á_2) = \underline{FF(75,á_1á_2)} * TM(75,á_2),\ á_1,á_2 \in Á. \quad /4/$$

Ujra hasonló szituációba jutottunk. Az ilyen helyzetek egy lehetséges feloldása az, ha van olyan programunk, amely a kapott egyenletrendszert meg tudja oldani. Tegyük fel, hogy a VIPOT "ismeri" a lineáris egyenletrendszer fogalmát. A felhasználó utasitása tehát az lesz, hogy "próbáld meg a feladatot lineáris egyenletrendszerként megoldani".**

---

* Az aláhuzással azt jelöljük, hogy az illető változó adott. Itt és a továbbiakban feltételezzük, hogy $75 \in Él$

** A 2-4 lépésekben kapott egyenleteket egymásba helyettesitve

Ez pontosabban azt jelenti, hogy: "próbáld meg a levezetésből adódó

$$TM(75,á_1) = \sum_{á_2 \in Á} FF(75,á_1á_2) * TM(75,á_2) + VF(75,á_1) - \sum_{r \in R} IM(75,á_1 r)$$

egyenletrendszert $\sum_{j \in I} a_{ij} * x_j = b_i$, $i \in I$ alakra hozni!

Tegyük fel, hogy a VIPOT automatikusan képes erre, és produkálni tudja a következő megfeleltetéseket:

$$I \leftarrow Á ,$$
$$a_{i,i} \leftarrow 1 - FF(75,i,i) ,$$
$$a_{i,j} \leftarrow - FF(75,i,j), \text{ ha } i \neq j ,$$
$$b_i \leftarrow VF(75,i) - \sum_{r \in R} IM(75,i,r) ,$$
$$x_j \leftarrow TM(75,j) .$$

Sikerült tehát a feladatot lineáris egyenletrendszer megoldására visszavezetni.

A levezetés egészéből tehát a következő programspecifikáció áll elő:

- állitsd elő egy lineáris egyenletrendszer paramétereit a fenti megfeleltetések alapján,
- oldd meg a rendszert,
- ha a megoldás sikerült /erre eddig semmi biztositékunk nincs/, végezd el a

$$TM(75,j) \leftarrow x_j$$
$$TF(75,á,á_1) \leftarrow FF(75,á,á_1) * TM(75,á_1)$$

müveleteket.

A példában csak a VIPOT SPECIFICATOR komponenesének müködését illusztráltuk, ezt is csak elnagyoltan. Elhanyagoltuk ugyanis azt a kérdést, hogy hogyan oldjuk meg az ismert

adatokat. Egy valódi szituációban ezt is le kellene irni, és a programspecifikációban az adatokra való hivatkozást tárolási hivatkozásokra kellene visszavezetni.

Abstract

Combining the relational way of looking at data with the results of semantic networks and the method of algebraic program specifications a formal description tool is presented for any element of an information system. In the proposed system procedures of deductive reasoning are applied to support modelling, designing and programing activities.

Irodalom

[1] Schank, R., Conceptual Information Processing, North-Holland, 1975.

[2] A proposal for the development of a general algebraic modeling system /GAMS/. Revised Draft, Development Research Center, World Bank, Sept. 28,1977.

[3] Nijssen, G.M., Modelling in Data Base Management Systems, North-Holland, 1976.

[4] Bobrow, D., Winograd, T., and KRL Research Group: Experience with KRL-0. Proc.of JCA Vol.I, 213-222.1977.

[5] Michie, D./ed./, Expert Systems in the Micro-Electronic Age /Edinburgh, University Press, 1979/.

[6] Knuth Előd-Radó Péter-Tóth Árpád: Az SDLA előzetes ismertetése. MTA SZTAKI Tanulmányok 104/1980.

Asztalos Domonkos, Koltai Tamás, Krekó Béla
OTSZK, 1149 Budapest, Angol u. 27.

Bach Iván

ADA PROGRAMOK SZEMANTIKAI ELLENŐRZÉSÉNEK EGYES KÉRDÉSEI

A cikk főleg az identifikáció problémáit tárgyalja. A nehézségek azoknak a specialitásoknak tulajdonithatók, amelyek különböznek a klasszikus blokk-strukturáktól. Ezek: a package és a package törzsének elválasztása, a direkt láthatóság a use-clausokon keresztül, az implicit módon deklarált függvények tulterhelése.
Az ADA-manual néhány hiányosságát is megemlitjük és vázoljuk az azonositás menetének algoritmusait.
Kulcsszavak: szemantika, nyelvi strukturák.

A korszerü nyelvek szintaxisának leirásáraáltalában valamilyen CF gramatikát használnak. Erre a célra legtöbbször a jól bevált Backus-Naur-jelölést alkalmazzák. Igy tesz az ADA Manual is. Ennek alapján készült el az ADA fordító első menete. Minden helyes ADA programnak eleget kell tennie a szintaxis szabályainak. Ez azonban csak szükséges, de nem elégséges feltétel. A nyelv egy sereg olyan szemantikus követelményt tartalmaz - például a deklarálásra vonatkozó kötelezettség, a kifejezésekben szereplő objektumok tipusával kapcsolatos megkötések -, amelyek CF gramatikával nem irhatók le. Megfogalmazásuk természetes nyelven, mint az ADA esetében, vagy formális módszerekkel - denotációs szemantika, Wijgarden-féle kétlépcsős nyelvtan - történhet. Ezek a megkötések fordítási időben mindig ellenőrizhető részének, a statikus szemantikai előirásoknak ellenőrzése a fordítás második menetének feladata.
A szemantikus ellenőrzés feladatai között megkülönböztetett helyet foglal el az azonositás. Ez a müvelet sok esetben nem is választható el más szemantikus feltételektől, mint például

a tipus kompatibilitás ellenőrzésétől. Ugyanakkor az azonosítást követően más szemantikus szabályok vizsgálata mind logikai, mind progamozástechnikai szempontból általában lényegesen könnyebb feladat. Éppen ezért a második menetről szólva az azonositás lesz az a problematika, amelyet ismertetnünk kell.

A programszöveg terminális szimbólumainak szeparálása és kategorizálása - kulcsszavak, azonositók, operátorok, literálok, határolók - a lexikai elemző feladata.

A szintaktikai elemző egyrészt a határolók, másrészt a kulcsszavak többségének elhagyásával felépiti a szintaktikus fát, amely néhány pontatlanságtól eltekintve hüen tükrözi a program felépitését. Igy különválasztja az azonositók és operátorok definiáló és alkalmazott előfordulásait, de nem tesz, nem tehet különbséget például függvényhivások és tömbelem-hivatkozások között. Ezen bizonytalanságok feldoldása is a szemantikai elemzés feladata.

Végül is a második menet egy olyan szintaktikus fát kap, ahol az identikus szimbólumok összetartozását az jelzi, hogy azok a lexikai tábla azonos elemére mutatnak.

Az azonositás feladata abban áll, hogy egyrészt a definiáló előfordulások jogosságát ellenőrizze, másrészt alkalmazott előfordulások esetében megállapitsa, melyik definiáló előfordulására utalnak. Itt lényegében a konfliktusok feloldása jelenti a problémát.

A teljesség igénye nélkül tekintsük át vázlatosan az azonositást érintő nyelvi szabályokat. Azonositója lehet programegységnek /package, task, block/, rutinnak /procedure, function, entry/, ciklusnak /iteration/, tipusnak /type, subtype/, hibának /exception/, változónak /variable/, állandóknak /constans/, számoknak /number/ és cimkéknek /label/.

Az ADA nyelvben szigoru explicit predefiniálási kötelezettség van, amely alól csak a ciklusváltozó és a cimke kivétel.

Az ADA blokkstrukturáju nyelv és igy általában érvényesek itt is az ilyen nyelveknél szokásos szabályok. Egy belső programegységben deklarált azonositó elrejti a külső egységben deklarált vele identikus azonositókat, igy azokra a belső egység-

ből közvetlenül nem hivatkozhatunk. Másik szabály szerint, egy azonositó nem rejthet el ugyanabban az egységben deklarált azonositót.

Ezek alól az egyszerü, világos és a forditóprogram irójának életét nagyon is megkönnyitő szabályok alól azonban két kivétel van.

Bizonyos szimbólumok ugyanis nem rejtik el egymást, tulterhelhetőek /overloading/. Ezen rutinok nevei az operátorok és az enumerációs literálok. Az ilyen szimbólumok definiáló előfordulását tehát a név és a programstruktura önmagában nem határozza meg, ezt a szövegkörnyezet szabja meg. Helyes program esetében ennek a hozzárendelésnek egyértelmünek kell lennie.

Az ADA sziguru tipus kompatibilitási szabályai kiterjednek a formális-aktuális paraméter cserére, függvényeknél még az eredményre is. Ezen tulmenően az ADA nyelvben az ismert /például az ALGOL-60-ban szokásos/ rutinhiváson kivül lehetőség van a híváskor a formális paraméterek névvel történő azonositására /named association/. Végül a bemenő formális paramétereknek adhatunk opcionális kezdőértéket /default value/, híváskor az ehhez a paraméterhez tartozó aktuális paraméter lemaradhat.

Az ADA szabályai szerint ennek alapján, ha a rutinok paraméterének /és eredményének/ tipusai, nevei, sorrendje és kezdőértékkel való ellátottsága tekintetében bármilyen különbség van, akkor identikus nevü rutinok nem rejtik el egymást. Ugyanez vonatkozik értelemszerüen az operátor szimbólumokra. Itt paraméternév és kezdőérték hiján csakis a tipus disztingvál.

Durván azt mondhatjuk, hogy két identikus nevü rutin akkor látható egyidejüleg, ha létezik olyan hivási mód, amelynél a szövegkörnyezet alapján egyértelmüen azonosithatunk. Ez persze nem jelenti azt, hogy adott esetben nem található két- vagy többértelmü hivás. Ilyenkor nem alkalmazhatjuk az azonositáskor a hagyományos, az elrejtésen alapuló láthatósági szabályokat, a program hibás. Ezzel szemben a fentiek szellemében meg nem különböztethető rutinok és operátorok láthatóságára a szokásos szabályok érvényesek, tehát elrejthetik egymást.

Hasonlóan tulterhelhetőek az enumerációs literálok. Persze egy

enumerációs tipushoz itt sem tartozhat két identikus literál. Enumerációs literálok nem rejthetik el egymást.
A hagyományos láthatóságot módositó másik lehetőség az use-clause. A csomag /package/ fogalmából következik, hogy a külső programrészek számára hozzáférhetővé kell tenni a csomag belsejében, pontosabban annak látható részében deklarált azonositókat. Ez a hagyományos láthatósági szabályok kis mértékü kiterjesztésével, szelektálással oldható meg. Az

Id1 . Id2

konstrukcióval, ahol Id1 egy csomagnév, az Id2 pedig a csomag látható részében deklarált azonositó, hozzáférhetünk a csomag belsejéhez. Minthogy csomagban is deklarálhatunk csomagot, a szelektorlánc több elemből is állhat. Ilyenkor közvetett láthatóságról beszélünk.
A kényelmetlen szelektálást oldhatjuk fel az use-clause segitségével. Ha valamelyik egység deklarativ részében use után csomagneveket szerepeltetünk, akkor ezen csomagok látható részében deklarált azonositókra közvetlenül, tehát nem csak szelekcióval hivatkozhatunk a szóban forgó egységen belül.
Az emlitett kivételek, amellett, hogy megkeseritik a forditóprogram irójának életét, egy sereg uj problémát vetnek fel. Szóljunk néhány szót ezekről is.
Eddig hallgatólagosan feltételeztük, hogy a nevek konfliktusa azonos jellegü deklarációk között lépett fel. Előfordulhat azonban, hogy különböző jellegü - tulterhelhető és nem tulterhelhető - identikus azonositók vegyesen összekeverve szerepelnek. Foglaljuk röviden össze az erre vonatkozó szabályokat.
Amennyiben a hagyományos láthatósági szabályok szerint legközelebbi deklaráció nem tulterhelhető, akkor a deklaráció nemcsak nem tulterhelhető testvéreit, de a vele identikus azonositóju tulterhelhető deklarációkat is elrejti. Ilyenkor az azonositás teljesen a szokásos módon történik.
Ugyanakkor, ha az első deklaráció t lterhelhető, ugy a nem tulterhelhető azonositók rejtve maradnak, de a megkülönböztet-

hető azonositók egyidejüleg láthatóak lesznek. Az egyértelmü azonositás ilyenkor a szövegkörnyezet alapján történik. Előfordulhat, hogy a hagyományos láthatósági szabályok szerint nem találunk deklarációt. Ekkor és csakis ekkor vehetőek figyelembe a use-clause segitségével közvetlenül láthatóvá tett deklarációk. A hagyományos, elsődleges láthatóság tehát "übereli" ez utóbbi, a use-clause szerinti másodlagos közvetlen láthatóságot.

Fontos, hogy másodlagos láthatóság esetében a programstruktura nem játszik szerepet, nincs elrejtés, minden, ami látszik, egyformán látszik.

A tulterhelhető és nem tulterhelhető azonositók itt sem keveredhetnek. Mivel elrejtés nincs, a nem tulterhelhetőből csak egyetlen lehet, a tulterhelhetőek pedig potenciálisan mind számba jöhetnek.

Ennyit az azonositás szabályairól, megjegyezve, hogy az ismertetés korántsem teljes. Ami a tipus-kompatibilitást illeti, egyelőre érjük be a szükebb tipusazonosság következményeivel.

A második menet célfüggvénye, hogy minden alkalmazott előforduláshoz megtalálja az érvényes deklarációt, és az eredetileg - az első menet végén - a lexikai táblára mutató nyilat erre a deklarációra irányitsa rá.

Az azonositás szempontjából fontos a láthatóság. Ennek gyors megállapitására minden definiáló előfordulásról nyilvántartjuk, melyik egység deklarációs részében van. Minden egységbe való belépéskor annak láthatóságát igaz értékkel inicializáljuk, az egységbe való kilépéskor viszont hamisra állitjuk.

Bizonyos komplikációt okoznak a rutinok, csomagok és folyamatok /task/, ahol a specifikáció elválhat, illetve az utóbbi két esetben szükségképpen elválik a törzstől. A szabályok szerint a törzs és a specifikáció egy egységnek számit, igy a törzsbe beépitve a specifikáció láthatóságát igazra kell állitani, a törzset elhagyva mind a specifikáció, mind a törzs láthatósága hamis lesz.

Ezzel az elsődleges láthatóság könnyen ellenőrizhető. Megkeressük a megtalált deklarációt magába foglaló egység deklaráció-

ját, és az ott található láthatósági bit adja az információt. Megjegyezzük, hogy szelekciónál a prefix, amely ilyen esetben mindig csomagnév, áttöri a szóban forgó csomag láthatósági korlátját.

A másodlagos láthatóság ellenőrzésére minden csomagspecifikációhoz egy use-számlálót rendelünk. Az egységek deklarációjában az use-listákat kigyüjtjük és az emlitett csomagok számlálóját inkrementáljuk. Az egységből való kilépéskor ismét átlapozzuk az use-listákat, csak most dekrementálunk. Ha valamelyik csomag use számlálója zérustól különböző, akkor az abban lévő deklarációk másodlagosan közvetlenül láthatók.

Az identikus azonositók deklarációja egy a lexikai tábla megfelelő eleméből kiinduló névláncra van felfüzve. Induláskor, mikor üres a lista, a pointer önmagára mutat.

Egyszerüsiti az azonositás munkáját, ha a látható deklarációk a programstrukturának megfelelő sorrendben vannak erre a listára felfüzve. Sajnos, ez a sorrend nem mindig egyezik meg a textuális sorrenddel.

A problémát a szétválasztott specifikációk és törzsek jelentik. A specifikációban deklarált azonositóval identikus, a specifikációt tartalmazó, de annak lezárása után deklarált azonositó a törzsbe való belépést követően textuálisan közelebb, de a láthatóság szempontjából messzebb van, mint a specifikációban deklarált. Éppen ezért célszerübb az adott példában a két deklarációt a szöveg szerintivel ellentétes sorrendben a névláncra felfüzni.

Ennek biztositására minden egységhez egy szintszámot rendelünk. Ha a névláncban megtalált identikus nevü deklarációt tartalmazó egység szintszáma nem nagyobb, mint az aktuális szintszám, akkor az uj deklarációt a keresés szempontjából az első helyre füzzük fel. Ha a tartalmazó egység szintszáma nagyobb az aktuálisnál, akkor a tartalmazási láncon, amely a programstrukturát tükrözi, visszalépünk az aktuális szintre. Ha az igy kapott tartalmazó egység nem azonos az aktuális egységgel, akkor ugy járunk el, mint előbb. Amennyiben a két

egység azonos, az uj deklaráció átlépi a most megtaláltat és az egész eljárást a névlánc következő elemével megismételjük.

Ezzel az elhelyezési algoritmussal elérjük, hogy a látható deklarációk a névláncban a hagyományos szabály szerinti láthatósági sorrendben vannak elhelyezve.

Az azonositó definiáló előfordulásának a névláncba füzésével még nem zárhatjuk le a szemantikai ellenőrzést. Meg kell vizsgálnunk, nincs-e ugyanabban a programegységben olyan deklaráció, amelyet az uj elrejtene. A névláncon tehát tovább kell keresnünk mindaddig, amig el nem hagyjuk az aktuális egységet.

Bizonyos bonyodalmat okoznak azok az entitások, amelyeknek két deklarációjuk van. Ilyenek a kétszer - először nem teljesen, másodszor teljesen - deklarált tipusok, a privát tipusok, végül a privát tipusu állandók. Itt mindkét deklaráció bekerül a névláncba, azzal, hogy a nem teljes deklaráció hivatkozik a teljesre. Az alkalmazott előfordulások arra a deklarációra utalnak, amely a szóban forgó helyről látható.

Sokkal nehezebb és kényesebb kérdés az azonositók alkalmazott előfordulásainak azonositása. Tekintsük ebből a célból példaként kifejezések elemzését, a teljesség és szabatosság igénye nélkül.

A kifejezés - mint minden szintaktikus konstrukció - fa formájában adódik át a második menetnek. Ennek levelei azonositók, operátorok, szimbólumok vagy literálok lehetnek. A második menet feladata az azonositók és operátor szimbólumok definiáló előfordulásának megkeresése, illetve a literálok tipusának meghatározása.

Az első feladatot kellően megvilágitottuk, szóljunk néhány szót a literálok tipusának meghatározásáról. Az ADA nyelvben az eredeti "beépitett" tipusokból kiindulva lehet tetszőleges számu uj tipust definiálni. Igy mód van például az INTEGER, tehát egész tipusból uj egész tipusokat definiálni, amelyek csak a tipus értéktartományában különböznek az eredeti tipustól. Egy egész literál, például 2, ezek szerint bármely olyan egész jellegü tipusnak eleme lehet, amelyben

a szóban forgó numerikus érték beleesik az értéktartományba. A tipus meghatározása, amelyet a környezet szab meg, fontos információ a kódgenerátor számára.
Az azonositás érdekében végighaladunk a névláncon és a korábbi szempontok figyelembevételével kigyüjtjük a potenciális definiáló előfordulásokat. Szerencsés esetben, ha éppen egy ilyen akad, vagy lehetséges, az azonositás megtörtént. Ugyanakkor az azonositó tipusa ismertté vált, ami a további elemzésnél fontos.
Amennyiben az azonositás adott pillanatban nem egyértelmü, a potenciális definiáló előfordulásokat és az eredmény potenciális tipusait egy-egy listára füzzük fel, egyre az elsődlegesen, egyre a másodlagosan láthatókat.
A kifejezés fáján a levelekkel szomszédos csomópontokra áttérve ezen potenciális lehetőségek egy részéről kiderülhet, hogy az adott szövegkörnyezetben nem megfelelőek. Például, ha függvényhivás esetén az aktuális paraméterek helyén álló kifejezés tipusa nem egyezik a megkivánt tipussal, vagy az operátor nincs értelmezve valamilyen tipuspárra.
Végül is egy farészletet teljesen feldolgozva a szövegkörnyezet az azonositási lehetőségeket és ezzel együtt az eredmény tipusválasztékát csökkenti.
Ha egy olyan ponthoz érünk, amelynek tipusa egyértelmü, vagyis a választék egyre csökkent, akkor az ehhez a ponthoz tartozó farészletet felülről lefelé megvizsgálva most már egyértelmüen kell tudnunk dönteni. Az eredmény tipusának ismerete ugyanis kiszürhet egyes alternativákat, sőt helyes program esetén szükségképpen csak egyetlen alternativa marad.
Ez az azonositási eljárás lényege. Algoritmizálása még a vázolt egyszerüsitett feltételek mellett is elég szövevényes. Néhány momentumra azonban felhivnánk a figyelmet, amely az egyébként sem egyszerü feladatot még bonyolultabbá teszi.
Az egyikről már szóltunk. Bizonyos objektumok, nevezetesen a numerikus literálok nem egyetlen tipushoz, hanem egy tipusosztály bármely eleméhez tartozhatnak.
A tipus képzési szabályai szerint tipusokból származtathatunk derivált tipusokat. Az ilyen tipusu objektumok ugyan nem kom-

patibilisek az eredeti tipus objektumaival, ugyanakkor azonban az eredeti tipus objektumára értelmezett operációk és függvények - bizonyos feltételek mellett - automatikusan, minden külön deklaráció nélkül értelmezettek a derivált tipus elemére is. Ennek nyomon követése meglehetősen szövevényes feladat.
Megjegyezzük, hogy a cimkék láthatósá ára egészen más szabályok érvényesek.
Egyes nem triviális kérdésekben a Manual szövege nem egyértelmü. Ilyen esetekben az ADA szellemében, pontosabban egy koncepcióval, amelyet mi az ADA szellemének hiszünk, hoztunk döntéseket.
Összefoglalva megállapitható, hogy statikus szemantika ellenőrzése és azon belül elsősorban az azonositás megvalósitása igen sokrétü és bonyolult feladat. Az az érzésünk, hogy a nyelvi lehetőségek növekedésével az algoritmus fokozottabban és talán nem is indokolhatóan válik szövevényesebbé.
Ez a jelenség nem uj a szakmában. Elsősorban és döntően ott jelentkezik, ahol egy nyelv tervezői és implementálói diszjunkt halmazokat alkotnak. Ha ugyanazok a tervezők és implementálók bármennyire skizoid természetüek is, a tervező a legihletettebb percekben is gondol az implementálással.

Abstratc: The paper mainly treats the problems of identification. Analyzes the difficulties due to the specialities different from the classical block structures. These are: the separation of package specification and package body, the direct visibility via use-clauses, the overloading and the implicitly declared derived function.
Some gaps of the Ada Manual is mentioned, and algorithms outlined for performing identification.

Bach Iván
MTA Számitástechnikai és Automatizálási Kutató Intézet
Budapest, XI. Kende u. 13-17.

Bárány Sándor–Langer Tamás

KORSZERŰ RENDSZERPROGRAMOZÁS ÉS CSOPORTMUNKA A CDL2 NYELVI RENDSZERBEN

A cikk bevezető részében összefoglalja a programozási nyelvre szabott fejlesztő rendszerek /ún. nyelvi rendszerek/ előnyeit összevetve a hagyományos forditási rendszerekkel. Szemléltetésképpen vázlatosan bemutatja a CDL2 nyelvi rendszert. Az irás második része arról ad áttekintést, miként támogathatja egy nyelvi rendszer /jelen esetben a CDL2 nyelvi rendszer/ a csoportmunkát.

Kulcsszavak: rendszerprogramozás, nyelvi rendszer, CDL2 programozási nyelv, forditási rendszer, csoportmunka.

A programozási munka folyamata igen sok lépésből áll, kezdve a program **meg**tervezésétől egészen a kész, hibátlannak vélt, hangolt program átadásáig, illetve még tovább a karbantartásig, továbbfejlesztésig. Kézenfekvő tehát, hogy a programkészitést jól használható segédeszközökkel /segédprogramokkal, rendszerprogramokkal/ támogassuk. A programok elkészitése általában a feladathoz legjobban megfelelő programozási nyelven történik. A hagyományos forditóprogramok a programozás folyamatát csak attól a lépéstől segitik, amikor már "kész" a program, /kezdődhet a "belövés"/, a programfejlesztés egészéhez nem nyújtanak elegendő támogatást /főként azért, mert egy-egy lépés elvégzését az operációs rendszer más programjaira bizzák/. Az 1. táblázat erről mutat be egy vázlatos felosztást a szokásos támogatásokkal együtt. A táblázatból kiderül, hogy elég sok olyan lépés van, amikor a programozó vagy semmilyen segitséget nem kap, vagy az őáltala választott nyelv ismerős fogalmait /pl. azonositó, változó, eljárás/ használva nem tudja az adott munkafázist elvégezni.

Egy adott programozási nyelven megoldandó részfeladathoz	van-e a nyelvhez szabott gépi támogatás
programterv készités, ellenőrzés	kivételesen
különforditási egységekre bontás	kivételesen
különforditási egységek kapcsolódásának ellenőrzése	az ellenőrzés később, a linkage editoron keresztül történik
részletesebb programvázlat készitése	kivételesen
teljes bekódolás	kódoláshoz segitség nincs
a gépelési /lyukasztási/ hibák, a lokális szintaktikus hibák azonnali ellenőrzése	egyszerübb v. speciális nyelveknél /BASIC, APL/ inkrementális, párbeszédes forditás; PL/I, FORTRAN esetén párbeszédes szintaxis ellenőrzés
globális szintaktikai, statikus szemantikai hibák ellenőrzése	általánosan van
a program javitása, szerkesztése	nincs
a program automatikus optimalizálása	van, pl. PLIOPT
/párbeszédes/ hibakereséshez alkalmas kód	van, pl. PL/I CHECKOUT, de ez kivétel
hangoláshoz, méréshez alkalmas kód	nagyon ritkán
különforditási egységeknek a nyelv szintjén való összerakása	nincs
forrásprogram-változatok nyilvántartása, kezelése és tárolása	nincs

1. táblázat

A programozási munka hatékonyságát nagymértékben lehet növelni, ha az elvégzendő feladatok mindegyikét olyan eszközzel végeztetjük, amely a nyelvhez és az adott feladathoz egyaránt illeszkedik. Az ilyen eszközök összességét az adott nyelvhez kötött nyelvi környezetnek, vagy nyelvi rendszernek nevezzük. Ez lényegesen több, mint a nyelv fordítóprogramja, hiszen a programozási munka folyamatát a kezdettől, megszakitás nélkül, egységes módon támogatja.

Néhány nyelvi rendszerről

A programozási nyelvek többségéhez még nincs nyelvi rendszer. Maga a gondolat azonban nem új. Az INTERLISP [1] rendszerben, illetve elődjeiben majd 15 éve megjelentek a nyelvi rendszer bizonyos elemei a LISP nyelv köré épitve.

Kevésbé speciális nyelvhez sokkal nehezebb nyelvi környezetet készíteni, ezért sokáig nagyrészt kisérleti rendszerek születtek erre a célra. Ezek közül a PASCAL nyelvet támogató programozási környezeteket emelnénk ki, amelyek a puszta nyelvi szerkesztőtől a programok belső formájú tárolását és feldolgozását is elvégző komolyabb rendszerekig jutottak el /pl. a MENTOR [2] rendszer/.

Ilyen előzményeket figyelembe véve alakult ki a programfejlesztést támogató párbeszédes ANSWER operációs rendszer CDL2 nyelvi rendszere. A rendszer elsősorban az alap- és célszoftver készitést /rendszerprogramozást/ kivánja támogatni, igy először egy megfelelő rendszerprogramozási nyelv után kellett nézni. Kedvező hazai tapasztalatok birtokában a CDL2 lett az a nyelv, amely köré a nyelvi környezet kiépült.

A CDL2 nyelvi rendszert a Nyugat-Berlini Müszaki Egyetemen fejlesztették ki a Nijmegeni Tudományegyetemmel és a Számitógépalkalmazási Kutató Intézettel együttmüködve. Jelenleg

Magyarországon a Siemens 7.755 gépen müködik üzemszerüen a BS2ooo-es operációs rendszerben, illetve IBM 3o31 és IBM 37o-es gépeken VM/CMS operációs rendszerben. Folyamatban van átvitele az R11-es gépre a VIDEOTON megrendelésére és integrálása az ANSWER rendszer többi komponensével.

A CDL2 nyelvi rendszer újszerü vonásai és szolgáltatásai

Alábbiakban megkisérelyük összefoglalni azokat az újszerü vonásokat, amelyeket a CDL2 nyelvi rendszer bevezetett vagy egységes keretek közé integrált. Néhány alapgondolat kifejtése már ugyanezen a konferencián három évvel ezelőtt megtörtént. [3,4] A moduláris programozással kapcsolatos bizonyos újabb gondolatokkal egy csatlakozó előadás keretei között foglalkozunk [5]. Ennek az előadásnak az első részében bemutatjuk a nyelvi rendszer jellemző vonásait, majd a második részben ismertetjük, hogy a CDL2 nyelvi rendszer hogyan támogatja a korszerü csoportmunkát.

Ellenőrzött programbeirás és javitás

A modulok beirása során a nyelvi rendszer folyamatosan ellenőrzi a program szintaktikus helyességét és azonnal jelzi, ha olyan konstrukciót kisérelünk meg bevinni, amely nem tesz eleget a nyelv /környezetfüggetlen/ szintaxisának. A folyamatos ellenőrzés a már beirt programrészek módositásakor, javitásakor is müködik, tehát a szintaktikusan helyes modult a javitás során nem lehet "elrontani". További segitség, hogy ha elfelejtjük, vagy elhagyjuk az egyértelmüen kötelező elemeket, a nyelvi rendszer ezeket helyettünk magától megirja.

Nyelvre szabott forrásjavitó

A javitás, törlés egységei,a beszúrás helye a nyelv fogalmaival jelölhetők ki: például "töröld ki az xy eljárást", "cse-

réld ki a zq azonositó" /azonositó és nem karaktersorozat!/ összes előfordulását valamire.

Egységes parancs- és programozási nyelv

A nyelvi rendszerben dolgozva lényegileg egyetlen nyelv segitségével tartjuk a kapcsolatot a számológéppel, akár programot, akár parancsot irunk. Sőt a programozó számára nem is különül el, hogy mikor ir parancsot és mikor programot. Egyetlen nyelv megtanulása elég a nyelvi rendszer használatához.

Emberközpontú ember-gép kapcsolat, megfelelő munkakörnyezet

A géppel való kapcsolat a nyelv ismerőinek igen természetes, mivel a parancsok "nyelvre szabottak", azaz a forrásnyelv szerkezetének megfelelő keresést, módositást, stb. tesznek lehetővé. Például egy next parancs bizonyos körülmények között a következő modult veszi, mig más nyelvi szinten pl. egy eljárás következő paraméterét. Mindig van egy aktuális programrészlet, amellyel éppen valami tennivaló akad /aktuális munkakörnyezet/, minden parancs alapértelmezésben erre vonatkozik, de az aktuális környezetből megfelelő kiválasztókkal bármely más programrészlet elérhető, és az aktuális munkakörnyezet paranccsal is módositható.

Hiányos programok kezelése

Lehetőség van hiányos programok ellenőrzésére /távlatilag a kipróbálására is/. Például a nyelvi rendszerben egy ajánlott munkamódszer, hogy előbb elkészitjük a programot alkotó modulok, majd a modulon belüli kisebb részek kapcsolatleirásait /interface-eit/, azaz egy kiindulási programtervet, ezeket ellenőriztetjük a nyelvi rendszerrel, és csak kifogástalan kapcsolatleirások birtokában látunk neki a részletek programozásának.

A moduláris programozás támogatása

Az előző pontban emlitett szolgáltatás segiti a programok modulokra való bontását és a modulok közötti kapcsolat ellenőrzését a programelőállitási folyamat megfelelő pillanataiban. A nyelvi rendszer kódelőállitási szolgáltatásai mentesitenek a moduláris programozás hátrányaitól, és tetszőleges programrészletből generálható a célnak legjobban megfelelő tipusú kód /lásd részletesen [5] a jelen kiadványban/.

A csoportmunka támogatása

A csoportmunkát a moduláris programozás támogatásán kivül egy egyszerü, de hatásos könyvtári szolgáltatás segiti /lásd részletesen az előadás második részében/.

Szelektiv hibajavitás

A program elemzése során észlelt hibák hibatipusok szerint vagy más elrendezésben együtt jelenithetők meg a hiba előfordulási helyével. A hibák osztályozása olyan, hogy különböző hibajavitási stratégiák követése lehetséges: pl. először arra törekszünk-e, hogy a modul a környezetébe hibátlanul beilleszkedjen, vagy arra, hogy önmagában hibátlan legyen /felülről-lefelé, vagy alulról felfelé programozási stilus/.

"Inkrementális forditás"

Mivel a nyelvi rendszeri szolgáltatások megvalósitásához az a legkényelmesebb, ha a forrásmodulokat kielemzett belső alakban tároljuk, igy ebből természetesen következik, hogy javitás esetén csak a javitott részt kell újraelemezni, ami jelentős mennyiségü gépidőt takarit meg.

Optimalizálás

Bátran irhatjuk programunkat a struktúrált programozás elveinek és a jól olvashatóság és karbantarthatóság követelményeinek megfelelően, mivel a szerkezeti optimalizáló a jól struktúráltságból fakadó összes hatékonyságvesztést kiküszöböli /eljárások nyilt behelyettesitésével, nem használt programrészek kihagyásával, a vezérlési szerkezet átalakitásával, stb./ Mivel a nyelvi rendszer az egy programhoz tartozó összes modult egyszerre "látja", az optimalizálás nem korlátozódik az egy modulon belüli dolgokra, a modul-határ nem határa az optimalizálásnak.

Kódgenerálás különböző gépekre

A jóldefiniált, körültekintően kimunkált kódgeneráló interface miatt a már meglévő /pl. R11-es, PDP 11-es, HwB/66-os és az újonnan elkészitendő kódgenerátorok könnyen illeszthetők a nyelvi rendszerhez. A kódgenerálás szintje is lehet különböző /assembly vagy bináris/ attól függően, hogy mi a tervünk a generált kóddal.

Forditóprogram helyett nyelvi rendszer

Az előzőekben számbavettünk néhány olyan vonást /a CDL2 nyelvi rendszer alapján/, amely az egy nyelv köré épült rendszert, a nyelvi rendszert jellemzi. A programozási munka teljes folyamatának egységes, gépi eszközökkel való támogatása a munka hatékonyságát kétségtelenül jelentősen növeli.

Ilyen nyelvi rendszer elkészitésének munkája sokszorosa egy forditóprogram elkészitésének. Ennek ellenére minden olyan környezetben, ahol nagyobb programozási társaság egy nyelven programozva hasonló tipusú feladatokat old meg, érdemes megfontolni ilyen nyelvi rendszer létrehozását. Itt elsősorban a

COBOL és a FORTRAN nyelvekre gondolunk, de érdekes lehet egy PL/I nyelvi rendszer is, bár ennek elkészitése lényegesen nagyobb feladat. Ha pedig új, egy adott feladathoz jobban illeszkedő nyelv megvalósitása merül fel, akkor mindenképpen érdemes gondolni arra, hogy forditóprogram helyett nyelvi rendszert készitsünk, vagy legalább a forditóprogram készitése közben gondoljunk arra, hogy később a forditóprogram részei egy nyelvi rendszer elemeiként felhasználhatók legyenek. Ebbe az irányba mutat, hogy az ADA nyelvhez kötött programozási környezet létrehozása több helyen elkezdődött, sőt az erre vonatkozó követelményeket együtt definiálták a nyelvre vonatkozó követelményekkel.

Csoportmunka a CDL2 nyelvi rendszerben

Továbbiakban a CDL2 nyelvi rendszer egyik újszerü, módszertani szempontból érdekes vonását mutatjuk be, kicsit részletesebben.

A csoportmunka és eszközei

Nagyméretü, bonyolult rendszerek fejlesztése esetén kulcskérdés, hogyan lehet elérni, hogy a - természetesen megfelelő minőségben és hatékonysággal elvégzett - egyéni munkák eredménye egységes egésszé, rendszerré álljon össze. Bonyolult rendszereket csak csoportmunkával lehet létrehozni.

Mennyiben több a csoportmunka a benne elvégzett egyéni munkák egyszerü összegénél? Vannak olyan feladatok, amelyeket nem lehet szétbontani, illetve egymástól függetlenül végezni. Ilyen mindenekelőtt az egész feladat részekre való bontása, az elkészült elemek egymáshoz illesztése és végül az egész rendszer összeépitése. A közösen végzett munkáknál döntően fontos, hogy a kapcsolódási pontok /interface-ek/ széleskörü és minden rendelkezésre álló információt figyelembe vevő ellenőrzésen essenek át. Összefoglalva azt mondhatjuk, hogy a jó csoport-

munka titka a zavartalan egyéni munka és a megfelelő összehangolás.

A hagyományos, nem egy-egy nyelvhez kötődő rendszerek az egyes programrészek kapcsolódási pontjairól nem tárolnak elegendő /nyelvi szintü/ információt. Igy ezen rendszerek használata esetén koordináció cimén megelégszünk azzal, hogy "csak" /az egyébként esetenként igen fejlett/ könyvtárkezelő szolgáltatásokat használjuk.

A csoportmunka eszközei a CDL2 nyelvi rendszerben

A nyelvi rendszer a programozók /a mi szóhasználatunkban: szerzők/ forrásprogramjait összegyüjtve egy ún. adatbázisban tárolja. Ez az adatbázis a szerzők elől el van takarva, azaz az egyes szerzők számára csak a saját forrásaik látszanak. A nyelvi rendszerbe a szerzőknek be kell jelentkezniük, és igy egy saját szerzői környezetbe kerülnek be. Itt tetszés szerint hozhatnak létre, módosithatnak és törölhetnek modulokat és programokat.

Ez a környezet teljesen független más szerzők munkakörnyezetétől. Ha egy feladat elég kicsi ahhoz, hogy egy szerző el tudja végezni, azt teljesen a saját környezetében maradva megteheti.

Bonyolultabb feladatok esetén az egyéni munkák összehangolására is szükség van. Az összehangolás sok részfeladat és tevékenység összefoglaló neve. Ilyenek például: verziók nyilvántartása és tárolása, nyelvi kapcsolódási pontok ellenőrzése, könyvtár és file-kezelés. Az összehangolás egyéb részfeladatai a programozás-technológia módszertani részéhez tartoznak.

A CDL2 nyelvi rendszer a "nyilvános könyvtár" eszközét kinálja

abból a célból, hogy a könyvtáron keresztül mások hozzáférjenek.

A szerzők saját moduljaikat is kétféleképpen tehetik a könyvtárba: "betehetik" /copy/ vagy nyilvánosságra hozhatják. A betevés azt jelenti, hogy bárki - akinek megvan a megfelelő joga - kölcsönözheti, kiveheti, a nyilvánosságra hozás pedig azt jelenti, hogy mindenkinek csak olvasást, kapcsolódást engedélyez a szerző.

Azokat a szerzőket, akik egy adott könyvtárhoz beirási joggal rendelkeznek, egy programozási csoport /team, project/ tagjainak tekinthetjük, akik között az elvégzendő munka fel van osztva /modulokra/. Mindenki a saját munkájáért /moduljaiért/ felelős, igy egy modul beirását is csak akkor engedélyezi a rendszer, ha olyan nevü modul a könyvtárban még /vagy már/ nincsen, ill. ha a szerzője a már létező modult új verzióra cseréli.

A csoport tagjainak több könyvtárhoz is lehetnek, méghozzá különböző hozzáférési jogaik: igy pl. létezhetnek feladatokra szabott segéd-könyvtárak - általánosan használatos alapmodulokkal, - amelyekhez egy egész programozási szervezet minden tagja olvasási joggal rendelkezik.

Ezek után a hozzáférési jogok három szinten befolyásolják egy szerző kapcsolatát egy adott modulhoz:

- van-e a szerzőnek a modult tartalmazó könyvtárhoz megfelelő joga - az erősebb jog biztositja a gyengébbet: a legerősebb a beirási jog, a leggyengébb az olvasási
- amikor a szerzője a modult beteszi, megszabhatja, hogy milyen módban lehet majd kivenni
- a modul átvételekor is megmondhatjuk, hogy olvasni vagy kölcsönözni akarjuk.

A csoportmunka módszerei

Az előzőekben vázolt eszköz egyszerüség folytán annyira rugalmas, hogy a legkülönbözőbb programozási módszerek gépi támogatására alkalmazható. Néha mindössze a terminológiát kell megváltoztatni, ahhoz, hogy ez láthatóvá váljék: pl. könyvtár helyett project library, szerző helyett project component, stb.

Nézzük például, a Chief Programmer Team módszer használata esetén hogyan lehet a nyelvi rendszerben dolgozni. A közös munka kezdetén a csoport vezetője megtervezi a program modulszerkezetét, megirja a modul-kapcsolatokat és az igy létrejött /majdnem teljesen üres modulokból álló/ programváz alapján szétosztja a feladatokat. Láttuk, hogy a nyelvi rendszer megengedi hiányos modulok kezelését is, igy a vezető programozónak lehetősége van a - még meg sem irt - program ellenőrző elemeztetésére és igy a kapcsolódások helyesbitésére. Ha a modul-kapcsolódásokat a szerzők nem változtatják meg /ami az összeillesztések során a nyelvi rendszerrel mindig ellenőrizhető/, a program ilyen értelmü helyessége az egész fejlesztést végigkiséri.

A szétosztott munka az egyéni, zavartalan környezetben hatékonyan folyhat, és eljuthat az ún. modul-tesztig, azaz a modul környezettől független bevizsgálásáig. Siker esetén a modul a csoport könyvtárába betehető.

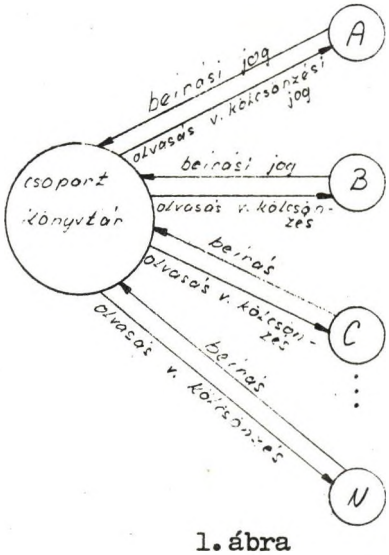
A csoport vezetője, vagy az e célra kijelölt szerző a program összeépitését úgy végezheti el, hogy a felépitő modulokat kikölcsönzi a könyvtárból. Igy biztositható az, hogy az összeépitéssel párhuzamosan elvégzett esetleges módositásoknak nincs váratlan, nemkivánatos hatásuk /majd csak a következő verzióra/. Az összeépitést végző szerző a felfedezett hibákat egyszerübb esetekben saját maga is ki tudja javitani, vagy a

hibás részeket ki is kapcsolhatja. Bonyolultabb esetekben a helyes munkamódszer az, hogy a hibák kijavitásáról az adott modul szerzője gondoskodik.

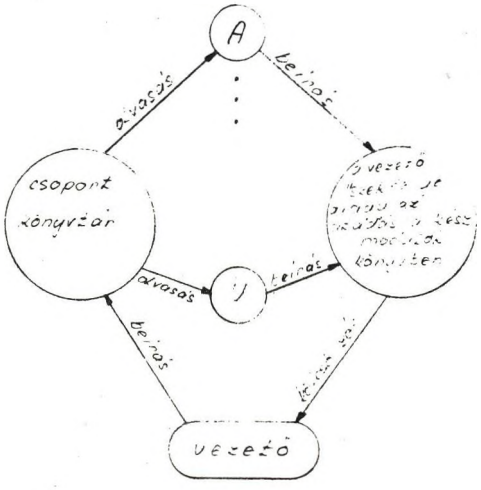
A fenti, alapvetően felülről-lefelé programozási stilus mellett elképzelhető az alulról-felfelé épitkező stilus is: pl. egy munka kezdődhet azzal is, hogy egy vagy több szerző "majd-később-jól-felhasználható" modulokat készit /meglevő alapelemeket módosit/, és a többiek ezekre a modulokra épitik fel sajátjaikat.

Ezek a stilusok keverhetők is, a munka egyes szakaszaiban igy, máskor másként dolgozhatunk.

Bemutatunk még két tanulságos példát. Az 1. ábra a "teljes szabadság" állapotát mutatja - nincs a csoportban vezető, mindenki egyenrangú, a koordinálás csak a rendszeren keresztül történik. A 2. ábra a "teljes diktatúra" állapota - a csoportnak diktatórikus vezetője van, és az összehangolás bürokratikus.



1. ábra



2. ábra

az összehangolás gépi támogatására. Egy ilyen könyvtár a szerzői környezethez hasonlit. Az úgynevezett általános célú könyvtárakhoz képest ez a nyilvános könyvtár a következőképpen működik:

- a könyvtárak csak modulokat tartalmazhatnak, nem tartalmaznak sem programokat, sem moduloknál kisebb egységeket,
- magában a könyvtárban semmilyen munkát nem lehet végezni, csupán tárolásra /és védelemre/ szolgál. A könyvtárba a szerzői környezetben kifejlesztett /hibátlan/ modult tehet be az illető szerző. A nyilvános könyvtárban tárolt modulokkal csak akkor tudunk dolgozni, ha azt a saját szerzői környezetünkbe valamilyen módon átvittük,
- a szerzők által a könyvtárakon, sőt az ott lévő modulokon elvégezhető müveleteket külön-külön hozzáférési jogok szabályozzák.

Egy modullal kapcsolatban háromféle hozzáférési jog létezik: olvasási, kölcsönzési és beirási jog.

Az olvasási, "kapcsolási" /link/ jog nem biztositja a könyvtárból egy szerzői környezetbe átvitt modul módosithatóságát, viszont azzal az előnnyel jár, hogy az átvitel során a modul fizikailag azonos marad az eredetivel, igy ha az könyvtárban kicserélődik egy újabb változatra, akkor kicserélődik a szerzői környezetben is.

A kölcsönzési, "kivételi" /copy/ jog a fentinek épp az ellenkezője: a modul átvétele a könyvtári példány lemásolásával történik. A modul igy megváltoztatható, de az eredetivel való kapcsolata megszakad. /Azonban a szerzői jogok nem adódnak át a kölcsönző számára!/

A beirási /update/ jog más értelemben különbözik a fentiektől: egyszerüen azt jelenti, hogy a szerzőnek joga van saját, hibátlan és teljes modulját a könyvtárba tenni, természetesen

Abstract:

The introductory part of the article summarizes the advantages of a programming-language-oriented, dedicated program development system compared with traditional compiling systems. As an illustration it briefly introduces the CDL2 Laboratory. The second part of the paper surveys how such a system can support group work.

Irodalomjegyzék

[1] Teitelman:
INTERLISP Reference Manual
XEROX PARC, Palo Alto, 1974.

[2] Donzeau-Gouge, Huet, Kahn, Lang:
The MENTOR program manipulation system /MENTOR Manual/
IRIA-Laboria, 1979.

[3] Bedő Á-Langer T:
A programozás technológiája az ANSWER operációs rendszerben.
Programozási Rendszerek'78 Konferencia, Szeged

[4] Bárány S-Bolgár G:
Nyelvre szabott interaktivitás az ANSWER operációs rendszerben.
Programozási Rendszerek'78 Konferencia, Szeged

[5] Bedő Á-Janni É:
A moduláris programozás támogatása a CDL2 nyelvi rendszerben.
Programozási Rendszerek'81 Konferencia, Szeged

[6] Bayer-Böhringer-Dehottay-Feuerhahn-Jasper-Koster-Schmiedecke:
Software Development in the CDL2-Laboratory
in: Software Engineering Environments, Proceedings of a simposium in Lahnstein, June 1980
North-Holland Publishing Co. 1981

[7] Bárány S:
Bevezetés a CDL2 nyelvi rendszer használatába.
SOFTTECH D 53, SZÁMKI, 1980

[8] Juhos A:
CDL2 nyelvi rendszer felhasználói kézikönyv
SOFTTECH D 54, SZÁMKI, 1980.

Szerzők:
Bárány Sándor-Langer Tamás
Számitógépalkalmazási Kutató Intézet
Budapest. I., Csalogány u. 30-32.

Bedő Árpád–Janni Éva

A MODULÁRIS PROGRAMOZÁS TÁMOGATÁSA A CDL2 NYELVI RENDSZERBEN

A cikkben bemutatjuk a modulok kettős szerepéből adódó ellentmondásokat: a moduláris programozás előnyeit és hátrányait. Bevezetünk egy új fogalmat: a programleirást.

Ezután bemutatjuk, hogy a CDL2 nyelvi rendszerben milyen programgenerálási technikákat alkalmazhatunk: különforditás, egybegenerálás, programrészek forditása.

Bemutatjuk, hogy a szerkezeti optimalizálás és az újfajta kódgenerálás együttes alkalmazása milyen új lehetőségeket ad: általános célú modulok specializálása, overlay-programok létrehozása.

Kulcsszavak: moduláris programozás, szerkezeti optimalizálás, programleirás, programvezérlés, overlay-programok, integrált programfejlesztő rendszer.

A moduláris programozás előnyei és hátrányai

A modulnak a hagyományos programozásban kettős szerepe van. Egyrészt logikailag egybetartozó, külön programozható részt testesit meg - azaz módszertani eszköz; másrészt a különforditás egysége - azaz technikai eszköz. Ha e két funkció szerinti felbontást külön-külön elvégezzük egy programra, általában nem ugyanahhoz a modulhalmazhoz jutunk. A CDL2 nyelvi rendszer az emlitett kettősségből fakadó ellentmondásokat igyekszik feloldani, és a moduláris forditási rendszereket lényegesen továbbfejleszteni; az ismert hátrányok csökkentésére, az előnyös tulajdonságok erősitésére törekszik. Legújszerűbb jellemzője, hogy egy adott modulcsaládból többféleképpen állitható elő futtatható program, illetve annak

szegmensei; igy a nyelvi rendszer használatával dinamikusan tudunk alkalmazkodni a különféle igényekhez.

A moduláris programozás egyik legfontosabb tulajdonsága a különfordítás; ez azt jelenti, hogy a modulokat önállóan programozzuk be, és a modulokból külön-külön generált kódot egy összeszerkesztő program /linkage-editor/ állitja össze futtatható programmá. Ennek a módszernek az előnyei általánosan ismertek:

- a modulok javitása, megirása egyszerübb
- a modulokat időben párhuzamosan is ki lehet fejleszteni
- olyan alapmodulokat lehet készitení, amelyek széles körben különböző tipusú feladatoknál is alkalmazhatók.

A jelenlegi, moduláris programozást támogató rendszereknek hátrányai is vannak:

- az egymástól függetlenül kifejlesztett modulok összeillesztése alacsony szintü összeszerkesztő program segitségével történik, igy a kapcsolódási hibák legjobb esetben is csak a késői, próbafuttatás előtti szakaszban derülnek ki /sőt sokszor csak futás közben/ - a nyelv fogalmaitól független hibaüzenetekből
- a több modulból álló programok rosszabb futási paramétereket tudnak csak elérni, mivel a modulok közötti hivások adminisztrációja sok esetben jelentős többletmunkát kiván
- nagyon nehéz a programokat úgy modulokra bontani, hogy a belőlük összerakott programok optimálisak legyenek: a modulok nem lehetnek túl nagyok, mert ezeknek valószinüleg csak kisebb részeit lehet egy adott programban felhasználni, a többiek feleslegesen terhelik a programot; ha viszont sok kicsi modult készitünk, akkor - bár felesleges részek nem kerülnek a programba - a modulok közötti hivások száma túlzottan felnőhet, és a modulok biztonságos kezelése is nehezebb.

A CDL2 nyelvi rendszer az emlitett hátrányokat egyrészt a modul-kapcsolatok állandó ellenőrzésével /lásd [1]/, másrészt a modulok programmá szervezésének, a futó kód előállitásának sokféle lehetőségével küszöböli ki. A továbbiakban ez utóbbi kérdéssel foglalkozunk részletesebben.

A program leirása

A CDL2 nyelvi rendszer lehetővé teszi mind a modulok különforditását, mind a több modulból álló program teljes egybefordi-tását csakúgy, mint a két határeset közötti egyéb megoldásokat. Ezeket a kódgenerálási technikákat a következőkben egy példán mutatjuk be.
Legyen egy képzelt forditóprogramunk, amelynek az első menetével foglalkozunk részletesebben. Az első menet feladata legyen a lexikai elemzés, a szintaktikus ellenőrzés és ezenkivül a program belső formájának /egy faszerkezetnek/ a létrehozása. Ezeknek a funkcióknak megfelelően szükségünk van a következő modulokra:

- lexikai elemző
- szintaktikus elemző
- fafelépitő.

Ezeknek a kiszolgálását /az alapvető aritmetikai müveletek és input-output tevékenységek definiálását/ végezze egy "alapok" nevü modul.

A CDL2 programokhoz ún. programleirás is tartozik, amely egyrészt megmondja,hogy az adott program milyen modulokból épül fel, másrészt megadja az ún. programvezérlést.

Példánkban a programleirás a következő lehet:

PROGRAM első menet.
PART alapok, lexikai elemző, fafelépitő, szintaktikus elemző.
PRELUDE lexikai elemző, fafelépitő.
ROOT szintaktikus elemző.
POSTLUDE fafelépitő.

A PROGRAM alapszó után a program nevét adjuk meg: a PART alapszót a programot alkotó modulok nevei követik. A PRELUDE-ben soroljuk fel azoknak a moduloknak a neveit, amelyekben előkészitő tevékenységek vannak /a lexikai elemzőben szokásos előkészités például az input file megnyitása és az első karakter beolvasása/. A program fő tevékenysége a "szintaktikus elemző"-ben indul /ez a program ROOT-ja/. A lezárási tevékenységeket tartalmazó modult adtuk meg a POSTLUDE-ban.

Különforditás

A programunk moduljait "külön fordithatjuk" a következő utasitássorozattal:
code MODULE alapok
code MODULE lexikai elemző
code MODULE fafelépitő
code MODULE szintaktikus elemző
code PROGRAM első menet.

Itt különforditáson modulonként való kódgenerálást értünk; a CDL2 nyelvi rendszerben bármilyen tipusú kódgenerálás az ún. elemzett formából történik, és csak a szintaktikus és szemantikus ellenőrzés után helyesnek talált modulokból, illetve programokból lehet kódot generálni. A különforditott moduljainkból olyan kód keletkezik, amelyben a modulhatárok még léteznek. A modulok közötti külső hivatkozásokat majd az összeszerkesztő program oldja fel . Az igy kapott modulszegmensek az összeszerkesztő program segitségével tetszőleges programhoz kapcsolhatók.

Egybefordítás

A programunkat egybefordithatjuk, ha a
code PROGRAM első menet
utasitást adjuk ki anélkül, hogy a programot alkotó modulokra előzőleg külön-külön "code" parancsot adtunk volna, vagy ha az előző "code" parancsok hatását
code nomore MODULE alapok
code nomore MODULE lexikai elemző
code nomore MODULE szintaktikus elemző
code nomore MODULE fafelépitő
parancsokkal megszüntettük a nyelvi rendszer számára.

Az ilyen tipusú teljes egybefordítás azt jelenti, hogy a programon belül megszünik minden modulhatár - ezt a szolgáltatást nevezzük magas szintü összeszerkesztésnek. Ebben az esetben egyetlen célkód-szegmens keletkezik, és a CDL2 fordítóprogram felhasználói előtt már jól ismert szerkezeti optimalizálás /lásd [2]/ az egész program optimalizálását jelenti: eltünnek a modulhatárok, az egész programból kimaradnak a felesleges részek, a tömöritést szolgáló helyettesitések az egész programban megtörténnek.

Programoknak a fenti módszerrel történő egybefordításával megvalósul az optimális moduláris programozás; a programszöveg elkészitése, javitása, ellenőrzése modulonként külön, jól kezelhetően, hatékonyan történik; az elkészült, hibátlan modulokból pedig olyan jó minőségü kód keletkezik, mintha a programot egyetlen részként irtuk volna meg. Ebben az esetben a modulok méretét is tetszés szerint választhatjuk meg; "túl" kicsi vagy "túl" nagy modulok irása sem befolyásolja a program végső minőségét.

Programrészek fordítása

Bizonyos esetekben szükségünk lehet több modul - de nem az

egész program - egybeforditására; ezt nevezzük programrészek forditásának. Példánknál maradva, a lexikai elemzőt "kiemelhetjük" a programból a következő utasitással:

code PROGRAM első menet PART lexikai elemző.

Ennek hatására kód készül a lexikai elemzőből és a program minden olyan részéből is, amelyik ebből a modulból elérhető /azaz amely ebből a modulból közvetve vagy közvetlenül meghivódik/ és amelyik részből eddig még nem generáltunk kódot /vagy "code nomore" paranccsal a kódolt állapotáról "elfelejtkeztünk"/. Programrészek egybeforditása például akkor hasznos, ha van már olyan stabil programrészünk, amelyben hosszabb ideig nem akarunk változtatni /a még munkában levő modulokat külön célszerü forditani/.

Ha például a következő utasitássorozatot adjuk ki:
code PROGRAM első menet PART lexikai elemző
code PROGRAM első menet PART szintaktikus elemző
code PROGRAM első menet PART fafelépitő
code PROGRAM első menet

akkor ennek hatására a teljesen különforditott esetnél jobb, de az egybeforditottnál rosszabb futási paraméterekkel rendelkező programot kapunk. Figyeljük meg, hogy az alapok nevü modul forditására már nem is volt szükség, mivel a három programrészbe bekerült minden olyan tevékenysége, amelyekre az egyes részeknek szükségük volt. Ilyen tipusú programrészgenerálásnál a programrész csak az adott programhoz kapcsolható.

A lexikai elemzőt kiemelhetjük ebből a programból úgy is, hogy másik program is felhasználhassa. Ehhez a
code separate PROGRAM első menet PART lexikai elemző
utasitást kell kiadnunk.

Megjegyezzük, hogy létezik a "code nomore" utasitással ellenkező hatású utasitás is. A "code later" utasitással azt mond-

hatjuk egy modulra, hogy bár most nem kivánunk belőle kódot generálni, de a rendszer tekintse úgy, mintha ez már megtörtént volna, azaz a programrészek generálásakor belőle ne szedje össze a szükséges részeket.

Általános célú modulok specializálása

Térjünk vissza a "túl" kicsi és a "túl" nagy modulok problémájára. A CDL2 nyelvi rendszert használva irhatunk külön modult minden olyan egységre, amelyet valamilyen szempontból logikailag önállónak érzünk, legyen az akármilyen kicsi, hiszen ezt bármely résszel egybegenerálhatjuk, igy semmi hatékonyságvesztés nem származik belőle. Bátran irhatunk nagy, általános modulokat is, például fogalmakat teljes általánosságban definiáló modulokat, hiszen a rendszer az ezeket felhasználó modulokhoz csak azokat a részeket generálja hozzá, amelyeket azok ténylegesen használnak.

Egy általános modult könnyen specializálhatunk explicit módon is. Csináljunk hozzá egy kis modult, amelynek semmi más feladata nincs, minthogy az általános modulból átveszi a speciális feladathoz szükséges algoritmusokat /CDL2 terminológiával: importálja őket/, és minden változtatás nélkül kiadja őket magából /exportálja őket/. Az ilyen specializáló modulokat nevezzük reexport moduloknak.

Például, ha van egy általános fakezelő modulunk, akkor az "első menet" nevü programunkban szereplő "fafelépitő" modul lehet ennek specializált változata, azaz a fafelépitő modul olyan reexport modul, amely csak az első menethez szükséges speciális tevékenységeket veszi át. Sőt, ha ezt a fafelépitőt még elég általánosnak találjuk, akkor a
code separate PROGRAM első menet PART fafelépitő
utasitással külön is generálhatjuk, amelynek hatására az általános fakezelő modulból kimetsződik a fafelépitő modul ál-

tal definiált rész, amely igy más program számára is felhasználhatóvá válik. Természetesen mindehhez a programleirásban a fakezelő modult is fel kell sorolni részként /PART-ként/.

Overlay programok létrehozása

A reexport modul fogalmát és a CDL2 nyelvi rendszer lehetőségeit fel lehet használni félig automatikus overlay-ezésre. A nyelvi rendszer kérésre rajzos javaslatot ad, hogy a program részeit hogyan célszerü overlay ágakra bontani. A javaslat alapján meg lehet tervezni a program overlay-ezhető részekre való bontását. Az egy overlay ághoz tartozó programrészeket reexport modullal gyüjthetjük össze. /Legyen egy ilyen modul neve "ovl"./ További szolgáltatásként, ha ebből a reexport modulból mint programrészből kódot generálunk a
code overlay PROGRAM ... PART ovl
utasitással, akkor nemcsak összegyüjti ezt az overlay ágat, de a programrészből kivezető, tehát már kódolt programrészbe mutató hivások elé generál egy "overlay ovl" nevü eljáráshivást, amely a megfelelő ág behozatalát elvégzi. Ilyen nevü eljárásról azonban nekünk kell gondoskodnunk az adott gép overlay lehetőségeinek az ismeretében.

Összefoglalás

A CDL2 nyelvi rendszer példa egy nagy, CDL2 nyelven irt moduláris programra. Ez a program kb. 7o modulból áll, amelyek mindegyikének mérete 2oo és 25oo sor között van.

A nyelvi rendszer első változatát a CDL2 forditóprogram által különforditott modulokból hozták létre, igy a program mérete kb. 1,2 Mbyte lett. Amikor az összetartozó modulokat programrészekbe gyüjtötték össze, a futtatáshoz szükséges memóriaméret kb. 7oo Kbyte-ra csökkent.

A nyelvi rendszer félautomatikus overlay-támogató szolgáltatá-

sát felhasználva létrehozták a CDL2 nyelvi rendszernek egy újabb változatát. Ez 25 overlay-ágból áll, amelyek mindegyike kisebb 5o Kbyte-nál, az állandó rész 7o Kbyte.

Az emlitett adatok azt mutatják, hogy a cikkben ismertetett kódgeneráló módszerek alkalmazása komoly gyakorlati eredményeket hoz. A CDL2 nyelvi rendszer a kisgépekre való szoftverfejlesztés fontos eszköze lehet.

Abstract

In this paper we show the contradictions arising from the two-faced role of modular programming: the pro's and contra's of modular programming.

Then we demonstrate the different program-coding techniques of the CDL2 Laboratory: separate compilation, program integration, coding of parts of programs.

We state the new possibilities coming from the mixed usage of the structural optimization and this new type of code generation: specialization of general purpose modules, overlay program-generation.

Irodalomjegyzék

[1] Bárány S.-Langer T:
Korszerü rendszerprogramozás és csoportmunka a CDL2 nyelvi rendszerben
Programozási Rendszerek '81 Konferencia

[2] Bedő Á:
A struktúrált programozás eszközei
I. Neumann János Kongresszus előadásai
Szeged, 1979

[3] Bayer-Böhringer-Dehottay-Feuerhahn-Jasper-Koster-Schmiedecke:
Software development in the CDL-Laboratory
in: Software Engineering Environments, Proceedings of a simposium in Lahnstein, June 1980
North Holland Publishing Co. 1981.

[4] Bárány S:
Bevezetés a CDL2 nyelvi rendszer használatába
SOFTTECH - D 53, SZÁMKI, 1980

[5] Juhos A:
CDL2 nyelvi rendszer felhasználói kézikönyv
SOFTTECH - D 54, SZÁMKI, 1980.

Csörnyei Zoltán

UNIVERZÁLIS MIKROPROCESSZOR

Az utóbbi években a mikroszámitógépek gyors fejlődésével a mikroszámitógépek software fejlesztésének hatékonysága rendkivül fontossá vált. Ebben az előadásban áttekintjük a mikroszámitógépek software fejlesztési lehetőségét /a cross-fejlesztő rendszereket és a gyártók fejlesztő rendszereit/, és megadunk egy egyszerü és gyors cross-assembler generálási módszert. Ezekben az univerzális cross-assemblerekben egy közös rész kezeli a processzor-független funkciókat, és a mikroprocesszorok utasitáskészlete egy-egy interpretativ leirásu adat file, amelyet a gyártók adatlapjai alapján könnyen el lehet készitení.

Kulcsszavak: assembler, cross-assembler, fejlesztő rendszerek, interpretativ leirás, makroprocesszor.

1. Bevezetés

A mikroszámitógép gyártás területén a következő tendenciák figyelhetők meg:

- a mikroszámitógépek /relativ/ mérete egyre kisebb lesz,
- a mikroszámitógépek ára csökken,
- az uj mikroszámitógép tipusok megjelenésének gyakorisága nagy.

A mikroszámitógépes software készitésére különösen a harmadik tendencia van hatással. A mikroszámitógépek felhasználóinak arra kell törekedniük, hogy az alkalmazásokhoz tartozó software-t minél gyorsabban létrehozzák. A software kidolgozásának meg kell előznie a processzor elavulását ahhoz, hogy a mikroszámitógépet komponensként tartalmazó termék korszerü legyen. Ezenkivül, a forgalomban lévő mikroszámitógépek nagy tipusszáma és a forgalomba kerülő uj tipusok miatt arra kell törekedni, hogy a mikroszámitógép tipusának megváltoztatása a software fejlesztő programoknak csak kismértékü megváltoztatását jelentse. Ezekből látható, hogy a mikroprocesszort tartalmazó termékek létrehozásának döntő feltétele lett a mikroprocesszoros software készitésének hatékonysága.

A mikroszámitógépes software fejlesztésére jelenleg a következő két módszert alkalmazzák:

- a programfejlesztés egy megfelelő konfigurációval körülvett mikroprocesszorral is megoldható. Ezeket a rendszereket mikroprocesszoros /prototipus/ fejlesztő rendszereknek nevezik. A programfejlesztést operációs rendszer támogatja, magasszintü nyelvek is rendelkezésre állnak.

- a mikroprocesszorok tömeges megjelenésének idejére a hagyományos számitógépek, elsősorban a miniszámitógépek használata általánossá vált. Ezek a számitógépek jelentős software bázissal rendelkeztek, mind a már létrehozott programokat, mind a programozói kapacitást tekintve. Célszerünek látszott a mikroszámitógépek programozására ezeket a számitógépeket és meglévő software hátterüket felhasználni. Igy alakultak ki a cross-fejlesztő rendszerek, amelyek szövegszerkesztő, cross-fordító és szimulátor prog-

ramokat tartalmaznak.

Mikroszámitógép tipusváltás vagy egy uj mikroprocesszoros fejlesztő rendszer megvásárlását, vagy a cross-fejlesztő rendszer programjainak ujrairását jelenti. A forgalomban levő mikroszámitógépek nagy tipusszáma, és elsősorban a fenti harmadik tendencia miatt, egy állandó termék darabszámot feltételezve, a mikroszámitógépet alkatrészként tartalmazó termékek árában a software fejlesztés és a fejlesztő rendszerek költsége nő. Ez a költség jelentősen csökkenthető olyan cross-fejlesztő rendszerek felhasználásával, amelyekben a tipusváltás nem teszi szükségessé a teljes rendszer ujrairását, azaz, ha a fejlesztő rendszer adaptálható programokból áll.

2. Univerzális, generálható programok

Minden program egy hardware és software környezetben müködik. Ezt a környezetet a program tárgykörnyezetének nevezzük. Egy program portábilis a tárgykörnyezetek egy halmazán, ha a programnak a halmazhoz tartozó tárgykörnyezetekben való futtatásához szükséges átalakitási költsége lényegesen kisebb, mint az ujrairási költsége. A portabilitás elérésének szokásos módszere az, hogy a fejlesztő rendszerek programjait magasszintü nyelven irják meg.

Minden programnak van tárgykörnyezete, a fordítóprogramoknak célkörnyezetük is van: az a környezet, amelyben a fordítóprogram outputja müködik. Azt mondjuk, hogy egy fordítóprogram adaptábilis a célkörnyezetek egy halmazán, ha a fordítóprogram a halmazhoz tartozó célkörnyezethez könnyen átalakitható, azaz a módositás költsége sokkal kevesebb, mint a célkörnyezethez való ujrairási költsége.

Az adaptábilis programokat univerzális programoknak, a célkörnyezethez való átalakitást generálásnak nevezzük.

A továbbiakban az assemblerek adaptábilitását vizsgáljuk, áttekintve a generálási módszereket is.

2.1. Az univerzális cross-assemblerek strukturája

Megfigyelhető, hogy minden assembler nagy része, kb. 80%-a a tárgy- és célkörnyezettől független. Az utasitásneveknek, a szimbólumok táblázatának a kezelése, az operandusban levő kifejezések feldolgozása, az assembler meneteinek szervezése, a hibakezelés mind ebbe a részbe tartozik. A megmaradó 20%-ból kb. 15% a célkörnyezettől függő rész, ide tartozik például az utasitásnevek, a direktivák táblázata, a cimzési módok leirása, a gépi kódok összeállitási szabályainak leirása. Az assemblernek csak kb. 5%-a függ a tárgykörnyezettől, ebbe a részbe tartoznak például az input/outputot végző programrészek.

Nevezzük az assembler célkörnyezettől független részét az assembler törzsének. Mivel minden assembler törzse azonos funkciókat végez, egy univerzális cross-assembler létrehozható ugy, hogy az assembler törzsét változtatlanul hagyjuk, csak a célkörnyezet leirásokat cseréljük. A célkörnyezet leirásának módszereit vizsgáljuk a következő részben, majd utána az assembler törzsének leirásával foglalkozunk.

2.1.1. Makroassembleres leirás

Olyan univerzális assemblerekkel, amelyekben a célkörnyezetet makrókkal irták le, már az 1960-as évek elején foglalkoztak.

Az 1970-es évek elején az univerzális programok létrehozásával kapcsolatban szinte ujból felfedezték a makrókkal való assembler leirás módszereit. A makrókkal leirt assembler feldolgozásához makroprocesszor kell, azonban kevés számitógép rendelkezik ilyen általános célu programmal. Viszont a számitógépek többségének van makroassemblere, és ez is jól használható univerzális cross-assemblerek létrehozására.

A tárgyszámitógép makroassemblere univerzális cross-assemblernek is tekinthető, ahol a célkörnyezet leirása a makrodefiniciók halmaza, az assembler törzse pedig a makroassembler. A módszer nagy előnye az, hogy a makroassemblert változtatás nélkül lehet használni, az univerzális cross-assembler törzse egy már kész, jól müködő program.

2.1.2. Táblázatos és interpretativ leirás

A különböző számitógépeken müködő assemblerek többsége táblázatos megadásu, azaz a célkörnyezet leirását táblázatok tartalmazzák. Az assemblerek generálása ezeknek a táblázatoknak, és a hozzájuk tartozó szubrutinoknak a cseréjét jelenti. A cserére nincs is minden esetben szükség, mivel a mikroszámitógép gyártók uj processzoraik utasitáskódjait ugy definiálják, hogy az a régiek bővitése legyen. Bár az ugyanazon funkciót megvalósitó utasitások gépi kódja különböző tipusu processzoraikban különbözik, az utasitások mnemonikja azonos. Ebben az esetben a mikroszámitógép assemblerének generálása csupán a már meglévő mnemoniktábla bővitését, és az utasitások gépi kódjának megfelelő megváltoztatását jelenti.

A továbbiakban azt vizsgáljuk, hogy hogyan lehet a generáláskor cserélendő adathalmaz méretét csökkenteni, azaz a táblázatokból, szubrutinokból mi vihető át az univerzális cross-assembler törzsébe.

Az utasitások feldolgozását bontsuk elemi lépésekre, modulokra. Észrevehető, hogy a különböző utasitások feldolgozásakor sok azonos funkcióju modult kell végrehajtani, azaz a gépi kód előállitásához szükséges különböző modulok darabszáma kicsi. A különböző értékkel azonos müveletet végző modulokat tekintsük egy paraméterekkel ellátott modulnak, igy a darabszám még csökkenthető is. Ezután a modulokból válasszuk le a célszámitógéptől függő adatokat. A modul csak az elvégzendő müveletek algoritmusát tartalmazza formális paraméterekkel, a célszámitógéptől függő adatok legyenek a modulok aktuális paraméterei.

Legyenek a modulok a tárgyszámitógép gépikódu szubrutinjai, az assembly nyelvü utasitások feldolgozása igy szubrutinhivások és paraméterek sorozatával, és a szubrutinok programjaival irhatók le. Mivel a szubrutinok áthelyezhetők az assembler törzsébe, ezzel a módszerrel a célkörnyezet leirása egyszerübb lesz, mint a fenti táblázatos megadással.

Egy többmenetes assembler mindegyik menetében használhatja ugyanazt a leirást, ha a különböző menetekben elvégzendő funkciót a menetszám függvényében a szubrutinok választják ki. Ezt megvalósitva elérhető az, hogy a leirások módositása vagy cseréje, azaz a generálás során csak egy adathalmazra kell koncentrálni, ami a változtatással jelentkező programhibák számát csökkenti.

A leirt módszerrel az univerzális cross-assemblernek a 2.1. pontban emlitett strukturája létrehozható: a célkörnyezet leirása és az assembler törzse szétválasztható, a célkörnyezet leirása könnyen elvégezhető. Egy jól megválasztott szubrutinkészlettel ugyanazzal a törzzsel egy mikroszámitógép gyártó különböző processzorainak assemblerei generálhatók. Ha egy másik mikroszámitógép gyártó által a processzorához definiált

assembly nyelv lényegesen különbözik az előbbitől, előfordulhat, hogy a szubrutinkészletet is át kell alakitani. Ilyenkor csak az uj szubrutinok megirása jelent plusz feladatot, a szubrutinkészlet átalakitása törléssel, cserével vagy bővitésssel egyszerüen megoldható.

A célszámitógép leirásának mérete még tovább csökkenthető. Ha a modulok darabszáma egy byte-ban ábrázolható és a szubrutin kezdőcimek legalább két byte-ot foglalnak el, lássuk el a modulokat sorszámmal, és a szubrutinok helyére irjuk ezeket a sorszámokat. Mivel a leirásokban a szubrutincimekre való hivatkozások száma nagy, ezzel a módszerrel a leirások helyfoglalása lényegesen csökkenthető. Az összes memória megtakaritás mértéke ennél kisebb, mivel a módszer alkalmazásakor kell egy olyan, eddig nem szereplő program, amely a sorszámokból a tényleges szubrutin kezdőcimeket előállitja.

A szubrutin sorszámokból a szubrutin kezdőcimeket és a szubrutin végrehajtását inditó programot interpreternek nevezzük. Az interpreter által értelmezett program a feladat interpretativ leirása, amely tehát a szubrutinokat reprezentáló kódokból és a paraméterekből áll.

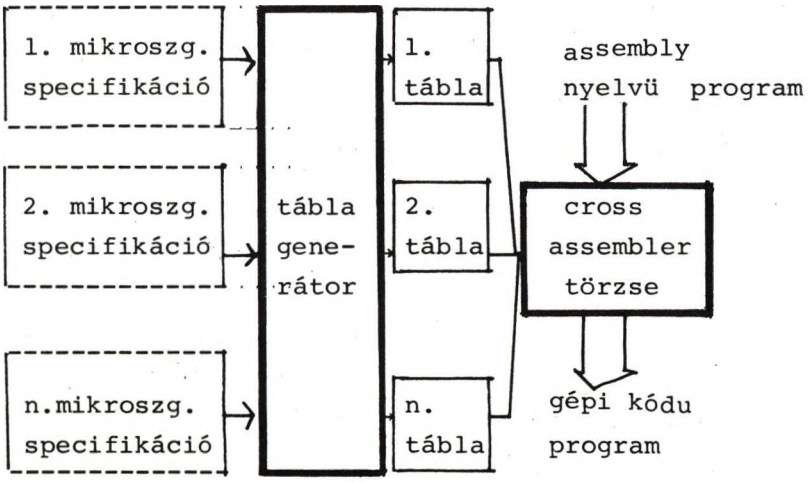
A fenti módszerrel az univerzális cross-assembler célkörnyezetet leiró táblái, azaz a mnemonikok és pszeudo utasitások táblázatai ugy alakithatók át, hogy az utasitást tartalmazó assembly nyelvü sort feldolgozó program interpretativ leirása az utasitás neve után álljon.

A leiró táblák létrehozásának módszereit vizsgáljuk a következőkben. Egy leiró táblába kerülő információ megadható vagy egy speciális, assembler leiró

nyelven, vagy a tárgyszámitógép valamelyik nyelvén. Az assembler leiró nyelvek forditóprogramjait, azaz a táblagenerátor programokat vagy az univerzális cross--assembler részeként, vagy egy külön programként használják. Az első esetben az univerzális assembler két üzemmódban dolgozik: az assembler épitő és a programfordító üzemmódban. A külön programként használt táblagenerátor a táblázatot egy különálló file-ként tárolja, és az assemblálás előtt kell a megfelelő táblázatot az univerzális cross-assemblerhez kapcsolni. Mindkét eljárás logikai felépitését a 2.1. ábrán láthatjuk.

A legegyszerübb módszer az assemblerek mnemonik és pszeudo utasitás táblázatainak létrehozására az, amikor a táblázatokban szereplő adatokat a tárgyszámitógép assembly nyelvén irják le. Az adatok adatelhelyező utasitásokkal adhatók meg. Igy egy tábla generálása csupán egy assembly nyelvü program megirásából és leforditásából áll, a táblagenerátor ugyanaz az assembler, mint amellyel a cross-assembler törzsét kell forditani.

Ez a módszer különösen egyszerüvé teszi az interpretativ leirásu táblák létrehozását. A szubrutin sorszámok megadása szimbólikusan



2.1. ábra

történhet, egy értékadó utasitással a szimbólikus szubrutinnév és a szubrutin sorszám összerendelhető. A paraméterek megadásánál jól használhatók a tárgyszámitógép assembly nyelvének számformátumai bit, byte vagy szó konstans megadására, a változókat, azonositókat tartalmazó kifejezések használata pedig leegyszerüsiti a paraméterek értékének definiálását.

A táblázatos leirás fenti módszere nem csak assembly, hanem magasabb szintü nyelvek leirására is alkalmas, ezáltal az univerzális cross-assemblerrel magasabb szintü nyelvek is fordithatók.

3. Az assembler törzse

Egy rövid áttekintést adunk az assemblálási folyamatról, megvizsgálva az assemblerek gépfüggetlen tulajdonságait. Leirjuk azokat a funkciókat, amelyeket minden cross-assembler törzsének meg kell valósitania.

Az assembler feladata az, hogy a programok assembly nyelvü formájából előállitsa a program gépi kódu formáját. Ha B a lehetséges nem-üres bitsorozatok halmaza, és E az üres bitsorozat, akkor az L nyelv assemblere a

$$L \longrightarrow B \cup \{E\}$$

leképezést valósitja meg. Ehhez fel kell ismernie az assembly nyelvü programban a szintaktikus egységeket, azaz a szimbólumokat, meg kell határoznia a szimbólumok bináris megfelelőjét, és ezekből kell összeállitania a gépi kódu programot.

Az assembler a következő alapvető funkciókat végzi el:

- értéket ad a szimbólumoknak,
- kiszámitja a kifejezések értékét,
- ezekből az értékekből meghatározza a gépi kódu programot.

3.1. A szimbólumok értékének meghatározása

Az assembly nyelvü programban három különböző tipusu szimbólumot különböztetünk meg: azonositó, konstans, és operátor szimbólumot. Egy szimbólum értékét a szimbólumtól függően a következő módszerek egyikével határozza meg:

- a szimbólumnak nem ad értéket /pl. pszeudo utasitás, operátor szimbólum /,
- a szimbólum értéke független a leforditandó assembly nyelvü programtól /pl. konstans szimbólum, mnemonik/,
- a szimbólum értékét az assembler definiálja a

a szimbólum előfordulási helye alapján /pl. cimkék/, vagy az assembly nyelvü programban levő értékadó utasitás alapján /pl. SET, EQU direktivával meghatározott szimbólumok/.

Azokat az azonositó szimbólumokat, amelyeknek az értékét értékadó pszeudo utasitás határozza meg, paramétereknek nevezzük. A paraméterek értéke egy programon belül is változhat. Van egy kitüntetett paraméter, az elhelyezés számláló, amelynek az értékét nemcsak az assembly nyelvü programban lehet megadni, hanem az assembler is megváltoztatja az utasitások leforditása után. Az azonositó szimbólumok között külön csoportot alkotnak a cimkék, egy cimke az utána következő utasitás elhelyezési cimét reprezentálja.

3.2. A kifejezések értékének meghatározása

Ha a kifejezésben szereplő azonositó szimbólumok értéke ismert, a kifejezés értékének meghatározása nem bonyolult feladat, mivel az operátor szimbólumok általában csak a szokásos aritmetikai vagy vagy logikai müveleteket irják elő.

Problémát az okoz, ha a kifejezésben levő szimbólum értéke még nem definiált, ez is elsősorban akkor, ha a kifejezés egy paraméter értékét meghatározó utasitás operandusában szerepel. Ebben az esetben a nem-definiált értékü szimbólumok darabszáma is növekszik. Az ilyen kifejezések és paraméterek értékét az assembler csak több menetben tudja meghatározni.

Legyen a feldolgozáskor ismert értékü paraméter postdefinitási foka $\emptyset$. Ha egy p paraméter az i-ik sorban a cimkezónában szerepel, akkor itt a pd/p,i/1//-vel jelölt postdefinitási foka legyen az operandusmezőben levő szimbólumok postdefinitási fokának maximuma. Ha ez a p paraméter a k-ik sor operandusmezőjében szerepel, akkor itt a postdefinitási foka legyen pd/p,k/3// = pd/p,i/1//, ha $k > i$, és

pd/p,k/3// = pd/p,i/1// +1, ha k $\leq$ i. Az Assemblálás Alaptétele [4] kimondja, hogy ha egy paraméter post-definitási foka j<∞, akkor a paraméter értéke a j+1-ik menetben határozható meg.

3.3. A gépikódu forma

Az assembler az assembly nyelvü program gépi kódu alakját a szimbólumok, kifejezések értékeiből állitja össze. A gépi kódu utasitások strukturája a számitógép-hardware által teljesen meghatározott, ezért az assembler számára a gépi kódu utasitások összeállitása jól definiált feladat, a gépi kódu utasitás az utasitáskódtól és az operandusoktól függ.

4. Hatékonyság

Egy táblázatos leirásu R10-es univerzális cross-assemblerrel két mikroprocesszor, az Intel 8008 és az Intel 8080 assemblerét adaptáltuk. Az elsővel együtt készült el az assembler törzse, ezután az Intel 8080 assemblerének elkészitése már kevesebb, mint egy munkanapot vett igénybe, az Intel Corp. kézikönyve alapján a munka könnyen elvégezhető, "gépies" volt.

Összehasonlitásul az univerzális cross-assemblerekről közlünk néhány publikált adatot:

publikáció	módszer	adaptált mikroprocesszorok száma	egy processzor adaptálási ideje
1.	táblázat	8	2 nap - 2 hét
2.	makró	4	2 nap
5	makró	-	1 nap
6	táblázat	22	1/60 - 1/100-a hagyományos módszernek
7	táblázat	21	néhány nap
8	táblázat	7	3 nap - 1 hét
9	táblázat	6	< 8 óra
10	makró	5	8 óra
11	táblázat	-	1 nap

Abstract

During the past few years microcomputers have evolved rapidly and the efficiency of development software had became an important issue. This paper describes the microcomputer software development facilities / cross--development systems and manufacturer's development systems/, and gives a method for simple and rapid generation of cross-assemblers. The main part of the universal cross--assembler manages the processor-independent tasks, while the instruction set for a paticular microprocessor is an interpretiv data file, which can be prepared from the manufacturer's data sheets with little manpower.

Irodalomjegyzék

[1] Caluwaerts, L.J., Peperstraete, J.A.: Universal cross-assembler for microcomputers. MIMI-77, Zürich, 1977. pp. 35-38.

[2] Cohen, H.A., Francis, R.S.: Macro-assemblers and macro-based languages in microprocessor software development. Computer, Vol.12, No.2, Feb.1979. pp.53-64.

[3] Csörnyei Z.: Egy módszer assemblerek előállitására. MTA SZTAKI Közlemények, 1978/19. 69-78. old.

[4] Csörnyei Z.: Univerzális mikroprocesszor assemblerek. Doktori értekezés. Budapest, 1980.

[5] Ferguson, D.E.: Evolution of the meta-assembly program. CACM Vol. 9, No.3, March 1966.

[6] Fujita, S., Yoshida, K.: A general purpose cross--assembler for microprocessors. Euromicro Newsletters Vol.3, No.4, Oct. 1977. pp. 88-89.

[7] Johnson, G.R., Mueller, R.A.: Automated generation of cross-system software for microcomputers. Computer, Vol.10, No.1, Jan. 1977. pp. 23-31.

[8] Johnson, G.R., Mueller, R.A.: A generator for microprocessor assemblers and simulators. Proc. of the IEEE, Vol. 64, No.6, June, 1976. pp. 921-931.

[9] Malcolm, M,A., Sager, G.R., Stafford, G.J.: A portable assembler writing kit. MIMI-76, Toronto, 1976. pp. 78-81.

[10] Seim,T.A.: Assembling microprocessor software with minicomputers. Symposium on Trends and Applications, Micro and Mini Systems. Gaithersburg, Maryland, USA. pp. 94-102.

[11] Taeymans, J., Tiberghien, J.: DASS8 - a cross assembler suitable for most of the 8 bit microcomputers. Euromicro Newsletters, Vol. 3, No.2, Apr. 1977. pp.80-83.

dr. Csörnyei Zoltán
tudományos munkatárs
ELTE TTK Numerikus és Gépi Matematika Tanszék
Budapest, VIII., Muzeum krt. 6-8. 1 0 8 8

Dettrich Árpád–Bánkfalvi Judit–Orosz Judit

COBOL ELJÁRÁS LOGIKAI KIFEJEZÉS INTERPRETÁLÁSÁRA

A COBOL nyelven programozók gyakorlatában igen gyakran felmerülő probléma az input fájl rekordjainak logikai ellenőrzése. Ez a tevékenység lényegében egy logikai kifejezés kiszámitása. A kifejezés operandusai az ellenőrizendő rekord egyes mezőire vonatkozó relációk.

Az előadásban bemutatunk egy általános - és egyben optimális - COBOL alprogramot, amely egy vektorban elhelyezett logikai kifejezés jelsorozatát elemzi, és egyidejüleg az értékét is kiszámitja.

A megvalósitásban nagy gondot forditottunk a tervezés módszerére, melynek alapjául a CDL2 nyelvet választottuk. Igy az algoritmus már a funkcionális leirás szintjén is tesztelhető.
Előadásunkban a feladatmegoldás teljes ciklusát bemutatjuk.

Kulcsszavak: COBOL, CDL2, logikai kifejezés, optimalizálás, tervezési módszer, logikai terv, tesztelés.

A feladat meghatározása

Az alapvető probléma az volt, hogy igen nagyszámú input rekord logikai ellenőrzését kell elvégezni. A különböző szempontok szerint történő ellenőrzést különböző logikai kifejezésekkel végezték, amelyeket minden újabb feladatnál újra meg kellett tervezni és megvalósitani. Ezért elkészitettünk egy általános célú COBOL alprogramot, amely egy teljesen általános logikai kifejezés értékének kiszámitását végzi. A logikai kifejezést kiszámitó algoritmust úgy terveztük meg, hogy bemenő paraméterként kapja a logikai

kifejezést tartalmazó pufferra vonatkozó információkat, míg kimenő paramétere a kifejezés értéke /0 vagy 1/. /Ezen a szinten a vizsgálandó rekordokkal nem foglalkozunk./

Először a pufferben lévő kifejezés irásának szabályait /szintaxisát/ definiáljuk, majd arra épitve leirjuk az elemző és értelmező algoritmust. Az algoritmus leirásához a CDL2 nyelvet használjuk [STAHL'78] , [BEDŐ'79] . Az igy keletkezett "logikai tervet" letesztelve módszert mutatunk az algoritmus COBOL -ba való - csaknem automatikus - átirására. Ezek után néhány szóban elmondjuk a COBOL szintü program tesztelésének problémáit. Végezetül bemutatjuk, hogy az elkészitett alprogramot hogyan használtuk fel valós környezetben.

Funkcionális leirás

Szintaxis

Kifejezés értéke: kifejezés, pont.
kifejezés: tag, (c: vagy, tag, :c;) .
tag: tényező, (c: és, tényező, :c;) .
tényező: nem, elemi kifejezés;
elemi kifejezés.
elemi kifejezés: nyitózárójel, kifejezés, csukózárójel;
reláció.

A "pont", "vagy", "és", "nem", "nyitózárójel", és "csukózárójel" primitivek rendre: . , V, Λ, ¬, (,) jeleket takarják, míg a "reláció"-primitiv definicióját nyitva hagyjuk.

A CDL2 nyelv előnye, hogy a fenti szintaxis-leirás egyben a forditóprogram vázát is adja - vagy más szóval egy magasszintü rendszertervét - , amelyet lépésenként bővitve /finomitva/ jutunk el a forditóprogram végleges formájához.

Szemantika:

```
kifejezés értéke + eredmény>:
    kifejezés + eredmény, pont.
kifejezés + eredmény> - operandus:
    tag + eredmény, (c: vagy, tag + operandus,
    logikai összeadás + eredmény + operandus, *c;).
tag + eredmény> - operandus:
    tényező + eredmény, (c: és, tényező + operandus,
    logikai szorzás + eredmény + operandus, *c;).
tényező + eredmény>:
    nem, elemi kifejezés + eredmény, negálás + eredmény;
    elemi kifejezés + eredmény.
elemi kifejezés + eredmény>:
    nyitózárójel, kifejezés + eredmény, csukózárójel;
    reláció + eredmény.
```

Ahol a "logikai összeadás", "logikai szorzás", és "negálás" makrók.

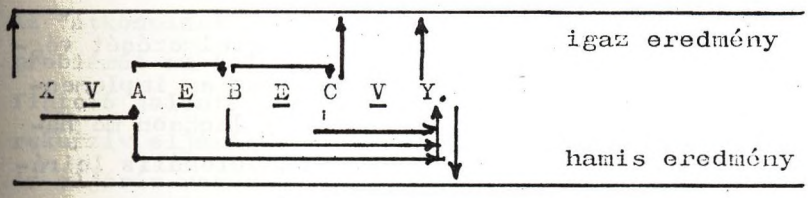
Optimalizálás

Az optimalizálás gondolatát néhány példával szemléltetjük. Az A V B V C diszjunktiv kifejezésben /logikai összeadás/ A,B és C képviselik a relációkat.

A kifejezés igaz, ha legalább egy reláció igaz. Tehát ha az elemzésben balról jobbra haladunk, akkor az első igaz relációnál befejezhetjük az értékelést, mert az eredmény a többi operandus értékétől függetlenül igaz.

Az A E B E C konjunktiv kifejezés /logikai szorzás/ értéke akkor igaz, ha valamennyi operandus igaz. Ezért balról jobbra haladva az értékelésben az első hamis értékü relációnál már tudjuk, hogy a további relációk értékétől függetlenül a kifejezés értéke hamis.

Vegyes kifejezésben - az operátorok precedenciája miatt - először a konjunkciót számoljuk, ezután a diszjunkciót. A következő ábrán a felül rajzolt nyilak az elemzés következő lépését vagy az eredményt mutatják, ha a kérdéses operandus igaz, az alul rajzolt nyilak az elemzés következő lépését, illetve az eredményt mutatják az operandus hamis értéke esetében.



A zárójelezés lényegében nem bonyolitja az elemzést, mert a zárójelben is egy kifejezés van, amelynek kiértékelése után a kapott eredményt használjuk fel a továbbiakban operandusként. A zárójelezés elvben bármilyen mélységben megengedett.
A fentiek figyelembevételével módositjuk a korábbi CDL2 leirás - rendszerterv - "kifejezés" és "tag" szabályait.

```
kifejezés + eredmény> - operandus:
   tag + eredmény,
   (c: vagy, ha igaz az + eredmény, a kifejezés végét keresi;
       vagy, tag # operandus,
       logikai összeadás + eredmény + operandus, *c ;).

tag + eredmény> - operandus:
   tényező + eredmény,
   (c: és, ha hamis az + eredmény, a konjunktiv tag végét
       keresi;
       és, tényező + operandus,
       logiaki szorzás + eredmény + operandus, *c ;).
```

A "ha igaz az " és "ha hamis az" predicate típusú eljárások; "a kifejezés végét keresi" eljárás a kifejezést határoló zárójelet (!) vagy pontot keresi a jelsorozatban; "a konjunktiv tag végét keresi" eljárás pedig vagy a kifejezés végét keresi, mint az előbbi, vagy ugyanazon a szinten az első "V" operátort.

A logikai terv tesztelése

Ezzel elkészitettük a logikai kifejezés értelmezését végző CDL2 program fő részét. Ez a programrész az implementálandó algoritmus rendszerterve - vagy Jackson meghatározását használva [JACKSON'75] - funkcionális leirása.

Annak előnye, hogy a funkcionális leirást CDL2 -ben adtuk meg elsősorban az, hogy a program már a tervezés szintjén is tesztelhető. A SIEMENS gépen a BS 2000 operációs rendszer alatt működik egy CDL2 rendszer [STAHL'78]. Ezen végeztük el a tesztelést. A teljes program három szekcióból épül fel. A 'SECTION' MACRO azokat a makrókat tartalmazza, amelyek az I/O tevékenységeket valamint a logikai műveleteket végzik. A tényleges kifejezés kiértékelés a harmadik szekcióban van /'SECTION' MEMBERSHIPS/, amely a második szekció /'SECTION' INTERFACE/ szabályain keresztül hivatkozik a makrókra.

A tesztelést olyan logikai kifejezésekkel végeztük, amelyekben a relációkat /operandusokat/ nulla vagy egy értékek megfelelő variációival helyettesitettük. Ezzel a módszerrel - természetesen - nem végezhettük el a teljes verifikálást, de a teszt-fájl összeállitásánál mind a kifejezések megválasztásával, mind az operandusok megfelelő variációival figyelembe vettük a lehetséges "kritikus" eseteket.

Implementálás COBOL nyelven

A CDL2 -ről a COBOL -ra való áttérés problémái

A funkcionális leirásból - tulajdonképpen további finomitással - könnyen eljuthatunk a COBOL nyelvü programig. /Ha valaki assembly nyelven akarja megvalósitani az algoritmust, akkor a COBOL nyelvü "leirást" is a rendszerterv egy magasabb szintjének tekintheti, amelynek tovább finomitásával eljuthat az assembly nyelv szintjére./

Az "átkódolást" csaknem automatikusan végezhetjük, egyetlen problémát az "elemi kifejezés" szabályban lévő rekurziv definició jelent. /Az ANSI COBOL -ban nincsen megengedve a rekurziv eljáráshivás./

Az eljárás rekurziv hivásának szervezéséhez két adatot kell vermelni. Az egyik adat a funkcionális leirásban szereplő "eredmény", a másik a szukcesszív leszállásnál a hivási pontok sorszáma, amelyekkel egy kapcsoló GO TO -n keresztül visszatérhetünk a hivási pont után következő programrészre /a megvalósitásban a K1, K2, K3, K4, VEGE pontokra/. Könnyü belátni, hogy az eredményt nem kell vermelni, elég egyetlen változó, mert az előző szintről visszaadott érték alapján a kiválasztott program-ág egyértelmüen rendelhető az igaz vagy hamis értékhez.

A COBOL program lényeges részének leirása

Erőforrás: /az előző szinten definiáltak mellett/

- PHOSSZ a logikai kifejezést tartalmazó puffer hossza;
- P a soros jel pointere, kezdőértékét is az eljárás állitja be;
- VEREM egyjegyü számok tárolására alkalmas vektor, a mérete egyben definiálja a skatulyázás mélységét is;
- TETO a verem tetejét mutatja, ahol az utoljára beirt kapcsoló van, kezdőértékét is az eljárás állitja be;
- KAPCSOLO egyjegyü szám az elágazás szervezéséhez.

Első lépésben megmutatjuk, hogyan lehet leképezni a funkcionális leirást COBOL nyelvre, majd azokat a szervezési problémákat elemezzük, amelyek a végső megoldáshoz vezettek.

```
KIFEJEZES-ERTEKE.
      PERFORM KEZDO-ERTEKEK.
      MOVE 5 TO KAPCSOLO  PERFORM VEREMBE.
KIFEJEZES.
      MOVE 1 TO KAPCSOLO  PERFORM VEREMBE  GO TO TAG.
K1.
      IF JEL = "V" AND EREDMENY = 1 THEN
         PERFORM KIFEJEZES-VEGE-FELD  GO TO VISSZA
      ELSE IF JEL = "V" THEN
         PERFORM KOV-JEL  GO TO KIFEJEZES
      ELSE GO TO VISSZA.
TAG.
      MOVE 2 TO KAPCSOLO  PERFORM VEREMBE  GO TO TENYEZO.
K2.
      IF JEL = "E" AND EREDMENY = 0 THEN
         PERFORM KONJUNKCIO-TAG-IGN  GO TO VISSZA
      ELSE IF JEL = "E" THEN
              PERFORM KOV-JEL  GO TO TAG
      ELSE GO TO VISSZA.
TENYEZO.
      IF JEL = "N" THEN
         MOVE 3 TO KAPCSOLO  PERFORM VEREMBE
         PERFORM KOV-JEL.
ELEMI-KIFEJEZES.
      IF JEL = " " THEN
         MOVE 4 TO KAPCSOLO  PERFORM VEREMBE
         PERFORM KOV-JEL  GO TO KIFEJEZES.
RELACIO.
      PERFORM RELACIO.
K3.
```

```
      IF EREDMENY = 0 THEN
         MOVE 1 TO EREDMENY
      ELSE MOVE 0 TO EREDMENY.
      GO TO VISSZA.
K4.
      IF JEL = " " THEN
         PERFORM MOV-JEL  GO TO VISSZA
      ELSE MOVE 1 TO HIBAKOD  GO TO KILEP.
VISSZA.
      MOVE VEREM  TETO   TO KAPCSOLO
      SUBTRACT 1 FROM TETO
      GO TO K1, K2, K3, K4, VEGE DEPENDING ON KAPCSOLO.
VEGE.
      IF NOT JEL = "." THEN
         MOVE 2 TO HIBAKOD  GO TO KILEP.
      IF P > HOSSZ THEN
         MOVE 3 TO HIBAKOD  GO TO KILEP.
```

Könnyü követni, hogy a MOVE ... TO KAPCSOLO PERFORM VEREMBE szervezéssel a következő szint hivását oldjuk meg, mig a visszatérés a "VISSZA" paragrafus kapcsoló GO TO utasitásán keresztül történik a funkcionális leirásban a zárójel után következő részre.

A tervezés következő szintje a hivatkozott paragrafusok kifejtése. A jelzett paragrafusnevekkel lefedett tevékenységek a programtervezés módszertana szempontjából különösebb érdekességet nem tartalmaznak, ezért bemutatásától eltekintünk. Ezeknek a paragrafusoknak a leirása a program végső megoldásában megtekinthető [DETTRICH'80].

Tesztelés

A funkcionális leirás alapján elkészült COBOL nyelvü programot először közösen megvizsgáltuk a formális megfeleltetések szempontjából. Itt elsősorban a tervezés és a megvalósitás szintje közötti kölcsönös és egyértelmü kapcsolatra ügyeltünk, mert az igy dokumentált program megkönnyiti a hibakeresést /pl. az olyan paragrafusnevek melyekre nincs

hivatkozás, de megfelelnek a funkcionális leirás egy-egy szintjét képviselő neveknek/.

A második lépésben a "száraz tesztelés" módszerével megvizsgáltuk a programnak azokat a részeit, amelyek a funkcionális leirásban a CDL2 miatt lefedve maradtak /elsősorban a vermelés szervezése/, illetve amelyek csak az implementáció szintjén kerültek kifejtésre. Itt egyetlen lényeges hibát találtunk: a puffer-pointer állitásába több helyen belebonyolódtunk, mert a szóközök "elnyelését" először a megfelelő szintaktikus kategóriák feldolgozásánál akartuk megoldani. Az elemzésből automatikusan következett, hogy ezt a tevékenységet a legalacsonyabb szintre kell helyezni, vagyis ahol a következő jel előállitása folyik /lásd KOV-JEL paragrafus/.

Az alprogram gépi teszteléséhez a tesztágyat a következő megfontolások alapján állitottuk össze.

A SIEMENS gépen a BS 2000 operációs rendszerben a COBOL programok teszteléséhez rendkivül hatékony interaktiv környezet áll rendelkezésre.

Emiatt a tesztelést végző paragrafus a következő feladatokat látta el:

1. Display -ről beolvasott egy jelsorozatot, amely egy jó vagy hibás logikai kifejezés volt.
2. Display -ről beolvasta az operandusok logikai értékeinek egy variációját.
3. Felhivta a kifejezés kiértékelését végző paragrafust.
4. Kiirta az eredményt vagy értelmezte az észlelt hibát.
5. Ismételt 2. -től, ha ugyanazt a kifejezést akartuk tesztelni más értékekkel.
6. Ismételt 1. -től, ha másik kifejezést akartunk tesztelni.

Ebből a szervezésből látható, hogy a tesztelésnek a technikáján változtatni kellett. Igy a logikai kifejezésben az operandusokat egy-egy sorszám képviselte, amelyek egy olyan hosszú bitvektor megfelelő elemére hivatkoztak, mint a ki-

fejezésben lévő különböző operandusok száma.

Valós környezet

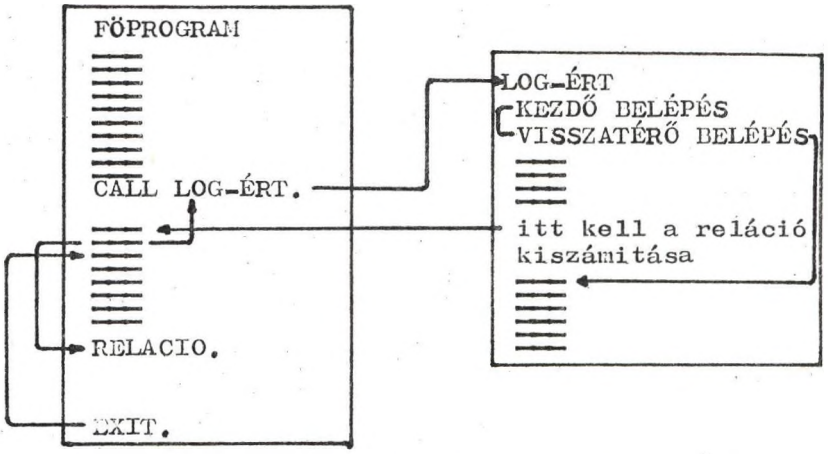
Az eddigiekben megmutattuk, hogy a specifikációban leirt feladatot hogyan lehet megtervezni és COBOL nyelven implementálni.

A tervezés és kódolás folyamán egyetlen dolgot hagytunk - szándékosan - figyelmen kivül: a logikai kifejezés operandusának, a relációnak kiértékelését. A logikai kifejezést kiszámitó alprogramot úgy készitettük el, hogy az csak a kifejezés értelmezésével foglalkozik, a reláció értékének kiszámitása tőle függetlenül történik. Ez a körülmény a következő szervezési problémát veti fel.

A hivó programban deklarált pufferben van a logikai kifejezést adó jelsorozat. A "LOG-ERT" alprogram paraméterként kezeli ezt a puffert, s mindaddig olvassa belőle a jeleket, amig relációhoz nem ér. Ekkor kellene felhivni a reláció kiszámitását végző alprogramot. Ezt az alprogramot azonban a hivó programnak kell aktivizálnia, mert csak ott lehet ismerni, hogy melyik tipusú reláció-számitási algoritmusra van szükség. Ezért a "LOG-ERT" visszaadja a vezérlést hivó programnak és ha az megoldotta a relácló kiszámitását, akkor újra felhivja a "LOG-ERT" alprogramot. Biztositani kell a "LOG-ERT" -ben, hogy a visszatérés után a megfelelő ponttól folytatódjon az elemzés. Amint látható, ez a megoldás egy normál korutin-szervezés azzal - az algoritmusból adódó - egyszerüsitéssel, hogy reláció kiszámitására csak egyetlen pontban van szükség, hiszen a rekurziv leszállásból adódó többszörös hivást a vermeléssel lekezeljük. Ez a szervezés megkivánja, hogy a "LOG-ERT" a hivó programot értesitse a visszatérés okáról:

- a logikai kifejezés értékét kiszámitotta,
- a következő reláció kiszámitását kéri,
- szintaktikus hiba van a logikai kifejezésben.

A "JELZO" és "HIBAKOD" paramétereken keresztül folyik a szervezéshez szükséges információcsere.
A következő ábra a hivó program /"FÖPROGRAM"/ és a logikai kifejezést számitó alprogram közötti funkcionális kapcsolatot szemlélteti.



Az eljárás felhivását a következő programrész szemlélteti:

```
LOGERT.
    ENTER LINKAGE.
    CALL LOG-ERT USING E LOGFOR JELZO HIBAKOD.
    ENTER COBOL.
    IF-NOT HIBAKOD = 0
       THEN GO TO HIBA1, HIBA2, HIBA3 DEPENDING ON HIBAKOD.
    IF NOT VEGE THEN  PERFORM  RELÁCIO GO TO LOGERT.
```

Az utolsó 2 paraméter jelentését már korábban megadtuk. Az "E" az eredményt tartalmazza, a "LOGFOR" pedig a pufferre vonatkozó információkat tartalmazó struktúra /csoportadat/.

Befejezés

Miután a programtervezés és megvalósitás problémáit bemutattuk, röviden szólunk az elkészült alprogram sorsáról és a felhasználása körül - a gyakorlatban - felmerült problémákról.

Az alprogram a Pénzügyminisztérium Számitástechnikai Intézetének SIEMENS gépén könyvtárazva megtalálható. Az alkalmazás során azt tapasztaltuk, hogy kis adattömegre irt ellenőrző feladatokban rendkivül jól használható, mert megkiméli a programozót az adatellenőrzési rész megirásától. Nagyobb adathalmazra azonban a program lassúnak bizonyult /többszázezer rekordból álló fájl esetén/, amelynek oka elsősorban az alprogramok közötti többszörös kapcsolásban rejlik. Ilyen esetben javasolhatjuk, hogy az adatfeldolgozást vágző főprogramba ezt az alprogramot, továbbá a relációt kiszámitó alprogramját is mint egy szekciót épitse be a programozó és akkor PERFORM utasitással hivatkozhat rájuk. Amennyiben a gyorsaság még igy sem megfelelő, akkor - mint azt már korábban jeleztük - a COBOL leirást egy magasszintü programtervnek tekintve elkészithetjük a logikai kifejezés értékét kiszámitó algoritmus assembly nyelvü implementációját.
A bemutatott módszertani megoldáshoz még annyit lehet hozzátenni, hogy ha a megvalósitás nyelvének olyan eljárásorientált nyelvet választunk, amelyben megengedett a rekurziv hivás is /pld. PL1/, akkor a CDL2 -ben megadott funkcionális leirásból az áttérés teljesen formálissá tehető.

ABSTRACT
The practice of COBOL programers often occurs the problem to check logically the records of an input file. It needs practically the evaluation of a logical expression. The operands of the expression are relations concerning the fields of the record which we want to check.
In the lecture we present a COBOL subprogram of a general and optimal algoritm. This algoritm recognises a logical expression represented by a string in a vector and its value is given.
A very important point of the realisation is the method of the system planning. We would like to demonstrate that

choosing the CDL2 language as a planning tool of the algoritm makes easy to test the method in the level of the functional description.
The complete life-cycle of the solution is presented in our lecture.
The verification of the functional description and the implementation of the COBOL subprogram is realised in the SIEMENS BS 2000 operating system.

I R O D A L O M J E G Y Z É K

[BEDŐ'79] Bedő Á. - Herényi I. - Langer T. - Szeredi P.: PROGRAMKÉSZITÉSI MÓDSZEREK, Közgazd. és Jogi Könyvkiadó, 1979.

[JACKSON'79] A.Jackson: PRINCIPLES OF PROGRAM DESIGN, Academic Press 1975.

[GRIES'71] D.Gries: Compiler Construction for Digital Computers. John Willey, 1971.

[DETTRICH'80] Dettrich Árpád - Orosz Judit COBOL eljárás logikai kifejezés interpretálására /SZÁMKI Tanulmányok 6./

[STAHL'78] Hans Michael Stahl CDL2 Towards SIEMENS BS 2000 ASSEMBLER under BS 2000 /Nymegen University, 1978/

Szerzők:

Bánkfalvi Judit	tudományos munkatárs	SZÁMKI
Dettrich Árpád	tudományos főmunkatárs	SZÁMKI
Orosz Judit	rendszerprogramozó	PSZTI

Dióslaki Ferenc

ÖSSZETETT ARITMETIKAI MŰVELETEK NAGYSEBESSÉGŰ SEJTPROGRAMJAI

Cikkünk néhány az aritmetikai számitások során előforduló alapvető feladat sejttérben való megoldását tárgyalja. Ezek a feladatok: bináris szorzás, X^n, n! számitás, vektorszorzás, polinom kiértékelés, könyvtári függvények közelitése, aritmetikai kifejezések kiértékelése. Bemutatjuk az ezen feladatok megoldására szolgáló sejtalgoritmusokat, és az őket végrehajtó sejtszerkezetek müködésének elvi lényegét. Az ismertetett megoldások többsége pipe-line, erősen párhuzamositott müködésü szerkezet. Bennük elsősorban az alkalmazott sejttér /időben és térben inhomogén 16 állapotu/ azon sajátosságait használtuk ki, amelyek lehetőséget adnak az alapmüveletek erősen bitpárhuzamos elvégzésére. Cikkünk végén konkrét példákon mutatjuk meg szerkezeteink számitási sebességének alakulását.

Kulcsszavak: aritmetikai müveletek, paralell algoritmusok, sejtalgoritmusok, pipe-line müveletvégzés.

1. Bevezetés

A sejtautomaták számitási rendszerekben való alkalmazása lehetőséget biztosit számitási feladatok igen széles körének erősen párhuzamositott végrehajtására. Cikkünk néhány gyakran előforduló aritmetikai feladatnak a Legendi Tamás által kidolgozott sejtprocesszor architektura [1] és sejtprogramozási metodológia [2],[3] keretein belül való megoldását tárgyalja. A bemutatott algoritmusok elsősorban a fenti architekturának az alapmüveletek átlapolására és bitpárhuzamos végrehajtására nyujtott lehetőségeit használja ki.

A jelen cikk célja, hogy bemutassa az emlitett feladatok megoldására szolgáló algoritmusokat és ismertesse az őket megvalósitó sejtszerkezetek paramétereit, valamint müködésük

elvi vázlatát.

Nem célunk az egyes helyeken előforduló - amüködés lényegét nem érintő - kizárólag technikai jellegü részletek ismertetése. Igy nem térünk ki az egyes átmenetfüggvények minden részletre kiterjedő preciz leirására, sem helyességük egzakt bizonyitására.

Az alkalmazott sejttér négy szomszédu /Neumann szomszédság/ tizenhat állapotu tér, amely a klasszikus sejtterektől több vonatkozásban eltér:

1; Térben inhomogén: ez azt jelenti, hogy a tér különböző sejtjei vagy sejtcsoportjai egy adott billenésen belül más-más átmenetfüggvény szerint billennek.
2; Időben inhomogén /ciklikus/: vagyis az egymást követő billenések során a sejtek /ciklikusan/ más-más átmenetfüggvényt valósitanak meg.

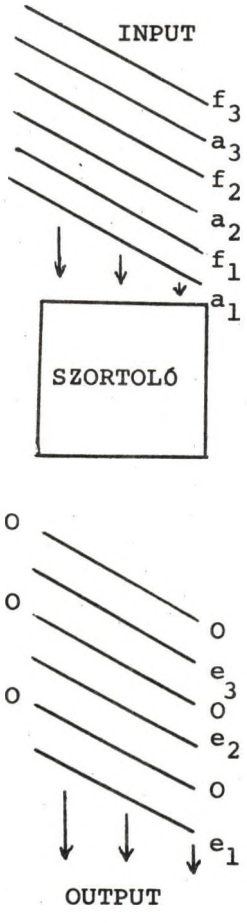
Az [1]-ben ismertetett sejtprocesszor architektura a klasszikus terek merev korlátait feloldva ezen két fontos tulajdonságával lényegesen kiterjeszti a processzor alkalmazhatósági körét és megkönnyiti a programozását.

Megjegyzés: Az alábbiakban használni fogjuk a bitcsatorna fogalmát, ez az itt használt erősen leegyszerésitett formában a következőket takarja: irjuk fel a tér sejtjeinek állapotát egy négy bites bináris szám formájában; i-edik bitcsatornán a sejtek igy felirt állapota i-edik bitjeinek összességét értjük.

2. Az összeadó-szorzó sejtszerkezet

A következőkben ismertetésre kerülő valamennyi sejtszerkezet egy igen gyors, erősen átlapolt és párhuzamos müködésre képes összeadó-szorzó sejtszerkezetre épül. Ennek a szorzónak a részletes leirása [7]-ben megtalálható, ezért az alábbiak-

ban csak input és output specifikációjának ismertetésére szoritkozunk.



1. ábra

Input: az 1. ábrán látható módón ferdén eltolt bináris számok, amelyek folyamatosan haladnak a szorzó irányába. /Billenésenként egyet lépnek lefelé./

f_i: a szám az egyes sejtek felső bitcsatornáján helyezkedik el.

a_i: a szám az egyes sejtek alsó bitcsatornáján helyezkedik el.

A helyiértékek jobbról balra növekednek.

Output:

Formátuma az inputéval megegyező a számok között egy $\emptyset$ sorral. /Az ábrán ezt O——O-val jelöltük./

$e_1=a_1$,

és minden $i\neq 1$-re:

$e_i=e_{i-1}f_{i-1}+a_i$

Helyigény: n^2 sejt, ahol n az eredmény megengedett hossza. Ha a számitások során ezt a hosszat valamely szorzat tullépi, tulcsordulásjel generálódik a négyzet bal alsó sejtjén.

Időigény: Az input fogadási sebessége billenésenként egy a_i vagy f_i adat, output: 2 billenésenként egy e_i. Egy a_i,f_i számpárhoz tartozó e_i eredmény az input első bitjének a szorzóba érkezésétől számitott 2n+2 billenés után alakul ki.

Először a fenti szerkezet néhány triviális alkalmazását mutatjuk be.

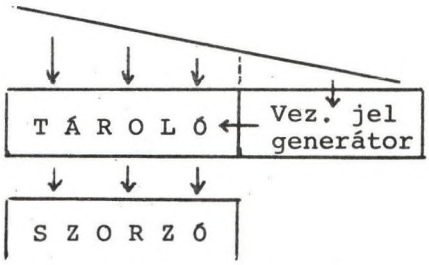
X^n kiszámitása:

X^n értékének kiszámitásához /ahol n természetes szám/ a szorzónak az alábbi inputot kell biztositani:

$a_1=1$, és minden $i\neq 1$ esetén $a_i=0$

$f_1=f_2=f_3=\ldots=f_n=X$

Ezt a következőképpen érjük el:



2. ábra

A szerkezet inputja /X és n/ az ábrán látható formában egymás mellett ferdén eltolva érkezik.

Müködése: Az input beérkezésekor $a_1=1$ generálódik a tároló jobbszélső sejtjén, x beiródik a tárolóba és n a vezérlőjel generátorba. A tároló X-et minden második lépésben outputra adja 2n billenés után a vezérlőjel generátortól kapott jel hatására a tároló tartalma törlődik. Igy a szorzó számára a fenti input áll elő. A szorzó ennek hatására a következő outputot szolgáltatja: $e_1=1$, $e_2=X$, $e_3=X^2$ x x x $e_n=X^n$

Időigény: Az input fogadásának a sebessége 2n+2 billenésenként egy-egy uj X és n érték. Az eredmény kialakulása az input X első bitjének az érkezésétől számitott $2(k+n)+2$ billenés, ahol k az alkalmazott szóhossz.

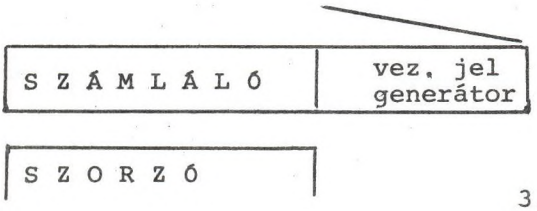
n! számitása

A feladat megoldásához ezuttal a következő inputot adjuk a szorzónak:

$a_1=1$, $a_i=0$ minden $i\neq 1$-re.
$f_i=i$ minden i-re.

Megvalósitása:

Az előző pontban szereplő tároló szerkezetet itt egy bináris számlálóra cseréltük fel:



3. ábra

Input: az n bináris szám, mely az ábrán látható módon fordén eltolve érkezik a vezérlőjel generátorba. Az előtte elhelyezkedő 1 jel kötelező /ez inditja a számlálást és a_1 képzését/.

Müködése:
Az input beérkezésekor n beiródik a vezérlőjel generátorba és a számláló jobb szélső sejtje $a_1=1$ outputot képez. A számláló [4] tartalma /amely induláskor 0/ két billenésenként 1-el növekszik és értéke mint f_i outputra adódik. 2n billenés után a vezérlőjel generátortól kapott jel hatására a számláló tartalma törlődik és a számlálás a következő input beérkezéséig szünetel.

Output: a szorzó outputjaként: $e_i=i!$ minden i-re. Egyéb paraméterei megegyeznek az X^n-et számitó szerkezetével.

Megemlitjük még, hogy a szerkezet használható két vektor skaláris szorzatának valamint polinomok értékének egy adott x helyen való kiszámitására is.

A vektorszorzásnál a szorzó az alábbi inputot kapja:

$a_{2i}=v_i$, $f_{2i}=u_i$ v_i és u_i a két vektor i-edik eleme

$a_{2i+1}=\emptyset$, $f_{2i+1}=\emptyset$ minden i-re.

Ekkor a számitás eredménye:

$$e_{2i}=v_i,\ e_{2i+1}=v_i u_i$$

$$e_{2i+2}=e_{2i+1}f_{2i+1}+a_{2i+2}=v_{i+1}$$

Ezután már csak az igy keletkezett szorzatokat /csak az e_{2i+1}-ként előálló eredményeket/ kell egy összeadóval összegezni.

A polinom kiértékelőnél az input az alábbi

$a_i=a_i$, $f_i=X$ minden i-re.

ahol a polinom Horner-elrendezés szerint felirt alakja: $(((a_1x+a_2)x+a_3)x+\ldots+a_{n-1})x+a_n$

A számitás eredménye e_i=a fenti i-edik zárójelpáron belül elhelyezkedő kifejezés értéke. A végeredményt e_n adja. A fenti néhány példa természetesen csak egy része annak az igen sok feladatnak, amelyeket egyszerüen a szorzó szerkezetünk inputjának célszerü megválasztásával megoldhatunk. A továbbiakban két olyan feladatot tárgyalunk amelyre megfelelően hatékony algoritmus kidolgozását a szorzó gyors müködése tette lehetővé.

3. Könyvtári függvények számitása

Igen sok számitás során szükség van bizonyos gyakran használt függvények /trigonometrikus, exponenciális stb/ értékének /közelitő/ kiszámitására. Célszerü erre a feladatra egy külön sejtszerkezetet alkalmazni, amely egy /rögzitett/ függvény értékét tetszőleges X helyen számitja ki. Az alábbiakban egy ilyen megoldást ismertetünk.

Az alkalmazott algoritmus lényege a következő: Tegyük fel, hogy n hosszuságu bináris számokat várunk a függvény argumentumaként. A számot osszuk két részre ugy, hogy az első /a magasabb helyiértékü/ k bitet, a másik n-k bitet tartalmazzon. Az első k bit lehetséges értékeihez tartozó függvényértékeket /a többi n-k bitet Ø-nak tekintve/ egy sejtprogrammal megvalósitott asszociativ memóriában [4] tároljuk. A számitás során az inputon érkező x felső k bitjéhez tartozó függvényértéket az asszociativ memóriából visszakeressük, az alsó n-k bit alapján pedig lineáris interpolációt végzünk. A számitás módja tehát:

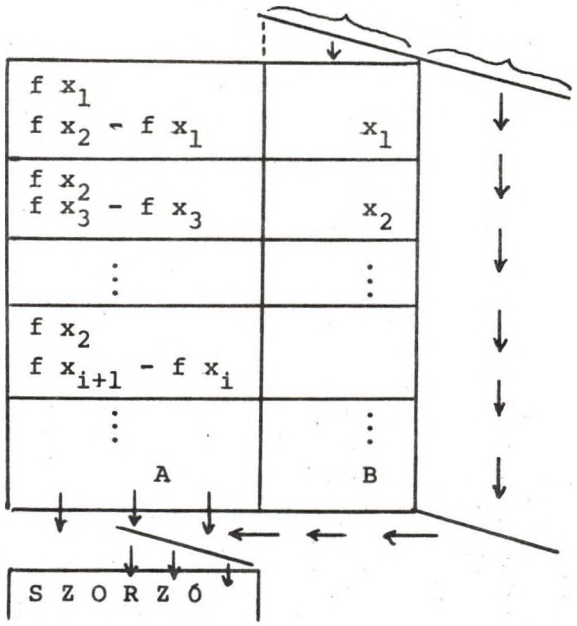
$$f(x) \approx f(x_i) + x'(f(x_{i+1}) - f(x_i))$$

ahol $f(x_i)$ a memóriából visszakeresett függvényérték,

x' az alsó n-k bit értéke

Az $f(x_{i+1}) - f(x_i)$-t pedig a szerkezet müködésének meggyorsitása érdekében célszerü szintén az asszociativ memóriában tárolni és $f(x_i)$-vel együtt visszakeresni.

A szerkezet elrendezése a következő:



4. ábra

A szerkezet müködésének megértéséhez szükséges az asszociativ memóriában lezajló tevékenységek vázlatos ismertetése: Az ábrán A-val és B-vel jelölt szerkezet különböző feladatokat lát el. A "B" részben tároljuk az alappontokat, egy sejtsorban egy x_i érték helyezkedik el. A "B"-n áthaladó input számot minden sejtsor összehasonlitja a benne tárolt x_i értékkel és ha egyenlőséget talál vezérlőjelet ad "A"-nak a vele egy sorban levő sejtsora felé. "A"-ban a függvényértékeket tároljuk, x_i-vel egy sorban a hozzá tartozó $f(x_i)$ és $f(x_{i+1})-f(x_i)$ értékeket. /lásd a 4. ábrát/ Az outputot "A"-nak az a sejtsora képezi amelybe a vezérlőjel érkezik.

A függvény számitó szerkezet inputja az ábrán jelölt módon pozicionálva és ferdén eltolva érkezik.

Müködése:

Az input első k bitjét az asszociativ memória "A" része összehasonlitja a tárolt alappontokkal és a megfelelő sorban vezérlőjelet ad "B" felé, aminek hatására az kiadja: először az inputhoz tartozó $f(x_i)$, majd két billenés mulva az $f(x_{i+1})-f(x_i)$ értéket. Ezalatt x', /az input alsó n-k bitje/ az ábrán jelölt uton halad és az asszociativ memória két outputja közé ékelődik.
Ez a három szám aztán a szorzó felé halad a következő formában:

$$a_1=f(x_{i+1})-f(x_i)$$
$$f_1=x'$$
$$a_2=f(x_i)$$

Ennek hatására a szorzó outputja: $e_2=(f(x_{i+1})-f(x_i))x'+f(x_i)$ tehát a függvény közelitő értéke az x helyen.

Helyigény: $2mn+2n^2$ ahol m: az alappontok száma
n: az alkalmazott szóhossz.

Időigény: Az input követési távolsága: 4 billenés, az inputhoz tartozó függvény érték az első bit beérkezésétől számitott 4n+m billenés mulva adódik outputra.

4. Aritmetikai kifejezések kiértékelése

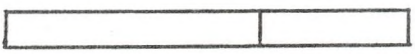
Az alábbiakban ismertetendő szerkezet feladata egyszerübb, *, + müveleti jeleket tartalmazó zárójel nélküli aritmetikai kifejezések kiértékelése /a számitások elvégzésekor a * müveletnek prioritása van/.

Input: A kifejezés formája megegyezik a szokásos matematikai irásmóddal, tehát váltakozva operanduszok és müveleti jelek sorozata:

$$op_1 \{{}^{+}_{*}\} op_2 \{{}^{+}_{*}\} op_3 \{{}^{+}_{*}\} \ldots \{{}^{+}_{*}\} op_4$$

Minden operandusz és müveleti jel egy-egy külön sorban /továbbiakban: input szó/ egymás felett helyezkedik el, ferdén eltolt formában, és a kifejezés felirásának sorrendjében billenésenként jutnak a szerkezetbe.

Egy input szó felépitése:



bennük az operanduszok és müveleti jelek elhelyezkedése a következő:

operandusz: a 0 - n biteken egész bináris szám, legkisebb helyiértékü jegye az n biten.
Az n+1, n+2 bitek mindig Ø-k.

müveleti jel: a 0 - n bitek mindig Ø-k
Az n+1, n+2 biteken *-nál 10. +-nál 01.

Az alkalmazott algoritmus lényege a következő: A kifejezés először egy szorzó szerkezeten halad keresztül, ahol:

1/ a * müveletek elvégződnek és eredményük egy vezérlőjellel ellátva outputra adódik,

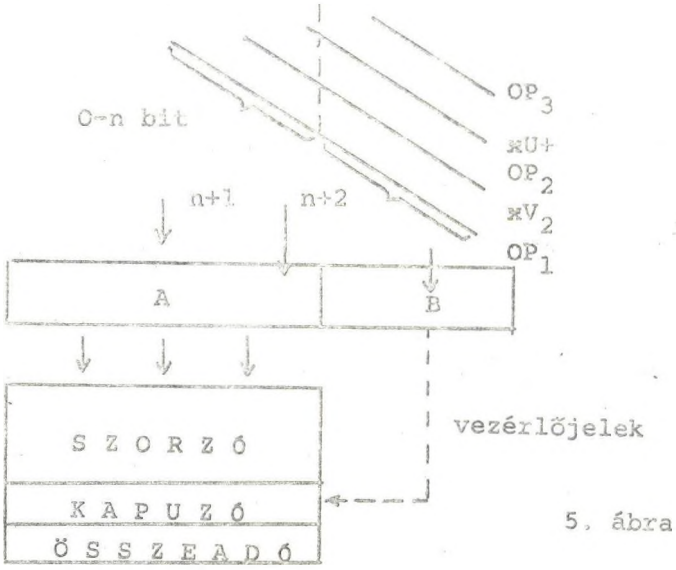
2/ a csak + müveletben szereplő tagok változtatás nélkül vezérlőjellel megjelölve kerülnek outputra.

Az igy megjelölt számsorozatot egy összeadóval összegezzük. Például: szerkezet input: 8*9*3 + 3 + 7*5 +4

szorzó output /összeadó input/: 216+3+35+4

összeadó output /eredmény/: 258

A szerkezet elrendezése a következő:



5. ábra

Müködése: Az input először az ábrán A, B-vel jelölt szerkezeten halad keresztül. Ennek a térrésznek a feladata a szorzó inputjának kialakitása. Tegyük fel, hogy a kifejezés i-edik input szava érkezik, és ez müveleti jelet tartalmaz. Ekkor B a jelet A-ba továbbitja, amin a vizszintesen keresztül haladva a következő inputot állitja elő a szorzó számára:

* jel esetén /az előző szorzatot vagy operanduszt szorozni kell/:

f_i= az i+1-edik input szóban levő operandusz

$a_i = \emptyset$

Ekkor a szorzó output $e_i=e_{i-1}f_i$ lesz.
e_{i-1} nem lesz vezérlő jellel jelölve.

+ jel esetén /az előző szorzatot vagy operandust hozzá kell adni az összeadó tartalmához/:
$f_i=\emptyset$ /B ekkor vezérlő jelet generál!/
a_i= az i+1-edik input szóban levő operandust.

Ekkor $e_i=e_{i-1}f_i+a_i=a_i$.

e_{i-1} vezérlő jellel jelölve lesz.

Az igy keletkezett e_i-k és vezérlőjelek ezután a kapuzó sorba érnek, amely csak azokat engedi az összeadóba jutni, amelyek vezérlőjellel vannak ellátva /a jel végig halad a soron/

Output: Az összeadóban keletkezik /egy 11 müveleti jellel törölhető/

Helyigény: $(n+3)^2$ sejt; n az alkalmazott szóhossz.

Időigény: A végeredmény az első jel beérkezésétől számitott k+2n+2 billenés mulva áll elő /k a kifejezés hossza/. A kifejezések egy törlő jellel /11/ elválasztva folyamatosan követhetik egymást.

A cikkünkben ismertetett sejtprogramok az emlitett sejthardvare megépitése esetén jelentősen leröviditik feladataink végrehajtási idejét. A jelenleg rendelkezésre álló hazai technológia felhasználásával megépitett sejtprocesszor egy sejttér billenést kb 1 μs alatt valósithat meg /nemzetközi technológiával ez az idő 5-10-szer kevesebb/. Ezt az értéket alapul véve az alábbi táblázatban néhány konkrét példa segitségével mutatjuk be szerkezeteink müködési sebességének alakulását:

szerkezet	paraméter	input fogadási sebesség	teljes számitási idő	idő/+1
X^n	n=8	18 μs	50 μs	2 μs
N!	n=10	22 μs	54 μs	2 μs
vektorszorzó	20 elemü vektorok	4 /elem	112 μs	4 μs
polinom kiért.	12 foku	26 μs	58 μs	2 μs
f(x)	100 alappont	4 μs	164 μs	1 μs
ar.kif.kiért.	11 operandus és müveleti jel	11 μs	45 μs	1 μs

Idő/+1 jelzésü rovat azt jelzi, hogy a paraméter oszlopban feltüntetett értékek eggyel való növelése mennyivel nyujtja meg a végrehajtási időt. A szerkezetek egy fontos tulajdonsága, hogy az alkalmazott szóhossz az input fogadási sebességét és az idő/+1 rovat értékét egyáltalán nem befolyásolja. A teljes számitási időt pedig csak egy additiv konstans értékkel nyujtja meg, ami nagy számu input esetén elhanyagolható. /Itt 16 bites szavakkal számoltunk./

Végezetül megemlitjük, hogy az utolsó két szerkezet fejlesztése tovább folyik. Kidolgozás alatt áll egy olyan szerkezet amely több különböző függvény értékét megadott szakaszokon polinom-közelitéssel számitja ki. Ugyancsak foglalkozunk egy a jelenleginél lényegesen általánosabb aritmetikai kifejezés kiértékelő elkészitésével.

Abstract:
This paper treats cellular programs for basic arithmetic computations, as: binary multiplication, evaluation of X^n, n! scalar product and polynomial evaluation.
Cellular algorithms and their realization are shown. Most of the solutions work on a bitparallel and/or pipe-line working principle. The supposed cellular space is a 16 state, time-

-space inhomogeneous one [1] with properties maintaining the bitparallel programming. At the end of the paper concrete examples with time estinations are given.

Irodalomjegyzék:

[1] Legendi T.: Cellprocessors in computer architecture /Computational Linguistics and Computer Languages, v. XI, 1977, pp. 147-167/

[2] Legendi T.: Programming of cellular processors /Proceedings of the Braunschweig Cellular Meeting, 1977, jun. 2-3./

[3] Legendi T.: Cellular algorithms and their verification /A CONPAR'81 konferencia kiadványa, Nürnberg, 1981./

[4] Katona E.: Sejtalgoritmusok. Válogatás az MTA Automataelméleti Kutató Csoportnál elért eredményekből /Neumann János Számitógéptudományi Társaság kiadványa, 1981. Budapest/

[5] Katona E.: Binary addition and multiplication in cellular space /Acta Cybernetica, Szeged, 1981./

[6] Katona E.: The application of cellprocessors in conventional data processing /A III. Magyar Számitástudományi Konferencia kiadványa, Budapest, 1981./

[7] Dióslaki F.: Sejtprocesszorok software eszközei /Diplomamunka, JATE, Szeged, 1979./

[8] Vollmar, R.: Algorithmen in Zellularautomaten /B.G. Teubner, Stuttgart, 1979./

Domán András

FUNKCIONÁLIS SZEMANTIKA A SOROS ÉS PÁRHUZAMOS PROGRAMOZÁSBAN[1]

Egy magasszintű applikatív nyelv

Amíg a hagyományos nyelvek főbb vonásaiban a /Neumann-elvű/ számítógépek architektúrája tükröződik, addig az applikatív nyelvek egyszerű matematikai /funkcionális/ szemantikára alapulnak. Ezek a nyelvek nem tartalmaznak sem programváltozót /mint a felülírható memória reprezentációját/, sem vezérlési struktúrát, amely a végrehajtási sorrendet önkényesen meghatározná. A funkcionális progeamok ugyanakkor függvényszerű leírás segítségével az objektumok között tartós összefüggéseket definiálnak. Az alapelvekből levezethető számos olyan tulajdonság, amely eleget tesz a modern programozási nyelvekkel szemben támasztott főbb követelményeknek, különösen a párhuzamos programozás területén: a nyelv és a programozás egyszerűsége, egyszerű helyességbizonyítás, dokumentálhatóság, párhuzamos algoritmusok egyszerű leírása párhuzamosító programkonstrukciók szükségessége nélkül, szemantikai mellékhatás-mentesség, determinizmus stb. A tanulmányban ismertetésre kerülő applikativ nyelv a PARAFLOG dataflow párhuzamos számítási rendszerekben különösen jól alkalmazható - a hardver szinten elérhető maximális párhuzamosságot magasszintű nyelvi eszközökkel engedi feltárni - ugyanakkor előnyei megmutatkoznak a soros programozásban is.

Kulcsszavak: párhuzamos programozás, dataflow nyelvek, funkcionális programozás, applikativ nyelvek.

Bevezetés

"Amikor a programozás, mint a rendezetlen komplexitással szembeni küzdelem jelentkezik, teljesen nyilvánvaló, hogy

[1] A tanulmány része az OMFB támogatásával folyó, "dataflow számítási rendszerek" kutatásának.

bárki ahhoz a gondolati diszciplinához fordul, amelynek a célja évszázadok óta az volt, hogy effektiv struktúrálást alkalmazzon, az egyébként rendezetlen komplexitáson. E gondolati diszciplina többé-kevésbé ismert számunkra, ez a matematika." /Dijkstra [1] /

A funkcionális szemantikára alapuló applikativ nyelvek azt a fontos, talán nem meglepő tényt példázzák, hogy a matematika is, amely tekintélyes erővel, történettel áll a rendelkezésünkre, lehet programozási nyelv - igen természetes módon. Ez nyilvánvalóan azt az igényt is maga után vonja, hogy /programozási/ problémáinkat, mint matematikai problémákat fogalmazzuk meg, megfelelő eszközök támogatásával.

Tanulmányunk célja kettős, egyrészt az általunk kidolgozott és implementált funkcionális nyelv, a PARAFLOG bemutatásával szemléltetni kívánjuk a magasszintü programozásra alkalmas egyszerü matematikai leirás lehetőségét, másrészt megmutatjuk, hogy a funkcionális nyelvek természetes algoritmus-leirási eszközt jelentenek nemcsak soros, de a párhuzamos végrehajtás számára is anélkül, hogy a felhasználónak a párhuzamosságot, mint különleges tényt kellene kezelni akár algoritmus- akár programozási szinten.

Applikativ nyelvek

Történetileg a programozási nyelvek az embernek a számitógép vezérlésére irányuló igényét szolgálták egyre növekvő pontossággal. A korai nyelvek a preciz és formális szintaxist hangsulyozták, az informális szemantika pedig inkább a rendelkezésre álló számitógép müveleteihez idomult. A fejlődés során egyre több kétség merült fel a hagyományos programozási nyelvek alapjaival kapcsolatban. "1968-ban Dijkstra kifogásolta a GO TO utasitást, 1973-ban Wulf és Shaw kifogásolta a globális változók használatát. 1973-tól Bauer, Berkling és mások kezdték kétségbevonni a programváltozók használatát, Backus pedig a programozási nyelvekben még a matematikai változók haszná-

latát is megkérdőjelezte. Néhány olyan nyelv megjelenése, mint a LISP, vagy más applikativ nyelvek kezdték komolyan kifogásolni az értékadó utasitásokat is" [2] . A programozási nyelvekben mindezek a "nem-matematikai" konstrukciók /GO TO, értékadás, változó, globális változó, cimke stb./ egyre növekvő mértékben hozzájárultak a nagy programok szoft ver bonyolultságához, a homályossághoz, a nehézkes helyesség bizonyitáshoz, a magas szoftver költségekhez.

A problémák még tovább fokozódnak a párhuzamos számitási rendszereknél. A komplex szinkronizációs és ütemezési mechanizmus elkerülése szempontjából lényeges aszinkron párhuzamos nyelvek nem férnek össze a felülirható memóriarekesz koncepciójával. Ha a számitások értékekre alapulnának, az értékek helyére utaló cimzés helyett, akkor az irás-olvasás versenyhelyzete is megszünne. Ez az "érték szerinti" számitás az egyik alaptézise az applikativ nyelveknek, amelyek a fent leirt helyzetből egyfajta kivezető utat jelentenek. Az érték szerinti mechanizmus az un. Single Assignment elvben fogalmazódik meg:

> minden változó a programben legfeljebb egyszer kaphat értéket, de az akárhányszor felhasználható.

Az applikativ nyelvek szemantikai alapja a funkcionalitás, azt fejezi ki, hogy a program transzformációi: argumentumaikon operáló függvények, és a számitási eredmény nem más, mint a függvényértékek halmaza. A funkcionalitásnak köszönhető a szemantikai mellékhatás-mentesség is: a függvények argumentumaikon kivül más hatást, pl. globális változót, vagy tárolt belső állapotot, nem vesznek figyelembe, ugyanakkor a környezetre is csak a függvényértéken keresztül hatnak.

Igen fontos az applikativ nyelveknek egy másik tulajdonsága a definitiv jelleg, amely az értékadás megváltozott szerepével kapcsolatos.

A hagyományos, imperativ nyelveknél az értékadás egy kifejezés értékét adott végrehajtási fázisban hozzárendeli egy változóhoz, melynek értéke /tartalma/ ujabb értékadással megváltoztatható. A funkcionális nyelvekben azonban, az utasitások a program egész terjedelmében érvényes definiciók, igy az objektumok, változók jelentése invariáns. Ez biztositja azt, hogy az utasitások sorrendje az ilyen definitiv nyelvü programban tetszőleges. nem befolyásolva a tényleges végrehajtási sorrendet. A definitiv nyelvek esetében a programhelyesség bizonyitása is rendkivül egyszerüvé válik; az állitások, amelyeket a bizonyitásban alkalmazunk azonosak a program definicióival. Nem kell végigkövetnünk a vezérlési folyamot - mint imperativ nyelveknél - ahhoz, hogy megállapitsuk, mely pontokban igaz egy állitás.

Dataflow számitás

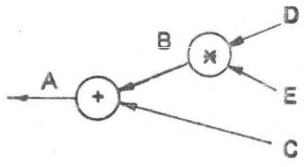
Applikativ nyelvü program végrehajtása a definiciókban szereplő transzformációk végrehajtását jelenti és bármely transzformáció legkorábban akkor hajtható végre, ha az objektumai már definiáltak. Az elemi definicióknak ez a logikai kapcsolata - az adatfüggés /data dependency/ - szemléletesen reprezentálható egy irányitott gráffal, az un. dataflow gráffal. /Ez a dataflow gráf egyben a különböző dataflow gépek absztrakt gépi nyelvének is tekinthető./ A gráf csúcsai az operátorok, amelyek elemi transzformációt reprezentálnak, mig az irányitott élek az adatpályák megfelelői.

A dataflow gráf /program/ végrehajtása minden csúcsban az operandus rendelkezésére állásán /availability/ alapul. Ha egy operátor minden bemeneti élén értékkel rendelkezik, akkor a müvelet végrehajtódik és a kimeneti éleken az eredményértékek megjelennek, miközben a bemeneti értékek eltünnek /"elfogyasztódnak"/. Ez a folyamat vég nélkül ismétlődhet.

A dataflow gráf és az applikativ nyelvü program ill. annak definiciói egyértelmü megfeleltetései egymásnak. A változók az éleknek, a transzformációk az operátoroknak a megfelelői, a funkcionális alkalmazás pedig az operátorok összekapcsolása a gráfban.

pl.

$$\left.\begin{array}{l} A = B + C \\ B = D * E \end{array}\right\} \quad A = D * E + C$$



A dataflow rendszerek egyik legfontosabb előnye a maximális /alapmüvelet-szintü/ párhuzamos végrehajtás lehetősége - a párhuzamosság /szinkronizáció/ explicit előirásának szükségessége nélkül. A felhasználó párhuzamosan futó programot készithet anélkül, hogy bármiféle elemzést vagy "párhuzamositás"-t végezne. A javasolt PARAFLOG magasszintü nyelv, mint definiciós nyelv az objektumok közötti tartós kapcsolatokat definiálja és minden programnak egyértelmüen megfeleltethető egy adatfüggési vagy dataflow gráf. A gráf csúcsaiként reprezentált operátorok /operációk/ folyamatosan, az availability szabály alapján müködnek egymástól függetlenül aszinkron módon. Igy a végrehajtás párhuzamosságát nem a programozó irja elő, hanem az adatfüggést tükröző dataflow végrehajtási mechanizmus. Egyprocesszoros /soros/ végrehajtás esetén a végrehajtó /enable/ operátorok közül egyszerre csak egy - közömbös, hogy melyik - müködtethető. Ez mintegy szimulációja a természetes párhuzamos végrehajtásnak.

A PARAFLOG NYELV INFORMÁLIS LEIRÁSA

Az általunk elfogadott filozófia szerint bármely program P-egyenletekből áll, amelyek a jobboldal és a baloldal tartós azonosságát definiálják /definitiv jellegg/. Igy a - hagyományos nyelvek utasitás-szintü egységeinek is tekinthető - P-egyenletek a program bármely részén szerepelhet-

nek, nem befolyásolva a program helyességét. Az egyenlet-/utasitás/ sorrendnek ez a programbeli invarianciája együtt jár azzal, hogy a program nem tartalmaz vezérlési struktu-rát sem: nem irhatjuk elő önkényesen az utasitások végre-hajtási sorrendjét. A végrehajtási mechanizmus a dataflow elvre alapulva, az algoritmus szükségszerü adatfüggéseit tükröző végrehajtási sorrendet definiál /soros ill. párhu-zamos végrehajtásban/.
Mint funkcionális nyelvnek további lényeges - a hagyományos nyelvektől eltérő - vonása a változó-mentesség. Nem tartal-maz a nyelv olyan objektumot, amelynek jelentése, ill. tar-talma a programvégrehajtás során változna. A Single Assignment elv, igy át is fogalmazható "Zero Assignment" elvre. Eszе-rint, ha a változóhoz az egyszeres értékadással egyetlen transzformáció rendelhető, akkor elégséges a függvényérté-ket ezzel a transzformáló függvénnyel azonositani. Az újsze-rü nyelvi megközelités a programirás mellett a programhelyes-ség bizonyitását is jelentősen megkönnyiti, ugyanakkor olyan programok irhatók ezen a nyelven, amelyek egyaránt futtat-hatók soros vagy párhuzamos dataflow számitógépen, a pár-huzamositó programkonstrukciók szükségessége nélkül. /A felhasználónak nem kell tudnia a végrehajtási mechanizmusról vagy a végrehajtó közegről./

P-Egyenletek

A program bármely egyenlete megegyezik abban, hogy jobbolda-la egy függvény - leggyakrabban összetett függvény - baloldа-la pedig a függvényértéket reprezentáló kifejezés. /Az egyen-let szintaktikus lezárása pontosvessző./
A P-egyenleteknek két alapvető tipusa van:

- applikációs egyenlet
- funkcionális kompozició

Az applikációs egyenlet meghatározott azonositóhoz /összetett/ függvényértéket rendel. Összetett függvény a funkcionális alkalmazás /applikáció/ segitségével képezhető, amely megengedi, hogy függvényargumentumként ujabb függvény, vagy függvényértéket reprezentáló azonositó szerepeljen. /A függvények a nyelvben egységesen prefix alakban szerepelnek./

Pl. az y = (a + b). z algebrai kifejezés két ekvivalens kifejezése a nyelvben:

```
x = ADD (a,b);
                        azaz      y = MPY (ADD (a,b),z);
y = MPY (x,z);
```

Az egymásba ágyazás eredményeként gyakran egyetlen egyenlettel is igen bonyolult program irható. Mivel a függvény argumentumai lényegében ujabb függvények, ezért a többszörös alkalmazásnál kialakul egy lánc, az un. applikációs lánc. Helyesen müködő programnál ennek az applikációs láncnak mindig lezártnak kell lennie adatgenerátor /input vagy konstans/ függvénnyel. /Az applikációs láncok szemléletesen mutatkoznak a dataflow gráfban is./

A P-egyenletek másik tipusa a funkcionális kompozició arra szolgál, hogy a felhasználó a meglévő függvények segitségével tetszőleges ujabb függvényt definiálhasson /később részletezve/.

Példa a komplex számokat összegző függvény definiálására:

```
COMPLEXADD (x,y) = DEF (REALPART, IMMAGPART) &
        REALPART = ADD (FIRST (x),FIRST (y)) &
     IMMAGPART = ADD ( LAST  ( x),LAST ( y));
```

Struktúrák

Ahogy bármely programbeli adatértéket függvény állithat elő, úgy adatstruktúrát függvénystruktúra hoz létre, ezzel azonositható. A függvénystruktúrák, amelyek speciális függvényekkel /pl. ARG, REPEAT, DEF/ képezhetők, lényegében vektorértékü függvények. /Alapvetően ezek hiánya miatt nem valósitható meg a hagyományos nyelvekben a funkcionális programozás./

A PARAFLOG struktúrái listák: olyan elemek listái, melyek maguk is tetszőleges függvénystruktúrák, vagy alapfüggvények. Ezeknek a kezelésére un. struktúrakezelő függvényeket definiál a nyelv. Pl. az ARG függvény /a LISP nyelv CONS függvényéhez hasonlóan/ listára füzi argumentumait, az i-THARG /i-edik struktúraelemet kiválasztó/ ill. FIRST, LAST függvények pedig szelektor függvények:

Pl.

A = ARG (C,ARG (NOP(B),F,ARG(G,SUM(X,Y)))) ;
egyenlet által definiált struktúra /lista/:

A = C.(NOP(B).F.(G.SUM(X,Y).NIL).NIL).NIL

és

X = LAST(LAST(A)) ; X ≡ G.SUM(X,Y).NIL
Y = FIRST(X) ; Y ≡ G

A struktúrákra vonatkozóan, a funkcionalitás ugyanúgy érvényes, mint egyszerü értékekre, függvényekre. Bármely struktúrához definiálható olyan függvény, amely argumentumként, ill. függvényértékként tekint ilyen struktúrát. Tekintsünk egy n-elemü vektort beolvasó és kiiró függvénystruktúrát:

```
a = n - REPEAT(INP(C:ACT));
b = n - REPEAT(OUT(ACT-THARG(a)));
```

Az "a" struktúra /lista/szerkezete:

INP(C:1). INP(C:2). ... . INP(C:n). NIL

Függvénystruktúrák alkalmazása amellett, hogy növeli a kifejező erőt és az egyszerüséget, komplex fogalmak egyszerü, természetes kezelését biztositja. Különösen szabályos struktúrák, vektorok, mátrixok kezelése könnyü. Igen egyszerüen definiálható pl. olyan mátrixösszeadási vagy akár szorzási függvény, amelynek argumentumai mátrixstruktúrák:

```
A = MATRMPY(C,MATRADD(E,F)) ;
```

/A struktúrák az értékeken túl magukban hordozzák saját szerkezeti leirásukat, méretüket anélkül, hogy ezt specifikáltuk volna./

Adattipusok

A PARAFLOG nyelvben az adatok igen speciális formában jelennek meg. Miden adatobjektum elemi adatok végtelen hosszúságú sorozata, stream-je. Ez úgy is értelmezhető, hogy az adatobjektumoknak mintegy a teljes története jelenik meg a számitásban és ennek megfelelően a függvények is adatstream--eken ill. adatstream-ek struktúráin vannak definiálva. Adatsorozatot /un. adatgenerátor függvények hoznak létre; ilyenek az input függvények, amelyek tetszőleges hosszúságú bemneti stream-et generálnak ill. a konstans függvény, amely azonos értékek végtelen sorozatát képezi.

A PARAFLOG-ban az elemi adatok részben tipizáltak, ami azt jelenti, hogy nem kell tipusdeklarációt adni, bár a nyelv 3 tipust megkülönböztet: egész, karakter és string tipusokat. Ezek a rajtuk alkalmazható függvényekben különböznek egymástól; bizonyos függvények nem minden tipuson definiáltak. Értelmetlen lenne pl. két string aritmetikai szorzása, de ugyanakkor az INP, OUT, NOP, EQU, IFTHELSE stb. függvények minden tipuson értelmezhetők.

Függvények

A függvények alapvető jelentőségüek a PARAFLOG-ban: minden transzformációt a programban ezek valósitanak meg, egységes

szemléletü nyelvi kezeléssel, értékekre alapuló számitással. A függvények adatstream-ek struktúráin operálnak ilyan módon, hogy a stream-ek elemein /stream-ek metszetein/ egymás után, egymástól függetlenül végeznek feldolgozást, függvényértékként ugyancsak adatstream-et eredményeznek. /A metafüggvények osztálya az egyetlen, amely eltér ettől./ Legyen pl. x és y adatstream-et reprezentáló függvény:

$$x = \langle x_1, x_2, \ldots x_n \rangle \quad \text{ill.} \quad y = \langle y_1, y_2, \ldots y_m \rangle$$

ekkor a z = ADD(x,y); egyenletben a z függvény értéke:

$$z = \langle (x_1+y_1), (x_2+y_2), \ldots (x_k+y_k) \rangle \quad \text{sorozat, ahol } k=\min(m,n)$$

A függvények stream-feldolgozó képessége következtében a nyelvben igen egyszerü, természetes módon irhatók le stream-feldolgozó, pipeline, real-time algoritmusok, valamint ilyen struktúrára transzformálható nem kisszámú, egyéb algoritmus.

A számitási folyamatban betöltött funkció alapján két alaptipusba sorolhatók a nyelv függvényei. Vannak aktiv függvények és un. struktúrakezelő függvények. /A különválasztást főleg a dataflow végrehajtás indokolta./

Aktiv függvények

Az aktiv függvényeket, amelyek futási időben tényleges adattranszformációt végeznek, azaz elemi dataflow operátorokat reprezentálnak a dataflow gráfban, a következő 3 osztály alkotja:

a./ adattranszformáló függvények: ADD, MIN, MPY, DIV, NOP, LESS, EQU, SQRT, ABS,...

b./ vezérlő függvények: IFTHELSE, IFTHEN,...

c./ <u>kommunikációs függvények</u>: INP, OUT, CONS,...

Nem okoznak változást a számitási térben, csupán a számitási tér és a környezet közötti interface-t valósitjá meg. Az INP és CONS adatgeneráló függvények, mint lezáró függvények argumentumként nem függvényt, hanem azonositót fogadnak el /a CONS emellett a generálandó értéke is /. Ez az azonositó arra szolgál, hogy a tetszőleges sorrendben érkező adatstream-eket, amelyek ilyen azonositóval rendelkeznek, hozzárendelje a megfelelő INP vagy CONS operátorokhoz.

Az adatobjektumok stream reprezentációjával kapcsolatos az a sajátos lehetőség is, amellyel az aktiv függvényekhez kezdeti érték, un. kezdeti jelölés /marking/ rendelhető. Az ilyen függvény kimeneti adatsorozatának nulladik eleme - függetlenül az argumentumoktól - a kezdeti érték lesz. A következő példa a MPY szorzási függvény jelöletlen /a/ és jelölt /b/ alakját, valamint a kimeneti adatsorozatokat mutatja:

a./ z = MPY(x,y); ⟶ $z = \langle (x_1 \cdot y_1) , (x_2 \cdot y_2) , \ldots \rangle$

b/ z = 7 INIT MPY (x,y); ⟶ $z = \langle 7, (x_1 \cdot y_1) , (x_2 \cdot y_2) \ldots \rangle$

/INIT initial alapszó, amely az aktiv függvényekhez a kezdeti értéket hozzárendeli és ezzel új függvényt definiál./

A függvényjelölésnek az a célja, hogy adatstream különböző sorszámú /"életkorú"/ elemei közötti müveletvégzés lehetőségét biztositsa /stream-tolás/. Szemléletes példával szolgál erre a visszacsatolt strukturájú, az y'=f(x) egyenlet egyszerü közelitő megoldását /integrálját/ kiszámitó program /részlet/:

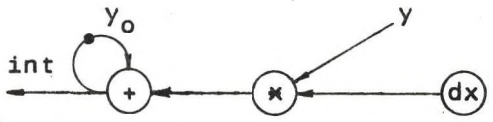
mat.formula: $int_i = y_0 + \sum_{j=1}^{i} y_j . dx$ y_0=kezdeti érték

rekurziv alak: $int_i = int_{i-1} + y_i . dx$ $i \geq 1$, $int_0 = y_0$

program: int = int_0 INIT ADD(int,MPY(y,CONS(dx))) ;

függvényérték /stream/: int = $< y_0, y_0+y_1 . dx, y_0+(y_1+y_2) . dx, \ldots >$

dataflow gráf:



A kezdeti jelölés itt szemléletes matematikai értelmet nyer azáltal, hogy a kezdeti érték az integrál kezdeti értékével egyezik. Megállapitható, hogy más esetekben is mindig megtalálható a kezdeti jelölés matematikai jelentése.

Struktúrakezelő függvények

Mig az aktiv függvények a tényleges elemi feldolgozást reprezentálják, a struktúrakezelő függvények az aktiv függvényekből épithető függvénystruktúrák statikus /forditásidejü/ manipulációját, leirását szolgálják. Ezek a függvények szintén 3 jól elkülönithető függvénycsoportot alkotnak:

a./ konstrukciós függvények: ARG, i-REPEAT,..
Aktiv függvényekből vagy struktúrákból újabb függvénystruktúrát hoznak létre.

ARG - argumentumait listára füzi /lásd LISP, CONS fv./

i-REPEAT - argumentumának i db. egymás után generált példányát listára füzi; a hagyományos FOR ciklushoz hasonlitható /i - metaérték/.

b./ szelektor függvények: i-THARG, FIRST LAST PREV, THIS, CURR,..

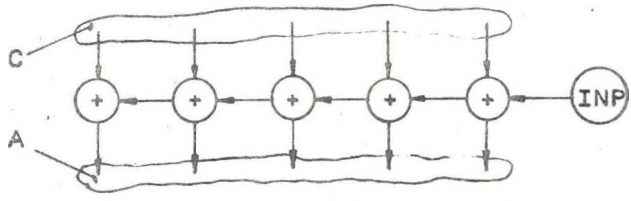
Adott függvénystruktúrának egy elemét - ami maga is tetszőleges struktúra lehet - adja a kimeneten.

i-THARG - a struktúra i-edik elemét választja ki

FIRST, LAST - a struktúra első ill. utolsó eleme

PREV, NEXT - a REPEAT függvény argumentumának függvénykifejezésében szerepelhetnek; az ismétlődések során a megelőző ill. rákövetkező függvénystruktúrát /argumentumpéldányt/ adják. Ezek a függvények az iteráció elején és végén lezárást kivánnak, amit a FBY /followed by/ és PBY /preceeded by/ alapszavak biztositanak, elválasztva a lezáró kifejezést az iteráció során érvényes argumentumtól. Példa:

```
A = 5-REPEAT(ADD(INP(B) FBY PREV, CURR(C)));
```



CURR - az ismétlési függvény i-edik iterációjában az i-euik elemét adja az argumentumban szereplő függvénystruktúrának
/ CURR(x) = ACT-THARG(x) /

c./ metafüggvények: LENGTH, ACT, konstans,..

Metaértéket, egésszámot adnak az i-REPEAT, i-THARG, stb. metafüggvényekhez /i-érték/

LENGTH - struktúra mérete /listahossz/

ACT - az ismétlési függvénynél az iteráció aktuális sorszáma; a ciklusváltozó megfelelője

konstans egésszám.

Funkcionális kompozició

A felhasználó a meglevő függvények alkalmazásával új függvényt definiálhat. A definiáló egyenlet baloldalán a definiált függvény a formális argumentumokkal szerepel, mig a jobboldalon álló DEF függvény argumentumai a definiált függvénystruktúra komponensfüggvényeit alkotják. Argumentumként nem szükséges a teljes függvénykifejezést szerpeltetni, lehetséges ezt azonositóval helyettesiteni, majd azt "&" szimbólummal elválasztott külön egyenletekkel definiálni.
Például vektorok, mint n-elemü struktúrák összeadására a következő definiáló egyenlet irható:

```
VECTORADD(x,y) = DEF(c) &
  n = LENGTH(x) &
  c = n-REPEAT(ADD(CURR(x),CURR(y))) ;
```

Végrehajtási mechanizmus a PARAFLOG-ban

A PARAFLOG funkcionális nyelv, melynek egyenletei egyszerü matematikai formulákkal kifejezett definiciók. Bármely program ill. annak definiciói egyértelmüen megfeleltethetők egy speciális irányitott gráfnak az adatfüggési vagy dataflow gráfnak, amelyet egy dataflow kiértékelési mechanizmus interpretál. /Az adatfüggési gráf végrehajtásának ismert más megközelitése is, pl. igényvezérelt - demanddriven./ Mind a fordító, amely a forrásnyelvü prggramból dataflow gráfot generál, mind pedig a gráfot végrehajtó szimulátor PROLOG /logikai alapú/ nyelvben lett implementálva. A dataflow gráfot igen egyszerüen reprezentálja az operátorok listája ugy, hogy minden operátor a funkcióleirást és a kapcsolodó operátorok hivatkozását tartalmazza. A szimulátor a listaelemeken ciklikusan sorbahaladva minden olyan operátort aktivizál, végrehajt, amelyre az availability feltétlenül teljesül. A dataflow gráf párhuzamos végrehajtására itt nem térünk ki, de megjegyezzük, hogy a bázis gépi nyelv

szintjéig nem különbözik egymástól a nyelv 2-féle implementációja.

MEGJEGYZÉSEK A NYELVRŐL

Az itt bemutatott nyelv a felhasználó számára bizonyos problémákat generálhat, egyrészt a struktúrák statikus kezelése miatt, másrészt a nyelv új, függvényszerű szemléletével, a változó-mentességgel és egyéb olyan vonásokkal, amelyek a funkcionális szemantikából adódnak. Mindazonáltal úgy érezzük, hogy a nyelv koncepciója szervesen beleilleszkedik az applikativ nyelvek irányzatába, amely az egyszerű matematikai szemantika által elérhető programozási előnyök mellett főként a párhuzamos rendszerekben való alkalmazásra koncentrál. A PARAFLOG-nak is fő motivuma a párhuzamos, dataflow számitási rendszerek, igy a PARADOCS dataflow számitógép [3] számára alkalmas magasszintü nyelv létrehozása.

Abstract: Unlike traditional programming languages, whose main features reflect the architecture of /Neumann-type/ computers, applicative languages are based on simple mathematical /functional/ semantics. These languages are variable-free avoiding the updateable memory concept/ and don't include control structure to define the execution order arbitraryly. At the same time functional programs, by means of function-like description, define permanent relation between the objects. From the basic concepts one can deduce a number of properties meeting the requirements in connection with modern programming languages, in particular on the field of parallel programming: Simplicity of the language and of programming, simple proving of correctness, easy description of parallel algorithms without the need of parallel programconstructs, freedom from semantic side effect, determinism, etc. In the study, an applicative language called PARAFLOG will be presented. It is well

suited for parallel dataflow systems - the greatest potential parallelism at the hardware level is allowed to exploit by high level language facilities - and its advantages may by experienced in sequential systems as well.

Keywords: parallel programming, dataflow language, functional semantics, applicative language.

Irodalom

[1] Dijkstra, E.W.: On a methodology of design MC-25 Information Symposium, Amsterdam 1971.

[2] Organick, E.I.: New Directions in Computer Systems Architecture. Euromicro Journal 1979.

[3] Domán, A.: PARADOCS: A Highly Parallel Dataflow Computer and its Dataflow Language. Microprocessing and Microprogramming. 1981/1.

Domán András
Számitástechnikai Koordinációs Intézet

Fabók Julianna–Hermann Gábor–Kovács József–Krammer Gergely

A GESAL NYELV PORTÁBILIS IMPLEMENTÁCIÓJÁVAL KAPCSOLATOS TAPASZTALATOK

Az MTA SZTAKI-ban 1976-80 között létrejött egy rendszerprogramozásra és alkalmazási programok irására egyaránt alkalmas kisgéporientált programozási nyelv, a GESAL, amelyet 5 gépre átvittünk. Az előadás a nyelvről és portábilis implementációjáról szól. A nyelv rövid ismertetése után a kompilerről, a közbülső kódról, a kódgenerátorokról, a portábilitás nehézségeiről és a munka tanulságairól van szó.
Kulcsszavak: portábilitás, programozási nyelvek, GESAL, intermediate code.

1. BEVEZETÉS

A GESAL nyelv fejlesztése 1976-ban kezdődött az MTA SZTAKI-ban /Gerhardt [1]/. A nyelvvel kapcsolatos fontosabb célkitüzések a következők voltak:

- magas szintü nyelv,
- hatékony tárgykóddal,
- rendszerprogramozásra alkalmas,
- alkalmazási rendszerek programozására alkalmas,
- portábilis nyelv: a benne irt programok és a kompiler maga is "könnyen" átvihető más gépre.

Az Intézet foglalkozik hardware/software rendszerek /folyamatirányitás, szerszámgépvezérlés stb./ és alkalmazási rendszerek fejlesztésével /gépipari NC-nyelv feldolgozása stb./, amelyeket különböző kisgépekre, nagygépekre és mikroprocesszszorokra kell elkésziteni.

Egy portábilis nyelv elősegiti a programozók átállását, a félkész programtermékek /file kezelés, nyelvgenerátor, grafikus alapsoftware stb./, esetenként egy-egy teljes programrendszer átvitelét egyik gépről egy másikra.
Az induláskor elsősorban a 16 bites kisgépek egy szük családját vettük figyelembe /TPA70, PDP11, R10/, és szükségtelennek tartott általánositásoktól /pl. különböző szóhossz/ eltekintettünk. Későbbi döntés folytán ugyanezt a nyelvet implementáltuk IBM 3031 CMS alatt és INTEL 8080-ra is.
A nyelvet a kifejlesztő osztály 5-10 programozója kezdte használni 1977-ben /ez a szám egyre nőtt/; segitségével többek közt egy adatbázis-kezelő rendszer, grafikus alapsoftware, BASIC interpreter készült.
Az Intézet fentebb emlitett feladataiból kiindulva, célszerü általánosan használható software termelőeszközök beszerzése vagy előállitása. Ebből a szempontból hasznosnak látszott a GESAL nyelv szélesebb körü használatra alkalmassá tétele. /Világos az is, hogy egy nyelv nem old meg minden problémát, és a nyelv kiválasztásában sem volt - természetszerüleg - összintézeti egyetértés./
A munkát a nyelv kis átalakitásával kezdtük /az ortogonális és a pragmatikus párt állandó vitái közben/, majd a gépfüggetlen IMC /intermediate code/ és a forditóprogram gépfüggő részeinek /pl. input-output/ specifikációja után előbb három kódgenerátor /TPA70, PDP11, R10/ irása kezdődött meg, majd ezt követte további kettő /IBM 3031, INTEL 8080/.
Ebben az előadásban az öt gépre implementált GESAL nyelv portábilitásával kapcsolatos tapasztalatokról számolunk be.

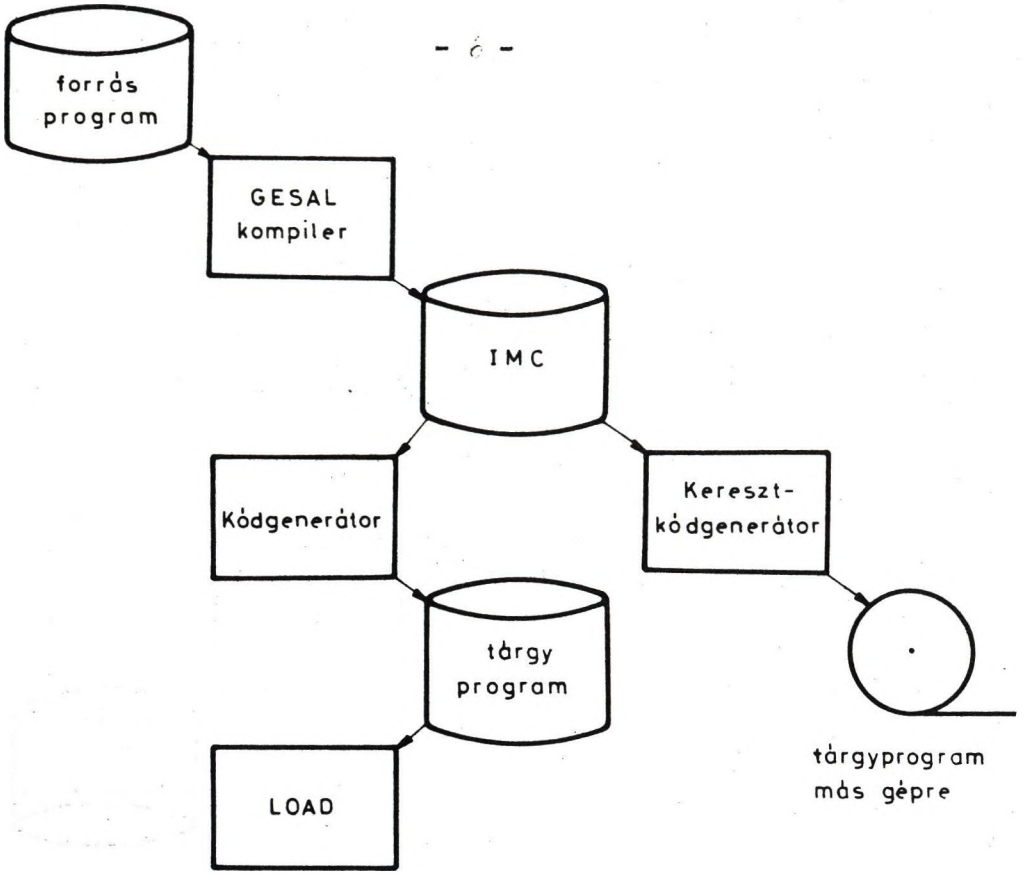
2. A GESAL NYELV ÉS FORDITÓPROGRAMJA

A nyelv alapvető tulajdonságait a PASCAL [4], ALGOL-68 [3], a C [2], a MARY [5] és a programozói gyakorlat tapasztalataiból kölcsönözte.
A GESAL nyelvü program önállóan forditható egységekből: szekciókból áll. A szekcióban deklarált objektumok hatásköre a

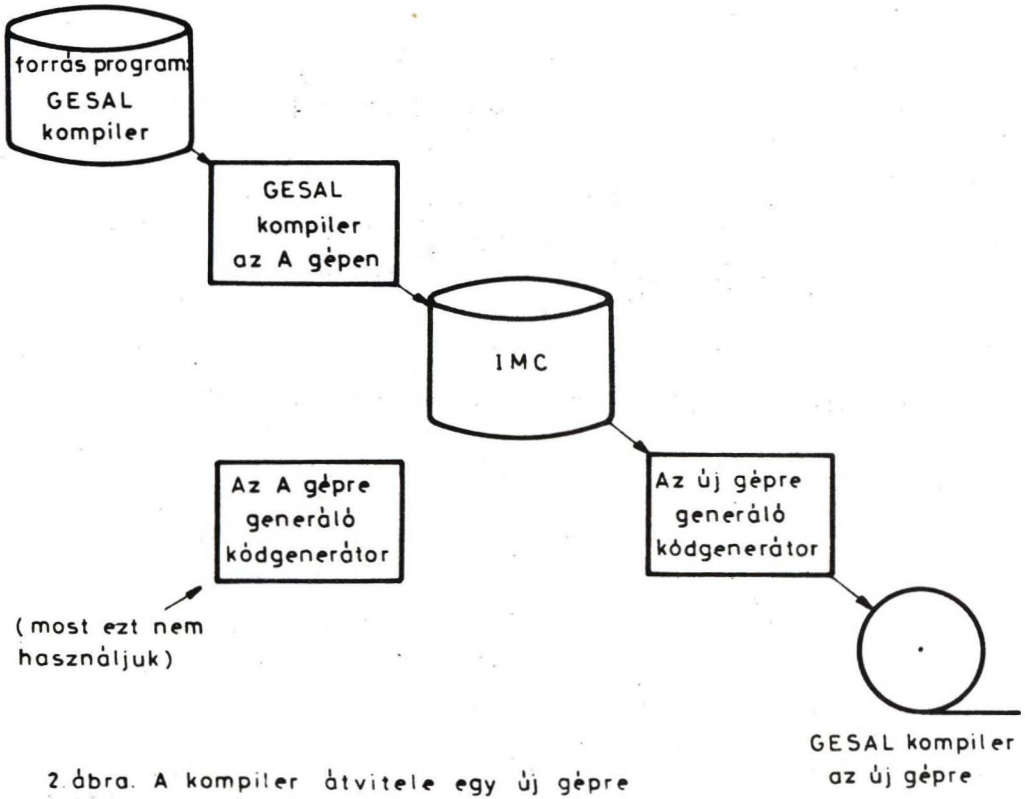
szekciókra korlátozódik; a más szekciókból látható objektumok deklarációját ENTRY prefixszel kell ellátni, ill. EXT prefixszel specifikálni kell a külső /más szekciókba tartozó/ objektumokat.
Célszerü egy szekcióba gyüjteni valamilyen közös feladatot ellátó adatokat és eljárásokat. Egyes szekciókat assembly nyelven is meg lehet irni, ha eközben betartjuk a nyelv konvencióit /pl. paraméter-átadás/, ill. GESAL szekciók bármikor lecserélhetők ekvivalens specifikációju assemly szekcióra. Ilyen szekció formájában irható meg például egy program input-outputja, vagy ilyennek foghatók fel egy operációs rendszer supervisor call-jai is.
A kompiler egyszerüsége érdekében a szekción belül nem adható meg ujabb szekció, mint ahogy eljáráson belül sem deklarálható eljárás. A szekcióban változók, konstansok és eljárások deklarációja adható meg, valamint uj tipusok definiciója; ezek a globális, az egész szekcióra érvényes deklarációk. Az eljáráson belül lokális, csak az eljárásra érvényes deklarációk szerepelhetnek.
Az eljárásokat lehet rekurzivan hivni, ill. az eljárások lehetnek rekurzivak. A szekciókból összeállitott, futtatható programban van egy kitüntetett eljárás, ahol a program futása kezdődik.
Az adatok a hat alaptipus valamelyikébe /karakter, byte, integer, longinteger, real, label/ vagy a program által definiált tipusok valamelyikébe tartoznak. Az utóbbiak az enumeráció, set, tömb, struktura, unió, vagy referencia tipusosztályokba tartoznak.
Az enumeráció és set tipus értelmezése a szokásos.
Bármely tipusu elemnek definiálható egyindexes tömbje; a többindexes tömb tömbök tömbje.
Az unió tipus ad lehetőséget arra, hogy ugyanazt a memóriahelyet különböző tipusunak tekintsük szükség szerint /ennek azonban gépenként eltérő következményei lehetnek/. A struktura a PASCAL rekordnak felel meg, mezőire minősitett névvel lehet hivatkozni. Változó hosszuságu struktura nincs /de

bármelyik mező lehet unió tipusu/.
A referencia tipusu változók objektumokra mutató pointerek.
Csak azonos tipusuak referenciái azonos tipusuak.
A program cimkéi label tipusu konstansok, a label tipusu változók, ill. ilyen tömbök elemei GOTO-ban szerepelhetnek.
A formális paraméterek száma és tipusa különbözteti meg a proc tipusokat. Deklarálható például az elemi függvények /egy valós be, egy ki/ tipusa, továbbá ilyenek ref-jének tömbje.
A skalár tipusokra a szokásos aritmetikai müveleteken kivül az assembly nyelvek logikai müveletei is alkalmazhatók /bitenkénti and, or, xor, not/.
Logikai tipus nincs, relációk használhatók kontroll-strukturákban. Ez utóbbiak
if-then-fi, if-then-else-fi,
if-then-elsif-...else-fi, case, while, until, for.
GOTO van, de a jó kontroll-strukturák mellett ritkán használja az ember.
A kompiler általában csak azt csinálja, amit a programozó leir: nincs automatikus tipus konverzió /csak explicit, a konverziós operátorokkal/; szigoru tipusellenőrzés van. Nincs automatikus dereferenciálás. Eljárás paramétere csak értékével átadott skalár lehet; összetett tipusu paraméternek a loc-ját /ref tipusu, tehát skalár/ kell átadni.
Az összes deklarált objektum helyfoglalása a forditás alatt eldönthető /bár a lokális objektumok csak futás közben foglalnak helyet/.
A programok forditását különböző gépekre és magának a forditóprogramnak az implementálását különböző gépeken egy gépfüggetlen közbülső kód /IMC = intermediate code/ és a bootstrap technika alkalmazásával érjük el /1., 2. ábra/.
A közbülső nyelveknek három fő fajtája van: a program fa reprezentációja, a postfix reprezentáció,
és a háromcimü utasitások formájában való leirás.
A GESAL IMC-nél postfix reprezentációt választottunk. Ezáltal a forditás nagyobb része végezhető el az általánosan megirt kompiler részben /mint a fa reprezentációnál/, és ez

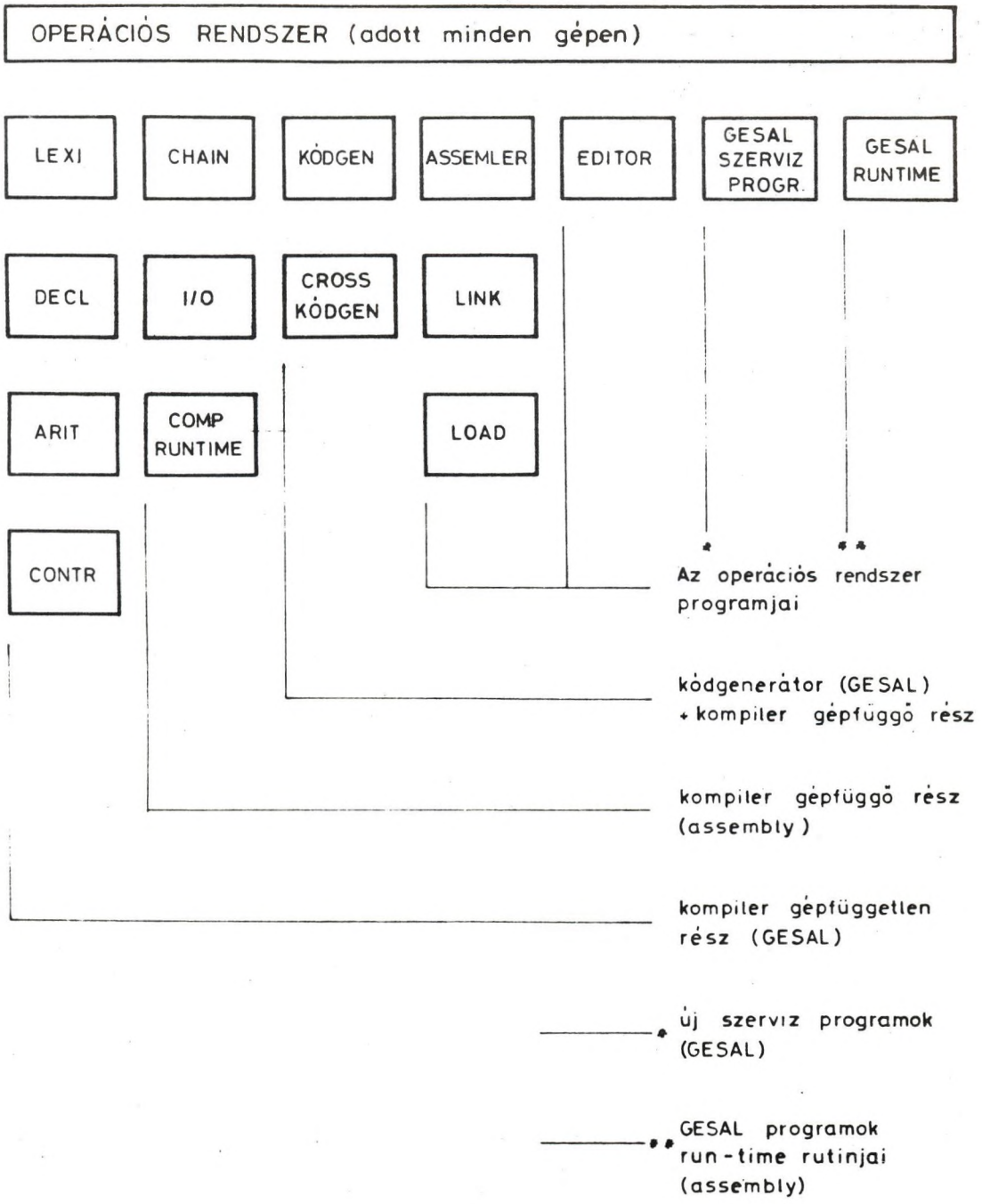
- ugy éreztük - közel áll egy általunk megcélzott kisszámitógép-tipus architekturájához.
Az IMC egy olyan gép utasitásrendszerének fogható fel, amely a postfix jelöléssel leirt kifejezések operandusait egy stacke tárolja /tipusukkal együtt/ és a müveleteket a stack tetején hajtja végre. A program strukturáját /feltételes utasitások, cimkék stb./ is ez a stack tárolja, csupán a FOR és CASE szerkezetek tárolására szolgál egy külön kiegészitő stack.
Az IMC program két részből áll: a deklarációs részből és végrehajtó részből. A deklarációs rész memóriaosztályokba /global, local, static-local, paraméter/ sorolva adja meg a deklarált objektumokat és esetleges kezdő értéküket.
A portabilitás másik feltétele a gépfüggő és gépfüggetlen részek szétválasztása. A portábilis kompiler ennek megfelelően négy GESAL szekcióból és egy assembly nyelvü szekcióból áll; az utóbbiba gyüjtöttük össze az összes gépfüggő eljárást és konstanst: elsősorban az input-output eljárásokat, és a kompiler fázisváltásait végrehajtó láncoló eljárást.
A kompiler átvitelekor egy uj gépre meg kell irni a kompiler gépfüggő részeit /amelyek specifikációja egy GESAL szekció specifikációjának formájában van megadva/, és kell irni egy kódgenerátort valamely létező kompiler mellé. Ez a kódgenerátor is irható GESAL-ban a kompiler gépfüggő részeinek felhasználásával.
A GESAL programok önálló forditási egysége a szekció. A leforditott szekciók összefüzésénél az adott gép software elemeit /Assembler, Link-editor/ használjuk. Ennek következtében a szekciók közötti hivatkozásoknál csak név szerinti illesztés /cim-hozzárendelés/ történik, tipusellenőrzés nem.
A kódgenerátor iró dönthet ugy, hogy az IMC egyes utasitásait nem végrehajtható utasitásokra, hanem run-time rutin hivásokra forditja. /A kompiler által is használt ilyen rutinok benne vannak a kompiler gépfüggő részében./



1. ábra. A GESAL program fordítása és kereszt-fordítása



2 ábra. A kompiler átvitele egy új gépre



3 ábra A GESAL programozási rendszer elemei

3. A GESAL NYELV A KÜLÖNBÖZŐ GÉPEKEN

A gépek és az operációs rendszereik közötti eltérések következtében a GESAL nyelv egyes tulajdonságait természetesen eltérő módon kell megvalósitani az egyes gépeken.
A gépek közötti eltéréseket négyféleképpen kezeljük. Az eltérések egy részét a programozó tudatosan önálló szekciókba lokalizálja, amelyeket minden gépre külön-külön meg kell irni, általában assembly nyelven /ilyen például az input/output/.
Az eltérések legnagyobb részét a kódgenerátor veszi figyelembe: a gépfüggetlen IMC-t gépfüggő kóddá alakitja. Ennek ellenére néhány géptől függő döntést már a kompilerbe be kell épiteni, igy már a programok IMC alakja is függhet a tárgygéptől /például a lebegőpontos konstansok ábrázolása, szavak és duplaszavak páros cimre illesztése stb./.
A nyelv definiciójával együtt csak azt irtuk elő, hogy a szekciók önállóan fordithatók és a program szekciókból összeállitható. Ennek megvalósitása gépenként különböző:
- a TPA70 DOST alatt a kompiler assembly kódot állit elő, és az assembler egyben összefüző /linkeditor/ is,
- az R10 IDOS alatt a kompiler egy közbülső, szerkeszthető formára fordit, és az összefüző assembly kódot állit elő,
- a PDP11 RSX11 alatt a kompiler Macro-assembly-re fordit,
- az IBM-en assembly kódot állit elő.

A mondottakba belegondolva kiderül, hogy például a TPA70-en /egyelőre/ nem lehet könyvtárból automatikusan kikeresni a hiányzó eljárásokat.
Több felhasználós gépeken - érthető módon - nem lehet fix cimre becimezni programrészeket. Igy azután a kompiler passok által közösen használt és interface file-on keresztül átadott deklarációs tábla, amely ref tipusu értékeket /cimeket/ is tartalmaz, a PDP11 multitask operációs rendszere alatt kellemetlen perceket szerzett a kompiler iróinak.
A szavas müveletek nagy része minden gépre egyértelmüen egy gépi utasitásra fordítható. Ezzel szemben a byte-müveletek /pl. az R10-en/ és a longint-müveletek /INTEL/ nem minden gé-

pen találhatók meg. Ilyen esetben a kódgenerátor irója választ az utasitás értelmező run-time rutinok, és a megfelelő makró kód kifejtés között.
A PDP11 és az INTEL8080-on egy szó két byte-ja forditott sorrendben helyezkedik el. Minden program, amely unióba hoz egy byte-os és egy szavas tipusu objektumot, a szavas értékadás után forditva veheti csak elő a két byte értékét.
A kompiler gépfüggő részében lévő globális változó tartalmazza a gazdagép számkódját; innen tudhatja meg a kompiler, hogy milyen gépen fut.
A tárgygép feltételezés szerint ugyanez, ellenkező esetben azt a kompilerrel közölni kell. Ettől függően választja meg a kompiler a lebegőpontos számok ábrázolását, az alaptipusok hosszát /a ref tipus az IBM-et 4 byte, másutt 2 byte/ és "alignement kódját". Az utóbbi szerint kell esetleg szavas objektumokat páros cimre, duplaszavakat 4-gyel osztható cimre helyezni; ezeket már a kompiler elintézi.
A kódgenerátor dönti el az egyes memóriaosztályok tárolásmódját. Kézenfekvő a lokális változókat stackre helyezni - ahol ilyen van. A TPA és a PDP gépen a deklarált objektumok deklarációik sorrendjében helyezkednek el, az R10 offset tartomány kis mérete miatt külön kerülnek a skalárok és az összetett /tömb, struktura/ objektumok /ezért nem irja elő a nyelv a deklarációk sorrendjében való tárolást/.
Eltérő módon kell kezelni a regisztereket is az egyes kódgenerátorokban. Az INTEL 8080 négy darab byte-os regisztere nem ad lehetőséget különösebb fejtörésre. Az IBM-en lehet válogatni az összetett adatok cimzése, és a stack tetejének szimulálása között. TPA1140-en a kódgenerátor nyilvántartást vezet a regiszterek pillanatnyi tartalmáról, hogy a felesleges LOAD/STORE utasitásokat elkerülje.
Különleges megszoritásokat eredményez a ROM /read-only-memory/ használata az INTEL 8080 esetén.
Az ilyen programoknál külön szekcióba kerülnek az eljárások és konstansok, ill. a globális változók;az előzőek ROM-ba kerülnek /és igy nincs szükség betöltésre/ az utóbbiak viszont a közönséges memóriába.

A karakter konstansok az IMC-ben a gazdagép kódolásának megfelelően jelennek meg, ezért a kódgenerátoroknak szükség esetén az ASCII-EBCDIC konverzióról is gondoskodni kell keresztfordítás esetén.

4. BEFEJEZÉS

Az induláskor kritikusaink azt kérdezték, hogy miért nem BCPL, BLISS, Mary vagy PLll /ami egészen más/. Az utolsó lendületvételkor /1979/ azt kérdezték, miért nem MODULA, PASCAL, vagy miért nem várunk, amig az IRONMAN életre kel. Ma azt lehetne kérdezni, miért nem ADA vagy C. Ugy érezzük, hogy a GESAL ezek közül legközelebb a C-hez van /ill. az ADA egy nagyon leszükitett részéhez/. GESAL-nak bizonnyal hátránya, hogy nem világszerte ismert publikációs nyelv. Előnye, hogy sajátunk, ugy bővitjük, ahogy akarjuk.
Nyilvánvalóan más megközelitése ez a programozásnak, mint a CDL2-ANSWER irányzat.
Érdemes volt? - kérdezzük magunktól. Azt hisszük, igen.
A döntés 1976-ban - azt hisszük - helyes volt. Hasonlóan 1979-ben; csupán a megvalósitás huzódott el tul soká. Ha ma kellene kezdeni, valószinüleg másként döntenénk.
Végezetül megemlitünk még néhány, nem szabványos GESAL kinövést.
Készült egy virtuális tárkezelő kódgenerátor és szekciókönyvtár a TPA70-re. Ez lehetővé teszi, hogy a programszekcióknak mindig a legutolsó változata használódjon, annak is csak a szükséges lapjai és a cimek run-time cimződnek be.
Készült egy egyszerü multi-task GESAL, amely eljárás hivásokon és konvenciók önkéntes betartásán alapul. Annak következtében, hogy nincsn GESAL I/O, eddig már legalább négyfajta assembly nyelvü I/O szekció készült. A tapasztalat szerint ezek megirása egy másik gépen nem jelent elvi problémát.
Dinamikus memóriagazdálkodás, multitasking csak ext eljárások /és változók/ meghivásán keresztül valósitható meg.
A szekciók közötti kapcsolat a forditást követő összefüzéskor

jön létre. A kompiler ellenőrzi az ext objektumok helyes használatát azok ext specifikációja szerint a szekció önálló fordításakor. Az összefüző összerendeli a cimeket az ext és entry nevek alapján. Ekkor azonban tipusellenőrzés már nincs.

Az előadás szerzőin kivül a GESAL kompiler kialakitásában még a következők vettek részt:
Gerhardt Géza/ó-GESAL nyelvdefinició és kompiler/,
Farkas Ernő /nyelvdefinició/, Déri Gábor /INTEL 8080/,
Váradi Tamás /TPA1140, VIRTUAL/, Megyeri László /R10/.
Köszönet illet mindenkit, aki vállalta a félkész kompiler kinjait, közülük is elsősorban Andor László kollégánkat, aki az átlagnál is nagyobb részt vállalt a hibakeresésben.

Abstract: A portable programming language, GESAL, proper for both system and user programming of minicomputers was developed and transported to five machines in the Computer and Automation Institute of Hungarian Academy of Sciences, between 1976-80. This paper is about the GESAL language and its portable implementation. After a short review of the language, the structure of the compiler, the intermediate code, the code generators,the special difficulties of the portability and experiences of the project will be discussed here.

6. Irodalomjegyzék

[1] G. Gerhardt: Design and Implementation of a High Level System Programming Language, Proc. of the Second Hungarian Computer Conf. 1976.

[2] D. M. Ritchie: C Reference Maual, Bell Telephone Laboratories Murray-Hill, New York, 1974.

[3] A. van Wijngarden, B.J. Mailloux, J.E. Peck, C.H.A. Koster: Report on the Algorithmic Language ALGOL 68, Num. Math. 14, 1969.

[4] N. Wirth: The Programming Language PASCAL Acta Informatica, 1. 1971.

[5] M. Rain et al.: MARY - A Portable Machine Oriented Programming Language, Machine Oriented Languages Bulletin 3, 1973.

Fabók Julianna, Hermann Gábor, Kovács József, Krammer Gergely
MTA Számitástechnikai és Automatizálási Kutató Intézet,
Budapest, XI. Kende u. 13-17.

Farkas Ernő

AZ ADA FORDITÓ KÓDGENERÁLÁSI MÓDSZERE

Ez a tanulmány áttekintést ad az ADA nyelv implementálásánál választott kódgenerálási módszerünkről.
Módszerünk alapja az, hogy bevezettük az A-gép fogalmát. Az A-gép egy közbülső szint az ADA és a konkrét gépek között. Módszerünk ujszerüségét az adja, hogy formális definiciót adunk az A-gép kódgenerátorára és magára az A-kódra is. További érdekesség az interface függvények és az algoritmikus interface alkalmazása.
A bevezetés az alapelveket ismerteti. Az első fejezet a forditó előző meneteivel való kapcsolódással, a 2. fejezet az absztrakt kódgenerátorral, a 3-4. fejezet az A-géppel, illetve annak konkrét gépi implementációjával foglalkozik.
Kulcsszavak: programozási nyelvek, ADA, kódgenerálás, szemantika definició.

BEVEZETÉS

A harmadik menet feladata az, hogy a második menet által létrehozott attributumos fa alaku programból futtatható programot állitson elő. Ez a feladat két problémát vet fel: először a rendelkezésünkre álló nagy mennyiségü, de hiányos és ellentmondásos dokumentumból meg kell állapitanunk az ADA nyelv szemantikáját, majd ezt igen eltérő gépeken meg kell valósitanunk. Az utóbbi problémát ugy oldottuk meg, hogy bevezettük a hipotetikus A-gép fogalmát, amely az ADA nyelv igényeihez közel álló memória, task és exception kezeléssel rendelkezik. Első lépésben erre a kódra forditunk, majd ezt a kódot fordithatjuk tovább a konkrét célgépekre vagy interpretálhatjuk a célgépen.

Az A-kód tehát nemcsak hipotetikusan létezik, hanem van A-kódot /amely szimbolikus assembly-szerü/ előállitó gódgenerátorunk is. Ennek az az előnye, hogy az A-kód továbbforditásával vagy interpretálásával az ADA nyelvü programok további gépeken is végrehajthatók lehetnek.
A kódgenerátort először absztrakt matematikai formában un. absztrakt kódgenerátorként irjuk le. Ez a leirás egyben egy olyan teljes formális nyelvleirásként szolgál, amely a korábbi definicióktól eltérően a parallelizmust is leirja. Magára az A-kódra informális és formális definiciót adunk. Igy az A-kód generátora mintául és pontos definicióul szolgálhat más /közvetlenül a célgépek kódjára fordítható/ kódgenerátorok elkészitéséhez.
A kódgenerátor technikai érdekessége, hogy az egyes programrészek a müködésükhöz szükséges információkat nem közvetlenül adat formájában veszik át és adják tovább, hanem interface függvényeken keresztül. Ez egyrészt lehetővé teszi az információ magasabb szintü kezelését, másrészt meglehetős szabadságot ad a programrészek közötti információk adatként való tárolásában, végül bizonyos esetekben lehetőség nyilik arra, hogy ezeket az információkat ne tároljuk le, hanem a programrészek rutinhivásokkal kapcsolódjanak egymáshoz és az információkat paraméterként adják át. Ezt az utolsó esetet algoritmikus interface-nek nevezi az irodalom [6].

1. A KÓDGENERÁTOR KAPCSOLÓDÁSA AZ ELŐZŐ MENETEKHEZ

A fordító első két menete a programot egy attributumos fává transzformálja. Az attributumos fa egy olyan szintakszisfa, amelyben az egyes szintaktikai egységeket jelképező csomópontokban más, elsősorban szemantikus tulajdonságok is fel vannak tüntetve.
A kódgenerátor szempontjából a fa reprezentációja érdektelen, mert a fához és a fában elhelyezett attributumokhoz csak az interface függvényeken keresztül férünk hozzá. Ez a fa kezelésében magasabb szintet és nagyobb rugalmasságot biztosit, különösen változtatások, javitások esetén.

A fa kezelésére a következő függvényeket használhatjuk:

a/ Context-free interface függvények

A context-free interface függvények egyrészt fabejáró függvények, amelyek lehetővé teszik, hogy egy szintaktikai egységet jelképező csomópontból eljussunk egy annak részét jelentő csomópontba. Másrészt olyan predikátumok, amellyel rá lehet kérdezni egy csomópont szintaktikai kategóriájára. Pl.:
is-object-name (prefix (NODE))
Ha a NODE egy "selected component" tipusu szintaktikai egység, akkor ő egy prefixből és egy szelektorból áll. A "prefix" egy "object name" vagy egy "function call" tipusu szintaktikai egység lehet. A fenti példában erre kérdeztünk rá.
A kódgenerátor egy olyan fán dolgozik, amelyből a program szövege szemantikailag ekvivalensen rekonstruálható. Feltételezzük, hogy a korábbi menetek olyan standardizálásokat hajtanak végre a szövegen, amelyek lehetővé teszik, hogy különbözőképpen leirt, de azonos jelentésü konstrukciókat a kódgenerátor egységesen kezelhessen.

b/ Lexikai interface függvények

Az előző menetek olyan információkat is elhelyeznek a fában, mint pl. a szintaktikai egység forrásnyelvi poziciója, az azonositók karakteres alakja stb. Ezeket felhasználhatják pl. magas szintü nyomkövetés generálásához. Ilyen függvények pl.:
a-line-number (NODE)
a-source-code (NODE)

c/ Context-sensitive interface függvények

A statikus szemantika ellenőrzése során megállapitásra került az objektumok tipusa, feloldódik a függvénynevek tulterhelése, megtörténik a statikus kifejezések kiszámolása. A context-sensitive interface függvények az igy megállapitott eredmények visszanyerését teszik lehetővé. Pl.
is-static (NODE)
is-int-number (NODE)
a-value (NODE)

Itt például megállapitjuk a NODE-ról, hogy statikus kifejezés, ezen belül egészszám és lekérdezhetjük az értékét.

d/ Célgép- és implementáció-függő interface függvények

Ahhoz, hogy egy ADA nyelvü programból a célgépre kódot generáljunk, két különböző munkát kell elvégezni. Egyrészt meg kell határoznunk a program bizonyos szintaktikai egységének célgépfüggő attributumait, pl. a változók cimét, a mezők rekordon belüli elhelyezkedését, a tipusok gépi ábrázolását és hosszát bitekben stb. Ezeket az attributumokat egyrészt a célgép adottságai szabják meg, másrészt optimalizálási megfontolásokon alapuló implementációs döntések. Másrészt ezek alapján ki kell generálnunk a célgép kódját.

Az ADA nyelv olyan, hogy a statikus szemantikai ellenőrzések nem végezhetők el annak ismerete nélkül, hogy a programot milyen célgépre akarjuk leforditani és ott hogyan rögzitettük a predefiniált standard tipusok ábrázolását. Ezért a gépfüggő attributumok egy részét már a második menet kitölti a fában. Ilyen pl. a skalár tipusok gépi ábrázolása és hossza. A második menet azonban csak a statikus szemantika vizsgálatához szükséges attributumokat tölti ki.

A harmadik menet egy önálló része, a III/A fázis foglalkozik azzal, hogy a többi implementáció-függő attributumot beirja a fába. Erre kétféle lehetőség van. Egyrészt lehetséges egy olyan megoldás, amikor a kódgenerálás előtt a fát bejárjuk és az összes attributumot kitöltjük. Másrészt az ADA nyelv azt is lehetővé teszi, hogy mivel kódgenerálás céljából ugyis bejárjuk a fát, minden egyes csomópontban akkor töltsük ki az attributumokat, mielőtt kódot generálunk belőle. A kisebb gépeken az első, a nagyobb gépeken a második megoldás látszik előnyösebbnek.

A következő fejezetben, amikor a kódgenerálásról szükebb értelemben beszélünk, feltételezzük, hogy amikor egy szintaktikai egységből kódot generálunk, a gépfüggő attributumok ki vannak már töltve.

A célgép és implementáció-függő attributumok kitöltését és

kezelését az un. implementáció-függő modul végzi. Ez a modul jól el van választva a fordítóprogram többi gépfüggetlen részétől. A modul része a második és harmadik menetnek is.

2. AZ ABSZTRAKT KÓDGENERÁTOR

2.a/ Az absztrakt kódgenerátor felépitése

Az absztrakt kódgenerátor matematikailag egy függvény, amely bemenetül egy forditási egységet leiró fát kap és kimenetül egy célgépi kódsorozatot /esetünkben pl. egy A-kódu modult/ ad.
Ennek a függvénynek a definicióját egy egyenlet segitségével adjuk meg. Az egyenlet jobb oldalán a következő függvények szerepelnek:

"gen" függvények
olyan generátor függvények, amelyek egy szintaktikai egységből generálnak kódot. Maga a kódgenerátor tulajdonképpen a gen-compilation-unit függvény.

"is" függvények
olyan predikátumok, amelyekkel attributumokból következő predikátumok teljesülésére vagy nem teljesülésére lehet rákérdezni.

"a" függvények
olyan segédfüggvények, amelyek az attributumok értékét adják vissza.

"put" függvények
olyan függvények, amelyek célkódot állitanak elő és tesznek outputra.

Természetesen a jobb oldalon álló függvények egy része is eléggé bonyolult és igy azokat ujabb egyenletekkel definiáljuk, mindaddig, amig olyan definiciót nem kapunk, amelynek jobb oldalán csak az 1. fejezetben emlitett basic-interface függvények és "put" függvények szerepelnek.
Általában igaz, hogy amikor egy egyenlet bal oldalán egy szintaktikus egység szerepel paraméterként, akkor a jobb oldalon is csak ugyanerre a szintaktikus egységre, illetve ennek részegységére hivatkozhatunk.

2. b/ Az absztrakt kódgenerátor formalizmusa

Egy "gen" függvényt leiró egyenlet jobb oldalán egy if-then--elsif-then-...-else alaku konstrukció áll, amelyet igy jelölünk:

x → y; u → v; ...; z.

Az x és u helyén egy-egy predikátumnak kell állnia, az y, v, z helyén pedig "gen" és "put" függvények sorozata áll, amelyek az adott sorrendben kódot generálnak az outputra. A sorozat egy eleme maga is lehet egy feltételes kifejezés is. Az "a" függvények definiciójában y, v, z helyén "a" függvények, az "is" függvények definiciójában "is" függvények /illetve ilyen jellegü feltételes kifejezések vagy konstansok/ állnak.

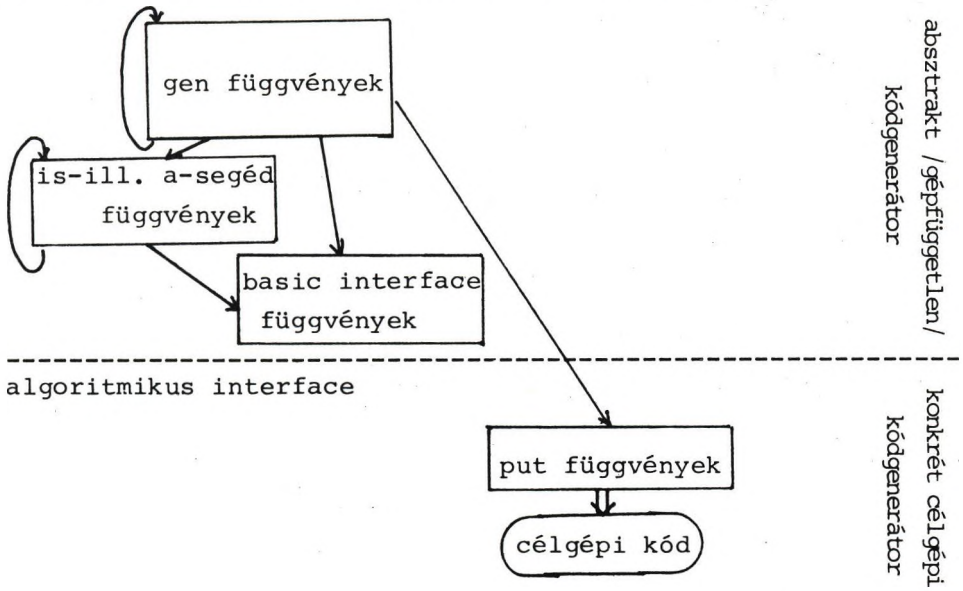
Az olvashatóság kedvéért kommentárokat is alkalmazunk, ezeket idézőjelbe tesszük. Pl.

```
gen-value (STATIC EXPRESSION)  =
is-scalar-or-access(a-type(ST.EXP))→
   put(LOAD-DIR, a-mode(a-type(ST.EXP)),
                          a-value(ST.EXP));
is-array(a-type(ST.EXP))→
   gen-load-addr(a-value(ST.EXP)),
   gen-load-addr(a-descriptor(a-type ST.EXP)));

"is-record"
   gen-load-addr(a-value(ST.EXP)),
   put(LOAD-DIR, length, a-length(a-type(ST.EXP))).
```

Ez azt jelenti, hogy ha egy statikus kifejezés értékét kell venni, ezt a következőképpen tesszük. Ha skalár, akkor egy direkt operandusu utasitással betöltjük az értékét. Ha tömb, betöltjük a kezdőcimét és a tömbleiró cimét. Végül, ha rekord, betöltjük a kezdőcimét és hosszát. A cim betöltése maga is többféleképpen történhet, attól függően, hogy hol helyezkedik el az adat, ezért nem put függvény hivás, hanem egy ujabb gen függvény hivás lesz.

A használt formalizmus definitiv jellege ellenére nem esik távol a CDL2 konstrukcióitól, a CDL2-be való átirás többé-kevésbé mechanikus munka.

Az absztrakt kódgenerátor felépitésének vázlata:



3. AZ A-KÓDGENERÁTOR, AZ A-KÓD ÉS AZ A-GÉP

Az A-kódgenerátor az A-gépnek mint célgépnek a kódgenerátora. Az A-kódgenerátor az absztrakt kódgenerátorból ugy jön létre, hogy az utóbbit leiró matematikai egyenleteket CDL2 programmá alakitjuk, és a put függvényeket ugy irjuk meg, hogy A-kódot generáljanak.

Amikor A-gépről beszélünk, tulajdonképpen nem egyetlen gépről van szó, hanem egy gépcsaládról. Ezek a gépek szervezésükben, memóriakezelésükben, utasitáskészletükben teljesen azonosak, különböznek azonban egymástól abban, hogy a gépi tipusokat milyen bit képben, milyen bithosszban ábrázolják, van-e kerekités vagy nincs stb. Ezeket a gépfüggő attributumokat már a második és harmadik menetnek ismernie kell és ezek figyelembevételével állitjuk elő az A-kódot a konkrét A-gépre. Igy elképzelhető, hogy egy adott A-gépre generált program a másik A-gép számára teljesen használhatatlan lesz, noha szintaktikailag tökéletesen helyes.

Az A-kódnak rögzitett a szintakszisa. /Tudjuk, hogy melyik utasitásnak hány, milyen jellegü, milyen sorrendü paramétere van./ Az A-utasitások szemantikáját először szövegesen irtuk le. Később, hogy egész pontosan definiáljuk, egy ADA nyelvü interpreterrel formálisan is megadtuk a müködést. Ez különösen a bonyolult utasitások esetén hasznos.
Az utasitások nagy része egyszerü, azaz egy másik számológépen 1-2-3 utasitással megvalósitható, de van néhány bonyolultabb is. Ide tartoznak a memóriakezelést és parallelizmust megvalósitó utasitások, amelyek még azzal a tulajdonsággal is rendelkeznek, hogy végrehajtásukat nem lehet megszakitani. Vannak továbbá olyan bonyolult utasitások is, amelyeket vissza lehetne vezetni egyszerübbekre, de gyakran előfordulnak, ezért implementációs szempontból célszerünek látszott bevezetésük. Ilyen pl. az indexelés, choice-listában való keresés stb.

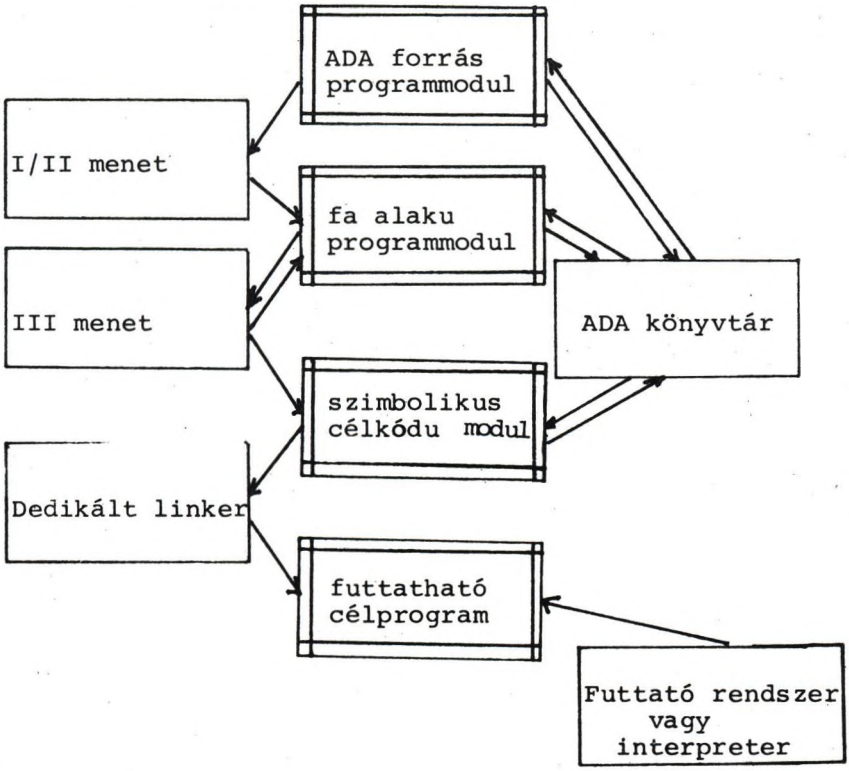
4. AZ ADA NYELV MEGVALÓSITÁSA A KONKRÉT GÉPEKEN

Az ADA nyelv kódgenerátora minden szóba jövő gépen az absztrakt kódgenerátoron alapszik, de a konkrét megoldások különbözők lehetnek.
Az egyik megoldás olyan, hogy az A-kódot az összeszerkesztés során egy tömör bináris alakra hozzuk és interpreter segitségével ezt hajtjuk végre a célgépen.
Egy másik megoldás az, hogy a "put" függvényeket ugy irjuk meg, hogy A utasitások helyett megfelelő célgépi utasitásokat, illetve futtató rendszer hivásokat generáljanak.
Egy harmadik megoldás az lehet, hogy az absztrakt kódgenerátor mintájára egy teljesen uj kódgenerátort irunk a fakezelő basic interface teljes megtartásával.
Az R11-es és TPA11/40-es gépen az első interpreteres változatot választottuk, legalábbis első közelitésben.
Az IBM implementáció első közelitésben a második megoldást választotta. Az ADA nyelvü programból IBM gépi kódu célprogramokat kivánnak előállitani, amely egy futtató rendszer segitségével lesz végrehajtható. A futtató rendszer végzi majd a memóriagazdálkodást, taskkezelést stb.

A kódgeneráláshoz kapcsolódóan kidolgoztunk egy szemétgyüjtő algoritmust is, amely az A-gépen alapul és opcionálisan a futtatási rendszerhez lesz kapcsolható.

5. ÖSSZEFOGLALÁS

Az ADA programoknak a célgépre való leforditását és futtatását a következő ábra szemlélteti.



Abstract: This study surveys our code generating method for ADA.
Our method is based on the idea of the A-machine. The A-machine represents an intermediate level between ADA and the concrete computers. The novelty in our work is that we give a formal definition for the generator of the A-code and for the A-code, too. Also an interesting point is the extensive use of the interface functions and the algorithmic interface.

The Introduction outlines the basic ideas. The first chapter deals with the connections to the previous passes. The second reviews the abstract code generator. Chapters 3 and 4 present the A-machine and its realisation on concrete computers.

Irodalomjegyzék

1 Reference Manual for the Ada Programming Language. United States Department of Defense, July 1980. Proposed Standard Document.

2 Formal Definition of the Ada Programming Language. Honeywell Inc and Cii Honeywell and Inria, Nov. 1980. Preliminary Version for Public Review.

3 xxxxxx
Ole Dommergaard 1980-08-31.
/Ada definition in VDM/

4 Diana Reference Manual
Institut Feuer Informatic II, Universität Karlsruhe and Computer Science Department, Carnegie-Mellon University March 1981

5 ADA compiler validation implementators' guide
Softtech, Inc. October 1, 1980.

6 Description of an Interface to the Machine-Dependent Code Generator...
Afdeling Invoermatika, Katolieke Universiteit Nijmegen, Nederland 21/3-1979.

Füle Károly–Tóth Tamásné

ÜZEMELŐ ALKALMAZÁSI RENDSZEREK FELÜLVIZSGÁLATA ÉS KORSZERŰSÍTÉSE

(Beszámoló egy konkrét vállalkozásról)

Ebben az előadásban egy olyan vállalkozásról számolunk be, amelynek tárgya egy konkrét alkalmazási rendszer felülvizsgálata és korszerüsitése volt egy hazai vállalatnál. Korszerüségen itt elsősorban a programok erőforrásigényének csökkentését, üzemeltetési hatékonyságának javitását, ezen belül is főképpen futási idejük röviditését értjük.

Programok hatékonyságának javitásáról már ezen a fórumon is esett szó /1/. A vállalkozás újdonsága abban van, hogy a hatékonyság javitását nem soroljuk a programfejlesztési munka funkciói közé, hanem olyan önálló feladatnak tekintjük, amelyet már régóta üzemelő programokon végzünk el. Ilyen hozzáállás mellett a programok hatékonyságának javitása valós és megbizható üzemelési adatokra támaszkodhat, gazdaságossági számitással is értékelhető megtakaritásra törekedhet.

Kulcsszavak: alkalmazási rendszerek, program hatékonysága, optimalizálás, tyuning

1. Helyzetelemzés

A müködő rendszerek felújitása összhangban van a korszerü programfejlesztési technológiák alapelveivel. Ezek szerint először müködő rendszert kell megcélozni, és csak azután kell a rendszer "optimalizálását" müsorra tüzni. Esetünkben a müködő rendszert már nem kellett létrehozni, és a rendszer mü-

ködéséről szerzett alapos információk alapján láthattunk neki a felújitásnak.

A korszerü programfejlesztési kultúra terjesztése minden olyan intézmény hivatásszerü feladata, amelyik technológiák kidolgozásával és alkalmazásával foglalkozik. Stratégiai szempontból az ilyen rendszerkorszerüsitési vállalkozások pillanatnyilag azért előnyösek, mert fokozatosan felkészithetik a vállalati számitóközpontokban folyó programfejlesztési munkát a korszerü technológiák által igényelt minőségi ugrásra.

Az üzemelő alkalmazási rendszerek felülvizsgálata és korszerüsitése olyan új profilnak tekinthető, amelyet a számitástechnika ellentmondásos fejlődése hivott életre. Ezzel kapcsolatban három "miért"-re próbálunk meg választ adni:

- Miért szükséges?
- Miért lehetséges?
- Miért érdemes?

1.1 Miért szükséges

Mindenekelőtt el kell ismernünk, hogy lényegében igaza van annak, aki azt hangoztatja, minek bántani az olyan programot, amelyik évek óta hibátlanul üzemel? Aki ezt mondja, az is jól tudja, hogy üzemeltetését illetően elég sok a kimondott és a kimondatlan kifogás az ilyen "jó" programmal szemben is. Ahol sok ilyen programot és régóta üzemeltetnek, főleg ott érzik jól, hogy egyik-másik program a többihez képest valahogy túl lassú, kiszolgálása túl körülményes, fölösleges adatokat is igényel, hasznositatlan eredményeket is produkál. Az ilyen program fölöslegesen sok erőforrást igényel, illetve köt le, nehézkes átfutása adott esetben vállalatirányitási problémákat is fölvet.

1.2 Miért lehetséges?

Közhely, hogy nincsen olyan program, amelynél jobbat nem lehetne irni; csak az nem biztos, hogy érdemes. Amikor átadja a kész programot, a programozó mindig azt érzi, hogy máris sokkal jobban irná meg, ha újra kezdhetné. Soha nincsen elég idő arra, hogy egy programot becsületesen irjon meg az ember.

Másik közhely, hogy a rendszerszervezőt szidjuk, mondván, hogy rossz specifikációból csak rossz programot lehet irni. Való igaz, hogy programjainkon érzik a valamikori erőlködés nyoma, hogy a gyenge specifikáció ellenére is elfogadható programot irjunk. A rendszerszervező bizonyos mértékig vétlen is, hiszen ma még igazából senki sem tudja a jó specifikáció általános receptjét; és ha tudná is valaki, akkor sem tudna a jobbára bizonytalan megrendelőtől minden szükséges információt kicsikarni. Valahogy úgy van ez, hogy a végleges specifikáció csak akkor alakul ki, amikor a program már régen üzemel.

Mindebből úgy tünik, hogy a program hatékonyságának javitása igazán csak akkor lehetséges, amikor a programnak már múltja is van. És ez esetünkben éppen igy van.

1.3 Miért érdemes?

Ami szükségesnek látszik, lehetségesnek is tünik, azt még nem feltétlenül érdemes megcsinálni. A programfejlesztési munka mai költségei mellett csak igen indokolt esetben érdemes egy programot újrairni, vagy akár csak gyökeresen átdolgozni is.

A programok felújitása kapcsán általában elzárkózunk a szolgáltatások átdolgozásától, azaz a programok továbbfejlesztésétől vagy módositásától. Bizonyos mértékig azonban az ilyen igényeknek elébe megyünk, amikor a nagyobb hatékonyság céljából éppen a programszerkezetet tesszük világosabbá vagy egyszerübbé.

A felújitás során nagyjából kétféleképpen avatkozunk be a program müködésébe:

- Igyekszünk finomitani a leggyakrabban végrehajtódó utasitáscsoportok és/vagy a leginkább időigényesnek ismert utasitások megfogalmazásán. Ehhez iránytüként a SZÁMKI-ban használatos MOZ-ART/MOZaik Alkalmazási Rendszer Technológia/ elnevezésü technológia /2//3/elveit és tapasztalatait használjuk, különös tekintettel a nyelvi ajánlásokra.
- Arra is törekszünk, hogy ráérezzünk, milyen fölösleges üzemeltetési, sőt szolgáltatási funkciókat lehet kis beavatkozással a kész programból kiiktatni, avagy a meglévőket milyen célszerübb funkciókkal lehet helyettesiteni, netán bőviteni. Ez már rendszerszervezőnek való feladat, sok gyakorlati tapasztalatot igényel.

Amikor néhány konkrét példát később ismertetünk, az is világossá lesz mindenki előtt, hogy az ilyen felújitási munka meglehetősen veszélytelen. És ami nagyon fontos, a megrendelő nem türelmetlen, addig is üzemel a régi változat.

2. A vállalkozás ismertetése

Megrendelő: Budapesti Tejipari Vállalat /BTV/
Számitóközpontja
Vállalkozó: Számitógépalkalmazási Kutató Intézet /SZÁMKI/
Programozási Rendszerek Főosztálya /PRF/
Programozásmódszertani Osztály
Időpont: 1980. IV. negyedév - 1981. I. negyedév

A vállalkozás ismertetése kapcsán az alábbi témaköröket részletezzük:

- Felkészülés és felmérés
- Tipikus beavatkozások
- Végrehajtás és értékelés

2.1 Felkészülés és felmérés

A BTV a SZÁMKI első partnere volt ebben a profilban. A vállalkozást a kölcsönös tájékozódás hivta életre.

A BTV a hatékonysági problémák meglehetősen széles spektrumát vázolta fel. "Bemelegitésként" a célokat végülis úgy korlátoztuk, hogy
- elsősorban a programok futási idejét kell csökkenteni;
- másodsorban a programokat áttekinthetőbbé kell tenni, erősen különböző megoldásaikat egymáshoz közeliteni kell.

A BTV által rendszeresen üzemeltetett alkalmazási rendszerek közül "első nekirugaszkodásra" az ún. értékesitési alrendszer havi modulját választottuk, ki. Ez a rendszer, amelyet havonta egyszer futtatnak, 5 nagyobb és 1 kis programból áll, a szabványos SORT/MERGE programokat nem számitva. A kiválasztás olyan szempontok szerint történt, hogy
- a rendszer jól képviselje a vázolt problémákat;
- ne legyen túl nagy; és
- gyakran előforduló tipusprogramokból álljon.

A részletes felülvizsgálathoz a BTV - a kért teljességgel - az alábbi adatokat szolgáltatta:
- rendszerdokumentáció/rendszerkoncepció, rendszerterv, felhasználási-programozási-üzemeltetési dokumentáció, I/O minták, forditási listák, joblisták, üzemeltetési leirások/;
- üzemeltetési statisztika/ I/O átlagos tételszám, rendezési kulcsok szerinti átlagos eloszlás, átlagos start/stop idő, átlagos CPU-idő, futtatási gyakoriság, átfutási problémák/.

Az átdolgozást az alábbi feltételek szerint terveztük:
- Az R20-as gépi konfiguráció, a DOS operációs rendszerkörnyezet, a D-szintü PL/I programnyelv, a programok inputja és outputja változatlan marad.
- A javasolt változtatásokat a vállalkozó forrásprogrambeli javitások formájában fogalmazza meg. A gyökeres átdolgozást

lehetőleg kerüljük.

- A javasolt változtatásokat a kijelölt programozóval előzetesen megbeszéljük. A javitások végrehajtása, az újraforditás, a tesztelés a programozó feladata. Az új programváltozatok esetleges forditási és futási hibáit a programozóval utólag megvizsgáljuk.
- Az új programváltozatokat a kijelölt programozók az aktuális élesanyagon is lefuttatják, és a felújitás hatását a régi és az új változat hatékonysági jellemzőivel mérik.

2.2 Tipikus beavatkozások

A szemléltetés kedvéért néhány jellegzetes esetet kiemelünk:

- Igen gyakori funkció alkalmazási programoknál az elemi gyüjtés, egy vagy több összegfokozat szerint, az illető fokozatnak megfelelő elemi változó/k/ba. Több esetben a program futási idejét azáltal sikerült jelentősen csökkentenünk, hogy a gyüjtőként használt változók deklarációját PICTURE tipusuról FIXED DECIMAL tipusura cseréltük le.
- Előfordult az egyik programnál a táblázatos gyüjtés is, két, sőt háromdimenziós tömbökbe. Ezek a tömbök már FIXED DECIMAL tipusuak voltak, a program viszont az input rekordból PICTURE tipusú változókba beolvasott értékeket használta indexekként. Itt nyilván FIXED BINARY tipusú segédváltozókat kellett használnunk az indexelésre, és ezzel is számottevő futási időt sikerült megtakaritanunk.
- Több program közös funkciója volt a törzsadatállomány memóriabeli tárolása és felhasználása. A nem túl nagy - 3oo tételből álló - törzsadatállományt kártyáról a memóriába olvasta be a program, mindjárt a futás legelején. A tárolás mindig struktura-tömbben történt, a hozzáférés vagy egy tömbelemnek megfelelő segédstrukturából előzetes átvitel után, vagy közvetlenül a struktúra-tömbből indexelt struktúra-elemek révén. Először is, az aritmetikai müveletekhez használt törzsadat-értékeket deklaráltuk át PICTURE tipusuról FIXED DECIMAL tipusura, az index-müveletekhez használt törzsadat-értékeket pedig FIXED BINARY tipusura. A törzs-

adat-értékek ismételt felhasználása esetén a konverzió igy elmarad, illetve a beolvasás és a struktúra-tömb feltöltése kapcsán szükséges egyszeri konverzióra korlátozódik. Másodszor pedig, segédváltozókat használtunk ott, ahol a program többször is számolt ugyanazzal az indexelt törzsadat-értékkel.

- Szinte mindegyik alkalmazási program többszáz lapnyi listát készit. A listázási funkció megoldásánál is sok lehetőség kinálkozott a gyorsitásra. A programozók előszeretettel alkalmazták a PUT EDIT utasitást, ami sokat nyomtató program esetében nem a legszerencsésebb dolog. Ehelyett azt a legcélszerübbnek látszó megoldást javasoltuk, amely szerint a listában előforduló valamennyi sortipusra - beleértve a fejlécet, oszlopfejlécet, láblécet, stb. is - külön-külön struktúrát használjunk. Ilyenkor ugyanis az illető sortipus vizszintes tagolódását egyszer dolgozza ki a program, az állandó mezőket - szóközöket vagy szöveges részt - elég egyszer betöltenünk, és a változó mezőket is csak akkor és csak ott irjuk felül, amikor és ahol azok valóban változnak. A struktúrákban az egyes mezőket úgy is deklarálhatjuk, hogy a számitáshoz használt változóval történő értékadáskor a szükséges szerkesztés is megtörténjen. Az illető sortipusnak megfelelő struktúrát - a változó mezők átirása után - WRITE FROM /strukturanév/ vagy PUT LIST/STRING/str.név// utasitással irjuk ki. Ebben a megoldásban lényegében a segédváltozó alkalmazásának ötletét teljesitettük ki a bonyolult listázási feladatra. Ugyancsak a listázáshoz kapcsolódik az a javaslatunk is, hogy a lapváltást és a soremelést lehetőleg az érdemi sor kiiratását végző utasitásban oldjuk meg. Ilyesmire ne irjunk külön utasitást /PUT PAGE, PUT SKIP/, csak akkor, ha ez föltétlenül szükséges /pl. a fejléc utolsó sora után/.
- Igen gyakori alapja volt a beavatkozásnak az a körülmény, hogy a program futását bizonyos utasitások fölösleges végrehajtása lassitotta. Az egyik tipikus jelenség: az összetett IF...THEN...ELSE utasitásokban a programozó figyelmen kivül hagyja azt, hogy a feltétel ellenőrzése is időbe ke-

rül, nemcsak az utasitás végrehajtása, Ahol a program egy változó értékétől függően nem kétfelé, hanem többfelé ágazik el, ott az újabb IF...THEN utasitás előtt az ELSE használata szinte kötelező. Arról nem is szólva, hogy az ilyen "case" tipusú feltételes szerkezeteknél a feltételeket legcélszerübb abban a sorrendben felirni, hogy először mindig a legvalószinübb esetre kérdezünk rá, aztán a kevésbé valószinüre, stb. Másik gyakori jelenség: gyüjtés közben, amikor kiirni még nem kell, fölösleges állandóan azokat az értékeket is kiszámitani, amelyek majd csak a kiiráshoz kellenek, például nem szabad árral szorzott mennyiséget gyüjteni, amikor az ár nem változik.

- Néhány esetben az alkalmazható lehetőségek felületes ismerete volt az oka az ügyetlen megoldásnak. Negálni úgy szinte tilos, hogy /-1/-gyel szorzunk, hanem az A = -A értékadást használjuk. Hasonlóan, a 1o, 1oo, 1ooo-rel való osztást is előnyösebb o.1, o.o1, o.oo1-del való szorzás útján elvégezni. Ugyanigy, negativ mennyiséget sem illik úgy gyüjteni, hogy először negáljuk, majd pozitivként gyüjtjük, hanem lehet a GY = GY - MENNY alakú utasitással is gyüjteni; avagy negativ mennyiséget PUT EDIT utasitásban pozitivként úgy is ki lehet irni, hogy a változólistába irjuk a negativ előjellel ellátott mennyiséget, fölösleges ezért külön segédváltozóban előállitani a pozitiv mennyiséget. Végül, a beépitett függvényeket sem célszerü úgy használni, mintha végrehajtásuk időt nem is igényelne; ha többször van szükségünk tökéletesen ugyanarra a függvényhivásra, érdemes az első hivás értékét segédváltozóba tenni, és azon keresztül többszörösen felhasználni.
- Szinte minden egyes program kinált olyan gyorsitási lehetőséget, amely az eljárások irása és hivása problémakörhöz sorolható. Igyekeztünk rámutatni arra, hogy az eljárás hivása is elég tekintélyes időt igényel, különösen több paraméter és automatikus változók használata esetén. Ezért több egymásbaskatulyázott eljáráshivást irtottunk ki, több közel azonos eljárást egyetlen közös eljárássá vontunk össze, sőt

a hivó program némi átalakitásával az eljárás törzsét az egyetlenre csökkentett hivási helyre közvetlenül beirtuk. Törekedtünk az eljárásparaméterek felszámolására is, ha ezek alkalmazása nem volt feltétlenül szükséges. Az eljárások automatikus változóit - bár ez elvileg vitatható - vagy statikus változókká deklaráltuk át, vagy pedig deklarációjukat áttettük a hivó programba. Az olyan megoldásokat, ahol az eljárás a meghivása után rögtön egy feltételtől függően általában kilép, csak speciális esetben dolgozik, átjavitottuk úgy, hogy a feltételt a hivó program ellenőrizze, és csak akkor hivja meg az eljárást, amikor annak valóban dolgoznia kell. Ilyesmire példa az olyan eljárás, amely lapot vált és fejlécet ir, de csak akkor, ha a lap már betelt.

3. Végrehajtás és értékelés

A BTV a vázolt munka lebonyolitásához a lehető legkörültekintőbben állt hozzá. Nemcsak biztositotta az extra munkához szükséges munkafeltételeket a programozói számára, hanem megfelelő anyagi elismerést is kilátásba helyezett. Megvalósitotta azt az elképzelését is, hogy a kitüntetett rendszer minden egyes programjának legyen gazdája, aki nemcsak az átdolgozást hajtja végre, hanem később az esetleges hibajavitáshoz vagy továbbfejlesztéshez is kéznél van.

A SZÁMKI részéről mindent megtettünk a programozók jóindulatának megnyeréséért. Többnapos tanfolyamot tartottunk részükre, amikor már alaposan megismertük a szóbanforgó programokat és gyenge pontjaikat. A tanfolyam első felében "szőr mentén" vázoltuk egyik-másik program alapstrukturáját a Jackson-féle programtervezési módszer szerint, összehasonlitva a hagyományos és a korszerü megoldás előnyeit-hátrányait. A tanfolyam másik felében a PL/I programozási stilus jegyében elemeztük a többször is előforduló stilushibákat és javitásukat. Az egyénenkénti megbeszélést ezután ugy szerveztük meg, hogy a programozók "házifeladatként" felkészülhessenek és elébe jö-

hessenek az általunk elővezetendő programjavitásoknak. Végül, a bizalom jegyében rájuk hagytuk, hogy a megbeszélt javitásokból mennyit és hogyan valósitanak meg.

A vállalkozás eredményességét a vizsgált programrendszer viszonylatában a mellékelt táblázat mutatja be. Az eredeti és a javitott változat jellemző élesanyagon történt futásának időadatai azt mutatják, hogy az alrendszer egyetlen futását 174 percről 129 percre sikerült csökkenteni, ami 45 perc, azaz 26 % megtakaritást jelent start/stop időben. A maximális megtakaritást az ERTEK programnál értük el, ahol a futási idő 15 perccel, azaz 5o %-kal csökkent. /A minimális futási idejü LE35AT-nevü segédprogrammal egyáltalán nem is foglalkoztunk./ Az alrendszer egyéves üzemeltetésénél igy 9 óra gépidőt takaritottunk meg, ami hetente vagy naponta futtatott rendszer mellett nyilván sokkal többre adódott volna. Ezt a nyereséget csaknem minden programnál nemkevés formális és egynéhány logikai javitással értük el. Csupán egyetlen program /FOLYO/ került ujrairásra, ez is inkább csak a jobb áttekinthetőség céljából. A vállalkozás eredményessége az áttekinthetőség-egyöntetüség szempontjából itt nem demonstrálható, de a forrásprogramokon határozottan meglátszik.

A BTV Számitóközpontjában a vállalkozás igazolta az elvárásokat, olyannyira, hogy az együttmüködést egy újabb alrendszerre vonatkozólag a jövőben folytatni kivánják. Szerintük a náluk folyó programozási munka most mozdult ki az ad hoc megoldások holtpontjáról és egy ujabb közös munka eredményeképpen a javulás várhatólag már el fogja érni a szükséges /ha nem is elégséges/ mértéket. Ilyen értelemben számukra csaknem közömbös a megtakaritott gépidő, a korszerüsitésre beruházott összeg, illetve a programrendszer várható élettartama alapján számitható megtérülési idő, annál is inkább, mert a felhasznált gépidő értéke náluk költségként közvetlenül nem jelentkezik.

Programnév	A módositás jellege és mértéke	Input tételszám	Futtatási idő módositás előtt	Futtatási idő módositás után	Futtatási idő megtakaritás időben	Futtatási idő megtakaritás %-osan	Futtatási idő megtakaritás 1 évre vonatkozóan
SZAMLA	formai kismértékü logikai	54612	69'	56'	13'	19 %	2 ó 36'
ESTAT	formai kismértékü logikai	59551	29'	22'	7'	24 %	1 ó 24'
FOLYO	újrairás	4141	12'	1o'	2'	17 %	ø ó 24'
ERTEK	formai kismértékü logikai	59551	3o'	15'	15'	5o %	3 ó øø'
LE35AT	formai kismértékü logikai	59551	6'	6'	-	-	-
JARLIS	formai kismértékü logikai	156726	28'	2o'	8'	29 %	1 ó 36'
Összesen:			174'	129'	45'	26 %	9 ó øø'

1. táblázat: Az átdolgozás eredményessége

A vállalkozás a SZÁMKI részéről is eredményesnek mondható. Egyértelmüen jogosnak bizonyult az a feltételezésünk, hogy van mit javitani a régóta üzemelő programokon, és hogy viszonylag kevés és veszélytelen beavatkozással elég látványos eredményeket lehet elérni. Szerintünk ebben a profilban "van fantázia", igy hasonló vállalkozásokba szivesen bocsátkozunk más vállalatoknál is.

Első vállalkozásról lévén szó ebben a profilban, messzemenő általánositásra nem vállalkozhatunk. Belső szándékunk szerint egy olyan "hangolási /tyuning/ technológia" csiráit keressük, amellyel már kész, sőt üzemelő programokat kézenfekvően hozzá tudunk igazitani az üzemi feladatok és a gépi környezet adottságaihoz. Egyelőre azonban csak ad hoc megoldásoknál tartunk és csupán többé-kevésbé ismert programozói fogásokról tudtunk beszámolni. Törekszünk arra is, hogy valamiféle "spefikációs technológia" fogásaira is ráérezzünk, nevezetesen arra, mikor és mennyiben nem elég csupán a megoldandó feladat algoritmusának és I/O adatainak mibenlétét előirni és hogyan lehet az üzemi feladat és a gépi-vállalati környezet egyéb adottságait megfogalmazni. Ilyen tekintetben munkánk sokban hozzájárulhat a MOZ-ART technológia kiterjesztéséhez azokra a munkafázisokra is, amelyeket ez a technológia egyelőre még érdemlegesen nem fed le.

Abstracts

This paper aims to report a project with the object of revising and modernising an application system in a Hungarian enterprise. By modernising a program we mean here principally the reduction of resource requirements, the improvement of the operational efficiency, practically the shortening of the run-time in most cases.

On this subject there was a similar paper read on the previous forum. The novelty of this project is in the fact, that the improvement of the efficiency is considered to be a stand-alone task performed on programs that have already been functioning for a long time rather than as one of the functions of the program development process. With this attitude, the improvement /or optimalization/ phase can be based on realistic and reliable operational statistics and can be controlled and properly limited by the use of conventional economical calculations.

Irodalomjegyzék

/1/ Dr. Obádovics J. Gyula: Programok hatékonyságának javitásáról, NJSZT, I. Országos Konferencia, Szeged, 1979.

/2/ Esztergár Zs., Havass M., Molnár M, Sunavecz M: A MOZ-ART programozási technológia, SZÁMKI, Budapest. 1981.

/3/ Sunavecz M: A MOZ-ART technológia alkalmazási tapasztalatai /esettanulmányok/, SOFTTECH D 45, SZÁMKI, Budapest, 1980.

Gilicze László

HONEYWELL REMOTE BATCH TERMINÁL KIFEJLESZTÉSE TPA (PDP) 11/40-EN

A közelmultban készült el a PDP 11/40 és a Honeywell számitógépek távadatátviteli illesztését megvalósitó szoftver rendszerünk. A fejlesztés a Budapesti Geodéziai és Térképészeti Vállalat és a KSH Államigazgatási Számitógépes Szolgálat együttmüködése keretében jött létre és első alkalmazásaként megteremti az alapszoftver feltételeket ahhoz, hogy az országos térképészeti adatbank osztott rendszerként kialakitható legyen. A kész szoftver nemcsak a PDP-n, hanem kompatibilitásuk miatt TPA 11/40-en hasonlóan müködik.

A PDP-n, RSX 11/M operációs rendszer alatt, felhasználói feladatként futó, a Honeywell-hez való illesztést biztositó szoftver egy teljes remote batch /távolsági kötegelt/ adatvég állomást helyettesit /emulál/. Kifejlesztése egyik eredménye az intézetünkben folytatott, a kisszámitógépes funkciók integrálására irányuló fejlesztéseinknek [6]. Ezt a rendszert, valamint fejlesztésének és próbaüzemelésének néhány tapasztalatát szeretném az alábbiakban bemutatni.

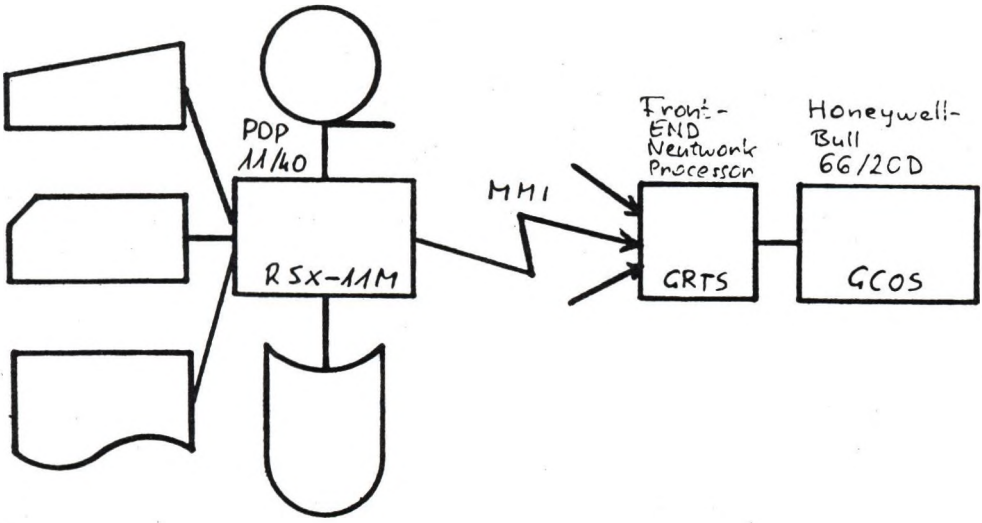
A remote batch terminál létrehozásának szempontjai

A két gép illesztésére vonatkozó igény felmerülésekor erre alkalmas software létezéséről nem volt tudomásunk. A mágnesszalagos adathordozón keresztül történő /off-line/ adatátvitel pedig nehézkes, lassú és igen megbizhatatlan volt. /Később

felmerült egy IBM 2780 terminál emulátor használatának lehetősége, de ezt az alternativát a várható hátrányok miatt elvetettük./

A remote batch terminál rendszertervének kialakitásakor nemcsak a jelenlegi, hanem a várható alkalmazási igényeket is figyelembe vettük. Legfontosabb volt a távoli job futtatás és a file-átvitel lehetőségeinek megteremtése a PDP 11/40-en, az üzemszerü alkalmazások lehető legkisebb mértékü zavarása mellett. Emellett természetesen alkalmazkodnunk kellett a meglévő hardware és software feltételekhez is.

A terminál rendszer hardver-szoftver környezetét az alábbi ábra szemlélteti:



Feltétel volt az is, hogy a Honeywell oldalon nem szabad változtatni, hiszen a nagygépen folyamatos üzemelés folyik, a front end hálózatvezérlő felügyelőprogramja /GRTS/ pedig enélkül is túlterhelt, a nagyszámú és különböző tipusú terminálok miatt. Adatátviteli protokollként ezért a Honeywell-en meglévő vonali interface-ek közül választottunk, mégpedig az

un. Multi Message Interface-t /MMI/ [1]. Ezzel a választással a kiindulási igények kielégitésén túlmenően jelentős további előnyökhöz jutottunk.

Az MMI két jelentős előnye a hazánkban használt hasonló /remote batch/ célú adatátviteli protokollokhoz képest, hogy:

a./ fizikailag ugyanazon a vonalon egyidőben kötegelt /batch/ és párbeszédes /interaktiv/ fogalmat is képes lebonyolitani. Igy lehetőség van rendszerünket a párbeszédes /pl. time-sharing/ terminálok koncentrálása funkcióval is bővíteni.

b./ fizikailag ugyanazon az adatátviteli vonalon több, ún. logikai vonal is forgalmazhat, és igy több felhasználó futtathatja jobját egyidőben rendszerünkről, úgy, mintha saját külön terminálnál dolgozna.

Ezeket a szolgáltatásokat az MMI hazai feltételek mellett /V24 interface/ képes megvalósitani.

A TPA ill. PDP 11/40 operációs rendszerei közül az RSX 11/M-et választottuk [2], amely jó arányt biztosit a valósidejü, többfelhasználós szolgáltatások és ezek erőforrásigénye között. Ez a megoldás a PDP-n keresztül sem szükiti le a Honeywell központi rendszeren rendelkezésre álló szolgáltatások körét, sőt bővíti és kényelmesebben felhasználhatóvá teszi azokat.

A GCOS és az RSX operációs rendszerek ilymódon történő összekapcsolásával lényegében olyan új környezet jön létre, amelyben egy RSX 11/M terminálnál ülő felhasználó szabadon dönthet arról, hogy például FORTRAN vagy BASIC programját a helyi kisgépen vagy a távoli Honeywell-en futtatja-e, és melyik gépen végzi el egy információs rendszer adatainak bevitelével, ellenőrzésével, az adatfile-ok átalakitásával, az adatbázis kezelésével kapcsolatos müveleteket.

A remote batch terminál rendszer felépitése

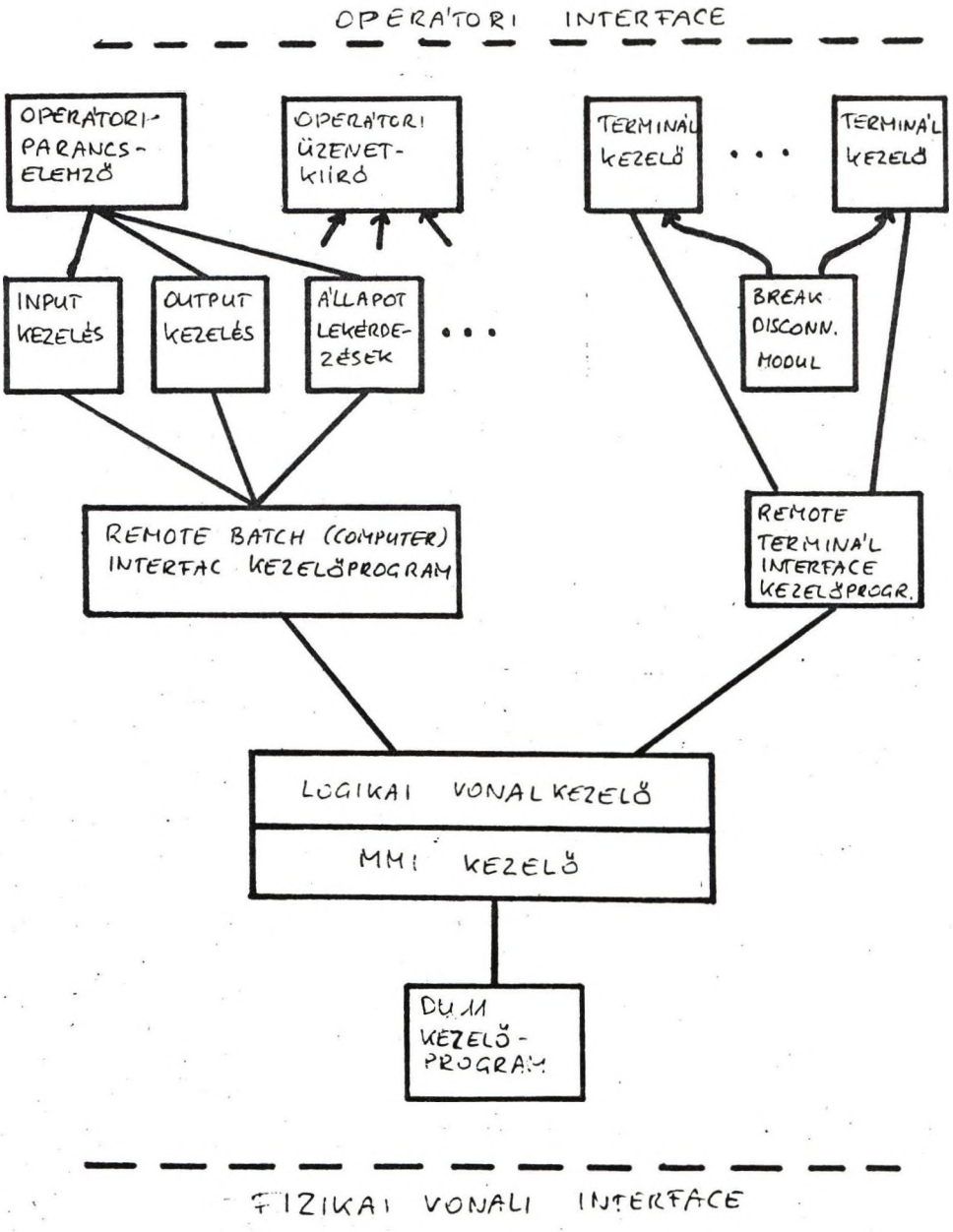
A terminál rendszer kb. 25 kisebb-nagyobb programból áll. A rendszer teljes mérete mintegy 50 Kszó, amely a pufferterület és a teszteléshez használt nyomkövetési rendszert is tartalmazza.

A rendszer - bonyolultsága ellenére - biztonságos, mert a részeit alkotó modulok a multi-tasking operációs rendszer alatt önálló task-okként futnak. A task-ok egymástól izoláltak,környezetük egymás ellen védett, igy az RSX védelmet nyújt a külső task-ok hibáival szemben. Sérülékeny pontot csak a közös dinamikus pufferterület és szubrutin-könyvtár jelent a rendszerben, amely a task-ok közötti üzenetáramláshoz szükséges. A task-ok egymással való koordinálásához az RSX megfelelő eszközöket biztosit [3], és csak néhány esetben kellett új megoldást keresnünk.

Ilyen új megoldás a dinamikus pufferkezelés, amelyet a hatékonyság érdekében külön kidolgoztunk, a pufferkezelésben előforduló kritikus szakaszok védelmére pedig - a párhuzamos futtatások várható kismértékü zavarásának elkerülésére - nem a megfelelő RSX eszköz /event flag/ [3], hanem saját szemafor szolgál.

Uj megoldás a szinkron vonali adapter /DU-11/ kezelőprogramja is.

A rendszer fő programjainak kapcsolatát az ábra mutatja:



A rendszer további jellemzői a könnyü kezelhetőséget, a párhuzamosságot, hatékonyságot és a fejlesztésekkel szembeni nyiltságot szolgálják.

A rendszer összméretéből is érzékelhető, hogy törekedtünk a legszélesebb körü igények kielégitésére. A jobok küldésére--fogadására nemcsak a kártyaolvasó és a nyomtató, hanem a mágneses perifériák is használhatók. A kiterjedt utasitáskészlet a jobok futtatásának teljes irányitását lehetővé teszi /állapotlekérdezések, abortálás, output átirányitás/, lehetőséget nyújt a kezelési és vonalhibák korrigálására és különböző statisztikák készithetők az üzemmenet követésére.

A kezelési lehetőségek kialakitásakor nagy előnyt jelentett az intézetünknél nagy alkalmazói népszerüségnek örvendő, hasonló funkciójú, eredeti Honeywell gyártmányú kisszámitógépen, a DN 707-en, négy éves üzemeltetési tapasztalatunk.

A viszonylagos nagy összméret ellenére rendszerünk már 16 Kszó memóriával is képes müködni, hiszen a multitasking megoldás következtében a rendszer egyes részei csak szükség esetén töltődnek be és a futó taskok is helyet adhatnak nagyobb prioritású taskok számára /roll-in-roll-out/ [4]. A sürgősebb helyi futtatásoknak is előny biztositható a prioritások helyes megválasztásával.

A rendszer hierarchikus felépitése lehetővé teszi a továbbfejlesztést, ez a task-hierarchia lényegében megfelel az ISO Nyilt Rendszer Architectura ajánlás [5] szintjeinek is, amely a távadatátviteli rendszerek szabványául tekinthető.

A rendszer elkészitésének tapasztalatai

A feladat felmerülésekor a szükséges feltételeknek csak egy töredék része állt rendelkezésünkre. Szükség lett volna többek között néhány tapasztalt rendszerprogramozóra, szakismeretekre a Honeywell, a távadatfeldolgozás, a PDP /TPA/ és az RSX témákban, valamint egy jól hozzáférhető /PDP/ TPA 11/40 kisszámitógépre lehetőleg a Honeywell környezetében. Hogyan sikerült mégis a szakterületek többségén kezdő kollek-

tivának - a menetközben felmerült további nehézségek ellenére - két év alatt megvalósitania a feladatot?

Legfontosabb eredménynek tartom, hogy sikerült kialakitani egy fiatal szakemberekből álló közösséget, akik érdeklődése pótolhatta a tapasztalatok hiányát és lelkesedésük kitartott a géphezférés későbbi nehézségei ellenére is.

A programirási teljesitmény a munka két évére számitva, 8 munkatárs 50 %-os ezirányú leterheltségét feltételezve napi 20 belőtt assembler utasitás /gépi szavakban mérve/. Ez nem tekinthető rossznak, ha a körülményeket figyelembe vesszük: közel egy éves felkészülési és tervezési idő, változó gépi konfiguráció, a becsültnél nagyobb dokumentációbeli pontatlanság és rendszerhiba arány, stb.

A tervezés során top-down, a megvalósitáskor bottom-up megközelitést alkalmaztunk. Ez bevált, néhány akadály elhárítása után a rendszer hamar müködni kezdett. Sok időt elvesztegettünk viszont ezután a rendszer tartós megbizható müködéséig. Ennek oka egyrészt az on-line tesztelés körülményeinek nehézségeiből adódott, amin jobb szervezéssel menetközben jelentősen javitani tudtunk. Másrészt a rendszerterv néhány részlete nem volt kellően kidolgozva, információ és tapasztalat, vagy egyszerüen idő hiányában. A rendszerteszt közben végzett módositások sokkal nagyobb késedelmet okoztak, mint a rendszertervezés során okoztak volna.

Jó tapasztalatot szereztünk a CDL nyelv alkalmazásával. A rendszer legbonyolultabb programja készült CDL-ben, és a CDL gépi implementáció néhány hibájának kijavitása után ez lett a legkönnyebben javitható és tesztelhető modul. A CDL-ben irt program mérete sem nagyobb lényegesen az assemblerben irottakénál, három összehasonlitási adatunk szerint 1,5-2-szeres, a program jellegétől függően. A programozási idő viszont lényegesen kisebb.

Uj eszközt kellett kidolgoznunk a rendszer müködésének tesztelésére. Az RSX alatt használható ODT [4] csak egyes task-ok önálló "belövésére" alkalmas. Több task együttmüködése, időzitések, szinkronizálási problémák és a tartós üzemeltetés teszteléséhez egy közös nyomkövetési programot készitettünk, amely a programok jelentősebb pontjaiban feljegyzett információkat kivánságra kinyomtatja. Jól bevált lehetősége a nyomkövető programnak, hogy több megadott task müködését - üzemelés közben - külön képernyőkön nyomon lehet követni.

A próbaüzemelések tapasztalata szerint a rendszerrel szemben támasztott követelmények teljesültek. Képes párhuzamosan müködni más RSX alatti futtatással, képes több felhasználó párhuzamos kiszolgálására. Könnyen kezelhető. 1200 és 2400 Bdos vonalsebesség mellett megbizhatóan müködik. A távoli job futtatások mellett használható file-átvitelre is. De néhány újabb megoldással /pl. a pufferhasználat optimalizálásával/ a hatékonyságot és a helyfoglalást jelentősen javitani lehet.

I R O D A L O M J E G Y Z É K

[1] RNP/FNP interface, Honeywell Information Systems Inc., 1975.

[2] PDP11 Software Handbook, Digital Equipment Corporation, 1980.

[3] RSX-11M Executive Reference Manual, Version 3, Digital Equipment Corporation.

[4] Introduction to RSX-11M, Digital Equipment Corporation.

[5] Reference Model of Open Systems Architecture, ISO /TC97/ SC16.

[6] Cilicze L.: Kisszámitógépes funkciók integrálásának tapasztalatai. SZEGED, PROGRAMOZÁSI RENDSZEREK' 78.

[7] Cserna Csaba: Az ÁSzSz távfeldolgozó hálózata. Információ Elektronika 1979/6.

[8] Yourdon, E.: Design of On-line Computer Systems, Prentice-Hall, 1972.

[9] Davies, D.W., Barber, D.L.: Communication Networks for Computers, John Wiley Sons Ltd. 1973.

Hámori István

TERMINÁL KOMMUNIKÁCIÓ RÖGZÍTÉSE ÉS LISTÁZÁSA SVS (TSO) TCAM KÖRNYEZETBEN

Rövid kivonat:
Az előadás a KSH/SZK-ban kifejlesztett program rendszert ismertet, amely segitségével SVS/TSO/TCAM környezetben rögzithető, majd pedig listázható a displayes terminál kommunikácó.

Az I/O üzenetek listázásának intervalluma különféleképpen jelölhető ki, TSO és batch módon egyaránt.

A rendszer alkalmazása hatékonyabbá, gyorsabbá teszi a TSO alatti programfejlesztő munkát /pl. teszt/, ezenkivül adatvédelmi feladatok megoldását segiti elő.

Kulcsszavak:

SVS TSO TCAM
I/O üzenetek rögzitése
displayes kommunikáció listázása

BEVEZETÉS

A TSO display terminálok input, output üzeneteinek háttértárolón való rögzitése és utólagos kilistázásának igénye alapvetően két szempontból merült fel:

- a terminál felhasználók részéről /TSO alatti javitások, tesztek, adatterületek, listák utólagos ellenőrzése, feldolgozása/
- üzemeltetési oldalról /adatvédelem céljára szükség esetén utólag ellenőrizhető bármely terminál adott időszakbeli tevékenysége/

A célnak megfelelő szolgáltatásu program termékek léteznek, pl. az IBM SPF /Structure Program Facility/, az IBM MVS Session Manager, de a részünkre az évekkel ezelőtt megrendelt SPF program termék a mai napig sem érkezett meg. Az 1 MByte-os központi egységünk egyébként is jelentősen korlátozza a program termék felhasználhatóságát. Igy a KSH Számitóközpont Software Fejlesztési osztályán elkészitettük a célra orientált hatékony program rendszert, amely a jelenlegi körülmények között is alkalmas a terminál kommunikáció rögzitésével és listázásával kapcsolatos igények kielégitésére. A fenti problémák esetén elképzelhető, hogy hasonló rendszer bevezetésének igénye egyes ESZR gépeken is felmerül majd.

A rendszer elemeinek ismertetése

A KSH Számitóközpontjában müködő IBM 370/155 számitógéphez a telekommunikációs rendszeren keresztül jelenleg az 1. ábra szerinti perifériák kapcsolódnak az IBM 3704 vezérlő egységén keresztül.

A terminálok és a központi egység közötti kommunikáció a TCAM /Telecommunications Access Method/MCP /Message Control Program/ rendszerprogramon keresztül bonyolódik le. A TCAM MCP fogadja, sorolja a terminálok üzeneteit, továbbitja a felhasználó /application/ programnak, illetve ezzel párhuzamosan hasonló módon kezeli az ellenirányu üzeneteket is. /2.ábra/. A felhasználó program a jelen esetben csak a TSO /Time Sharing Option/ program, de egyéb alkalmazásoknál más alkalmazási programok kiszolgálására is szükség lehet /'TSO/TCAM mixelt' rendszer/.

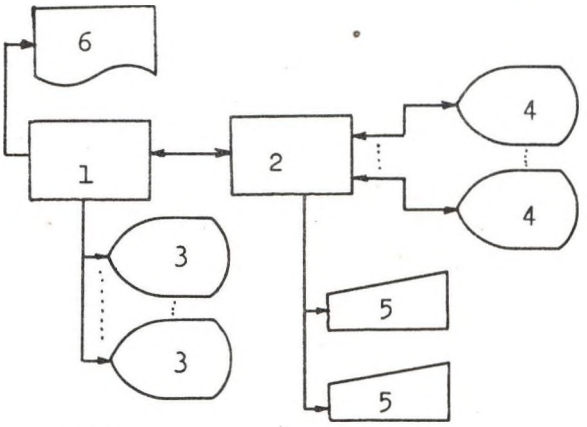
A TCAM rögzitett hosszuságu puffer egységekkel dolgozik, amelyból az üzenet hosszának megfelelő darabot láncol össze. Igy a TSO és a TCAM üzenetek darabszáma eltérő lehet. Pl: Egy hosszu TSO output üzenet több TCAM puffer egységbe kerül át, mig több rövid TSO output üzenet egy TCAM pufferbe kerülhet. Ezen kivül a TCAM üzenet tartalmaz még TCAM MCP belső információt is.

Minden terminálhoz tartozik egy QCB /Queue Control Block/, amely az adott terminál speciális adatait tartalmazza a TCAM MCP számára. Egy felhasználó QCB-je elérhető egyaránt a TSO és a TCAM oldalról is, igy a terminálokhoz rendelt QCB-k az üzenetek rögzitésével és listázásával kapcsolatos adatok TSO és TCAM MCP közötti átadásának fontos eszköze lett.

Feladat

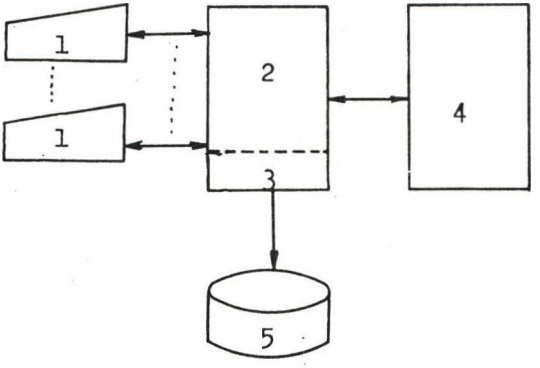
A programmal szemben támasztott főbb követelmények:

- minden terminál input üzenet rögzitésre kerüljön
- a terminál output üzenetek rögzitése terminálonként kapcsolható legyen
- a rögzitett kommunikáció TSO-ból és batchból is listázható legyen



1 IBM 370/155 CPU
2 IBM 3704 Communication controller
3 Local display /5 db/ /IBM 3275/
4 Remote display /16 db/ /IBM 3275,VDDS,VDT/
5 Remote CMC /2 db/

1. ábra



1 Local, Remote terminálok
2 TCAM MCP
3 Rögzitő program
4 Application program /TSO/
5 Disk

2. ábra

- adott illetőségi körön belül más terminál kommunikációja is listázható legyen
- a rögzitett kommunikáció archiválása
- a terminál I/O üzenetek eredeti formájukban kerüljenek listázásra

A fenti feladat megoldására két fő irány kinálkozik, vagy a TSO vagy pedig a TCAM oldalról való megközelités. Mi az utóbbit választottuk, tekintettel arra, hogy ez egy későbbi 'mixelt' - nem csak TSO alkalmazási program - TCAM rendszer esetén is alkalmas a telekommunikáció rögzitésére.

A kidolgozott rendszer három programból áll /3. ábra/
- rögzités kapcsoló program, amellyel az output rögzitést terminálonként be- és ki lehet kapcsolni
- rögzitő program, amely a TCAM üzeneteket megfelelően szelektálva és kiegészitve diszkre tárolja
- listázó program, amely a diszken rögzitett TCAM üzenetek válogatását, eredeti TSO üzenetre konvertálását és az outputra küldését végzi.

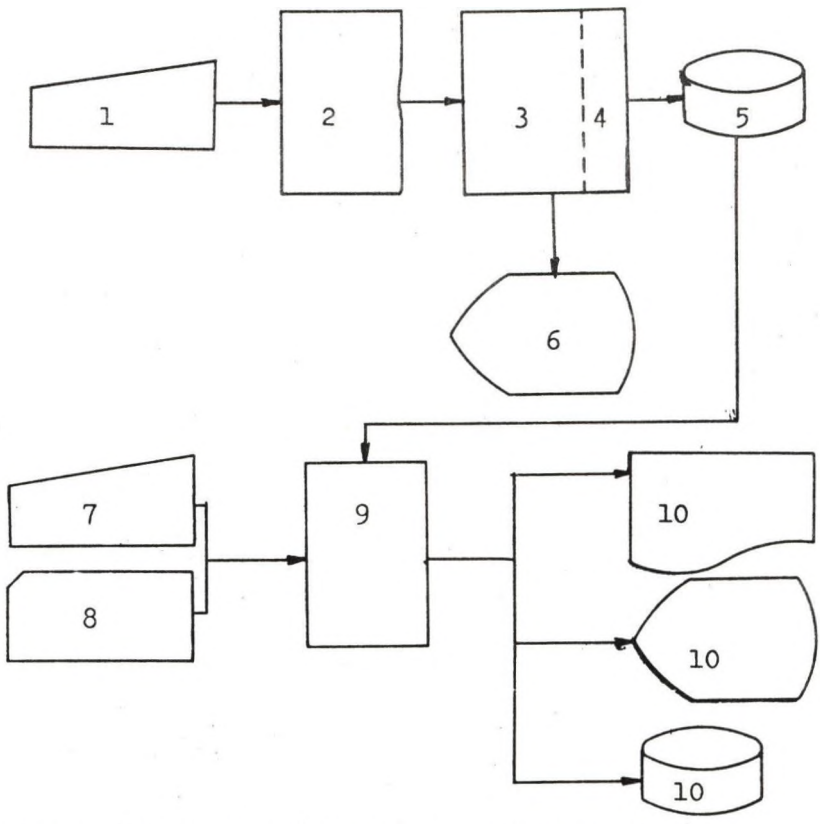
A rögzitést kapcsoló program

Ezzel a TSO utasitás feldolgozóval tudja a felhasználó az output üzenetek rögzitését be- és kikapcsolni. Az ehhez szükséges adatok beállitását és ellenőrzését végzi. TSO-ból a TCAM MCP számára.

A rögzitő program

Ez a TCAM MCP-be beépitett program az MCP-nek olyan pontján került elhelyezésre, amely lehetővé teszi az összes input és output üzenet ellenőrzését.

Minden egyes üzenetnél megvizsgálja a QCB-t és ha ott az üzenet tipus /I/O/ rögzitésére vonatkozó kérés van, akkor a TCAM puffer láncból az eredeti TSO üzenethez tartozó mindegyik puffer egységet kikeresi és egyenként ellátja a listázáshoz szükséges információkkal a terminálhoz rendelt QCB-ből.



1 Rögzitő bekapcsoló program /TSO/
2 TSO utasitás feldolgozó
3 TCAM MCP
4 Rögzitő program
5 Rögzitett állomány
6 Terminál output
7 Listázás kérés /TSO/
8 Listázás kérés /batch/
9 TSO utasitás feldolgozó és listázó
10 Listák

3. ábra

Például: - felhasználó azonositó /UID/
- üzenet dátuma, időpontja
- üzenet sorszáma a terminálon
- üzenet tipusa /input/output/,
- puffer egység tipusa /kezdő/folytató/befejező/

Ezután a fenti információkkal ellátott TCAM puffer egységek QSAM-mal az állandóan allokált és az első üzenetnél megnyitott 20 cilindernyi diszk állományba kerülnek rögzitésre. /Az eddigi üzemelés során ez az állomány elegendőnek bizonyult a napi 4-5 órai terminál kommunikáció rögzitésére./ A TCAM MCP a rögzitésre allokált állományt folytatólagos irásra nyitja meg, igy a nap közbeni esetleges ujabb TCAM inditások során rögzitett üzenetek az előző állományt folytatják.

A reggeli TSO inditás előtt kerül sor az előző napon rögzitett TCAM állomány szalagra másolására és diszkről való törlésére. /Archiválás tehát mágnesszalagon történik./

Hangsulyozni kivánjuk, hogy rögzitésre az eredeti TCAM puffer egységek kerülnek, kiegészitve a listázó program számára szükséges - már emlitett - információkkal. Ezen kivül a TCAM üzenet tartalmazza még a képernyőn nem látható kódokat is.

Felmerült az a gondolat, hogy a TCAM puffereket a rögzitő program szerkessze vissza a képernyőn megjelenő üzenetnek megfelelő formára és igy kerüljön rögzitésre.
Ezt a megoldást azért vetettük el, mert igy a szerkesztéssel járó többletmunka a véleményünk szerint kritikus TCAM programot terhelte volna. Ezzel szemben, ha a listázás során végezzük el ezt a tevékenységet, akkor elsősorban az adott TSO region-t terheljük és csakis a listázás lényegesen ritkább volumenében /a későbbiekben emlitésre kerülő batch listázásról nem is beszélve/.

Meg kivánjuk emliteni, hogy a próbaterhelések során bebizonyosodott, hogy még maximális rögzitési igény esetén sem okozza a rögzités a telekommunikációs rendszer észlelhető lassulását.

A listázó program

A listázó program TSO és batch módon egyaránt hivható utasitás feldolgozó programként készült el. A batch mód azért fontos, mert igy lehetőség nyilt a napi TSO időn kivül, utólagosan felmerült problémák tisztázására az adott display kommunikáció listázása révén.

A listázó program a kiválasztott rekordokból szerkeszti meg az eredeti TSO rekordokat; kiszürve belőle a képernyőn leolvasható információtól független adatokat.

TSO-ból történő hiváskor a listázás outputjaként választható az alábbiak közül:

- az adott display
- printer /X osztályu output/
- diszk állomány

Batch hivás esetén az output vagy printer /A osztály/, vagy pedig diszk állomány lehet. Printer esetén lehetőség van a remoteprinteren való listázásra is.

Terminálra történő listázás alatt - azon a terminálon - a kommunikáció rögzitése automatikusan szünetel a többszöri, felesleges rögzités elkerülése érdekében.

A listázás intervallumának definiálására a hiváskor az alábbi lehetőségek adottak:

- 'page'
- 'last'
- 'time'
- 'day'

'Page' forma esetén a megelőző 1-10 képernyő tartalom irható vissza.
'Last' forma esetén a megelőző 1-999 perc képernyő tartalma kerülhet kilistázásra.
'Time' forma esetén megadható a listázás kezdő és végső időpontja.
'Day' paraméter esetén pedig a listázás kezdő és befejező dátuma.

A 'page' és 'last' forma értelemszerüen csak TSO terminálról adható ki, mig a 'day'-nek csak batch-ban van szerepe, ugyanis az archivált szalagok üzemeltetés rend miatt TSO-ból nem érhetők el. Emiatt a néhány nappal előbb rögzitett kommunikáció csak batch uton listázható ki.

A listázási utasitásban megadható, hogy input és/vagy output sorok kerüljenek listázásra a rögzitett állományból.

Tekintettel arra, hogy kérésre a listázó az üzenetek elején kinyomtatja a rögzités időpontját, mód van arra, hogy először egy nagyobb időintervallumból csak a kevesebb input sort listázzuk ki, majd - megállapitva a szükebb időintervallumot - kerüljön a teljes kommunikáció kilistázásra.

Alkalmazás

- Az eddigi tapasztalatok szerint a terminál kommunikáció rögzitésének és listázásának hatása elsősorban a programok tesztelésénél jelentkezett. Tesztelési taktika változást jelentett az, hogy TSO idő után a program fázisok és adatterületek összefüggéseiben áttekinthetően "nyugodt" körülmények között álltak rendelkezésre.
- Egyes képernyőről inditott file müveletek adatainak utólagos ellenőrzése is lehetővé vált, sőt dokumentálható lett.
- Sor kerülhet nagyobb mennyiségü display-en editált input állomány megsemmisülése esetén az eredeti állománynak a rögzitett állományból való visszaállitására.

- Lehetővé vált a jelenleg fejlesztés alatt álló on-line adatbázis lekérdező rendszerrel kapcsolatos adatvédelmi feladatok ellátása. Az input üzenetek állandó rögzitése lehetővé teszi adott esetben egy adott terminál, ill. adott felhasználó tevékenységének visszamenőleges ellenőrzését.

Abstract

This paper describes a software program product of Hungarian Central Statistic Office Computing Center developped for logging and listing TSO display communication in SVS/TSO/ TCAM environment.

The listing interval of display I/O messages can be defined in various ways under TSO and batch mode.

The application of this software program product gives you more effectual and faster interactive program developping facilities /e.g. testing/ and helps in the data security system if needed.

Hámori István
KSH Számitóközpont
1024 Budapest
Budai László u. 1-3.

Hanák Péter–Rácz Gábor–Sarbó János

MIKROPROCESSZOROS SOFTWARE-TECHNOLÓGIAI RENDSZER

Az előadás 8-bites mikroprocesszorok, valamint PDP-11 számítógépek programozására alkalmazott software-technológiai rendszer kialakításának előzményeit és a rendszer alkalmazásának tapasztalatait ismerteti.
A technológiai rendszer a CDL (Compiler Description Language) nyelvre épül. Keresztfejlesztéses módszert alkalmaz, felhasználva a CDL-ben írt programok rendkívül jó hordozhatóságát.
Az ismertetett software-technológiát több éve alkalmazzuk; az eredmények azt mutatják, hogy az assembly programozásnál lényegesen jobb eredmények érhetők el vele.

Kulcsszavak:

mikroprocesszorok, CDL, software-technológia, programhordozás

A Budapesti Műszaki Egyetem Műszer és Méréstechnika Tanszékén 1977-ben kezdtük el egy software-technológiai rendszer kialakítását. A rendszert PDP-11 számítógépek, valamint (1979-től kezdve) 8 bites mikroprocesszorok (INTEL-8080, ZILOG-80) programozására alkalmazzuk. A fejlesztést támogató programok a Tanszéken levő PDP11/34 és PDP-11/45 számítógépeken futnak.

Cél

A technológiai rendszer megtervezésénél és kialakításánál alapvetően azt a célt tűztük ki, hogy a korábban alkalmazott assembly programozást korszerűbb, biztonságosabb módszerekkel váltsuk fől. A "technológiai" jelző arra utal, hogy a rendszernek egységes, elvi és gyakorlati támogatást kell nyújtania a mikroprocesszoros programfejlesztési folyamat minden fázisában - a programtervezéstől az élesztésen át a karbantartásig.

Problémák

Célberendezés

A Tanszéken nagy bonyolultságú és értékű, egyedi vagy kis-közepes sorozatban (10-100 példányban) gyártott mérőkészülékek, mérőrendszerek fejlesztésével foglalkozunk. A készülékprogramok mérete általában 8-20000 assembly sor közé esik. Gyakorlatilag minden készülékben van valamilyen speciális hardware, amelyet programból kell kezelni. A készülékprogramok egyedi jellege miatt a software karbantartásának (adaptálhatóság, módosíthatóság stb.) nagy jelentősége van.
A fejlesztendő készülékek software-je jelentős részben (40-60%-ban) általános célú rendszerprogram, amelyet egy processzortípus elavulása esetén szeretnénk átmenteni az új processzortípusra. A programok hordozhatósága ezért szintén nagy jelentőségű.

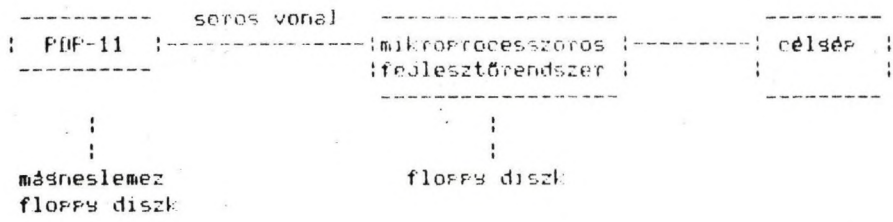
Programozási nyelv

A technológiai rendszerben alkalmazott programozási nyelvnek határozottan támogatnia kell a moduláris és strukturált programozást, valamint az öndokumentáló és kifejező programok írását. Az előbbi igényeknek megfelelő "magasszintű" nyelv ugyanakkor nem korlátozhatja a programozót a processzor, illetve az adott rendszer hardware-lehetőségeinek kihasználásában.

Fejlesztési módszer

Keresztmódszert kell alkalmazni a programfejlesztés minden fázisában, azaz nemcsak a programok fordításához, hanem a programélesztés nagy részét kitevő algoritmikus ellenőrzéshez is. Az élesztést alapvetően a magasszintű forrásnyelv szintjén (és nem gépi kódú szinten), lehetőleg interaktív eszközzel kell támogatni.

Hardware-környezet



1. ábra Hardware-környezet

Programhordozásra épített technológiai rendszer

A kifejlesztett software-technológia a CDL (Compiler Description Language) programozási nyelvre épül.
A CDL mint programozási nyelv önmagában is megoldja a programozás néhány fontos problémáját:

- határozottan támogatja a strukturált programozást, a lépésenkénti finomítást,
- konkrét alkalmazási tapasztalataink vannak arra vonatkozóan, hogy CDL-ben lehet közel assembly-hatékonyságú programot írni (mind a futási időt, mind pedig a memóriaterületet tekintve).

Az egész rendszer szempontjából a CDL-nek az a tulajdonsága a legjelentősebb, hogy a CDL-ben megírt programok hordozhatóak -- hatékonyságveszteség nélkül vihetők át az egyik processzortípusról egy másikra.
Ez a tulajdonság lehetővé teszi, hogy a programfejlesztés legidőigényesebb részét, a programelesztést egy kisszámítógépen végezzük el. A kisszámítógép méretei, sebessége és periféria-ellátottsága miatt alkalmasabb a fejlesztés hatékony támogatására, mint egy mikroprocesszoros fejlesztőrendszer (elsősorban az állandó újrafordítás igénye miatt).

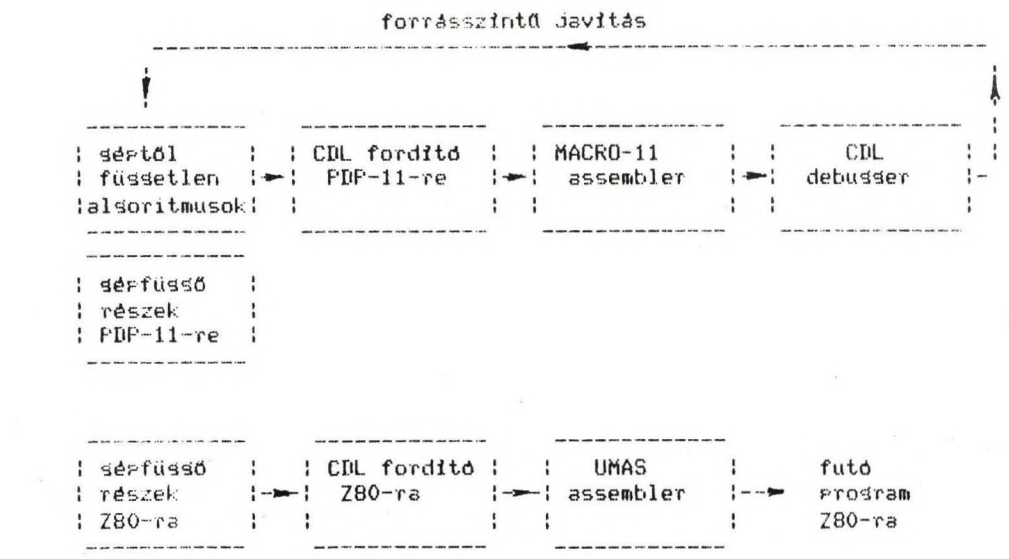
Software-környezet

PDP-11 (RT-11 vagy RSX-11M operációs rendszer)
segédprogramok: EDIT, LINK, PIP stb.
CDL fordítók: CDL --> MACRO11, CDL --> UMAS(Z80)
CDL-szintű debugger
Makróassembler (MACRO11)
Univerzális makróassembler (UMAS):
UMAS --> Z80, INTEL-8080, MOTOROLA-6800
Bázis I/O
Makrókönyvtárak

Mikroprocesszoros fejlesztőrendszer
Hardware-rel támogatott, gépikód-szintű debugger
Bázis I/O

Programfejlesztés

A mikroprocesszoros programfejlesztés folyamatát a 2. ábra mutatja:



2.ábra A mikroprocesszoros programfejlesztés folyamata

A programhordozásra épített technológia alkalmazásával a mikroprocesszoros programfejlesztés főbb lépései a következők:

1. A program megírása a fejlesztő (host) gépre.
2. A program felélesztése a host gépen (forrásnyelvi szinten, interaktívan).
3. A host gépen futó, algoritmikusan jó program ekvivalens változatának elkészítése a célgépre (azaz a gépfüggő részek átírása).
4. A program ellenőrzése a célgépen.

Mivel a célgép általában alkalmatlan software-fejlesztésre, sokkal egyszerűbb a célgépen egy már jó program átírt változatát ellenőrizni és az esetleges átírási hibákat megtalálni, mint egy még rossz program hibáit kideríteni.

Alkalmazások

A vázolt software-technológiát 1979. óta alkalmazzuk ZILOG-80 mikroprocesszorok programozására, s körülbelül 30 ezer sornyi CDL programot fejlesztettünk ki. A legfontosabbak ezek közül az alábbiak:

Megnevezés:	CDL-sor:	Makrósor:
operációs és file-kezelő rendszer	3500	1500
handler floppy diszkhez	1100	120
debugger a mikroprocesszoros fejlesztőrendszerhez	3000	800
makróassembler	10000	1500
szövegszerkesztő (text corrector)	4000	1500

A fenti programokon kívül CDL-ben készült a technológiai rendszer PDP-11-en futó minden eleme -- fordítók, I/O rendszer, nyelvi debugger, utóoptimalizálók stb.

Tapasztalatok

Programozás CDL-ben

Bár a CDL szintaxisa rendkívül egyszerű, néhány óra alatt megismerhető, a CDL által megkívánt gondolkodásmód elsajátítása akár 6-12 hónapig is eltarthat. Tapasztalatunk szerint gyakorlott programozók a nyelvet nehezebben szokják meg. Ez elsősorban annak tulajdonítható, hogy a CDL a hagyományos programozási nyelvektől eltérően nyílt nyelv, nem tartalmaz elemi algoritmusokat. Ennek megfelelően egy program elkészítésénél nem az a probléma, hogyan lehet valamit az adott nyelven kifejezni, hanem az, hogy hogyan lehet a szükséges nyelvet létrehozni. A CDL-ben való programozás során ugyanis alkalmas nyelvek (virtuális gépek) sorozatának létrehozásával oldunk meg feladatokat. E "fordított" gondolkodásmód elfogadása, illetve elfogadtatása lassú folyamat.

A CDL programozás során a gépfüggő algoritmusokat (a makrókat) assembly nyelven fogalmazzuk meg. A makrók egy része az egyes programokban közös, ezeket könyvtárazással vagy más módon a programok átvehetik. Az így kialakuló szabványos makrókészlet a program összes makró-algoritmusának rendszerprogramok esetén 70-90%-át, hardware-közeli programok esetén 40-60%-át is lefedheti. Ez nem jelenti a nyelv lezárását, hiszen a hiányzó makrókat esetenként, egyedileg kell megfogalmazni.

Az interaktiv debugger használata

A program élesztését a forrásprogram fogalmaival vezérelhetjük. Mivel CDL-ben kifejező programokat irunk, fontos, hogy a program elemeire a programozó által adott névvel lehessen hivatkozni. Ez elsősorban a vezérlési szerkezetre igaz. Mivel CDL-ben minden nem elemi adatot hozzáférési algoritmus kezel (amelyekről a debugger nem tud), az adatszerkezetre csak részben hivatkozhatunk név szerint.

A CDL programok vezérlési szerkezete szemléletesen és jól követhető. Mivel a hibák nagy része vésülis vezérlési hibává válik, a hiba ténye és körülbelüli oka hamar látszik. Ezután -- mivel a CDL algoritmusok (szabályok) kicsik és a program jól strukturált -- a hiba helyét is könnyű behatárolni, például felezéses algoritmussal. Részben a CDL programok áttekinthetőségéből adódik, hogy a módositások, javitások hatása, illetve kiterjedése jól kézbentartható. Jó esélyünk van rá, hogy egy hiba kijavitásánál nem teszünk újabbat a programba.

A moduláris forditás miatt a programok forditása elég gyors -- egy átlagos programnál 2-4 perc -- ezért a hibákat forrásszinten javitjuk (a debugger segitségével sem teszi lehetővé, hogy a memóriában bármit módositsunk). Igy a forrásszöveg mindig aktuális marad, ami utóbb sok bosszúságtól és felesleges munkától szabadit meg. A hibajavitásnak ez a gyors módja jelenleg csak a host (PDP-11) gépen áll rendelkezésre. A célrendszerben (a fejlesztőrendszeren) nincsenek meg a CDL technológia elemei (forditók, debugger stb.).

Az interaktiv debugger használatával a programok élesztési ideje drasztikusan lecsökkent. A kb. 10000 soros makróassembler 3 hét után már forditott, és újabb 3 hét múlva teljesen kész volt. Lényeges eredménynek tartjuk, hogy reális becslést tudunk adni az élesztési időre. Ezáltal ugyanis jelentősen megnőtt a munka biztonsága. Nellően nagy assembly programnál nem lehet megmondani, mennyi idő alatt élesztheto fel. A csúszás lehet fél év is -- legalábbis a mi gyakorlatunkban ez már többször előfordult. A CDL-technológia alkalmazásával az időbecslésünk sokkal pontosabb lett.

Programhordozás

A hordozás a makrók szemantikusan ekvivalens átirását jelenti. A program algoritmusainak csak kb. 10-30%-a makró. Az átirás mechanikus munka, éppen ezért könnyen elronthatjuk. Az átirt makrókat célszerű valakivel átnézetni, vagy szimulátoron (ha van) ellenőrizni. Ha a hordozás után a program a célrendszeren nem működik, akkor hiba csak a makrók átirásában lehet. Ez nagyon fontos kérdés, mivel a célrendszer

-- definició szerint -- nemigen támogatja a programfejlesztést, legfeljebb assembly szintű, primitív hibakeresési eszközzel rendelkezik.

A hibajavítás--újrafordítás--fizikai átvitel esetleg 1-2 órát is igénybe vehet, ezért az átvitt program ellenőrzése során a hibákat gépi kódban javítjuk, és általában csak 10-15 füsgetlen hiba behatárolása után módosítjuk a forrásprogramot. Ez a módszer a generált kód ismeretét és nagyobb programozói adminisztrációt igényel. Egy programot legfeljebb 2-4-szer kell ilyen módon javítani, mivel hiba már csak a CDL program makróiban, a hordozás következtében keletkezhetett.

Egy új fejlesztésnél előre tudjuk, hogy a programnak mikroprocesszoron (is) kell futnia. Ezért eleve úgy írjuk meg, hogy a hordozás egyszerű legyen. Már a fejlesztő (host) gépre készített programban figyelembe vesszük a célgép egyes hardware sajátosságait, pl. a byte-os műveletvégzést. Előfordulhat, hogy emiatt a host gépen futó program nem lesz hatékony, azonban nem is kell annak lennie!

A host gépről a célgépre file-okat fizikailag is át kell tudni vinni. Fontos, hogy ezek mérete minél kisebb legyen, ezért a keresztfordítás egészen gépi kódig történik. Másfelől szükséges, hogy a két gép közös perifériával rendelkezzék (a Tanszéken az átvitel a gépek között soros vonalon vagy floppy diszkkel oldható meg). A hordozási folyamat elég gyors, a korábban említett programok mindegyike 3-4 héten belül működött a mikroprocesszoron. A folyamat tovább gyorsul, ha a célgépen (a fejlesztőrendszeren) is elkészül a CDL debugger, amely a hiba behatárolását egyszerűbbé teszi.

Továbblépés: CDL2 laboratórium

Az eddigiekben ismertetett programfejlesztési technológiából a továbblépés legtermészetesebb iránya a CDL2 laboratórium. A CDL2 laboratórium (amely az ANSWER rendszer része) a programfejlesztés számára azonos elveken alapuló, egységes rendszert ad. A CDL2 laboratórium alapvetően három részből áll:

- nyelvi szerkesztő
- CDL2 fordítók
- interaktív próbapad

vagyis --magasabb szinvonalon -- ugyanazokat az elemeket tartalmazza, mint a vázolt technológia. Lényeges technikai különbség azonban, hogy egyetlen összeépített rendszerről van szó, amely ennek következtében csak igen nagy gépen futtatható.

A CDL2 laboratórium is a programhordozáson alapuló fejlesztést támogatja. Ismereteink szerint azonban nem foglalkozik azzal a fontos problémával, hogyan élesszük fel a hordozott programot a célrendszeren. Kétségtelen azonban, hogy a hordozás legveszélyesebb szakasza, a gépfüggő részek átírása valószínűleg sokkal nagyobb biztonsággal végezhető el a CDL2 laboratóriumban, mint egyéb rendszerekben.

Abstract

This paper describes a software technology developed for programming several types of 8-bit microprocessors and PDP-11 computers.
The technology is based on the CDL (Compiler Description Language). Due to the high portability of CDL programs, cross-method of development is used.
The described software technology has been in use for more than three years with good results, compared to formerly used assembly programing.

Irodalomjegyzék:

[1] C.H.A.Koster: A Compiler Compiler
Matematish Centrum Amsterdam, MR 127 (1971)
[2] J.P.Dehottay, H.Feuerhahn, C.H.A.Koster, H.M.Stahl:
Syntaktische Beschreibung von CDL2
TU Berlin, Informatik (1976)
[3] Hanák P.-Rácz G.-Sarbó J.-Tétényi I.:
A software technology based on program portability
MIMI'80, Sixth International Symposium, Budapest

Szerzők neve és címe:

Hanák Péter - Rácz Gábor - Sarbó János
Budapesti Műszaki Egyetem
Műszer- és Méréstechnika Tanszék

Hernádi Ágnes

TÍPUSABSZTRAKCIÓ ÉS IMPLEMENTÁLÁSÁNAK LEHETŐSÉGEI

Az absztrakt adattipusok lényeges szerepet játszhatnak megbizható, hatékony és könnyen kezelhető szoftver fejlesztésében. Az utóbbi időben sok javaslat született absztrakt adattipusok programnyelvekbe ágyazására. Bár ezek gyakran uj nyelvek tervezéséhez vezettek, ez nem feltétlenül szükségszerü. A dolgozat bemutatja a PL/I nyelv preprocesszor segitségével megvalósitott kiterjesztését. Bár ez a rendszer nem nyujt optimális tipusvédelmet, több előnye van: az alapnyelvet sok programozó ismeri, szinte mindenütt rendelkezésre áll fordítóprogram, és a kiterjesztés tartalmazza az adatabsztrakciós nyelvekhez ajánlott tulajdonságok legtöbbjét.

Kulcsszavak: absztrakt adattipusok, PL/I, preprocesszor

1. Bevezető

Az elmult évtizedben /a SIMULA 67 hatására/ több olyan programnyelv alakult ki, amely a szokásos adattipusok mellett lehetőséget ad a felhasználónak a munkájához szükséges speciális adattipusok definiálására. Az ilyen programnyelvek használata számos előnnyel jár. Ezek közül talán az a legfényegesebb, hogy az ilyen nyelven irt program nagyobb valószinüséggel tükrözi a létrehozásához vezető gondolatmenetet. Igy a program könnyebben olvasható és érthető lesz, különösen azok számára, akik nem vettek részt készitésében.

Az absztrakt adattipusok programnyelvbe épitésével kapcsolatos jelenlegi munkák legtöbbje hangsulyozza az erősen tipizáló és osztályszerü /ld. SIMULA 67/ szerkezetek használatát az absztrakt tipusok implementációinak elkülönitésére. Tipusellerőrzéssel megakadályozható, hogy egy adott tipusu objektumhoz "idegen eszközökkel" lehessen hozzányulni. Ha pl. nem csupán egy veremként használható adatstruktura, hanem egy Verem tipus definiálható, akkor a fordítóprogram meggátolhat-

ja, hogy a felhasználó a Verem tipus definiciójában megadott müveleteken kivül módosithassa vagy elérhesse a Verem tipus értékeit. Igy biztositott az adatstruktura sértetlensége, és a program reprezentáció-függetlensége.

Az absztrakt adattipus vagy tipusabsztrakció kifejezés ábrázolásfüggetlen specifikációval definiált objektumosztályt jelöl.

Ma már komoly nemzetközi munka folyik egy ilyen tulajdonságu, egységesen elfogadott nyelv /ADA/ megvalósitására. A tapasztalatok szerint egy uj nyelv általános elterjedése másfél-két évtizedet vesz igénybe. /Noha a PL/I jóval hatékonyabb, az adatfeldolgozási feladatok megoldásában a statisztikák szerint sokkal nagyobb a COBOL nyelven irt programok aránya./ Ebből a gondolatból kiindulva ésszerünek látszik már meglévő programnyelvek olyan adatabsztrakciós lehetőséggel való kiterjesztése, ami az adatabsztrakciótól megkivánt tulajdonságok legtöbbjével rendelkezik. Ezt a problémát fogom megvizsgálni előadásomban.

2. Absztrakt tipusok specifikációja

Az absztrakt tipusok programnyelvekbe ágyazásával foglalkozó ujabb munkák nagy része /pl. CLU, Alphard, Euclid/ a SIMULA 67 osztályszerkezetéből indul ki. Bár mindegyikük ajánl egy mechanizmust az operációk és egy tipust reprezentáló társtruktura összekötésére, de az alapnyelv szabta határokon belül nem adnak reprezentációfüggetlen eszközt az operációk viselkedésének leirására. Reprezentációtól függetlenül csak a különböző operációk értelmezési tartományai és értékkészletei adhatók meg. Pl. /egész számokat tartalmazó/ Queue tipus a következő operációkkal definiálható:

NEW: → Queue
ADD: Queue × Integer → Queue
FRONT: Queue → Integer

REMOVE: Queue → Queue
EMPTY: Queue → Boolean

A Queue és FRONT szavak jelentésétől eltekintve az operációk akár a Stack tipust is definiálhatnák. E két tipus esetén az értelmezési tartományok és értékkészletek leirásai izomorfak. Nevek jelentésének megérzésére támaszkodni még szokásos tipusok esetén is veszélyes, szokatlan tipusok esetén pedig csaknem lehetetlen. Ezért specifikálni kell a tipus operációinak szemantikáját.

Az absztrakt adattipusok szemantikai leirásai egy közismert osztályozás szerint két kategóriába sorolhatók: müködésjellemző /operational/ vagy definiciós leirások. A müködésjellemző specifikáció az absztrakt adattipusok tulajdonságainak leirása helyett megszerkesztésükre ad receptet. Néhány jól értelmezett nyelvből vagy diszciplinából kiindulva felépiti a tipus egy modelljét.

A müködésjellemző megközelités legfőbb előnye, hogy programozók által viszonylag könnyebben megszerkeszthetőnek tünik, mert nagyon hasonlit a programozásra. Kevés, közepesen egyszerü /azaz a modellező tartományban könnyen kifejezhető/ operációt tartalmazó absztrakt tipus esetén a müködésjellemző leirás elég szabatosnak tünik. Minél bonyolultabbak a leirandó operációk a modellező tartományhoz képest, annál hosszabbak a müködésjellemző specifikációk, tul hosszuak ahhoz, hogy az emberek közti kommunikáció megfelelő eszközének bizonyuljanak. Az operációk közti kapcsolatok nincsenek világosan kifejtve, és az operációk számának növekedésével egyre nehezebb kombinatorikusan következtetni rájuk. Ez tükröződhet a tipust használó programok nehézkes okfejtéseiben.

A müködésjellemző specifikációk legkomolyabb problémája az, hogy csaknem mindig az absztrakció tulrészletezésére kényszeritenek. Nem odatartozó részletek bevezetésével lényeg-

telen attributumokat kapcsolnak a tipushoz.

Egy definiciós leirás határozottan felsorolja az absztrakt tipust alkotó objektumok és operációk tervezett tulajdonságait. Ennek fő előnye, hogy a tipus teljesen általános definiálására törekszik, hiszen csak lényeges jellemzőket kell megadni. A specifikáció igy egy viszonylag nagy implementáció-osztályt lefedő absztrakció. A leirás általánosságának növelésén tul a nélkülözhető részletek elhagyása is világosabbá teszi a leirást. Ha a tipusnak sok operációja van, az a tény, hogy világosan meghatározhatók az operációk közti kapcsolatok, a specifikációt a formális érveléshez alkalmasabb eszközzé teszi.

Roppant sok formalizumussal lehet definiciós leirást készíteni. A programozásban a két legkiemelkedőbb módszer a Hoare féle axiomatikus leirás [HO'69] és Scott matematikai szemantikája [SC'70]. A továbbiakban az elterjedtebb axiomatikus definiciókkal foglalkozunk.

Az absztrakt tipusok specifikációjának két axiomatikus megközelitését fogjuk vizsgálni: a Hoare által javasolt módszert [HO'72], majd az algebrai axiómákat [GU'78a]. Ma a Hoare módszer változatai vannak tulsulyban, pl. [WU'76] és [LO'78], de ugy tünik, hogy az algebrai megközelités is kezd térthóditani [DA'78].

Napjainkban természetes igény, hogy a programok müködését szabatosan bizonyitani lehesen. Igy az absztrakció és a helyességbizonyitás nyelvi formáinak megteremtése összefonódik. A helyességbizonyitással azonban csak olyan mértékig foglalkozunk, amennyi a nyelvi mechanizmusok megértéséhez okvetlenül szükséges.

2.1. Hoare féle megközelités

Az alapötlet uj hatáskört /scope/ definiáló szerkezetek, osztályok használata. Egy osztálynak lehetnek saját lokális változói, más változók ilyen tipusunak deklarálhatók, de ez utóbbiak reprezentációja rejtve marad, és csak az osztály müveletei tudják kezelni. Az osztályt az különbözteti meg a már meglévő hatáskördefiniáló szerkezetektől, hogy ezeknek az eljárásoknak a nevei az osztályon kivül is ismertek. A tokbazárást /encapsulation/ nyujtó és tipusabsztrakciót segitő nyelvi mechanizmus más képet mutat felhasználóinak, és ismét más képet megvalósitóinak. A felhasználó csak a megengedett felhasználáshoz illő tulajdonságokat látja. Ezeket az "absztrakt" jellemzőket Hoare elő- és utófeltételekben fogalmazza meg.

A modul megvalósitójának azonban nemcsak a megvalósitandó objektumot definiáló felhasználó szemléletével, hanem a modul adatszerkezetével és rutinjainak testével is foglalkoznia kell. E között a két szemlélet között a konkrét térből az absztrakt térbe képező ábrázolásfüggvény a hid.

A fenti módszer alkalmazói többnyire valamilyen módon eltértek a Hoare által eredetileg használt jelöléstől, de az alapvető megközelités lényegesebb szempontjai megegyeznek. Bár egy tipus operációi bonyolult kapcsolatban vannak egymással, a tipus /absztrakt/ leirása e kapcsolatok közvetlen állitása helyett minden operációra különálló elő- és utófeltételeket tartalmaz. Ez egy harmadik, az operációk jelentésének megfogalmazására alkalmas tartomány bevezetéséhez vezet. A programozónak nyujtani kivánt absztrakciót a nyelv primitiv tipusaival valósitjuk meg, és egy /feltételezetten jól definiált/ harmadik tartományba, pl. a matematikai halmazokon vagy sorozatokon végezhető müveletek tartományában irjuk le. Ahhoz, hogy az absztrakció általunk adott megvalósitásának helyességét bizonyithassuk, egy nyelvi adatstruktúrát a bevezetett harmadik tartományra /matematikai halma-

zokra vagy sorozatokra/ kell képeznünk. Az absztrakt tipust használó programokat a bevezetett tartományon kell megindokolni. Ez komoly probléma lehet. Feltehetően elsősorban azért kerül sor egy absztrakció bevezetésére, mert ugy tünik, hogy előnyös ezzel az absztrakcióval gondolkodni. Bizonyára nem nagy veszteség, ha pl. az 1-100 tartományba eső egész számok halmazának absztrakciója helyett a matematikai halmazokkal való következtetésre kényszerülünk. Végeredményben igen hasonló absztrakciók. Az érvelés kedvéért azonban tegyük fel, hogy már jól megértett tipusaink tartománya nem foglalja magába a matematikai halmazokat. A programozó igy egészen más absztrakcióval való indoklásra kényszerül, mint amilyet programjába bevezetni kivánt. Ekkor viszont a tipusabsztrakció bevezetésének nincs sok értelme.

2.2. Algebrai megközelités

Egy absztrakt tipus algebrai specifikációja három részből áll: egy szintaktikus, egy szemantikus és egy megszoritásokat tartalmazó leirásból. A szintaktikus specifikáció adja meg azt a szintaktikus és tipusellenőrző információt, amit már sok programnyelv megkövetel: a tipushoz kapcsolt operációk neveit, értelmezési tartományait és értékkészleteit. A szemantikus leirás egy axiómahalmaz, ami az operációk jelentését definiálja, megállapitva egymás közti kapcsolataikat. A megszoritások felsorolása előfeltételeket és kivétel feltételeket tartalmaz. Az "algebrai" jelzőt az indokolja, hogy egy absztrakt tipus alapvetően értékek és operációk gyüjteménye, és igy egészen természetes absztrakt algebrai rendszernek tekinteni. Gannon és Zilles az algebrai megközelitést erősen hangsulyozva a heterogén /many-sorted/ algebrák alkalmazásaként kifejlesztette az absztrakt algebrák elméletét. E felfogás szerint az axiómák által definiált algebrák izomorfak. Ezért az absztrakt tipus összes elfogadható megvalósitásának egymással és az eredetivel izomorf algebrát kell adnia. Igy egy implementáció helyességének megmutatása nem

más, mint az egyik algebrát a másikba vivő izomorf leképezés létezésének igazolása.

Nézzük meg e módszer alkalmazását tetszőleges, de egy objektumon belül azonos tipusu elemeket tartalmazó Stack tipus, helyesebben szólva tipus-séma definiálására. Példánkban

- a csak kisbetüs szimbólumok a jelzett tartományokon értelmezett szabad változók,
- a csak nagybetüs szimbólumok müveletnevek,
- a nagy kezdőbetüs szimbólumok tipusnevek.

```
typeStack[elem-type:Type,n:Natural number] where()
syntax
 NEWSTACK: → Stack
 PUSH:     Stack × elem-type → Stack
 POP:      Stack → Stack
 TOP:      Stack → elem-type
 ISNEW:    Stack → Boolean
 REPLACE:  Stack × elem-type → Stack
 *DEPTH:   Stack → Integer                    (1)
semantics
 declare stk:Stack,elm:elem-type
 1/ POP(PUSH(stk,elm))=stk
 2/ TOP(PUSH(stk,elm))=elm
 3/ ISNEW(NEWSTACK)=true
 4/ ISNEW(PUSH(stk,elm))=false
 5/ REPLACE(stk,elm)=PUSH(POP(stk),elm)
 6/ DEPTH(NEWSTACK)=0
 7/ DEPTH(PUSH(stk,elm))=1+DEPTH(stk)
```

---

(1) A müveletet megelőző * azt mutatja, hogy a müvelet segédfügyvény /rejtett függvény/.

restrictions

pre(POP,stk) = ¬ISNEW(stk)

pre(REPLACE,stk,elm) = ¬ISNEW(stk)

ISNEW(stk) => failure(TOP,stk)

failure(PUSH,stk,elm) => DEPTH(stk) ≥ n

elem-type-ot egy bizonyos tipushoz, és n-t egy bizonyos egész számhoz kötve, pl. Stack[Real,18], a séma egy egyszerü absztrakt tipus leirására egyszerüsödik.

2.3. Néhány megjegyzés az axiomatikus leirásokról

Absztrakt tipusokat használó programok bizonyitásához mind az algebrai mind a Hoare-féle tipusleirás a program és funkcionális leirása közti ellentmondásmentesség megmutatására használható deduktiv szabályokat ad. Azaz az absztrakt tipusok axiomatikus definiciójának jelenléte mechanizmust ad annak bizonyitására, hogy egy program és funkcionális leirása nem ellentmondó, feltéve, hogy a program által használt absztrakt müveletek megvalósitásai sem mondanak ellent leirásaiknak. Igy a bizonyitás részekre bontásának módszere adott, mert az axiomatikus definiciók az absztrakció alacsonyabb szintjén annak leirásául szolgálnak, amit meg kell valósitani. Absztrakt tipusok ábrázolásának helyességbizonyitásához az axiomatikus leirások az igazolandó állitások minimális halmazát adják. Algebrai axiómáknak ill. Hoare-féle leirásoknak programok helyességbizonyitásában való alkalmazását [GU'78b] ill. [HO'72] és [WU'76] részletesen tárgyalja.

E két leiró módszer előnyeinek és hátrányainak összevetése szükségszerüen szubjektiv. Tetszőlegesen sok olyan példa lelhető, amire az egyik, vagy a másik megközelités nyilvánvalóan kényelmesebb. A Hoare-féle megközelitést előnyben részesitő példákat az alapul szolgáló leiró nyelvben elérhető tipusokkal közvetlenül összefüggő tipusabsztrakció választása jellemzi. Az algebrai módszert előnyben részesitő példák olyan tipusabsztrakciót választanak, amit nem áb-

rázol vagy modellez azonnal egy közismert tipus. A példáknak ez a két osztálya két tényt tükröz:

1. ha nagyon sokat tudunk néhány tartományról, és az adni kivánt absztrakció azonnal e tartományba képezhető, akkor sokat nyerünk e leképezés végrehajtásával, és a jobban megértett tartomány értelmében való okoskodással;
2. ha arra kényszerülünk, hogy a kivánt absztrakciót egy másféle tartományba képezzük, és ezután e tartomány értelmében okoskodjunk, akkor elveszitjük az absztrakció bevezetésétől remélt előnyöket.

Ezt a két tényt figyelembe véve, a tipusabsztrakció e két megközelitésének viszonylagos előnyét csak a tipusabsztrakció felhasználási módjának szubjetiv értékelésére támaszkodva lehet felbecsülni. Fel kell tenni a kérdést, hogy milyen absztrakciók bizonyulnak majd előnyösebbnek.
Ha a szokatlan tipusok igen ritkának bizonyulnak, a tipusabsztrakció elsődleges haszna az információ elrejtése lesz.
Ha a szokatlan tipusok gyakoribbak lesznek, akkor a tipusabsztrakció a megvalósitás részleteinek elrejtésén tul uj tartományokat is eredményez.

3. Tipusabsztrakciót támogató szoftver eszközök megvalósitása a PL/I-hez

A szocialista országokban jelenleg legelterjedetebb, az ESZR rendszer által támogatott legfejlettebb, és szinte mindenütt rendelkezésre álló nyelv a PL/I. Természetes, hogy elsősorban ez a nyelv vizsgálatunk tárgya. Ugy véljük, hogy a PL/I nyelv korlátain belül lehet olyan adatdefiniciós lehetőséget teremteni, ami az adatabsztrakciótól megkivánt tulajdonságok legtöbbjével rendelkezik, és amit a felhasználó könnyen és kényelmesen kezelhet.

Az általunk tervezett absztrakciós mechanizmussal a PL/I lehető legszükebb kiterjesztését akartuk létrehozni. A ki-

terjesztett nyelv megtervezésekor szem előtt tartottuk, hogy a felhasználó jól strukturált programokat irhasson benne. Ez a feltétel a preprocesszor által készitett programra - valószinüleg - már nem biztositható.

Az adatabsztrakció makrók alkalmazásával való megvalósitása [HA'74] névproblémához vezet, és nem nyujt kellő védelmet az illetéktelen hozzáférés ellen.

Az adatdefiniciók és müveletek csoportositására a PL/I eljárást használjuk. A PL/I-ben ENTRY utasitással egy eljáráson belül további belépési pontok definiálhatók, igy ezek segitségével fogjuk szimulálni egy absztrakt adattipus müveleteit. Módositott nyelvünkben egy adatabsztrakció szerkezete vázlatosan a következő:

```
typename: PROCEDURE;
     op1: ENTRY;
          •••
          RETURN;
     op2: ENTRY;
          •••
          RETURN;
     op3: ENTRY;
          •••
          RETURN;
     END typename;
```

Ebben az esetben "typename" az előállitott absztrakt adattipus neve és az op1, op2, op3 belépési pontok definiálják az uj adatokon végezhető müveleteket. Igy más eljárásokból typename "tipusu" változók adhatók át ezeknek a müveleteknek.

E szervezés alapvető problémája, hogy PL/I-ben rendes szekvenciális utasitás végrehajtása esetén egy ENTRY utasitás jelenléte - az üres utasitáshoz hasonlóan - nem befolyásolja a végrehajtási sorrendet, azaz egy ENTRY utasitáson"átfolyik" a szekvenciális vezérlés. Ha az absztrakt müvelete-

ket definiáló rutinok ily módon való átfolyását megengedjük, könnyen fordulhat elő hiba /elfelejtjük a következő ENTRY utasitás előtti RETURN-t/. Ez kiküszöbölhető, ha egy ENTRY utasitás előfordulásakor automatikusan előállitjuk a RETURN utasitást, hiszen ekkor minden ENTRY-RETURN pár külön függvényt definiálna /a GOTO utasitást lehetőleg elkerülve/. A SIMULA 67-hez hasonlóan az absztrakció-mechanizmus igy az eljárás bármely lokális STATIC változóján keresztül közös /shared/ információval rendelkező függvények alakjában definiált. Ha továbbá az adatabsztrakció ábrázolása csak erre az eljárásra korlátozott, akkor a felhasználó modulja csak a definiált müveleteken keresztül kezelheti az objektumot, és reprezentációját semmilyen más módon nem változtathatja meg.

Absztrakt tipusu adatobjektumok deklarálásához bevezettük a TYPE attributumot. Pl. egy programban a következő deklarációval definiálhatnánk a "typename" absztrakt tipus objektumait:

```
DECLARE X TYPE(typename),
        Y(100) TYPE (typename);
```

A fenti tervezetet megvalósitó preprocesszor Intézetünkben fejlesztés alatt van. A kisérletek során elért eredményeket az előadáshoz készitendő vetitési anyagban illusztrálni fogjuk.

Abstract:

Abstract data types can play a significant role in the development of software that is reliable, efficient, and flexible. There have been many recent proposals for embedding abstract data types in programming languages. Although including these ideas has often led to the design of a new language, that is not always necessary. This paper discusses the extending of PL/I using a preprocessor. While the resulting system does not have the optimal set of protection features, it does have several advantages : the base language

is known to a large class of programmers, there are many such compilers already written, and the system achieves almost as much protection as is needed.

Irodalomjegyzék:

DA'78 Dahl,O.J.: Can Program Proving Be Made Practical? institute of Informatics,Univ. Oslo, Norway, 1978

GU'77 Guttag, J.V.: Abstract Data Types and the Development of Data Structures
Commun. Ass. Comput. Mach., Vol. 20, pp.396-404, June 1977

GU'78a Guttag, J.V.,Horning,J.J.: The Algebraic Specification of Abstract Data Types
Acta Informatica, Vol. 10, pp. 27-52, 1978

GU'78b Guttag,J.V.,Horowitz,E.,Musser,D.R.: Abstract Data Types and Software Validation
Commun. Ass. Comput. Mach., Vol. 21, pp. 1048-1064, December 1978

GU'80 Guttag,J.V.: Notes on Type Abstraction /Version 2/
IEEE Trans. Software Eng., Vol. SE-6, pp. 13-23, January 1980

HA'74 Hansal,A.: "Software Devices" for Processing Graphs Using PL/I Compile Time Facilities
Information Processing Letters, 1974, Vol. 2, pp. 171-179

HO'69 Hoare,C.A.R.: An Axiomatic Basis for Computer Programming
Commun. Ass. Comput. Mach., Vol. 12, pp. 576-580, October 1969

HO'72 Hoare,C.A.R.: Proofs of Correctness of Data Representations
Acta Informatica, Vol. 1, No. 1, pp. 271-281, 1972

LI'77 Liskov,B.,Snyder,A.,Atkinson ,R.,Schaffert,C.: Abstraction Mechanisms in CLU
Commun. Ass. Comput. Mach., Vol. 20, pp. 564-576, August 1977

LO'78 London, R.L.,Guttag,J.V.,Horning,J.J.,Lampson,B.W., Mitchell,J.G.,Popek,G.J.:
Proof Rules for the Programming Language Euclid
Acta Informatica, Vol. 10, pp. 1-26, 1978

SC'70 Scott,D.: Outline of a Mathematical Theory of Computation Proc. 4th Annu. Princeton Conf. Information Science and Systems, 1970, pp. 169-176

WU'76 Wulf,W.A.,London,R.L.,Shaw,M.: An Introduction to the Construction and Verification of Alphard Programs IEEE Trans. Software Eng., Vol. SE-2, pp. 253-265, December 1976

ZE'78 Zelkowitz,M.V.,Larsen,H.J.: Implementation of a Capability-Based Data Abstraction IEEE Trans. Software Eng., Vol. SE-4, pp. 56-64, January 1978

Hernádi Ágnes
Számitógépalkalmazási Kutató Intézet
Rendszerfejlesztési Főosztály

Budapest, I. Csalogány u. 30-32. 1015

Horvai Mátyás–Koch Róbert–Kovács Kálmán–Tibor József

TPA 11 – INTEL 8080 (ICC) KERESZTRENDSZER

A TPA 11 - Intel 8080 keresztrendszer létrehozásával a mikroprocesszor korlátozott kapacitásu eszközei helyett a TPA 11 DOS-RV (RSX-11/M) többfelhasználós operációs rendszere áll az Intel 8080 szoftver fejlesztés rendelkezésére. A belövés kivételével a fejlesztés teljes egészében a TPA 11-en történhet. A fejlesztés különböző állapotainak megfelelő fájlok a DOS-RV segédprogramjaival processzálhatók.

Megfelelő hardver feltételek esetén Intel programok közvetlenül is áttölthetők DOS-RV fájlból az Intel 8080 memóriájába ill. Intel memóriatartalom küldhető át DOS-RV fájlba.

Kulcsszavak:

Intel 8080, kereszt makró assembler, kereszt szerkesztő, közvetlen betöltés, TPA 11 (PDP 11), DOS-RV (RSX-11/M), CP/M operációs rendszer.

Bevezetés

A több programból álló rendszer célja az, hogy a TPA 11 számitógépen lehetővé tegye az assembly nyelvü program-

fejlesztést Intel 8080 mikroprocesszort tartalmazó mikroszámitógépekre, mint például a KFKI-ban kifejlesztett intelligens CAMAC vezérlőre /ICC/. Mivel a TPA 11-en futó DOS-RV operációs rendszer több terminált kezel egyidőben, ezért a keresztrendszert is egyszerre többen vehetik igénybe.

1. A CPX keresztfordító

A forditandó forrásprogramban az eredeti Intel assembler nyelv utasitásai használhatók. A makró lehetőségek lényegében megegyeznek a Digital Research által az Intel 8080-ra kifejlesztett CP/M operációs rendszere alatt müködő makró assemblerének nyelvével. Igy - kis különbségektől eltekintve - ugyanaz a forrásprogram akármelyik rendszerben leforditható. Lényeges bővitést jelent a mikroprocesszoron futó rendszerekhez képest, hogy a keresztforditó nemcsak abszolut, hanem áthelyezhető kódot is tud generálni. Lehetőség van könyvtárkezelésre is. A forrásprogramok módositása a DOS-RV operációs rendszerben futó bármelyik szövegszerkesztővel történhet. A CPX keresztforditó lehetőségei közül néhányat az alábbiakban sorolunk fel:

a. Makró definiálási lehetőségek /ujradefiniálás, többszörös mélységü definició ill. hivás/
b. Makró könyvtári fájlok használata
c. Feltételes forditás kijelölése direktivákkal
d. Abszolut program ill. áthelyezhető modulok forditása
e. Globális azonositók használata az áthelyezhető modulokban

f. Betürendes azonositó táblázat és kereszthivatkozási táblázatok kérhetők
g. A fordító üzemmódjai előirhatók programon belüli direktivákkal és operátori paranccsal is.

2. CPY keresztszerkesztő

A CPY keresztszerkesztő bemenő fájlja a CPX keresztfordító áthelyezhető bináris kimenő fájlja. Feladatai:

a. Áthelyezhető bináris modulokat a megadott báziscimtől kezdődően egymás után szerkeszt, bitkép fájlt hoz létre
b. Feloldja a modulok kapcsolódását biztositó globális hivatkozásokat
c. Kiszámitja a programban szereplő nem abszolut tipusu kifejezések értékét
d. Könyvtár fájlokból szelektiv szerkesztést végez
e. Betöltési térképet és globális kereszthivatkozási listát készit
f. Szimbólum definiciós modult tud előállitani, amelyet későbbi szerkesztések során inputként meg lehet adni.

3. CPL könyvtárkezelő

A CPL könyvtárkezelő makró- és tárgyprogram könyvtárak létrehozására és karbantartására szolgál.

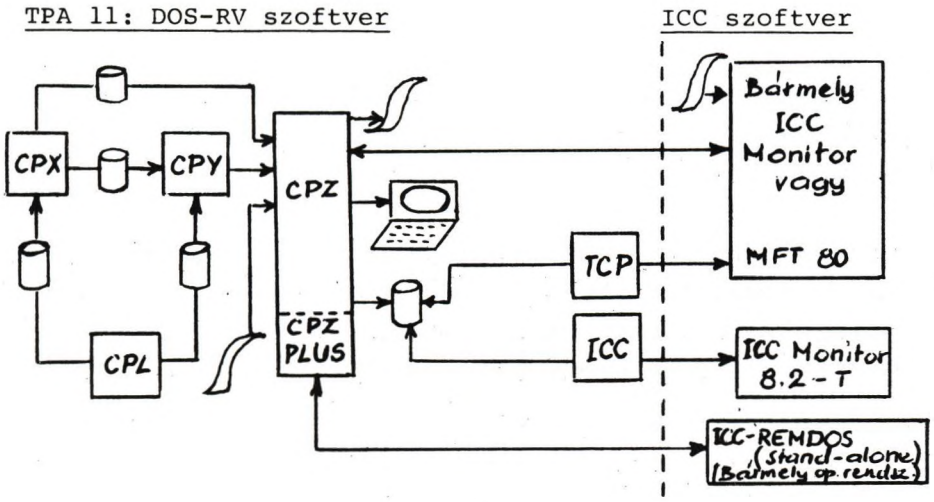
4. CPZ /CPZ-PLUS/ segédprogram

A CPZ segédprogram abszolut bináris ill. szerkesztett bitkép fájlból Intel szabványu hexadecimális /karakter/ formát állit elő. Bővitett változata a CPZ-PLUS program, amely a CPZ feladatain kivül közvetlen programbetöltést és Intel memóriatartalom elmentést tesz lehetővé. Munkánknak ezt a részét a KFKI-ban kifejlesztett intelligens CAMAC vezérlőre /ICC/ valósitottuk meg.

Az Intel 8080 mikroprocesszoron az ICC-REMDOS programnak kell futni, amely az ICC konzolját DOS-RV terminálként kezeli, ugyanis a CPZ-PLUS taszkot erről a terminálról kell inditani.

A hardver kiépitettségtől függően az ábrán látható további két lehetőség biztositja a programok ICC memóriába való töltését /TCP taszk, ha az ICC-nek nincs konzolja, vagy az ICC taszk speciális ICC monitorral/.

A keresztrendszerhez tartozó programok /taszkok/ egymáshoz kapcsolódását az ábra szemlélteti.



A CPZ programból való közvetlen átvitelt a lyukszalagos perifériákat helyettesitő paralel kábel-kapcsolat valósitja meg. Az ábrán látható többi összeköttetés soros vonalakkal realizálódik. A keresztrendszer komponenseit szabványos DOS-RV parancsokkal lehet aktivizálni, a közvetett parancs-fájl lehetőség is használható.

Tapasztalatok

A felhasználói tapasztalatok a várt eredményt hozták. A többfelhasználós rendszerek közvetlen használatával lényegesen lecsökkentek a programok belövésére forditott idők; lényegesen többen tudtak egyszerre Intel programokat fejleszteni.

A CPX keresztfordító nyelvének makró lehetőségei jól támogatják a programozási munkát.

A nyomkövetési funkciók megoldatlansága miatt azonban a relokációs lehetőségeket /keresztszerkesztés/ kevesebben használják. /Várható, hogy az év folyamán a nyomkövető komponenst is kidolgozzuk./

A közvetlen betöltési lehetőséggel kiküszöböltük a lyukszalag használatát ill. megbizhatóbbá tettük a programbevitelt. A betöltési idő a programok méretétől függően a másodperc - perc tartományba esik.

Abstract

Instead of the limited capacity of devices the TPA 11 - Intel 8080 cross-system provides for the Intel 8080 microprocessors' software development. Except the program verification /debugging/ the whole development of the Intel 8080 software can be done by means of the TPA 11 computer with its resources under the multi-user operating system DOS-RV (RSX-11/M).

The files of the different stages of the program development can be processed by different DOS-RV utility programs of the cross-system.

In case of special hardware conditions Intel programs can be loaded directly into the Intel's memory from DOS-RV files or Intel memory-dump can be sent into DOS-RV file.

Irodalom

1. Intel 8080 Assembly Language - MTA KFKI, Bp.
2. DOS-80 Operating System - MTA KFKI, Bp.
3. DOS-80 Assembler - MTA KFKI, Bp.
4. CAMAC Intelligent Crate Controller Family - MTA KFKI, Bp. Revised 1981.
5. DOS-RV operátori kézikönyv /megjelenés alatt/ - MTA KFKI, Bp.
6. TPA 11 - Intel 8080 /ICC/ keresztrendszer /megjelenés alatt/ - MTA KFKI, Bp.

Szerzők neve és cime:

Horvai Mátyás, Koch Róbert, Kovács Kálmán, Tibor József

Magyar Tudományos Akadémia
Központi Fizikai Kutató Intézet
Bp. XII., Konkoly Thege ut 29-33.
H-1525 Budapest 114. P.O.B. 49.

Horváth András–Ivanyos Lajosné–Papp Béla

KONKURRENS PASCAL IMPLEMENTÁCIÓ ÉS PASCAL KERESZTFORDÍTÓ A TPA 1140 KISSZÁMÍTÓGÉPEN

Cikkünk P. Brinch Hansen PDP 11/45-re irott Konkurrens PASCAL [1] rendszerének TPA 1140 kisszámitógépre való átvitelével foglalkozik, mely rendszer keresztforditóként is használható. Hansen rendszere konkurrens és szekvenciális PASCAL nyelveken, valamint gépi nyelven készült modulokat tartalmaz. Az alábbiakban a gépi modulok átszervezésével és változtatásával, a PASCAL programok módositásával, valamint az INTEL 8080-ra kifejlesztett Interpreterrel foglalkozunk.

Kulcsszavak: PASCAL keresztforditó virtuális kód.

A TPA 1140-en, RSX-11M operációs rendszer alatt futó változat elkészitésénél a következő célkitüzések vezettek:

- többfelhasználós RSX-11M rendszer felhasználása PASCAL nyelven való program fejlesztésre
- paralel programozást támogató PASCAL forditó implementálása
- a CPX-CPY-CPZ INTEL 8080-as keresztrendszerhez [9] illeszkedő szekvenciális PASCAL keresztforditó létrehozása
- szekvenciális és konkurrens PASCAL programok kipróbálásának biztositása

A PDP 11/45-ön megvalósitott rendszer, a 4K szónyi assembly nyelven megirt Kernel-Interpreter-ből, a kon-

kurrens PASCAL nyelven megirt - fejlesztést és kipróbálást támogató - SOLO operációs rendszerből, a szekvenciális és a konkurrens PASCAL fordító programokból, valamint a SOLO rendszerprogramokból áll. Az utóbbiak mind szekvenciális PASCAL nyelven iródtak, amely a Jensen és Wirth által 1974-ben specifikált PASCAL nyelvnek felel meg [2], némi eltéréssel.

Mindkét fordítóprogram egy virtuális gépre /Virtuális Stack Gép/ generál kódot /P kód, [6]. Ez a kód igen jó hatásfokkal értelmezhető a TPA 1140 tipusu számitógépen is, a kb. 1.2K szónyi Interpreter program segitségével, s az Interpreter más géptipusra is viszonylag könnyen és gyorsan elkészithető.

A TPA 1140 RSX-11M alatti implementációnál megtartottuk az eredeti rendszer strukturát, de az egészből egy RSX taszkot készitettünk. Használjuk a SOLO-nak egy némileg röviditett változatát, egyrészt azért, mert a fordítóprogramok igénylik a SOLO szolgáltatásait, másrészt a SOLO támogatja a leforditott programok kipróbálását.
A SOLO rendszerprogramok tárolására, valamint a fordítóprogramok munkaterülete számára megtartottuk az eredeti diszk strukturát, de nem egy teljes lemezt, hanem csak egy 500 szektoros folytonos RSX adatállományt bocsájtunk a SOLO rendelkezésére. A forrásnyelvi szöveget valamint a leforditott programokat /általában/ nem SOLO állományokban, hanem RSX állományokban tároljuk. A Kernel-Interpreter rendszermegoldásokat RSX szolgáltatásokkal helyettesitettük. Az RSX-hez fordulást SOLO szinten uj rendszerprogramok elkészitésével és a fordítóprogramok vezérlő részének módositásával támogatjuk. A végzett munkáról a következő csoportositásban számolunk be:

- a PDP 11/45-ön implementált rendszer PASCAL-ban megirt részén végzett módositások
- a Kernel - Interpreter átalakitása, a teljes rendszer RSX taszkká szervezése

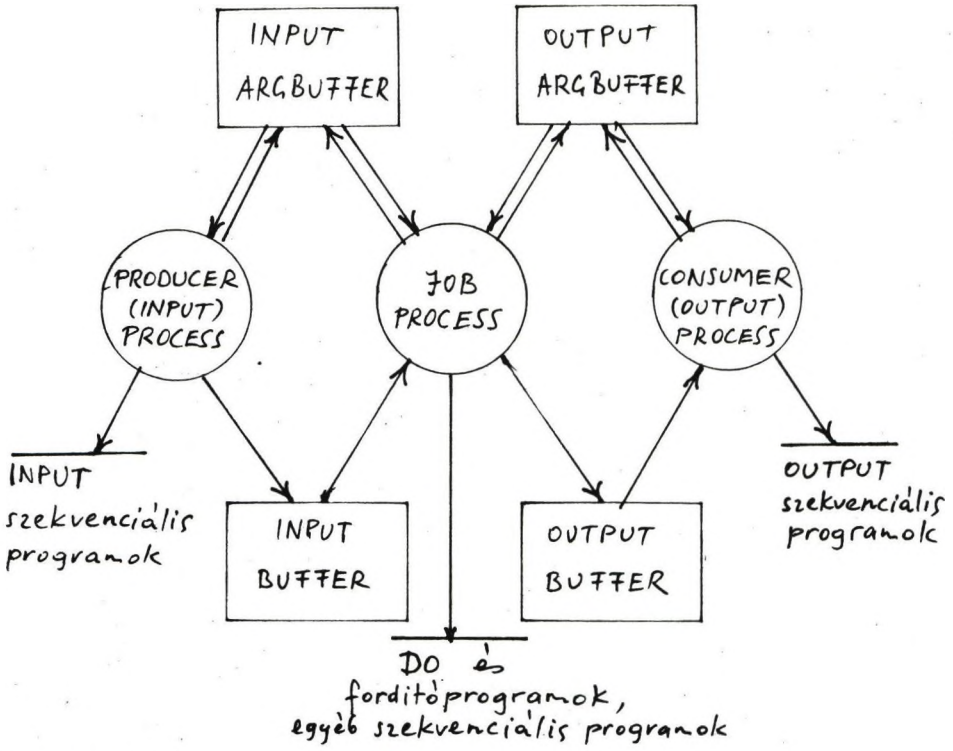
- az Interpreter az INTEL 8080-on.

Párhuzamosságot feltételező feladat megoldására - P. Brinch Hansen elgondolása szerint - egy konkurrens PASCAL programból és szekvenciális PASCAL programokból álló rendszert kell /és lehet/ létrehozni. A konkurrens PASCAL nyelv a szekvenciális PASCAL bővitése, benne definiálhatók un. rendszer-tipusok, PROCESS, MONITOR és CLASS tipusok. A rendszer-tipusok mindegyikében lényegében az adatstrukturákat és a rajtuk való müveleteket /mint PASCAL utasitások sorozatát/, valamint a hozzáférési jogokat definiáljuk. A PROCESS-ek egymással szimultán müködhetnek /taszk/, s rendszerprogramként nyilvántartott szekvenciális programokat futtathatnak, ha azok kód- és adatrésze /STACK-HEAP/ számára megfelelő területet foglaltak le. A szekvenciális programok ujabb szekvenciális programokat is hivhatnak, s PROCESS-ük szekvenciális kódterületén felváltva helyezkednek el. /Ha a szekvenciális programlánc egy tagja befejezte müködését, visszatöltődik az őt hivó program kódrésze ./ A MONITOR tipussal a PROCESS-ek közös adatstrukturáit definiáljuk. A MONITOR-ok biztositják a PROCESS-ek szinkronizálását és azt, hogy az adatokhoz a PROCESS-ek egymás kölcsönös kizárásával férjenek hozzá. A CLASS tipus egy PROCESS-hez tartozó adatstruktúrához való ellenőrzött hozzáférést nyujt.

A SOLO operációs rendszer a PASCAL program-fejlesztés és kipróbálás feladatára készült konkurrens program. Hat PROCESS-t definiál, ebből három alapvető fontosságu, a JOB, a CONSUMER és a PRODUCER PROCESS-ek. Mindegyik futtat szekvenciális programokat: a JOB a vezérlő PROCESS, futtatja az operátor kommunikációt lebonyolitó DO programot, az pedig a forditóprogramokat. A forditóprogramok vezérlő részből és hét forditó menetből állnak. A vezérlőrész határozza meg azt, hogy a JOB PROCESS milyen input-output utility-k használatára utasitsa a CONSUMER ill. PRODUCER PROCESS-eket.

RSX adatállományok használatát lehetővé tevő szekvenciális PASCAL programokkal bővitettük a SOLO rendszerprogramokat, és a forditóprogramok vezérlő részét módositva ezek használatát irtuk elő a CONSUMER ill. PRODUCER számára.

A SOLO-ból kiiktattuk a többi PROCESS-t és néhány felesleges CLASS-t ill. MONITOR-t. A módositott rendszer müködését nagy vonalakban a következő ábra szemlélteti.



Megjegyzés:

Az ARG BUFFEREK-ben küldi a rövid üzeneteket egymásnak a JOB és a PRODUCER ill. a JOB és a CONSUMER PROCESS. Pl. a JOB PROCESS ilymódon határozza meg az INPUT és OUTPUT szekvenciális programokat.

A PDP 11/45-ön implementált Konkurrens Pascal rendszer ugy is tekinthető, hogy alapvetően két részből áll; a Virtuális Stack Gépet szimuláló programból és értelmezendő kódból. Gépi szinten az Interpreter program kód-részletei valósitják meg a virtuális gépet, melynek regiszterei részben gépi regiszterek, részben memóriahelyek. Az un. „konkurrens kódok" /CLASS, MONITOR és PROCESS adminisztráció és adatforgalom/ interpretálása szoftvare interruptok kezelésével történik, kihasználva a PDP 11/45 automatikus regiszter-készlet cseréjét, mely a memória leképezésére is kiterjed. /Az executive-szintü géphasználatot engedélyező módról nevezik Kernel-nek a kezelő programot./ A rendszer áthelyezése RSX-11M alá szükségessé teszi a Kernel/Interpreter átirását.
A feladat első lépcsőjeként üzembe helyeztük az önálló rendszert a TPA 1140-en. A lebegőpontos processzor utasitásait a jóval szerényebb lehetőségü TPA 1140 lebegőpontos utasitások miatt szubrutinokkal helyettesitettük, valamint - interruptkor nem lévén automatikus regiszter-készlet váltás - a Kernelben meg kellett oldani a regiszterek elmentését.

A következő lépés a rendszer RSX-11M taszkként való implementálása volt. Figyelembe kellett vennünk, hogy Kernel módot, és közvetlen adatforgalmat nem használhatunk; a megfelelő funkciókat Executive Direktivák [10] segitségével kellett megvalósitani.

Az interpreter kódértelmezés alkalmával a memóriát két részre bontja: az összes PROCESS által hozzáférhető közös szegmensre és a PROCESS-ek saját szegmensére. PROCESS-váltás alkalmával a Kernel az uj PROCESS fizikai szegmensét leképezi a közös szegmens végétől kezdődő, megfelelő méretü virtuális cimtartományba, ilymódon az összes PROCESS számára külön-külön biztositva a teljes cimezhető tartományt.

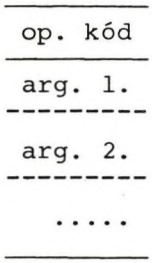
A közös szegmens elején foglal helyet az interpreter cimtáblázata, kihasználva a táblázat nullás cimen kezdődő voltát. Tekintettel arra, hogy egy RSX taszk headerje ezt a helyet már eleve lefoglalja, az Interpretert ennek figyelembevételével kellett átirni. Ilymódon a közös szegmens, melybe a Kernel/Interpreter is beletartozik, a taszk alapleképezésével dolgozik, a Kernel hivásánál EMT helyett szubrutinhivást alkalmaztunk és gondoskodtunk a PROCESS-ek számára szükséges virtuális tárkezeléséről.

A virtuális gépet realizáló taszk jelenleg ugy müködik, hogy a terminálon a többi forditóprogramnál szokásos módon megadott parancssor alapján az input, a tárgyprogram és a lista RSX-fájlok megnyitása után a diszken létesitett, szabvány RSX-fájl szerkezetü, de az eredeti SOLO diszk-rendszer strukturáját őrző véletlen elérésü adatállományt is megnyitja, s ebből a SOLO konkurrens kódját saját területére beolvassa, majd a PROCESS-ek tárigényének megfelelő méretü régiót foglal el a memória tetején. Az átdolgozott Kernel/Interpreter ezután a SOLO ill. az általa betöltött szekvenciális programok végrehajtásával folytatja müködését.

P. Brinch Hansen PDP 11/45-ön implementált rendszerének PASCAL forditóprogramjai egy meghatározott utasitáskészlettel rendelkező feltételezett gépre a VIRTUÁLIS STACK GÉP /VSG/-re generálnak kódot. /P kód, [6]/

A VSG egy olyan feltételezett gép, amely meghatározott utasitáskészlettel rendelkezik /virtuális utasitás készlet/ és a teljes memóriát stack-szerüen kezeli.

A virtuális utasitás mindig tartalmaz egy operációs kódot és ezt nulla vagy több argumentum követi:



Mind az utasitás kód, mind az argumentumok 2 byte memória helyet foglalnak. A virtuális kód referenciái /hivatkozásai/ a programon belül mindig relativek, azért a virtuális kódu program bárhova betölthető.

A VSG-ben a teljes rendelkezésre álló memóriaterületet mint stack-et kezeljük, ahol a helyfoglalás a magas cimektől az alacsony cimek felé növekszik. Ez a stack tartalmazza a virtuális kódu programot és a változókat, paramétereket is. Igy a fentmaradt hely a végrehajtás során szükséges stack és heap igényeket is optimálisan elégitheti ki.

A VSG müködéséhez szükséges regiszterek a következők:

- S - stack pointer,
- B,G - báziscim regiszterek,
- Q - virtuális utasitásszámláló, amely mindig a következő virtuális utasitásra vagy egy argumentumra mutat,
- X,Y,W - a virtuális utasitások végrehajtásához szükséges munka-regiszterek.

A Brinch Hansen-féle rendszerben leforditott program csak relativ cimzéseket használ a programon belüli vezérlésátadáshoz; s mind a globális, mind a lokális adatmezejét egy-egy báziscimhez képest relativ cimzéssel kezeli.

A leforditott PASCAL program végrehajtása a VSG realizálását kivánja meg azon a gépen, amelyen a programot futtatni kivánjuk [8] Ezt a feladatot teljesiti az INTEL 8080-as mikroprocesszorra irt Interpreter.

Az Interpreter az a gépi kódu program, amely elvégzi a leforditott program virtuális kódu utasitásainak értelmezését és végrehajtását.

Az Interpreter két fő része a különböző virtuális utasitásokhoz tartozó gépi kódsorozatok gyüjteménye és az un. operációs tábla, amely a gépi kódsorozatok belépési cimeit tartalmazza.

Az Interpreter feladata a virtuális utasitás számláló /Q/ léptetése és a soron következő virtuális utasitás 'végrehajtása' a kódhoz tartozó gépi kódsorozat segitségével. Minden egyes gépi kódsorozat végén az Interpreter egy indirekt vezérlésátadással választja ki és inditja el a soron következő kódsorozatot. Az Interpreter biztositja a kapcsolatot a már meglévő szoftver-rendszerekhez /pl. 'INTEL MONITOR' vagy 'MFT80'-hoz/. Ezenkivül az Interpreter részét alkotják azok a programszegmensek is, melyek pl. az Interpreter inicializálását, a rendszer rutinok hivását és rendszer rutinból való visszatérést, stb. biztositják.

Az Interpreter használja a lebegőpontos csomag szubrutinjait is, ezért betöltés előtt a leforditott Interpretert a lebegőpontos csomaggal össze kell szerkeszteni. Ugyancsak gondoskodni kell a rendszerrutinoknak az Interpreterhez való szerkesztéséről. Ehhez a TPA 1140 INTEL 8080 keresztrendszerének CPY programját kell felhasználni [9].

Abstract:

This paper considers the moving of the implementation of the P. Brinch Hansen's Concurrent PASCAL for PDP 11/45 to the TPA 1140 minicomputer. The using of the system as a cross-compiler is also described here. Hansen's system consists of programs written in the Concurrent and Sequential PASCAL languages and of assembly modules. The modification

of the PASCAL programs and the alteration and reorganization of the assembly modules are discussed as well as the development of the Interpreter for the Intel 8080.

Irodalom

1. P. Brinch Hansen, The Architecture of Concurrent Programs, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1978.
2. K. Jensen and N. Wirth, Pascal-User and Manual Report, Springer Verlag, Berlin, 1975.
3. P. Brinch Hansen and C. Hayden, Microcomputer Comparison, Software-Practice and Experience 9 /March 1979/.
4. A.C. Hartmann, A Concurrent Pascal Compiler for Minicomputers, Springer Verlag, Berlin, 1977.
5. P. Brinch Hansen, Concurrent Pascal Implementation Notes, Information Science, California Institute of Technology, 1976.
6. J.R. Bell, 'Threaded code', Comm. ACM 16,6, 370-372, /1973/.
7. E. Jul, Microcomputer Comparison, Euromicro Journal 6 /1980/ 144-148.
8. D. Neal and V. Wallentine, Experiences with the Portability of Concurrent Pascal, Software-Practice and Experience, Vol. 8, 341-353 /1978/.
9. Horvai - Koch - Kovács - Tibor
   TPA 11 - INTEL 8080 keresztrendszer
   KFKI kiadvány
10. RSX 11M Executive Reference Manual

Szerzők neve és cime:

Horváth András - Ivanyos Lajosné - Papp Béla
Magyar Tudományos Akadémia
Központi Fizikai Kutató Intézet
Bp. XII., Konkoly Thege ut 29-33.
H-1525 Budapest 114. P.O.B. 49.

Katona Endre

SEJTPROCESSZOROK ALKALMAZÁSA FIXPONTOS VEKTOR- ÉS MÁTRIXSZORZÁSRA

Vektorok skaláris szorzatának számitása és a mátrixszorzás természetes módon párhuzamositható müveletek, ezért a sejtautomaták egyik alkalmazási területét képezhetik. A jelen cikk a Legendi Tamás által kidolgozott sejtprocesszor architekturát alapul véve /lásd [1]/ részletesen kidolgozott sejtalgoritmusokat ismertet vektor- és mátrixszorzásra. A bemutatott algoritmusokból egyben kitünnek a sejtprocesszorok előnyei a szokásos mátrixprocesszoros gépekkel szemben:

- a sejttér tetszőleges szóhossz esetén teljes kihasználtsággal müködhet;
- nemcsak az egyes aritmetikai alapmüveletek hajtódnak végre egyidejüleg, hanem az alapmüveleteken belül is párhuzamos számitás történik /bitpárhuzamos müveletvégzés/;
- a bitpárhuzamos müveletvégzés lehetővé teszi az egyes alapmüveletek átlapolt végrehajtását, igy elérhető, hogy közel minden időpontban minden sejt hasznosan müködjön.

Kulcsszavak: sejtprocesszorok, párhuzamos algoritmusok, sejtalgoritmusok, mátrix müveletek

1. Az alkalmazott sejtprocesszor architektura

Az [1]-ben kidolgozott sejtprocesszor architektura egyfelől erősen leegyszerüsitette a sejthardware-t, ezzel reálissá téve viszonylag nagyméretü sejtterek megépitését, másrészt feloldotta a klasszikus sejtautomata-koncepció merevségét, igy a sejtprocesszor könnyebben és rugalmasabban programozhatóvá vált. Az alábbiakban röviden vázoljuk ezt az architekturát.

A sejttér egy téglalap alaku sejtmátrix, melyet a szélein un. bábsejtek határolnak. A bábsejtek biztositják a sejt-

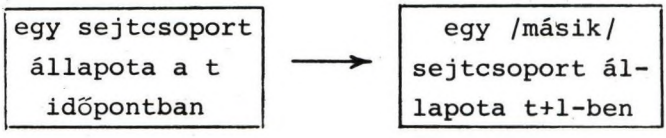
térnek a kűlvilággal való kapcsolatát: ezek a sejtek nem végeznek állapot-átmenetet, hanem állapotukat a külvilágból lehet lépésenként beállitani.

A sejttérben az un. Neumann-szomszédságot alkalmazzuk, vagyis minden sejt a négyzetrács szerinti négy szomszédjával van összekötve.

A sejteknek nincs rögzitett átmenetfüggvénye, hanem mikroutasitásokat kapnak egy központi vezérlő egységből /CCPU/ és egy meghatározott utasitássorozat /mikroprogram/ hatásaként áll elő a kivánt állapot-átmenet. Ezen szervezésmód következtében a sejtek tetszüleges átmenetfüggvénnyel müködhetnek /a mikroutasitás-készlet ezt lehetővé teszi/, sőt az egymás utáni lépésekben is más-más átmenetfüggvényt alkalmazhatunk /időbeli inhomogenitás/.

A sejttér térbeli inhomogenitással is rendelkezik abban az értelemben, hogy az egyes sejtek egyidejüleg más-más átmenetfüggvénnyel dolgozhatnak. Ennek biztositására minden sejtet belső állapottal látunk el. A belső állapotokat a sejtprocesszor müködése kezdetén lehet beállitani, és azok a müködés során nem változnak. A különböző belső állapottal rendelkező sejtek különbözőképp reagálhatnak ugyanarra a központi utasitásra, igy különböző átmenetfüggvénnyel müködhetnek.

Az átmenetfüggvényeket [3] alapján mikrokonfigurációs termekkel adjuk meg. Egy mikrokonfigurációs term sémája a következő:



Ha a jobb oldalon csak egy sejt szerepel, a bal oldalon pedig az öt szomszédja, akkor a mikrokonfigurációs term egy hagyományos értelemben vett termnek felel meg, egyébként több hagyományos termmel egyenértékü. Az inhomogenitás miatt egy mikrokonfigurációs term egyidejüleg több átmenetfüggvényt is definiálhat.

A jobb oldalon szereplő sejtek általában a bal oldalon is szerepelnek, itt kettős kerettel jelöljük őket az azonositás megkönnyitésére.

2. Bináris összeadás és szorzás sejttérben

Az összeadás és szorzás sejtalgoritmusát [4] és [6] tartalmazza részletesen, itt csak vázoljuk azokat.

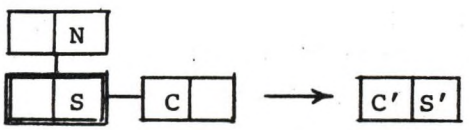
Az összeadó sejtalgoritmus lényege, hogy az összeadást párhuzamosan minden helyértéken végezzük a

$$[C_i S_i] := S_i + C_{i-1} + N_i$$

formula alapján, ahol S_i jelöli az i-edik helyértéken számitott /még nem végleges/ összegbitet /"sum"-bit/, C_i jelöli az i-edik helyértéken keletkező átvitelt /"carry"-bit/, N_i pedig egy uj összeadandó i-edik bitjét jelenti. A $[C_i S_i]$ irásmóddal azt fejezzük ki, hogy a C_i és S_i bitek uj értékét együtt, mint kétjegyü bináris számot definiáljuk.

A fenti un. carry-save összeadó algoritmus egy lépés során három számból /$[S_k \ldots S_1]$, $[C_k \ldots C_1]$ és $[N_k \ldots N_1]$/ kettőt képez, ezáltal lehetővé válik lépésenként egy uj összeadandó bevételezése, anélkül, hogy előbb az átvitelek végigfutását megvárnánk.

Az összeadó algoritmus sejttérbeli megvalósitása kézenfekvő. Az összeadó sejtszerkezet egy sornyi 2-bites /4 állapotu/ sejtből áll, melyek a következő átmenetfüggvénnyel müködnek:

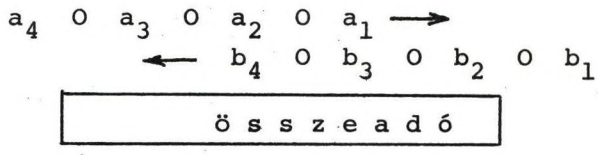


ahol $[C'S'] := S + C + N$

Kezdetben az összeadóban minden S és C bit értéke 0. Ha a felső sorba /bábsejtsorba/ lépésenként egy uj összeadandót irunk, akkor n db k bites szám esetén n+k lépés után alakul

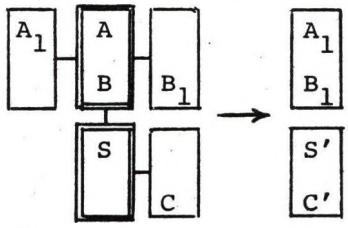
ki az összeg az összeadó S bitjein.

A szorzó sejtalgoritmus az összeadásra épül. Lényege, hogy egy összeadó sejtsor felett egymással szemben mozognak a szorzandók, és mindig az azonos pozición álló bitek szorzatából képzett szám kerül az összeadóba. A részletszorzatok helyes generálása érdekében a szorzandókat nullákkal fel kell higitani /1. ábra/. Ekkor ugyanis pl. a b_3 bit rendre találkozni fog az a_1, a_2, a_3 és a_4 bitekkel, ezáltal a $b_3 \cdot [a_4 a_3 a_2 a_1]$ részletszorzat az összeadóba kerül. Hasonlóan áll elő a többi részletszorzat egymáshoz képest egy helyértékkel eltolva.



1. ábra: A bináris szorzás sejtalgoritmusa 4-bites operanduszok esetén.

A fenti sejtalgoritmus két sor 2-bites sejttel realizálható, melyek a következő átmenetfüggvénnyel müködnek /a két függvényt együtt adjuk meg/:



ahol $[C'S'] := S + C + A \cdot B$

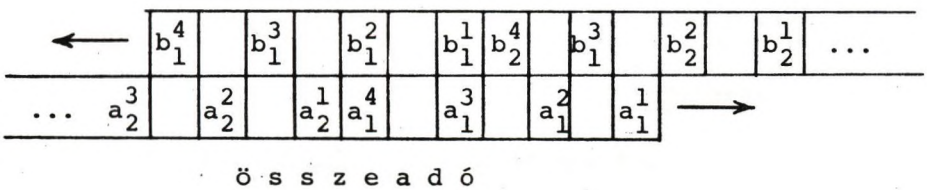
k-bites számok szorzása esetén természetesen 2k hosszuságu összeadó szükséges. Az operanduszok 2k lépés alatt haladnak el egymás mellett, az átvitelek végigfutásához további k lépés kell, igy a szorzás teljes időigénye 3k lépés.

3. Vektorszorzás

Ha az előző pontban leirt szorzó sejtszerkezet adatcsatornáira nem csak egy-egy operanduszt, hanem operanduszok sorozatát adjuk, akkor a páronként keletkező szorozatok az összeadóban összegződnek. A szorzó sejtszerkezetnek ez a tulajdonsága lehetővé teszi, hogy azt lényegében változtatás nélkül vektorszorzásra használjuk. Eljárásunkat az alábbiakban részletezzük.

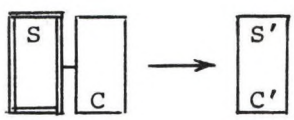
A feladat az $(a_1,\ldots,a_n)$ és $a(b_1,\ldots,b_n)$ k-bites számokból álló vektorok $a_1b_1+\ldots+a_nb_n$ skaláris szorzatának számitása. Ehhez elegendő az $a_1,\ldots,a_n$ számokat a szorzó "A"-val jelölt jobbra mozgató adatcsatornájára adni, a $b_1,\ldots,b_n$ számokat pedig a balra mozgató "B" adatcsatornára küldeni. Az operanduszok sorrendjének megfelelően minden a_i a neki megfelelő b_i-vel fog szorzódni, és a szorzatok az összeadóban összegződnek.

Ügyelni kell azonban arra, hogy az a_i szám a szorzón való áthaladása során ne kerülön fedésbe a b_{i-1} és b_{i+1} számok egyetlen bitjével sem, mert ekkor hibás részletszorzat-bitek kerülnének az összeadóba. Ilyen hibás részletszorzatok keletkezését a legegyszerübben ugy kerülhetjük el, hogy az egymás után küldött operanduszok között nem hagyunk hézagot /2. ábra/. Igy a_i csak b_i-vel találkozik szinkronban, b_{i-1}-nek és b_{i+1}-nek csak a felritkitó nulláival kerül fedésbe.



2. ábra: 4-bites számokból álló vektorok szorzása. a_j^i az a_j szám i-edik bitjét jelenti.

n elemü, k-bites számokból álló vektorok esetén a vektorszorzat maximális értéke $n(2^k-1)^2 \approx n \cdot 2^{2k}$, ennek megfelelően $2k+\log_2 n$ sejtből álló összeadót kell alkalmazni. /Kevesebb sejt esetén az összeadó bal szélére speciális átmenetfüggvénnyel ellátott "tulcsordulás-figyelő" sejt illeszthető./ Figyelembe kell azonban venni, hogy az összeadónak csak a jobboldali 2k-1 db sejtje képez részletszorzatokat, az ettől balra fekvő sejtek az adatcsatornákat figyelmen kivül kell hogy hagyják. Ezek átmenetfüggvénye:



ahol [C'S'] : = S + C

A vektorszorzás időigénye n komponensü, k-bites számokból álló vektorok esetén $2kn+k$ lépés. Ez kevesebb, mint n egymás utáni szorzás időigénye, mivel az egyes szorzások átlapoltan hajtódnak végre /az átvitelek végigfutását csak az utolsó szorzás után kell megvárni/. A sejtalgoritmus helyigénye $2(2k+\log_2 n)$ 2-bites sejt. 4-bites sejtek esetén a sejtszerkezet két sora összevonható, igy ebben az esetben fele annyi sejt szükséges.

Megjegyezzük, hogy 2-es komplemens kódolásu előjeles számokhoz is készithető a fentihez hasonló vektorszorzó sejtszerkezet. Itt az összeadó sejteket egy-egy vezérlőbittel kell ellátni /3-bites sejtek/, amelyek az előjelek speciális kezelését biztositják. Az algoritmus egy változatának részletes leirását 9 tartalmazza.

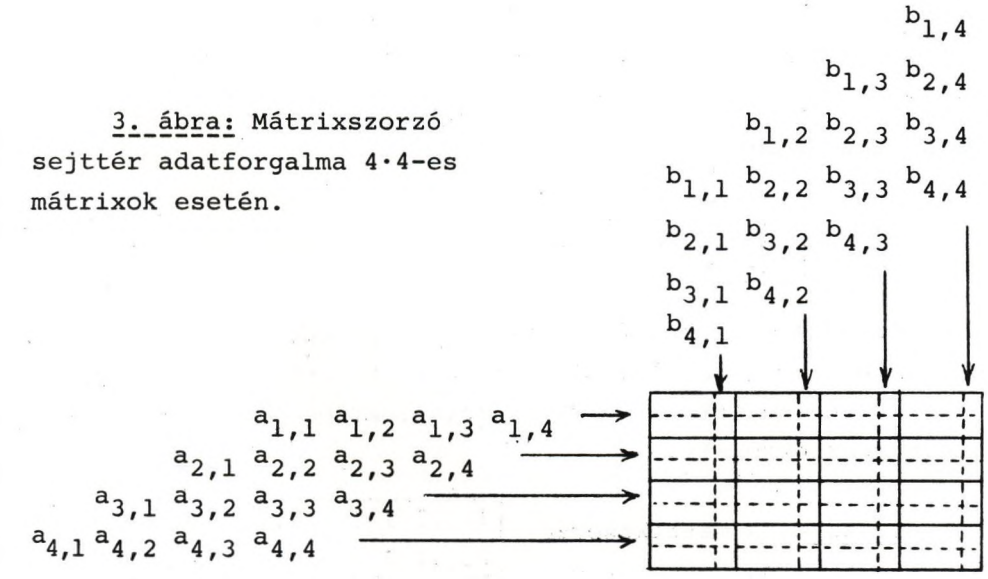
4. Mátrixszorzás

Legyen A és B két k-bites számokból álló n·n-es mátrix, feladatunk a C=A·B szorzatmátrix kiszámitása. Mivel a c_{ij} mátrixelem az A mátrix i-edik sorának és a B mátrix j-edik oszlopának skaláris szorzataként adódik, igy kézenfekvő a mátrixszorzást vektorszorzásra visszavezetni. Az ismertetés-

re kerülő sejtalgoritmus téglalap alaku mátrixokra is alkalmazható, pusztán a könnyebb tárgyalás érdekében szoritkozunk négyzetes mátrixokra.

A mátrixszorzó sejtszerkezet n^2 vektorszorzóból áll, melyeket n·n-es mátrix formájában helyezünk el a sejttérben. A cél olyan adatáramlást biztositani, hogy az (i,j) indexü vektorszorzón az A mátrix i-edik sora és a B mátrix j-edik oszlopa haladjon át megfelelő szinkronitásban, ezáltal a vektorszorzó épp a c_{ij} elemet fogja számitani.

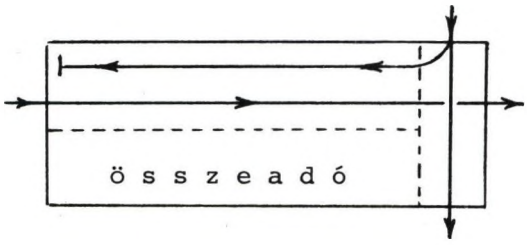
A megoldás lényegét a 3. ábra mutatja. Az A mátrixot soronként késleltetve a vektorszorzók jobbra shiftelő "A" adatcsatornájára adjuk, majd a mátrix egyik szorzóból a másikba áramolva balról jobbra áthalad a sejttéren. Ezzel egyidejüleg a B mátrixot oszoponként késleltetve a szorzók kö-



3. ábra: Mátrixszorzó sejttér adatforgalma 4·4-es mátrixok esetén.

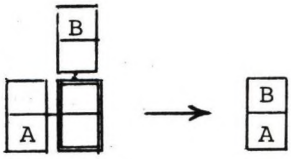
zött elhelyezett függőleges adatcsatornákra küldjük, amelyeken az lefelé áramlik, de minden egyes szorzóhoz érve a mátrixelemek bekanyarodnak /pontosabban elágaznak/ a szorzó "B" adatcsatornájára. Igy minden egyes vektorszorzóban a 4.

ábra szerinti adatáramlás megy végbe. A vektorszorzók bal végén a "B" adatcsatorna megszakad, az operanduszok nem lépnek át a szomszéd szorzóba.

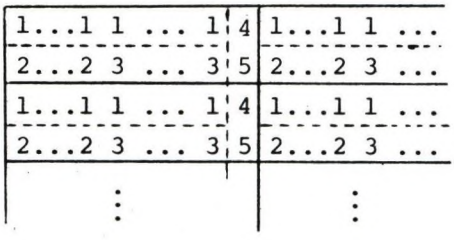


4. ábra: Egy vektorszorzó adatforgalma

A sejtszerkezethez szükséges inhomogenitást az 5. ábra mutatja. A 2-es belső állapotu sejtek passziv összeadó sejtek /az adatcsatornákat figyelmen kivül hagyják/, minden szorzóban $\log_2 n+1$ ilyen sejt szükséges. A függőleges adatcsatornáknak a 4-es és 5-ös belső állapotu sejtek felelnek meg. A 4-es sejteknél kereszteződik a jobbra shiftelő vizszintes és a lefelé shiftelő függőleges adatcsatorna, ezért ezek a sejtek "cross-over" /keresztező/ átmenetfüggvényt hajtanak végre:



Az 5-ös sejtek a felső állapotbiten egyszerüen lefelé shiftelnek, mig az alsó biten állandó 0 értéket tartanak, ezt a bitet ugyanis a velük balról szomszédos összeadósejt átvitelnek érzékeli.



5. ábra: A mátrixszorzó sejttér inhomogenitása /belső állapot mátrixa/.

A leirt szervezéssel elértük, hogy az (i,j) indexü vektorszorzón az A mátrix i-edik sora és a B mátrix j-edik oszlopa haladjon át. Ezután már csak azt kell biztositani, hogy az elsőnek beérkezett vektorelemek (a_{in} és b_{nj}) egyszerre érjék el a szorzót, hisz ekkor már minden további a_{ik} szám szükségképpen a neki megfelelő b_{kj}-vel fog szorzódni.

Ez a sejttérbe érkező sor- illetve oszlopvektorok megfelelő késleltetésével érhető el /3. ábra/. Az A mátrix első sora és B első oszlopa egyidőben kell hogy érkezzen az (1,1) indexü szorzó bemeneteire /pontosabban ugy, hogy az operanduszok jobbról a k-adik összeadósejt felett találkozzanak/. Az a_{1n} elem $2k+\log_2 n+1$ lépés után éri el az (1,2) indexü szorzót, a B mátrix második oszlopát tehát $2k+\log_2 n+1$ lépéssel kell késleltetni ahhoz, hogy az (1,2) indexü szorzóban a vektorok szintén szinkronban találkozzanak. Ugyanakkor a b_{n1} elem a függőleges adatcsatornán már 2 lépés alatt eléri a (2,1) indexü szorzó bemenetét, igy A második soránál csak 2 lépés késleltetés szükséges. Általában igaz, hogy ha az A mátrix i-edik sorát $(i-1)\cdot 2$ lépéssel, a B mátrix j-edik oszlopát $(j-1)\cdot(2k+\log_2 n+1)$ lépéssel késleltetjük, és az (1,1) indexü szorzóban szinkronban találkoznak a vektorok, akkor minden további szorzóban is szinkronban fognak találkozni. Ezzel a mátrixszorzás szinkronizálási problémáját megoldottuk.

Az A mátrix sorait ill. B oszlopait nyilván a vektorszorzásnál alkalmazott formátumban kell a sejttérbe küldeni /az egyes számok nullákkal felritkitva, de hézag nélkül egymás után/. A C szorzatmátrix az összeadókban alakul ki a szorzandó mátrixok teljes áthaladása után.

Időigény: Az A és B mátrixok teljes áthaladásának ideje $4kn+n(\log_2 n+2)-2$, a mátrixszorzás teljes időigénye tehát durván $4kn$ lépésnek tekinthető.

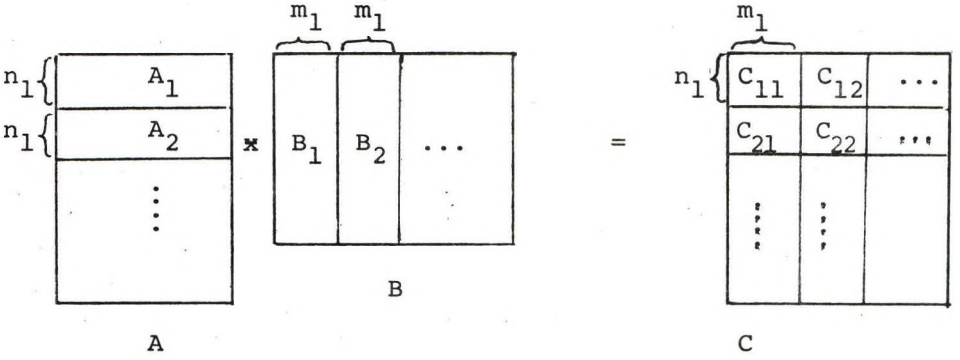
Helyigény: A sejtalgoritmus $2n^2(2k+\log_2 n+1) \approx 2kn^2$ 2-bites sejtet igényel.

Végül megjegyezzük, hogy n·p és p·m méretű téglalap alaku mátrixok szorzásához n·m vektorszorzóból álló sejtszerkezet szükséges, a p paraméter csak az időigényt befolyásolja. Maga a szorzási müvelet pontosan a fent leirtak szerint zajlik le.

5. Nagyméretü mátrixok szorzása

Az előző pontban feltételeztük, hogy a szorzandó mátrixok méretének megfelelő mátrixszorzó sejtszerkezet elfér a rendelkezésre álló sejttérben. Az alábbiakban megmutatjuk, hogy tetszőlegesen nagyméretü mátrixok esetén is alkalmazható az előző pontbeli sejtalgoritmus, ha a szorzandó mátrixokat megfelelően részekre bontva szorozzuk.

Az eljárás lényegét a 6. ábra mutatja. Tegyük fel, hogy a rendelkezésre álló sejtprocesszorban - a k operanduszhossz figyelembevételével - legfeljebb $n_1 \cdot m_1$ méretü mátrixszorzó sejtszerkezet fér el. Ekkor az n·p és p·m méretü szorzandó mátrixokat a 6. ábra szerint n_1 ill. m_1 szélességü csikokra, a C mátrixot pedig $n_1 \cdot m_1$-es részmátrixokra bontjuk. Könnyen belátható, hogy minden i,j-re $C_{ij}=A_i \cdot B_j$ teljesül. Mivel minden $A_i.B_j$ szorzást el tudunk végezni a rendelkezésre álló sejtprocesszorral, igy a C mátrix részletenként számitható.



6. ábra: Nagyméretü mátrixok felbontása.

6. Összefoglalás

A cikkben bitpárhuzamos algoritmusokat adtunk vektor- és mátrixszorzásra, amelyek az [1]-ben javasolt sejtprocesszor architekturában realizálhatók, és igen jó hatásfokuak /v.ö. pl. [11]/, mivel gyakorlatilag minden időpontban minden sejt hasznosan müködik.

A legfeljebb 16 állapotu mikro-sejtekből való épitkezés és a bitszintü müveletvégzés lehetővé teszi a szóhosszhoz való maximális alkalmazkodást, a sejttér mindig teljesen kihasznált lehet. Például 1-bites operanduszok /bitmárixok/ esetében egy mátrixelemre egyetlen /3-bites/ sejt jut a mátrixszorzó sejtalgoritmusnál /lásd [7]/.

Az 5. pontban kifejtett overlay elv lehetővé teszi a sejttér méretét meghaladó mátrixok szorzását is, változatlan sejtalgoritmussal.

Megjegyezzük, hogy másfél-kétszeres helyigénnyel és változatlan időigénnyel lebegőpontos vektor- ill. mátrixszorzó sejtszerkezet is konstruálható.

Az e cikkben bemutatott, valamint az MTA Automataelméleti Kutató Csoportnál eddig készült több mint 100 sejtalgoritmus /lásd [6]/ a sejtprocesszorok széleskörü alkalmazhatóságára utalnak.

Abstract

Vector and matrix operations form a natural application field of cellular automata. In this paper cellular algorithms are given for fixed-point vector and matrix multiplication, taking into account the cellprocessor architecture developed by T. Legendi /see [1]/. In this architecture each cell works as a bitprocessor /micro-cells/ and time-space inhomogeneity is ensured.

The presented algorithms show well the advantages of micro-cells, as follows:

Mátrixszorzó sejttér számitógépes szimulációja

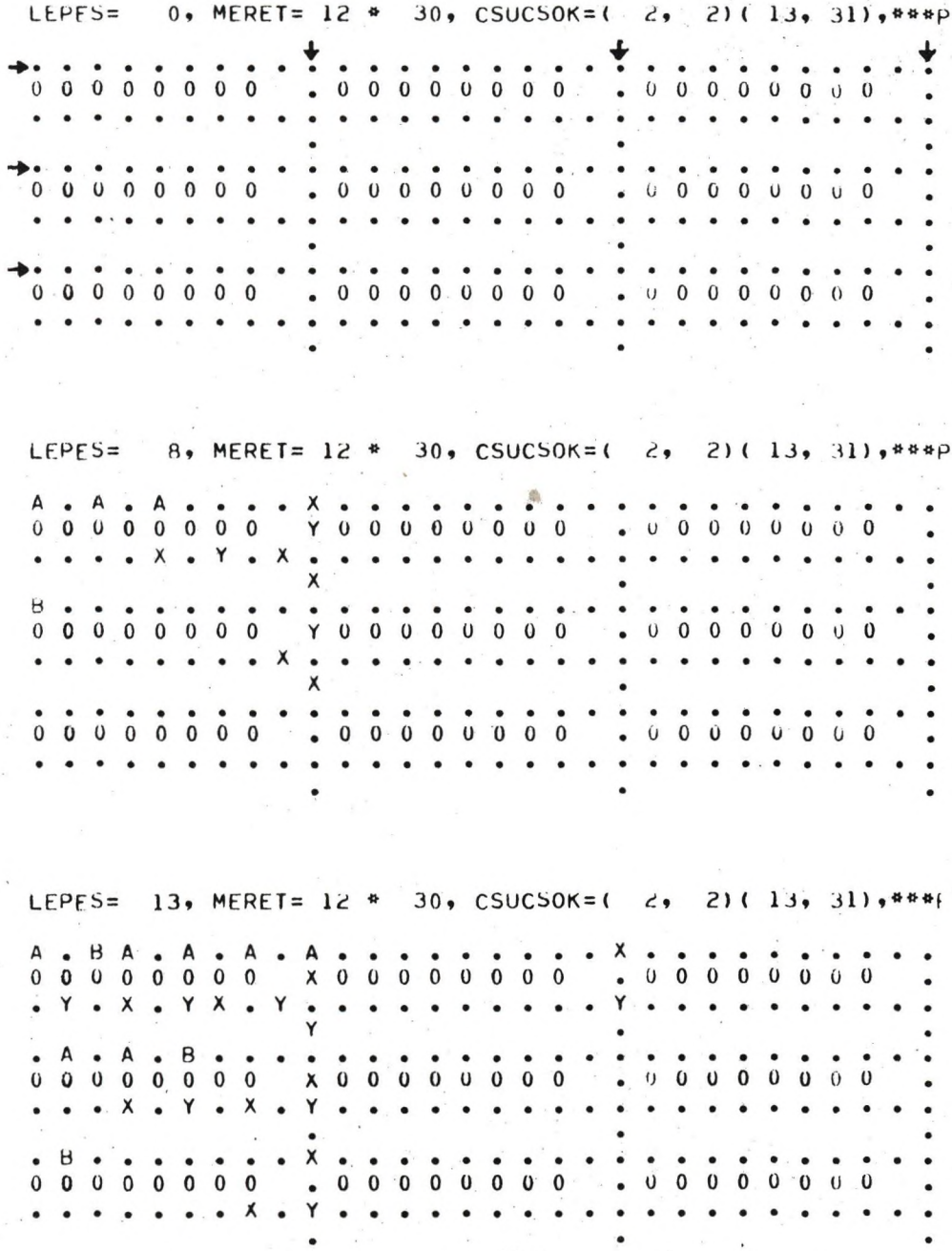
A mellékelt szimuláció a CELLAS sejttérszimulációs nyelv segitségével készült. 3·3-as mátrixszorzó sejtteret vizsgálunk, az elvégzett mátrixszorzás a következő:

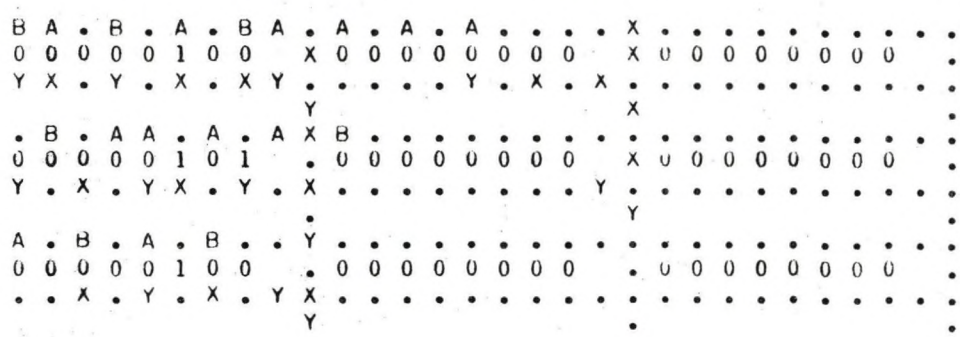
$$\begin{bmatrix} 1 & 5 & 0 \\ 0 & 10 & 1 \\ 2 & 5 & 5 \end{bmatrix} \times \begin{bmatrix} 10 & 5 & 8 \\ 4 & 0 & 4 \\ 5 & 8 & 10 \end{bmatrix} = \begin{bmatrix} 20 & 5 & 28 \\ 45 & 8 & 50 \\ 65 & 50 & 86 \end{bmatrix}$$

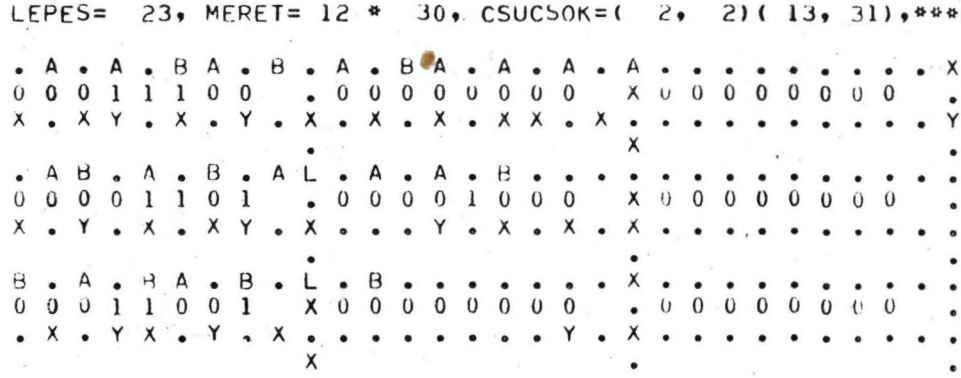
A B C

A szemléletesség érdekében a sejtállapotok megjelenitésére különféle karaktereket alkalmaztunk. Az összeadósejtek állapotai numerikusan jelennek meg, a 2 állapot például 1-es carry-bitet és 0 sum-bitet jelent. Az áttekinthetőség növelésére a vektorszorzók adatcsatornáit külön sejtsorokban helyeztük el: az "A" adatcsatornát az összeadó feletti sejtsorban, a "B" adatcsatornát az összeadó alatt. Az A mátrix bitjeit 0="A", 1="B" konverzióval, a B mátrix bitjeit 0="X", 1="Y" konverzióval jelenitettük meg. Az értéktelen /helykitöltő/ nullákat mindkét esetben pont jelöli. A sejttér állapotát csak a 8. lépéstől kezdődően, 5 lépésenként ábrázoltuk.

A kezdőállapotban a sejttér üres, az összeadósejtek és az adatcsatornák egyaránt 0 állapotuak. A szorzandó mátrixok a nyilakkal jelölt pontokon lépnek be a sejttérbe, és a leirt sejtalgoritmusnak megfelelően áthaladnak azon. A teljes áthaladás után /esetünkben a 63. lépésre/ az összeadókban kialakul a szorzatmátrix.


LEPES= 0, MERET= 12 * 30, CSUCSOK=(2, 2)(13, 31),***P
LEPES= 8, MERET= 12 * 30, CSUCSOK=(2, 2)(13, 31),***P
LEPES= 13, MERET= 12 * 30, CSUCSOK=(2, 2)(13, 31),***

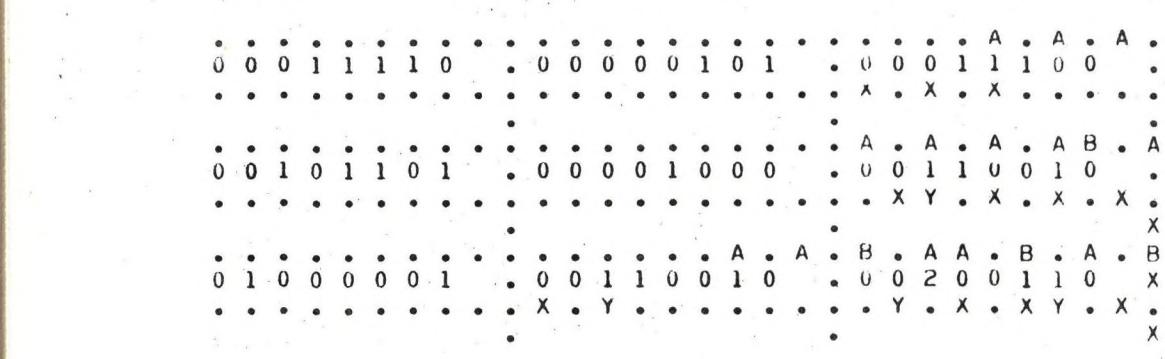

LEPES= 18, MERET= 12 * 30, CSUCSOK=(2, 2)(13, 31),***


LEPES= 23, MERET= 12 * 30, CSUCSOK=(2, 2)(13, 31),***

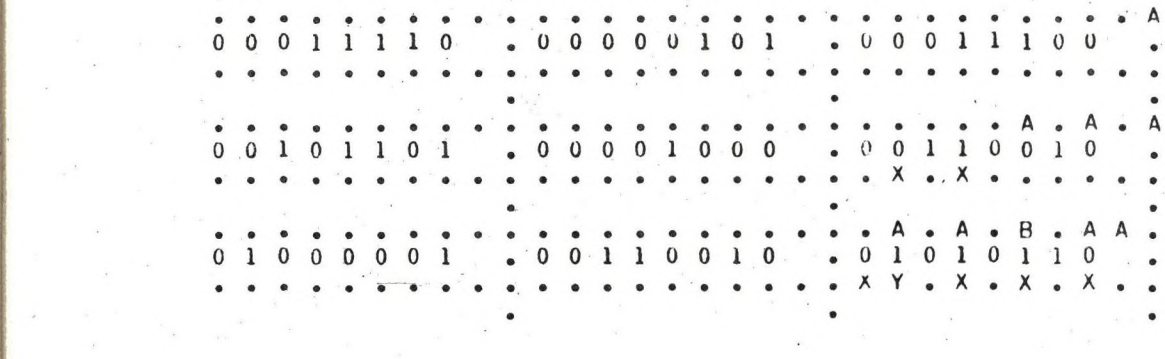

LEPES= 28, MERET= 12 * 30, CSUCSOK=(2, 2)(13, 31),***

```
LEPES=  33, MERET= 12 *  30, CSUCSOK=(  2,  2)( 13, 31),***(

. . . . . . . . . A . A . A . B A . B . A . B A . A . A . A
0 0 0 1 1 1 1 0   . 0 0 0 0 0 0 0 0   . 0 0 0 0 0 0 0 0   X
. . . . . . . . . X . X X . Y . X . Y . X . Y . X X . Y . .
                  .                   .                   Y

. . . . . A . A . A . A B . A . B . A K . A . A . B . . . .
0 0 1 0 1 1 0 1   . 0 0 0 0 1 0 0 0   . 0 0 0 0 1 0 0 0   X
. Y . X . . . . . | X . X . X . X X . Y . . . Y . X . Y . X
                  .                   .                   .

. A . A . B . A A . B . A . B A . B . K . B . . . . . . . Y
0 0 1 1 1 1 0 1   . 0 0 1 0 1 0 0 0   X 0 0 0 0 0 0 0 0   .
X Y . X . Y . X . . . X . X X . X . X . . . . . . . . Y . X
                  .                   X                   .

LEPES=  38, MERET= 12 *  30, CSUCSOK=(  2,  2)( 13, 31),***(

. . . . . . . . . . . . . A . A . A . B A . B . A . B A .
0 0 0 1 1 1 1 0   . 0 0 0 0 0 1 0 1   . 0 0 0 0 0 1 0 0   X
. . . . . . . . . Y . X . Y . . . . . X X . Y . X . X Y .
                  .                   .                   Y

. . . . . . . . . A . A . A . A B . A . B . A A . A . A X
0 0 1 0 1 1 0 1   . 0 0 0 0 1 0 0 0   . 0 0 0 0 1 0 1 0   .
. . . . . . . . . X X . Y . X . Y . X . Y . X X . Y . X .
                  .                   Y                   .

. . . . . A . A . B . A A . B . A . B A . B . A . B . . Y
0 1 0 0 0 0 0 1   . 0 0 1 0 1 0 0 0   X 0 0 0 0 1 0 0 0   .
Y . X . . . . . . . X . X . X X . Y . . . Y . X . Y . X X
                  .                   Y                   X

LEPES=  43, MERET= 12 *  30, CSUCSOK=(  2,  2)( 13, 31),***(

. . . . . . . . . . . . . . . . . . A . A . A . B A . B .
0 0 0 1 1 1 1 0   . 0 0 0 0 0 1 0 1   . 0 0 0 1 1 1 0 0   .
. . . . . . . . . . . . . . . . . . . X . X Y . X . X . X
                  .                   .                   .

. . . . . . . . . . . . . . . . . A . A . A . A B . A . B . A K
0 0 1 0 1 1 0 1   . 0 0 0 0 1 0 0 0   . 0 0 0 1 0 0 1 0   .
. . . . . . . . . . X . Y . . . . . . X . Y . X . X Y . X .
                  .                   .                   .

. . . . . . . . . . A . A . B . A A . B . A . B A . B . L
0 1 0 0 0 0 0 1   . 0 0 1 0 2 0 0 0   . 0 0 1 1 0 0 1 0   X
. . . . . . . . . X X . Y . X . Y . . . Y . X X . Y . X .
                  .                   .                   X
```

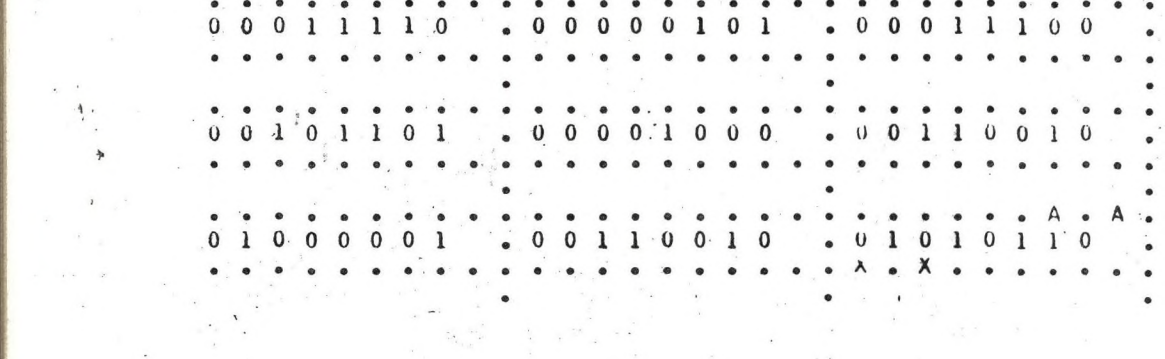
```
LEPES=  48, MERET= 12 *  30, CSUCSOK=(  2,  2)( 13, 31),***
```




```
LEPES=  53, MERET= 12 *  30, CSUCSOK=(  2,  2)( 13, 31),***
```




```
LEPES=  58, MERET= 12 *  30, CSUCSOK=(  2,  2)( 13, 31),***
```



Kerékfy Pál–Ruda Mihály

JAVASLAT EGY PROGRAMSZERKESZTÉSI MÓDSZERRE

Dolgozatunkban egy programszerkesztési módszert mutatunk be. Első lépéseinket ezen az úton az országos kórházi morbiditási adatok feldolgozása kapcsán tettük meg, a SIS77 statisztikai információs rendszer létrehozásával. Az itt szerzett tapasztalatok alapján dolgoztuk ki a GENERA programszerkesztő rendszert, és inditottuk meg a SIS77 továbbfejlesztett változatának (SIS79/GENERA) létrehozását. Legfőbb törekvésünk az volt, hogy a programgenerálást felhasználói szinten valósitsuk meg, lehetővé téve ezzel a felhasználó aktiv részvételét a futó rendszer kialakitásában.

Kulcsszavak: programgenerálás, befogadó nyelv, felhasználási hatékonyság.

1. Bevezetés

Napjainkban programozási nyelvek, programrendszerek és a hozzájuk tartozó elméletek áradata lepi el a felhasználót. Az egyre bonyolultabb feladatok sokasodása, a számitástechnikával szemben támasztott igények rohamos növekedése egyaránt segiti és indokolja a szoftver és a számitástudomány dinamikus fejlődését. Ebben a folyamatban - melyet esetenként inkább (szoftver) krizisnek, mintsem fejlődésnek neveznek [2] - fontos feladat azoknak a lehetőségeknek a megragadása, amelyek a rendelkezésre álló eszközök hatékony felhasználását segitik elő.

Mielőtt rátérnénk az előadásunk tárgyát képező programszerkesztési eljárás bemutatására, röviden érintjük a

szoftverfejlesztés és a felhasználás néhány, minket legközelebbről érintő kérdését.

2. A szoftverfejlesztés és a szoftverfelhasználás hatékonysága

Egy számitógépes rendszer hatékonysága számtalan tényezőtől függ, és nem csak a hatékony gépkihasználást jelenti. (Ilyen kérdésekkel a statisztikai adatfeldolgozás problémáihoz kapcsolódva, bővebben [8] foglalkozik.) Vegyünk sorra néhány fontosabb szempontot!

a./ Be kell látnunk, hogy az általánosság igénye nem mindig egyeztethető össze a kényelmes felhasználás lehetőségével. A teljes komfortot biztositó rendszerek általában merevek, egy adott célt szolgálnak, nehezen kiegészithetők.

b./ A különböző késztermékek közül választani, megállapitani, hogy a kiválasztott számitógépes rendszer a konkrét felhasználói igényeket kielégiti-e, a kiválasztott rendszert implementálni, alkalmazását elsajátitani rendkivül sok felhasználói energiát köt le.

c./ A felhasználás tudatosságának problémája. A bonyolult programrendszerek alkalmazásakor a legtöbb felhasználó előtt misztikus homályba burkolódzik a rendszer müködésének mechanizmusa. Igy részben sok értékes tulajdonsága kihasználatlan marad, részben hibásan értelmezett alkalmazások és eredmények születnek. Az elemi épitőkövekre lebontott rendszerek - mint a makró nyelvek, preprocesszorok (célnyelvek), program (szubrutin) könyvtárak - sokkal inkább kényszeritik a felhasználót a tudatos alkalmazási munkára, bár egyúttal több programozói munkát is igényelnek.

d./ Bármely müködő rendszernél jogos igény a továbbfejleszthetőség, kiegészithetőség, az hogy az eredmények más rendszerek számára is hozzáférhetők legyenek, stb. Ezt az igényt talán még az egyszerü preprocesszorok sem teljesitik.

Röviden összefoglalva a fenti problémákat, azt mondhatjuk, hogy a zárt rendszerek, melyek teljes felhasználói kényelmet biztositanak, elsősorban rutinfeladatok, illetve egy-egy jól körülhatárolt feladatkör megoldásánál alkalmazhatók sikeresen. Dinamikus felhasználói tevékenységet igénylő területeken reménytelen dolog előregyártott komplet rendszerek teljeskörü alkalmazhatóságában bizni. A folyton változó körülmények és igények, az előre pontosan meg nem határozható feladatok olyan eszközöket igényelnek, amelyek nem közvetlenül egy-egy konkrét feladat megoldását, hanem általában a szoftverfejlesztői munkát támogatják. A szerzők ilyen jellegü problémákkal találkoztak orvosi statisztikai adatfeldolgozások során [6]. Itt kell megemliteni a szoftverfejlesztésnek azt az oldalát, amely nem közvetlenül az alkalmazási rendszerek létrehozására, hanem azok leirására, a bonyolult fogalomrendszerek gépi reprezentálására irányulnak [5].

3. Javaslat egy szoftverfejlesztési módszerre

Az előző pontban felvetett hatékonysági problémák a következőképpen fogalmazhatók meg pozitiv követelmények formájában:

Olyan rendszer létrehozására törekedtünk, amely

a./ nem annyira általános, hogy használatának elsajátitása nagy munkaráforditást, üzemeltetése pedig irreálisan nagy gépkapacitást igényelne,

b./ a felhasználó számára áttekinthető, egyszerü szerkezetü,

c./ az elért eredmények megbizhatók,

d./ üzemeltetése nem igényel sok elemi hibalehetőséget magában rejtő részletmunkát,

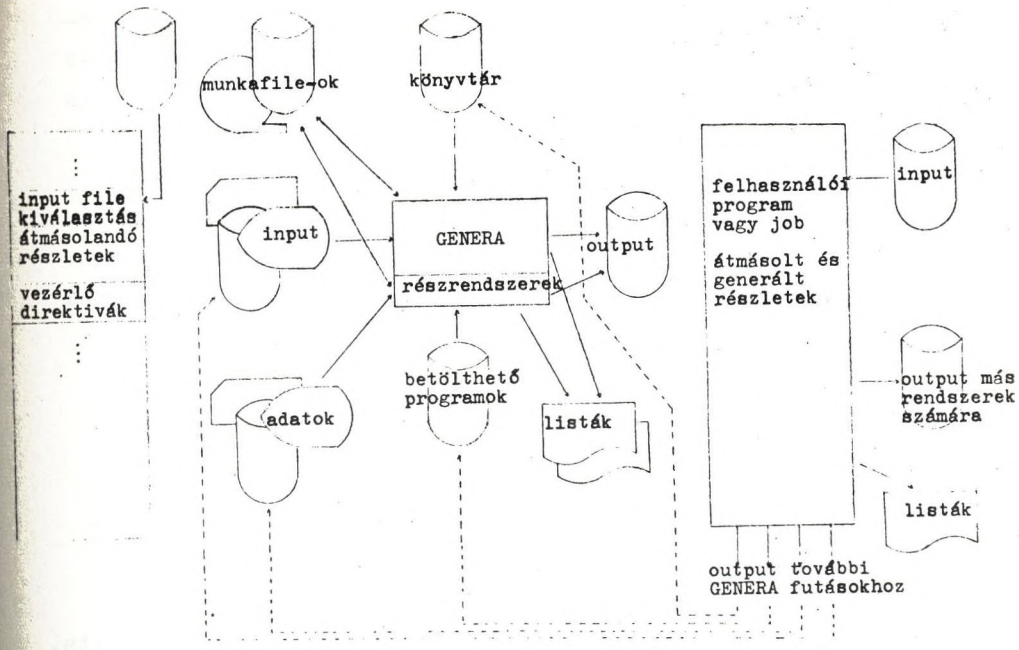
e./ más rendszerek irányába nyilt, könnyen továbbfejleszthető, alakitható.

Az általunk javasolt módszer lényegében egy egyszerü programszerkesztési eljáráson alapul [3], amelyet először

nagy adatrendszerek statisztikai vizsgálatánál alkalmaztuk [4], [6]. A programszerkesztés kezdetben [7] olyan programok létrehozását szolgálta, amelyek általános feltételek mellett is igen hatékonyan működnek. A hatékonyság azonban nem csak a hatékony gépkihasználást jelenti, hanem a rugalmas és kényelmes felhasználhatóságot is. A most bemutatásra kerülő GENERA rendszer éppen ezt az utóbbi célt szolgálja. A GENERA befogadó nyelvbe ("host language") ágyazott célnyelvi utasitások (eljárások) feldolgozását, szerkesztését, makró- és szubrutinkönyvtár használatát, illetve job-ok szerkesztését támogatja. Emellett a célnyelvi eljárások létrehozását és rendszerré történő szervezését is segiti.

Alapelvünk az volt, hogy az elemi feladatokat, melyek nagyon változatosak lehetnek, legcélszerűbb elemi uton, a szokásos programozási nyelvek valamelyikén leirni. A munkaigényes, de előre megfogalmazható tipikus feladatokat előre elkészitett eljárások oldják meg. Erre legcélszerűbb szubrutinkönyvtárakat, makrókönyvtárakat és magasszintü célnyelveket (preprocesszorokat) használni. Ezen a szinten azonban túlságosan sok szervezési feladat hárul a felhasználóra. Itt nyujt segitséget a GENERA.

A GENERA rendszer a következő részekből áll (ld. az alábbi ábrát). Tartalmaz egy fájlkezelő rendszert és egy előfeldolgozót (preprocesszort). Az ezek által feldolgozott input egy befogadó nyelven (pl. PL/1, FORTRAN) irt program és a generáló eljárásokat vezérlő direktivákat is tartalmazza. A direktivákkal kifejezett parancsokat az installációs eljárás során definiált részrendszerek hajtják végre. Ezen a ponton épülhetnek be könyvtári eljárások is. A feldolgozást meghatározó paraméterek részben a generálást vezérlő direktivákban, részben külön adatállományokban lehetnek. Ezzel a módszerrel a felhasználó teljes mértékben ellenőrizheti a rendszer működését, mivel a programok elemi részekből (a befogadó nyelv utasitásaiból) és generáló direktivákból állnak össze. A generált prog-



ram végrehajtásához szükséges környezetet (job control utasitások, program hivások, stb.) a rendszer az installáció során meghatározott vagy a felhasználó által az input fájlon adott módon épiti fel. Mivel a direktivák halmaza tetszőlegesen bővithető új részrendszerek beépitésével, és a felhasználó tetszőleges utasitásokat helyezhet el közöttük, a generált programok kapcsolatban állhatnak bármely más programtermékkel, és új generáló eljárások is létrehozhatók.

Legfőbb törekvésünk az volt, hogy a programgenerálást felhasználói szinten valósitsuk meg, lehetővé téve ezzel a felhasználó aktiv részvételét a futó rendszer kialakitásában [1].

A fenti ábra az input és output fájlokat és az adatáramlást mutatja egy a GENERÁ-ra alapozott rendszerben. A baloldalon az input állomány szerkezete látható. Ez átmásolandó programrészekből, a generáló eljárásokat vezérlő

direktivákból és az input állomány egyes külön tárolt részeit kijelölő utasitásokból áll. A munkafájlok azokat az eljárásokat és adatokat tartalmazzák, amelyek többször is felhasználhatók a generálás folyamán. A jobboldalon az output állomány szerkezete látható. Ez általában egy programot vagy egy job-ot tartalmaz, a végrehajtáshoz szükséges input és output definiciókkal együtt. A létrehozott program kapcsolatban állhat bármely más programmal vagy akár magával a generáló programmal is.

4. A GENERA alkalmazási lehetőségei

Programgenerálási eljárást először az országos kórházi morbiditási vizsgálatok adatainak feldolgozásánál használtunk. Létrehoztuk a SIS77 statisztikai információs rendszert [7], amelyet az ESZTIK 1977 óta folyamatosan üzemeltet. Generálási eljárások alkalmazását az indokolta, hogy a morbiditási vizsgálatok számos, egymáshoz nagymértékben hasonló, de az idők folyamán állandóan változó rutinfeladat megoldását igényelték (adatelőkészitési, ellenőrzési, válogatási, táblázási feladatok). Általános, paraméterezhető eljárások alkalmazása az adatok viszonylag nagy tömege miatt nehézkes lett volna. Ezért a legtipikusabb feladatokra eljárásgeneráló programokat készitettünk, amelyek mindig a pillanatnyi igényeknek megfelelő programokat generálták. Ilyen módon vált például lehetővé az országos adatok terminálos lekérdezése is.

Rendszerünk egyik gyengéje az volt, hogy felhasználói szinten nem fejleszthető tovább. Ezért hoztuk létre a GENERA rendszert, amely önmagában csupán a programszerkesztés adminisztrációját látja el - de igy a legkülönbözőbb területeken használható fel [1].

Végezetül bemutatunk egy példát a GENERA alkalmazására. Ez a SIS77-nek megfelelő SIS79 (Statistical Information System '79) rendszer használatát illusztrálja. Az alábbi program vegyesen tartalmaz FORTRAN és SIS79 utasitásokat.

```
#OPTION
 $PARAM LIST="ERROR",SYSTEM="FORTRAN"$
      INTEGER AGE,HYEAR,BYEAR,SEX,PROFS,
     *CODE,MAINCD,SUBCD,ERROR(1)
    1 CONTINUE
#LECTOR
 $PARAM FC=5,END=5ØØ,ERR=6ØØ$
 $DESCR PATIENT
        1 NAME 3ØX
        1 BYEAR I4
        1 SEX I1
        1 COUNTY 2X
        1 PROFS I4
        1 HDATE
          2 HYEAR I4
          2 HMONTH 2X
          2 HDAY 2X
        1 CODE I4
          2 MAINCD I3
          2 SUBCD I1
 $
      AGE = HYEAR - BYEAR
#GRAPH
 $PARAM GRAPH="AGE CODING",DATANA="AGE",
NEWDAT="CDAGE",SACKNO=1,LEVELS=1,UPPBOU=1ØØ$
      IF (NUMERR.NE.Ø) GO TO 1ØØ
#GRAPH
 $PARAM GRAPH="CONTROL",DATANA="CDAGE","SEX",
"MAINCD","SUBCD",SACKNO=34,LEVELS=1,1Ø,15,8,
UPPBOU=1Ø,1Ø*1Ø,896,788,1Ø*999,618,528,496,2*7,6*9,
LOWBOU(2)=1Ø*1,LOWBOU(15)=125$
      IF (NUMERR.NE.Ø) GO TO 1Ø1
      GO TO 1
  1ØØ WRITE (6,1Ø) ERROR(1)
   1Ø FORMAT(" ERROR IN AGE:",I4)
      GO TO 1
```

```
1Ø1 WRITE (6,11) ERROR(1)
 11 FORMAT(" ERROR:",I4)
    GO TO 1
6ØØ WRITE (6,12)
 12 FORMAT(" READ ERROR")
    GO TO 1
5ØØ STOP
    END
```

Az #OPTION direktiva deklarációs jellegü; a példában a listán csak az esetleges hibaüzenetek jelennek meg és az elkészült programot a FORTRAN fordító fogja leforditani (LIST="ERROR",SYSTEM="FORTRAN"). A #LECTOR direktivát követő $PARAM halmaz jelentése a példából önként adódik. A $DESCR alatt a PATIENT nevü rekord szerkezetét adjuk meg. A #LECTOR részrendszer gyors olvasó eljárást hoz létre a FORTRAN programban. A #GRAPH direktivát részletesen [4] tárgyalja. Itt csak annyit mondunk róla, hogy többváltozós függvények (például kiválasztási szempontok) leirására és kiértékelésére szolgál. A példában mutatott első #GRAPH direktiva az "AGE CODING" nevü eljárást hozza létre. A függvénynek egyetlen argumentuma van: AGE és a CDAGE változó kap értéket. A SACKNO és a LEVELS paraméter a függvény (vagyis egy a függvényt reprezentáló gráf) szerkezetét irja le, amely most egyetlen értéktáblázatból áll. A táblázatban az 1 és 100 közé eső argumentumok kapnak értéket (UPPBOU=100), például egy életkor-korcsoport leképezéshez. A második #GRAPH direktivában leirt eljárás neve "CONTROL", ennek négy változója van (a CDAGE, SEX, MAINCD és SUBCD változók - pl. korcsoport, nem, diagnóziskódok kapcsolatát vizsgálja). A függvényt 34 elemi értéktáblázat irja le (SACKNO=34). A négy argumentumnak rendre 1, 10, 15, 8 résztáblázat felel meg. Az argumentumok alsó és felső korlátjai a LOWBOU illetve az UPPBOU tömbben vannak. A NUMERR és az ERROR paraméter a hibakezelést szolgálja. Az értéktáblázatok tartalmát a szerkesztő eljárás külön adatállományból olvassa be (ld. az ábrát).

Abstract

A program generator method is described in this paper. At the very beginning this method was applied to build statistical information system SIS77 for processing of the national hospital morbidity data. Based upon the experiences obtained, program generator system GENERA was created and we set about building the improved system SIS79/GENERA. Our main ambition was to realise program generating on the users' level. So, the user can take part in development of the application system.

Hivatkozások

[1] J. Demetrovics, P. Kerékfy, M. Ruda, Some Remarks on Software Means in Computer-Aided Design, IFIP WG. 5.2 Working Conference on CAD System Frameworks /megjelenőben/.

[2] J. Foisseau, R. Jacquart, M. Lemaitre, M. Lemonie, J. C. Vignat, G. Zanon, Program development with or without coding, Software World, Vol. 12., No.1., pp. 9-12., 1981.

[3] P. Kerékfy, GENERA - A Program Generator System, Progress in Cybernetics and System Research, Vol. 11., Hemisphere, Washington /megjelenőben/.

[4] P. Kerékfy, A. Krámli, M. Ruda, SIS79/GENERA Statistical Information System, Progress in Cybernetics and System Research, Vol. 11., Hemisphere, Washington /megjelenőben/.

[5] E. Knuth, P. Radó, A. Tóth, Preliminary Description of SDLA, MTA SZTAKI Tanulmányok, 105/1980., Budapest.

[6] A. Krámli, M. Ruda, M. Csukás, M. Galambos, Large Sample Size Statistical Information System for HwB, Data Analysis and Informatics, ed. E. Diday, North-Holland, pp. 457-462., 1980.

[7] M. Ruda, Statistical Information System with Health Service Application, MTA SZTAKI Tanulmányok, 87/ 1978., pp. 167-172., Budapest.

[8] Ruda M., Optimalizálási kérdések a statisztikai adatfeldolgozásban, VIII. Magyar Operációkutatási Konferencia, 1978. /megj.: MTA SZTAKI Tanulmányok, 110/1980., pp. 5-18., Budapest/

Kerékfy Pál Ruda Mihály

Magyar Tudományos Akadémia
Számitástechnikai és Automatizálási Kutató Intézete
Budapest

Neumann János Számítógéptudományi Társaság

PROGRAMOZÁSI RENDSZEREK '81.

Konferencia előadásai

II. Kötet

Szeged
1981. december 2–4.

Szerkesztette:
Dávid Gábor

Kertész Ádám

TAPASZTALATOK EGY ADATFELDOLGOZÁSI RENDSZER KÉSZÍTÉSE SORÁN

Számitástechnikai folyóirataink, könyveink - hasonlóan a világ bármely táján megjelenő irásokhoz, konferencia-előadásokhoz - egyre erősödő hangon hivják fel figyelmünket az ún. szoftverkrizisre. Ez alatt általában azt a - jelenleg is még tágulónak tünő - szakadékot értjük, amely a nagyüzemi, termelékeny, hatékonyságában megsokszorozódó hardver-gyártást és az ebből adódó fokozott technikai lehetőségeket elválasztja az ilyen jelzőkkel még nem dicsekedhető szoftver-készitéstől.

Nem véletlen, sőt szándékos az előbbi mondatokban a többesszám első személy használata. Ez a figyelem-felhivás, ez a szakadék-felismerés - néhány tiszteletreméltó, de ritka kivételtől eltekintve - csak a számitástechnikát müvelők körét érinti, és szerencsére legfeljebb olvasmányi szinten riasztgatja a számitástechnikát alkalmazók: a felhasználók egyre szélesedő körét.

Azért szerencse ez, mert a felhasználók, akik érdekében / s nem öncéluan / valósitottuk meg a számitástechnika teljes körét, épp elég problémával, gonddal-bajjal küzdenek meg ahhoz, hogy a mienket már ne kivánják vállukra venni. Számukra a számitástechnika - processzoraival, csatornáival, bit-

jeivel és programjaival együtt - eszköz végső termelési céljuk jobb megoldása érdekében. Ettől - a mi szivünknek olyannyira kedves - termelési eszköztől sok mindent várhatnának el: gyorsaságot, pontosságot, néha még munkaerő-megtakaritást is; de elsősorban, mint minden eszköztől, azt várják el, hogy használható legyen. / Talán nem kell hozzátennem, hogy a magyar nyelv ereje miatt ez a hasznosságot is magában foglalja. /

Ez a"használható" fogalom első hallásra nagyon tágnak, nehezen definiálhatónak tünhet számunkra. Ha azonban megpróbáljuk a felhasználó szemszögéből nézni a dolgokat, alapvetően két tulajdonság együttes meglétét jelenti:

- azt a munkát, amit el akarunk ezzel az eszközzel végezni, könnyen, "jól kézzelfoghatóan" végezze el, és
- ha kicsit változik a megoldandó feladat, ne okozzon gondot szerszámunk átállitása.

Hadd nevezzem az első tulajdonságot emberségességnek és a másodikat módosithatóságnak. E két tulajdonság jelentőségét, fontosságát egy példával kivánom szemléltetni.

4

1979-től kezdve részt veszek egy adatfeldolgozási rendszer kialakitásában. A Magyar Nemzeti Bank Devizakiutalási Osztályán a bank külföldi valutáris tranzakcióit bonyolitják le. Ez a munka - a pénzintézetek szokásos biztonsági intézkedései miatt - többek között megköveteli azt is, hogy a tranzakciókban résztvevő bankokat mind táviratban, mind telexen értesitsék az átutalt összegekről, nem felejtve ki természetesen az MNB saját számitóközpontját sem. A többszörös adatgyüjtés és -kibocsátás manuális végzése számos hibalehetőségre adott alkalmat, s az előforduló hibák - a mozgatott értékek nagyságrendje miatt - komoly gondokat okozhattak. Természetes módon merült fel a gépesitès szükségessége, amely a hibák várható kiszürése mellett gyorsaságot és munkaerő-megtakaritást is jelentene. A jelenlegi számitógépes rendszer kötegelt jellegü feldolgozása nem bizonyult megfelelőnek e munka elvégzésére, ezért egy on-line, interaktiv rendszer bevezetése mellett döntöttek. Ugyanakkor nem állt szándékukban - legalábbis egy ideig - a gép, az alkalmazandó programozási nyelv, egyszóval a "mélyebb" számitástechnika megismerése: <u>csak</u> használni kivánták azt céljaik elérésére.

Komor Tamás–Molnár Máté

A MOZ–ART TECHNOLÓGIA KIALAKULÁSA ÉS ALKALMAZÁSA

Rövid kivonat: az előadás a MOZ-ART technológia kialakulását és néhány alkalmazását ismerteti. A MOZ-ART /MOZaik Alkalmazási Rendszer Technológia/ technológia az adatfeldolgozás területén dolgozó programozók munkáját támogatja.

A technológia fő összetevői:

- Jackson programtervezési módszer;
- Fagan-féle terv- és kódinspekció;
- nyelvi ajánlások /PL/I, COBOL/;
- csoportmunka.

Az előadás a technológia kialakulásának ismertetése után vázlatosan bemutatja a technológiai folyamatot. Beszámol néhány alkalmazásról, majd ezek alapján felhivja a figyelmet néhány szempontra, amelyekre az alkalmazásoknál ügyelni kell. Végül ismerteti a technológia alkalmazásának előnyeit.

Kulcsszavak: programozási technológiák, Jackson-módszer, programinspekciók, struktúrált programozás.

Bevezetés

Az elmúlt évtizedben a programkészités gyakorlata gyökeresen átalakult. A hardver árának drasztikus csökkenése miatt a felhasználók köre ugrásszerüen megnőtt; a szoftver eladásban hatalmas emelkedés tapasztalható. Az egyedi felhasználás helyett egyre nagyobb az igény a tömeges alkalmazásra, az egyedi programok készitése helyett az áruszerü, megbizható szoftver-termékek előállitása a feladat. A szoftver-készités áttérése a manufakturális szintről a tömeggyártásra nagyon sok kudarccal és megrázkódtatással jár. A számitástechnika nélkülözi azt a sokszázéves kutató-fejlesztő munkát és gya-

korlati tapasztalatot, ami ma minden hagyományos mérnöki munkában megtestesül.

Az ebből következő problémák a hazai szoftver-fejlesztésben is jelentkeznek. A rendszerszervező nagyon ritkán kapja meg a feladat pontos megfogalmazását a megrendelőtől, gyakran csak egy ellentmondásokkal teli, elemzéseket nélkülöző vázlat áll rendelkezésre. A gyakorlott programozó "szerencsére" tudja, hogy mit kezdjen az ilyen leírással. Kódolás és tesztelés közben saját maga próbálja az ellentmondásokat feloldani. Konkrét előírások hiányában munkája a szükségesnél erősebben intuitiv, megfelelő szerszámok hiányában termelékenysége igen alacsony. A program verfikálására csak az üzemeltetés beindulásakor kerül sor: ekkor derül ki, hogy a megrendelő nem azt kapja, amire szüksége van. A programot javitani kell, de a szerző éppen nem elérhető és mások számára a program nem hozzáférhető.

Ezek a problémák már több éve ismeretesek, de sajnos még mindig aktuálisak. Sőt, nemcsak itthon, hanem a számitástechnikai fejlődésben előttünk járó országokban is megfigyelhetők. Például az idén Karlsruhe-ban megtartott programozástechnikai konferencia [Go81] előadásai és kerekasztal vitája jól mutatta, hogy az NSZK-ban rendelkezésre álló fejlettebb hardver és szoftver eszközök ellenére a gyakorlati programfejlesztésben az itthonihoz hasonló problémákkal küszködnek.

A gondokat felismerve a Számitástechnikai Központi Célprogram keretében a KSH már a negyedik ötéves tervben több olyan kutató-fejlesztő tevékenységet inditott el, amelyek célja az ipari méretü szoftver-fejlesztés kialakitása volt [Ha77]. Ezek keretében 1977 közepén kezdődött meg a munka az egységes Alkalmazói Rendszer Technológia /ART/ kialakaitására. A kiinduló koncepciót 1978-ban az ART-csoport dolgozta ki, elsősorban a SZÁMKI, MHE-SzSzK, NOTO OSZV, SZÁMOK és SZKI kutatásaira támaszkodva. Az ART-csoport kétféle technológia - mozaik és monolit - bevezetését szorgalmazta. A monolit technológia egy-

séges, egy cég által forgalmazott rendszer /pl. PROTEE, METACOBOL/, a mozaik technológia pedig több, eredetileg különálló módszert és eszközt foglal össze egységes keretbe.

A MOZ-ART /Mozaik-ART/ a mozaik technológiák közé tartozik, a SZÁMKI és a SZÁMOK közösen dolgozta ki /lásd [Mé79]/. Az előadásban nem kívánjuk - nincs is módunk - a technológiát részletesen ismertetni. A MOZ-ART kialakitásának feltételeit tárgyaljuk röviden és az elmúlt évek során összegyült tapasztalatokat értékeljük.

A MOZ-ART technológia célja és felépitése

A MOZ-ART kialakitásakor 1979-ben a következő célokat kellett szem előtt tartanunk. A technológiának támogatnia, szabályoznia kell a programfejlesztési tevékenységet, elő kell segitenie jó minőségü programrendszerek készitését. A programok minőségét sok tényező befolyásolja /lásd pl. [Bo78]/, mi elsősorban a következő tulajdonságokat emeljük ki. A jó minőségü program

- korrekt, kielégiti a felhasználó igényeit, funkcionálisan és a hatékonyság szempontjából is;
- időben bevezetésre kerül, amikor a megoldandó feladat még nem vesztette el aktualitását;
- karbantartható, a változó körülmények miatt fellépő, vagy esetleges hibából származó módositásokat a rendszerben biztonságosan át lehet vezetni, szükség esetén a fejlesztőktől függetlenül is.

Ilyen programok készitését csak olyan technológia biztosithatja, amely szabályozza legalább a

- tervezési módszert;
- kódolási stilust;
- dokumentálást;
- csoportmunka végzését.

Természetesen a feladatkitüzéskor világos volt, hogy univerzális technológia kialakitása nem lehetséges. A MOZ-ART-ra vonatkozó korlátozó feltételek a következők voltak.

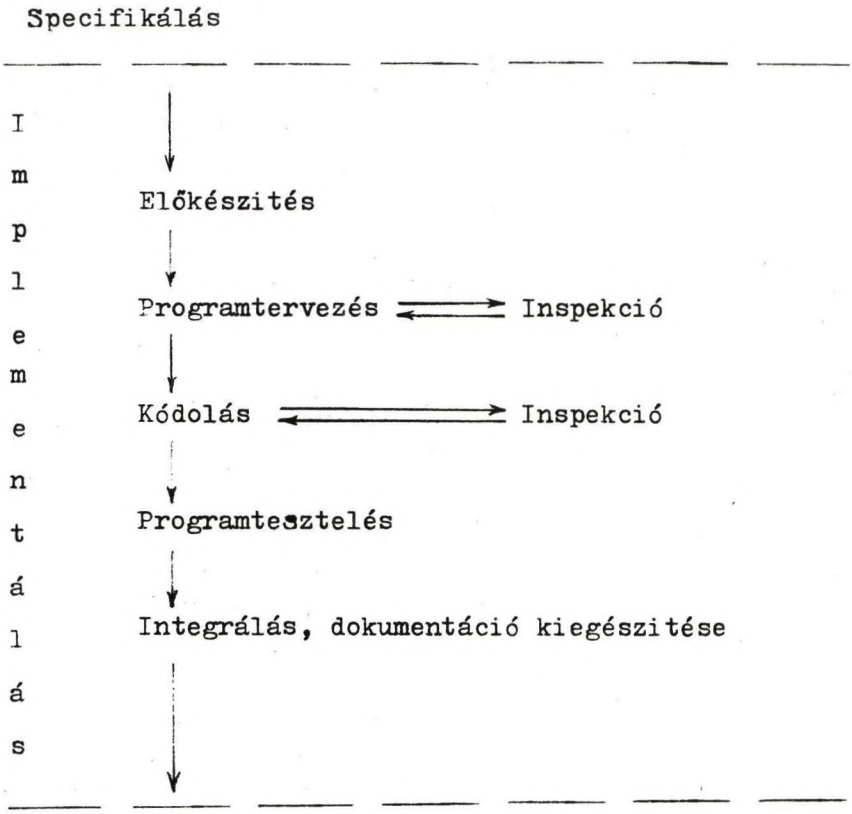
- Tipikus adatfeldolgozási programrendszerek kidolgozását kell támogatni.
- A rendelkezésre álló ESZR gépekre kell támaszkodni, figyelembevéve, hogy a DOS és OS rendszert itthon egyaránt széles körben alkalmazzák.
- A technológiát gyorsan kell létrehozni és a gyakorlatban alkalmazni.

Ez utóbbi feltételből következett, hogy csak meglévő, a - hazai vagy nemzetközi - gyakorlatban már bevált technológiai elemeket, módszereket és eszközöket vehettünk figyelembe. Ennek megfelelően a technológiát saját tapasztalatainkon kivül a következő alapokra épitettük fel:

- Jackson programtervezési módszer [Ja75],
- Weinberg és Kernighan-Plauger kódolási ajánlásai [We7o, KP74],
- ARDOSZ '79 dokumentációs útmutató [Ke79],
- Fagan inspekciós módszere [Fa76].

Az 1. ábrán vázoljuk a programfejlesztés folyamatát a MOZ-ART technológia szerint. A specifikálás magábanfoglalja a követelmény elemzést és rendszertervezést. E szakasz eredményeképp elkészül a rendszerterv, amelynek átvétele és ellenőrzése az előkészités egyik feladata.

Az előkészités során kell létrehozni a gépi környezetet /szabványos jobok, közös adatok leirása központilag kezelt könyvtárban, közösen használt eljárások és tesztsegédletek, stb./, ki kell osztani a feladatokat és fel kell készitenie a munkára a csoport tagjait. Mindezekre a feladatokra a technológia kézikönyve [Es81] ajánlásokat tartalmaz. A felkészités különösen akkor fontos, ha a csoport tagjai először alkalmazzák a technológiát.



1. ábra

A programfejlesztés folyamata a MOZ-ART technológiában

A programok tervezése a Jackson-módszer előirásai szerint történik. A módszer oktatása a felkészités része, új alkalmazók számára egyhetes intenziv tanfolyamot kell tartani a kidolgozott metodika szerint.

A tervet az inspekció során a csoport néhány tagja megvizsgálja. A hibák korai felfedezésén kivül ez lehetővé teszi az egységes stilus erősitését és a munka haladásának érdemi figyelemmel kisérését. Az inspekció előkészitését és lefolytatását, a kitöltendő formanyomtatványokat a technológia kézikönyvében szabályoztuk.

A kódolás a MOZ-ART ajánlások megtartásával történik, a szintaktikusan helyes kódot és a részletes tesztelési tervet vizsgálja az újabb inspekció.

A rendszer korrekt voltát a fejlesztők a megrendelő által előirt valódi tesztadatokon mutatja be, ezután a rendszert üzemeltetésre átadjuk.

A technológia alkalmazásai

Az alábbiakban leirjuk a technológia néhány tipikus alkalmazását. Részletesebb adatok [Su8o] -ban találhatók.

1. Oktatási információs rendszer felvételi alrendszere.

   Az alrendszer feladata a jelentkezési lapok ellenőrzése, a jelentkezők beosztása felvételi vizsgákra, a tesztek értékelése, a tanfolyami névsorok összeállitása különböző szempontok szerint.
   Adatok az implementációról:
   Az alrendszer felhasználja a SÁMÁN adatbáziskezelő rendszert.
   Csoportlétszám: hat.
   Méret: 24 program, 16 tabló, 1115o forrássor.
   Ráforditott kapacitás: 21 emberhó.
   Termelékenység: 24 sor/embernap.

2. FILTER általános inputellenőrző és javitóprogram.

A program feladata adatrögzités után keletkező fájlok ellenőrzése. A program felderiti a rekordokon belüli hibákat és rekordcsoportok hibáit, lehetőséget ad a hibás rekordok módositására, illetve törlésére.
Adatok az implementációról:
Csoportlétszám: 4.
Méret: 8 modul, 295o forrásnyelvi sor.
Ráforditott kapacitás: 6 emberhó.
Termelékenység: 22 sor/embernap.

3. Szállitási alrendszer.

Az alrendszer feladata menetlevelek ellenőrzése és feldolgozása. A menetlevelek alapján elllenőrzi és aktualizálja a járművek és vezetők adatait tartalmazó file-okat és különböző /teljesitmény, önköltség, üzemanyagfelhasználás, stb./ táblákat készit.
Adatok az implementációról:
Csoportlétszám: 6.
Méret: 23 program, 12 tabló, loloo forrásnyelvi sor.
Ráforditott kapacitás: 3o emberhó.
Termelékenység: 15 sor/nap.

Mire kell ügyelni a MOZ-ART technológia alkalmazásánál?

Az alábbiakban felsoroljuk azokat a fő szempontokat, amelyeket a technológia bevezetésekor figyelembe kell venni. Ezek elmulasztása lényegesen ronthatja a technológia eredményességét.

- Külső feltételek biztositása: hely a team számára, a teamtagok egyértelmü hozzárendelése a teamhez, megfelelő géphezfordulás, másolási lehetőség a tervinspekció anyagaira. Ezek a feltételek természetesnek tünnek, mégis volt olyan tapasztalatunk, hogy a biztositásuk néha komoly problémát jelent.

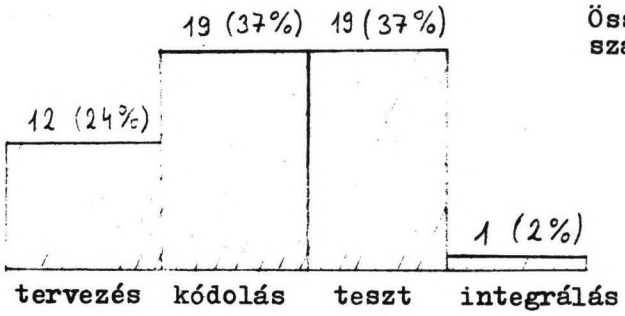
- Specifikáció biztositása. Sok esetben az implementáció inditásakor nem áll rendelkezésre a megfelelő minőségü részletes rendszerterv. Ilyenkor még a programtervezés elkezdése előtt a specifikációt át kell dolgozni. Az ehhez szükséges kapacitást és időt figyelembe kell venni a munka ütemezésénél.
- Integrációs teszt feltételeinek biztositása. Az integrációs teszt akkor jó, ha körülményei megközelitik a tényleges üzemeltetés körülményeit. Ehhez többek között viszonylag nagyobb mennyiségü tesztadat szükséges, amelyek előállitása munkaigényes feladat. Hasonló a helyzet a kapott futási eredmények ellenőrzésével. E feladatok megoldásában kivánatos a megrendelőnek /ill. a specifikáció készitőjének/ a részvétele, ugyanis az elkészült rendszert igy lehet a leghatékonyabban átvenni.
- Ügyeljünk arra, hogy semmi ne maradjon ki a technológiai folyamatból! Tapasztalataink szerint a problémák azokkal a programrészekkel vannak, amelyek valamely oknál fogva nem kerültek inspekcióra.
- Ne aggódjunk amiatt, hogy a kódolás és a tesztelés később kezdődik! A program megtervezésével és inspekciózásával a feladat jelentős részét már megoldottuk.

A MOZ-ART alkalmazásának értékelése

1. Az elkészült programok a szabályozott tervezési és kódolási stilus,valamint az inspekciók által biztositott közös ellenőrzés eredményeképpen karbantarthatóak, azaz relative hibátlanok, könnyen érthetőek, módosithatóak, optimalizálhatók. Példaként emlithetjük, hogy a FILTER program módositását-optimalizálását a fejlesztő team egy tagja a többiektől függetlenül sikeresen végezte el [Hb81].

2. A programkészités nehezen tervezhető és becsülhető szakasza általában a tesztelés, mert a programozó által elkö-

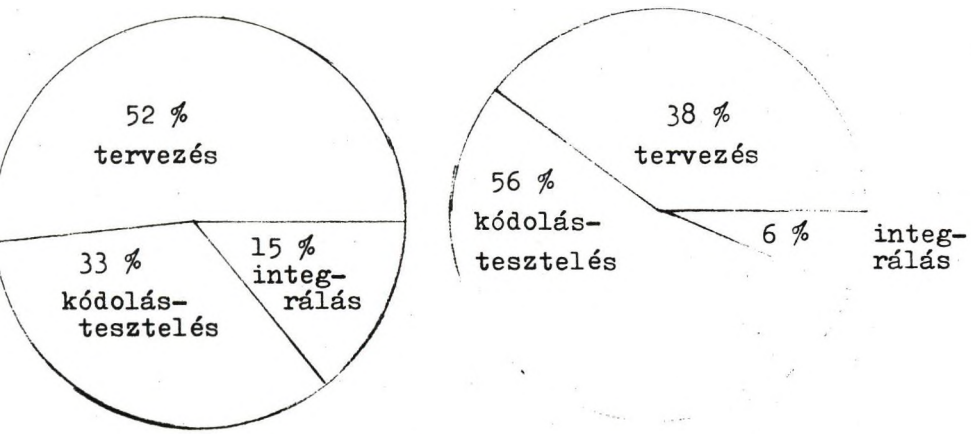
vetett ismeretlen mennyiségü hiba felderitése a feladata. A MOZ-ART technológia alkalmazásával azonban a hibák jelentős része már az inspekciók során kiderül, tehát kisebb "teher" hárul a tesztelésre. Ezt illusztrálja a hibák munkafázisonkénti eloszlása a FILTER projektben /2. ábra/, amelynek arányai a többi alkalmazásra is jellemző. A hibák 6o %-át még a tesztelés megkezdése előtt fedezték fel. Ez természetesen az átlagosnál lényegesen kisebb gépidőfelhasználást is eredményez.



2. ábra
Hibák megoszlása munkafázisokként a FILTER projektben

3. A munka állása /mit végeztünk már el? mi van előttünk?/ jól becsülhető. A becsléshez fontos tudni, hogy az egyes munkafázisok a teljes munka mekkora részét teszik ki. Erre vonatkozóan folyamatosan gyüjtjük az adatokat. A 3. ábra szemlélteti az egyes munkafázisok kapacitásigényének arányát két projektben. Az eltérések alapvetően az eltérő géphezfordulási lehetőségekből adódtak, rossz /napi egyszeri és kevesebb/ fordulás esetén még több feladat párhuzamos futtatása mellett is a programozó kapacitása

a kódolás és tesztelés idején részben kihasználatlan. Megjegyezzük, hogy a kapacitásarányokkal az időarányok csak az integrálásig egyeznek meg, mert az integrálásban már nem az egész csoport vesz részt, és igy az időegységre eső kapacitás ilyenkor kisebb.



3. ábra

Kapacitáseloszlás fázisonként a felvételi alrendszer és a FILTER program kidolgozásában

- A technológia rendszeres használata biztositja az új projektek jobb kapacitás és átfutási idő becslését. A MOZ-ART technológiához egy bizonylati rendszer tartozik, amely segitségével a megoldott feladatok jellemzőit, méreteit és a megoldásukra forditott kapacitást rögzitjük. Ezen adatok és az új projekteknek a korábbiakkal való összehasonlitása alapján biztonságosabban tudunk becsülni.

- A dokumentáció nagy része a tervezéssel párhuzamosan elkészül.

6. Az előállitott programok nem csak "személyes" termékek, hanem a csoport közös munkájának az eredményei /vö. "egoless programming" [We79]-ben/.

7. A felkészitő oktatás és az inspekciók eredményeképpen a munkatársak egységes stilusban dolgoznak és megismerik egymás programjait, igy a MOZ-ART team rugalmasabb a személyi változásokkal szemben /csoporttagok cseréje, feladatkiosztás változása/.

8. A feladatok ütemezésének aktualizálása az inspekciók eredménye alapján biztositja a tempós és egyenletes haladást.

Összefoglalás

Az elmúlt két év tapasztalatai azt mutatják, hogy a MOZ-ART technológia a kitüzött céloknak megfelel. A bevezetéséhez szükséges többletráforditás rövid időn belül megtérül olyan környezetben, ahol jó minőségü programokra tényleges igény van.

Ugyanakkor a technológia továbbfejlesztése is indokolt, két alapvető irányban is. Egyrészt a rendszertervezés módszertani támogatását erősiteni kell, másrészt a munka hatékonyságát további szoftver-eszközök alkalmazásba vételével kell javitani.

Abstract: the lecture presents the history and some applications of MOZ-ART technology. The MOZ-ART /in Hungarian: MOZaik Alkalmazási Rendszer Technológia, Mosaic Application System Technology/ helps the programmers working in data processing.

The main components of the technology are the following:

- Jackson design method;
- design and coding inspections /Fagan/;
- recommendations for reasonable coding in PL/I and in COBOL.

After the history of the technology, the lecture gives an outline of the technological process. Some applications of the technology and the conditions for successful application are also shown. At the end, the advantages of the technology are listed.

Irodalomjegyzék

Bo78 B.W. Boehm: Characteristics of software quality. North-Holland, 1978

Es81 Esztergár Zs. és tsai: A MOZ-ART programozási technológia. SZÁMKI, 1981

Fa76 M.E. Fagan: Design and Code Inspections to Reduce Errors in Program Development. IBM Systems J. Vol. 15 N. 1o, 1976

Go81 G. Goos/editor/: Werkzeuge der Programmiertechnik, Proceedings, IFB 43. Springer, 1981.

Ha77 Havass M: A nagyüzemi software gyártás. SOFTTECH D 1, SZÁMKI, 1977

Hb81 Haba A: Az ellenőrző program futási idejének csökkentése. Megjelenik a SZÁMKI-Tanulmányok 8. számában.

Ja75 M. Jackson: Principles of Program Design. Academic Press, 1976

Ke79 Kertész J. és tsai: Számitógépes információs rendszerek tervezése és dokumentálása /ARDOSZ '79/. SKV 1979

KP74 B. Kernighan-P.Plauger: The Elements of Programming Style. McGraw-Hill, 1974

Mé79 Mérey A./szerk/: A MOZ-ART programozási technológia. SOFTTECH D 44, SZÁMKI, 1979

Su8o Sunavecz M. és tsai: A MOZ-ART technológia alkalmazási tapasztalatai. SOFTTECH D 45, SZÁMKI, 198o.

We7o Weinberg G: PL/I Programming: A Manual of Style. McGraw-Hill, 197o

We79 G. Weinberg: A számitógépprogramozás pszichológiája. KJK, 1979

Szerzők: Komor Tamás, Molnár Máté

Számitógépalkalmazási Kutató Intézet
1536 Budapest, Pf. 227.

Kovács György–Vadócz László–Márton Mátyás

AUTOMATIKUS JOBLÁNCKEZELŐ RENDSZER – JOLÁN

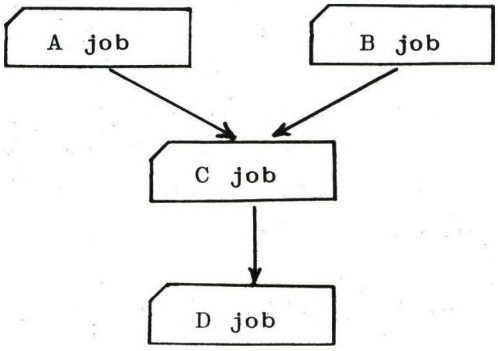
Az "Automatikus Joblánckezelő Rendszer" lehetővé teszi az egymással kapcsolatban álló jobok - jobláncok összefüggéseinek és indithatósági feltételeinek JCL szintü leirását. A jobok beolvasásakor dinamikusan fölépiti és adminisztrálja a lánchierarchiákat. Minden láncjob futása után kiértékeli a rá várakozó jobok indithatósági feltételeit. Adminisztrálja a feltételek teljesülését, ill. az összes feltétel teljesülése esetén a jobot előkészíti inditásra. A job kiválasztását a rendszer várakozási sorból és inditását a megfelelő initiátorok végzik. Minden lefutott láncjobot a rá várakozó jobok indulása követheti. Ez az eljárás - kedvező esetben - a teljes lánc feldolgozásához vezet. Az automatizmust olyan programtermékek támogatják, amelyek külső beavatkozást és információszerzést biztositanak a jobláncokkal kapcsolatban.

Kulcsszavak: jobkezelés, jobláncok, ütemezés

A hazai számitógép fejlesztés /ESZR/ és a jelenlegi gépi adottságok miatt a közeljövőben a leggyakrabban alkalmazott operációs rendszerek - OS/VS rendszerek lesznek, amelyek batch operációs rendszerek.

A batch munkák /jobok/ sok esetben nem függetlenek egymástól, hanem valamilyen feladatkör részfeladatait leiró un. jobláncok elemei.

pl. joblánc



Az IBM és ESZR OS rendszerek azonban nem támogatják az összefüggő jobok - jobláncok kezelését.

Jelenleg az ilyen jobláncok futtatása kisérőlap segitségével, operátori döntéssel és beavatkozással történik, amely körülményes, nagy hibalehetőséget rejt magában, igy a tevékenység nincs összhangban az egész operációs rendszer szinvonalával.

A számitástechnikai fejlesztésekkel kapcsolatban egyre több szó hangzik el a hatékonyságról. Ennek egyik komponense a jobkezelés hatékonyságának növelése és komfortosabbá tétele.

Ennek szellemében készült el a JOLÁN automatikus joblánc-kezelő rendszer.

A JOLÁN feladata:

- a lánchierarchiák kezelése
- a láncjobok indithatóságának figyelése
- kellő időben a láncjobok előkészitése inditásra

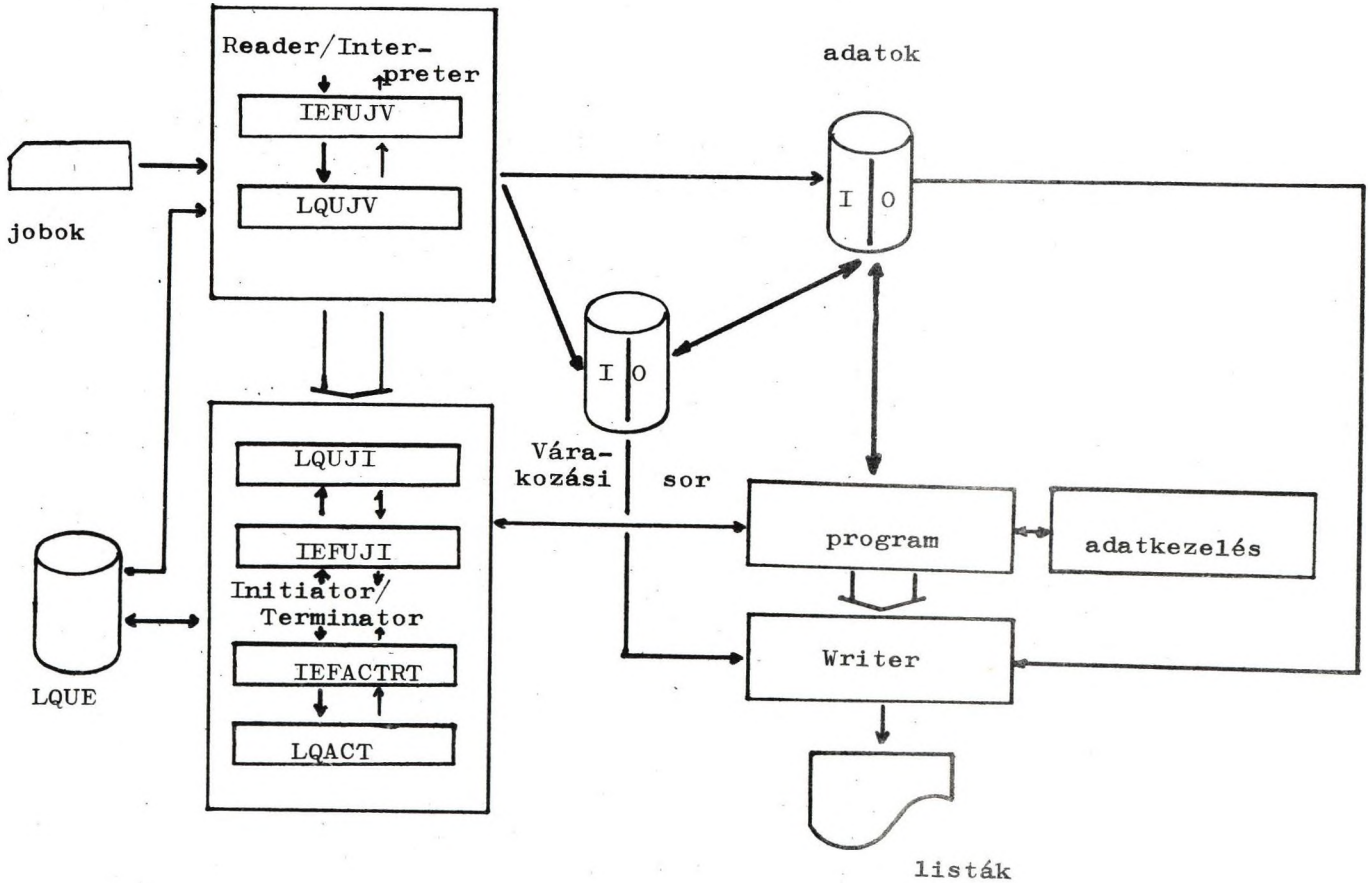
A JOLÁN által támogatott joblánc definiciója:

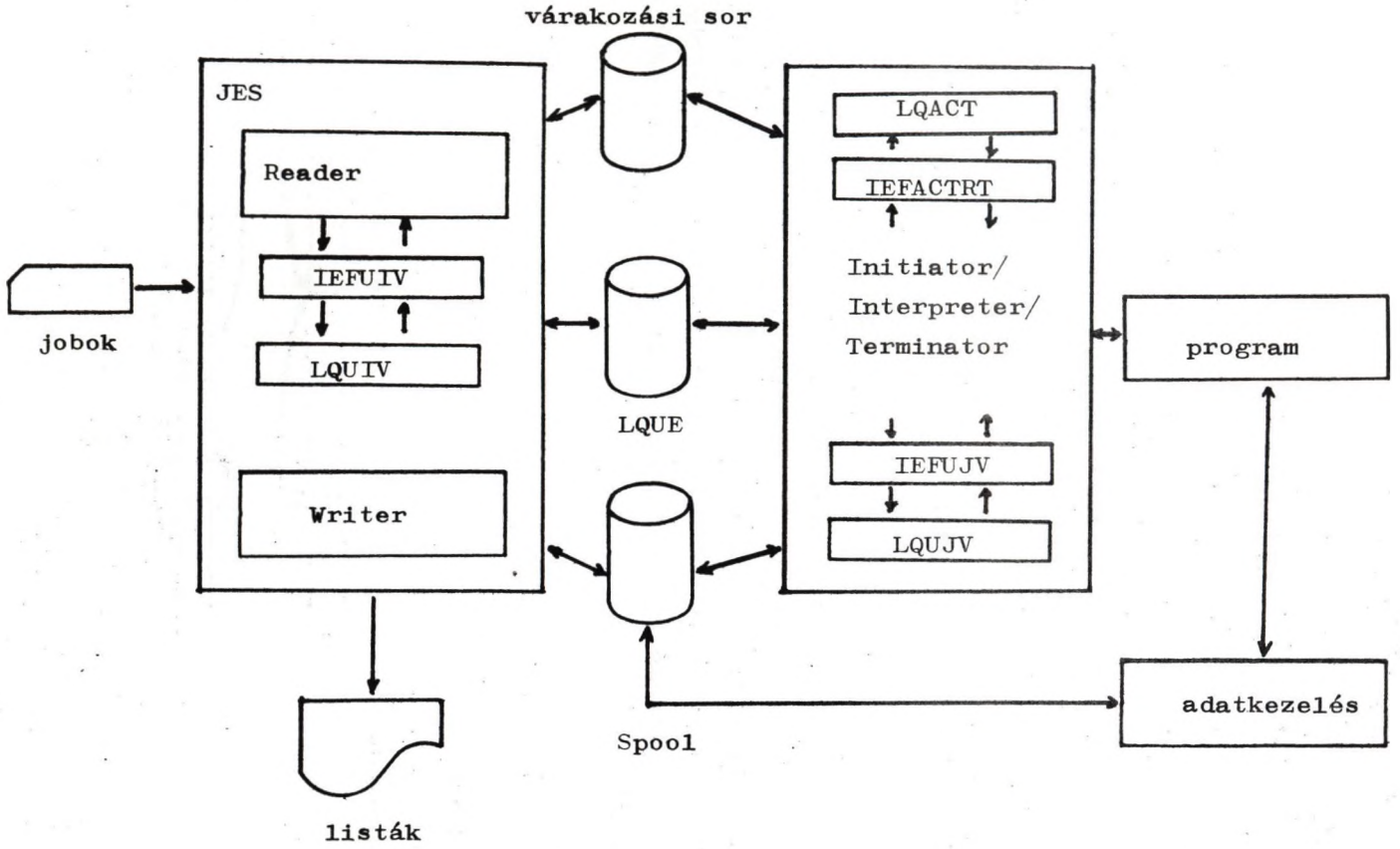
1. Az egymással feladatkapcsolatban álló jobok halmaza, amelyek indithatósága függhet más jobok /jobonként 2-2/ futásától
2. /más megfogalmazásban/ olyan irányitott gráf, melynek minden elemébe /job/ tetszőleges számu él futhat be, és minden eleméből max. két él indulhat ki.

A JOLÁN tulajdonságai:

- A láncdefiniciók /indithatósági feltételek/ a JOB-kártyán, annak komment mezőjében irhatók le;
- Mivel a láncparaméterek kommentként jelennek meg, a láncjobok JCL módositás nélkül futtathatók JOLÁN-t nem tartalmazó operációs rendszerekben;
- A láncjobok és nem láncjobok egyidejü kezelése zavartalan /jobnév konvenció/;
- A JOLÁN az SMF-en keresztül kapcsolódik a job management-hez;
- A JOLÁN nem szükiti az OS lehetőségeit /SMF rutinok használata/;

JOLÁN - OS/MVT


Reader/Inter-
preter
IEFUJV
LQUJV
jobok
adatok
I O
I O
Vára-
kozási
sor
LQUJI
IEFUJI
Initiator/
Terminator
IEFACTRT
LQACT
LQUE
program
adatkezelés
Writer
listák

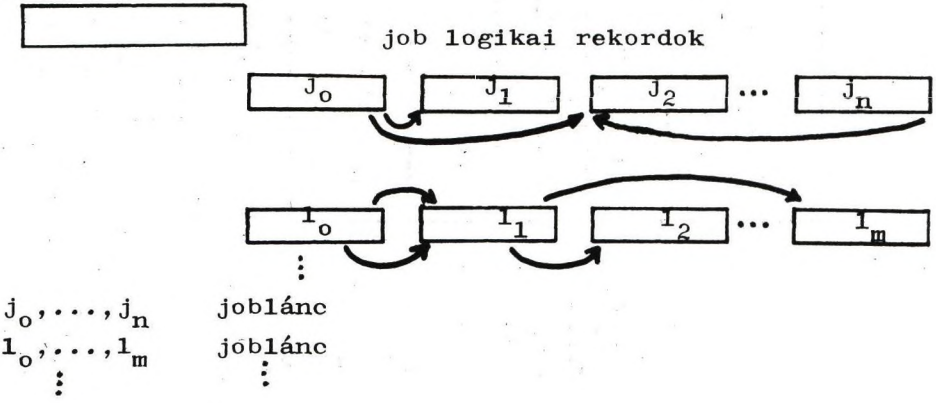

várakozási sor
JES
Reader
IEFUIV
LQUIV
Writer
jobok
LQUE
LQACT
IEFACTRT
Initiator/
Interpreter/
Terminator
IEFUJV
LQUJV
program
adatkezelés
Spool
listák

A JOLÁN és az OS kapcsolata

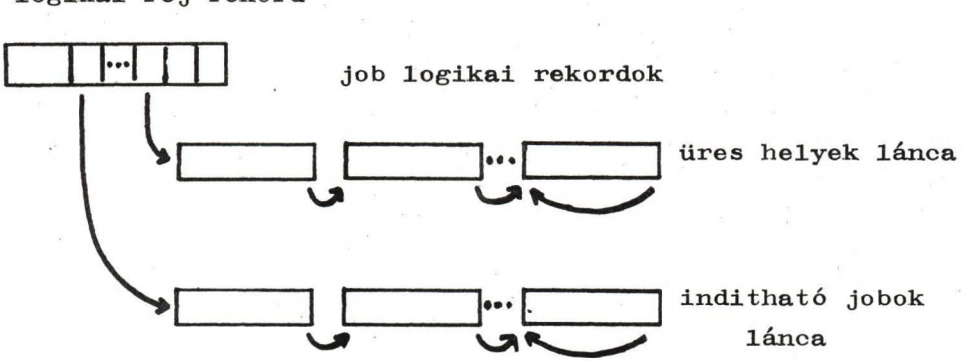
A JOLÁN elsődleges programkomponensei SMF exit rutinok. SMF exit rutinok irására általában elszámolásnál, felhasználó jogosultságának ellenőrzésénél, időkorlát figyelésnél, stb. van szükség. Azért, hogy ne foglaljuk le ezen rutinokat a joblánckezelő rendszer részére, bevezettük az LQUJI, LQUIV, LQUJV, LQACT modulokat, melyekre a megfelelő SMF exit rutinokból láncolunk. Igy teljes egészében megtartottuk az SMF exit rutinok felhasználhatóságát.

LQUE állomány tartalmazza a lánchierarchiákat. Az állományban minden láncjobot egy logikai rekord ir le. Ezen rekordok összeláncoltak egy-egy adott jobláncon belül, másrészt láncoltak a láncfeldolgozásban lévő állapotuk szerint. Ezek a láncolások a láncfeldolgozásban lévő egymásrahivatkozások kezelését, másrészt a feldolgozás gyorsitását szolgálják.

a./ LQUE állomány jobláncai



b./ LQUE állomány állapotláncai



A JOLÁN működése

Ha láncjob kerül beolvasásra /job név konvenció/ a joblánckezelő rendszer megvizsgálja, hogy az adott job indithatósága függ-e más joboktól /láncparaméterek/. Ha függ, a jobkártyán meghatározott jobosztály /PROFILE/ paramétertől függetlenül a job a rendszer várakozási sor egy kitüntetett /"tároló"/ jobosztályába kerül, melyre initiátor nem indítható, mig a más joboktól függetlenül indítható jobok a job kártyán meghatározott jobosztályba kerülnek. Ezzel egyidőben a beolvasott job lényeges adatai egy LQUE nevü adatállományba iródnak.

Az initiatorok elinditják az eredeti jobosztályukban lévő láncjobokat, melyeknek lefutása után a JOLÁN bejegyzi a jobok befejezési kódját LQUE-ba, és megvizsgálja azt, hogy a joblánchan az éppen lefutottra hivatkozó /várakozó/ jobok indithatósági feltétele teljesül-e. Ha teljesül, a várakozó jobnál az bejegyzésre kerül. A várakozó job minden feltételének teljesülésekor a JOLÁN átteszi a jobot a tá-

roló jobosztályból az eredeti osztályába, amiből a megfelelő initiator kiválaszthatja inditásra. Igy, ha a várakozó jobok feltételei teljesülnek, a teljes lánc lefut. Ha valamely feltétel nem teljesül, a lánc megakad, melynek feloldására különféle egyszerü módszerek állnak rendelkezésünkre.

Konvenciók:

Job befejezési kód

A láncjob befejezési kódja a legutoljára végrehajtott job lépés befejezési kódja, vagy - ha valamelyik job lépés abnormálisan fejeződött be - egy un. ABEND kód.

Jobnév

Az OS szintaktikának megfelelő max. 8 karakter, melynek utolsó helyén a JOLÁN installáláskor meghatározott jel áll.
Alapértelmezés: #
Ez jelzi azt, hogy az adott job láncjob.

Jobkártya

A jobkártya tartalmazza a láncjob indithatósági feltételeit

// jobnév JOB paraméterek [láncparaméterek]

láncparaméterek: láncfeltétel1 [,láncfeltétel2]
F1 F2

F1=(feltétel job1,reláció11,felt.kód11[,rel.12,felt.kód12])
F2=(feltétel job2,reláció21,felt.kód21[,rel.22,felt.kód22])

Az aktuális job akkor indulhat el, ha F1 és F2 által meghatározott feltétel jobok futása után a feltétel job_i befejezési kódja és a feltétel $kód_{i1}$ közötti reláció $_{i1}$ /vagy a feltétel job befejezési kódja és a feltétel $kód_{i2}$ közötti $reláció_{i2}$/ teljesül.

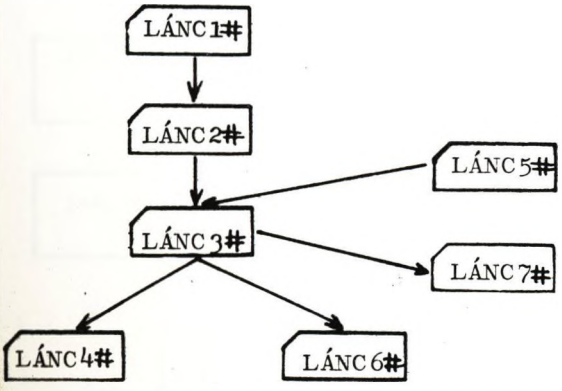
ahol i = 1,2

a relációk: EQ egyenló
LT kisebb
GT nagyobb
LE kisebb egyenlő
GE nagyobb egyenlő
NE nem egyenló

feltétel kódok:
- normál max, 4 jegyü dec. szám
- ABEND /ABEND befejezési kódtól függő indulási feltétel/
- EVEN /befejezési kódtól független indulási feltétel/

Minta joblánc

gráf



A jobkártyákkal reprezentált joblánc:

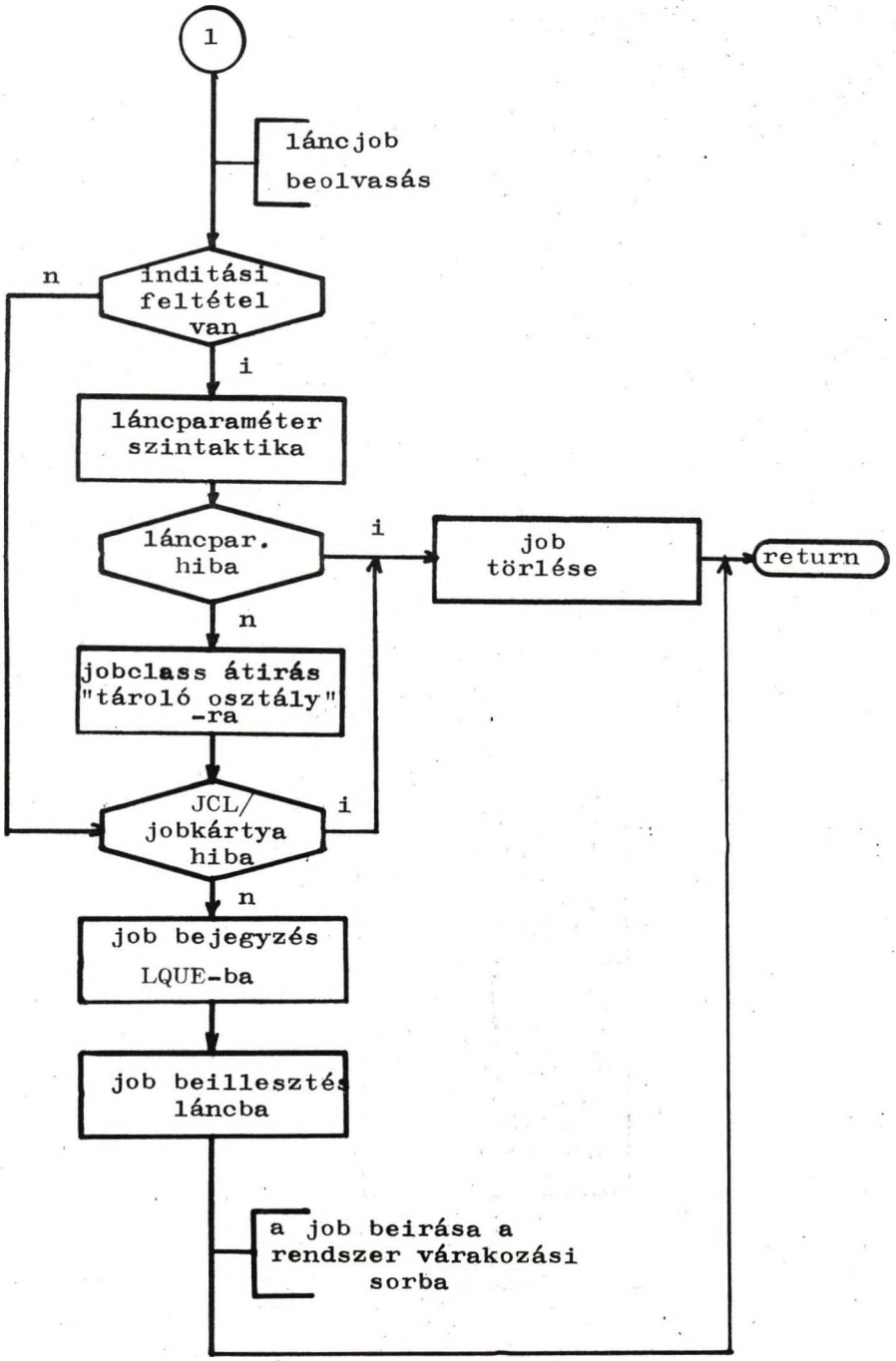
```
//LANC1#  Job param
//LANC2#  Job param   F1 = ( LANC1#,EQ,Ø)
//LANC3#  Job param   F1 = ( LANC2#,EQ,Ø), F2 = ( LANC3#,EQ,Ø)
//LANC4#  Job param   F1 = ( LANC3#,NE,ABEND)
//LANC5#  Job param
//LANC6#  Job param   F1 = ( LANC5#,EQ,ABEND)
//LANC7#  Job param   F1 = ( LANC5#,EQ,EVEN)
```

Magyarázat:

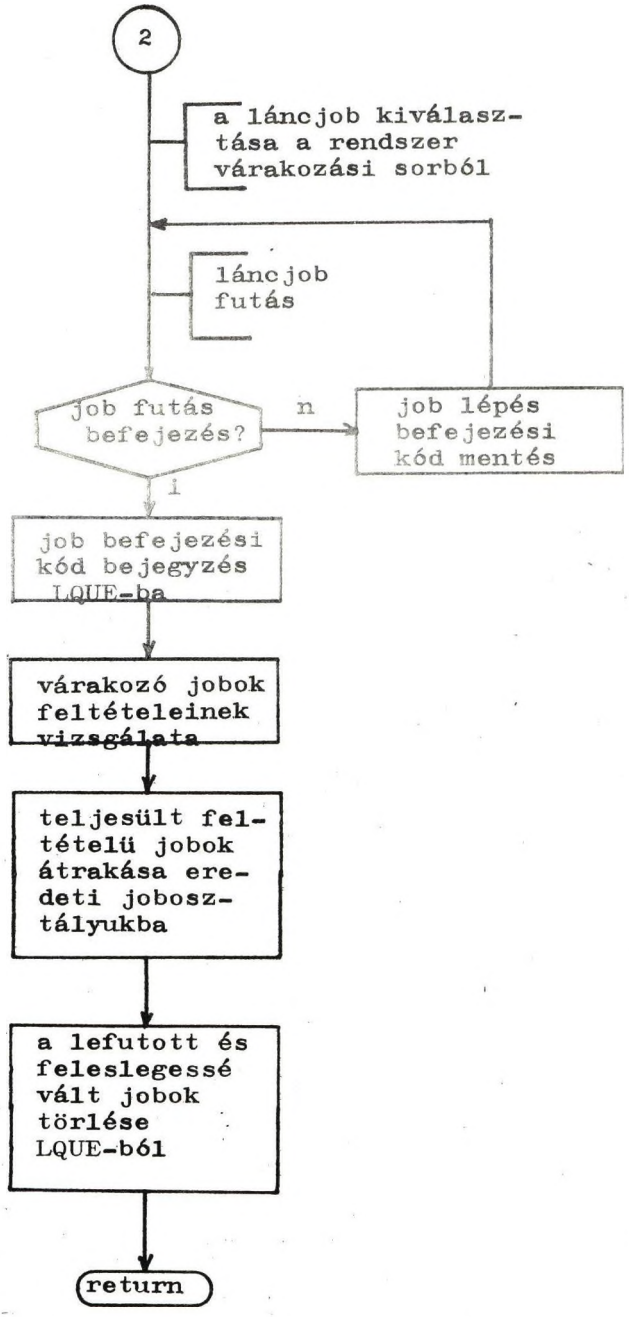
- a LANC3# job akkor indulhat el, ha LANC2# és LANC5# jobok lefutottak Ø befejezési kóddal.
- LANC7# job a LANC5# job tetszőleges eredményü futása után induljon el.

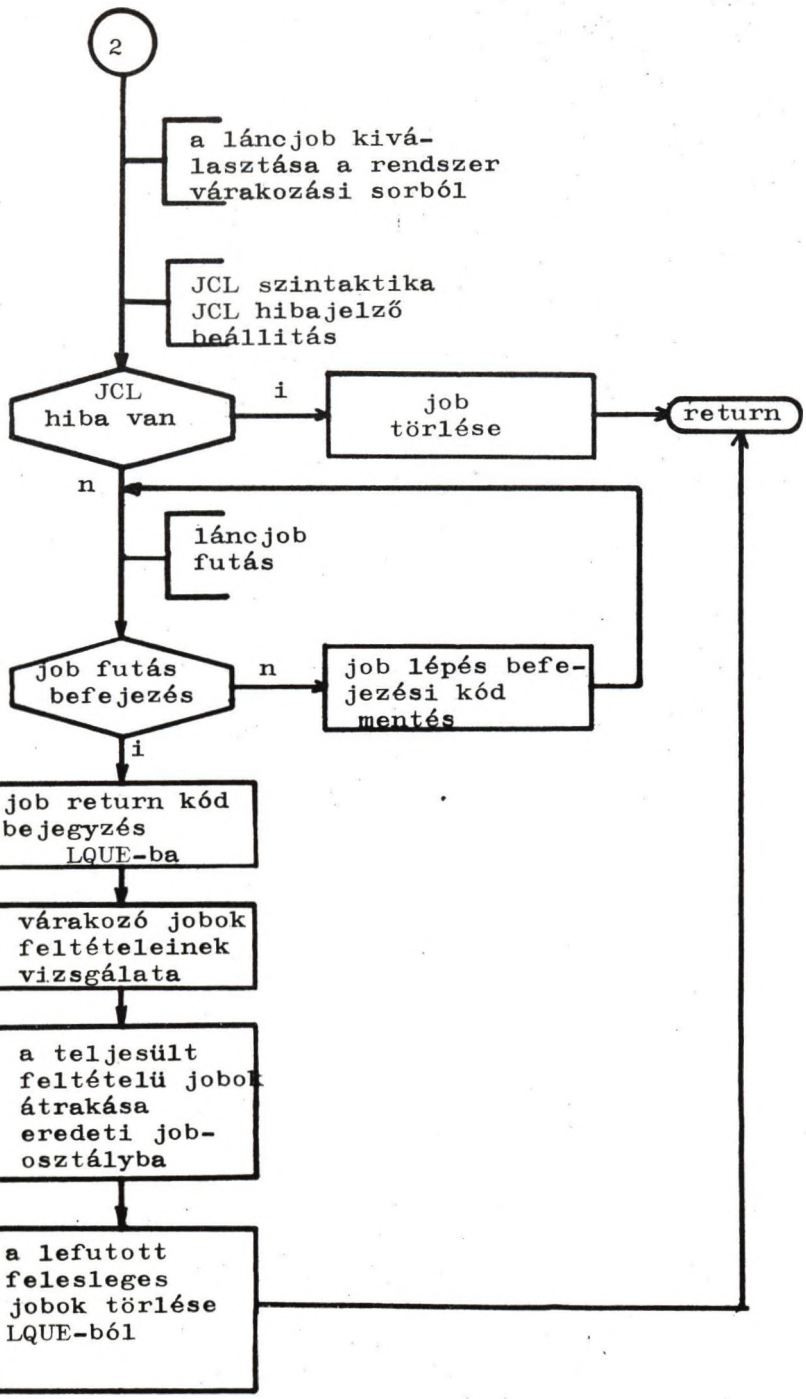
.
.
.

Joblánckezelés /OS MVT és VS1/



Jobláncкezelés OS/MVT




2
a láncjob kivá-
lasztása a rendszer
várakozási sorból
JCL szintaktika
JCL hibajelző
beállitás
JCL
hiba van
i
job
törlése
return
n
láncjob
futás
job futás
befejezés
n
job lépés befe-
jezési kód
mentés
i
job return kód
bejegyzés
LQUE-ba
várakozó jobok
feltételeinek
vizsgálata
a teljesült
feltételü jobok
átrakása
eredeti job-
osztályba
a lefutott
felesleges
jobok törlése
LQUE-ból

14

A lánckezelés információs és namuális beavatkozásra szolgáló eszközei

- A jobláncкezelés menetéről a JOLÁN folyamatosan ad közléseket a konzolon és a felhasználói programok sysout listáin
- lehetőség van konzolon és printeren történő nyomtatással bizonyos láncjobok és az összes lánc pillanatnyi állapotáról információt kérni
- láncakadásnál, amely ismételt és eredményes futással feloldható, lehetőség van a lánc többféle módon való továbbinditására, pl. tetszőleges befejezési kód beállitására
- külső beavatkozásra adnak lehetőséget - konzolról és jobból - a láncjobok és részláncok törlését biztositó eljárások.

Abstract:

The automatic chained job scheduler is a software facility, which enables the user to describe the logical interdependence of related jobs or job chains on a job control language level.
Whenever a cheined job is read in, the necessary references are dynamically created and evaluated. Similarly, at the end of execution all the references are updated and the jobs in the queue of chained jobs are tested if ready for running. The program examines all the conditions that have been set and if they are met then the appropriate job is scheduled for running.
The jobs are selected from the system queue and started by the appropriate initiator. After the execution of each chained job that has been waited, the jobs dependent on it can now be run automatically, which, in an optimum case, means that the whole chain of jobs will be processed.
This automatic facility is supported by functions that make operator intervention and information retrieval possible.

Irodalomjegyzék: OS/VS1 Job control language
OS/VS1 planning anduse guide
OS/VS1 job management logic
OS/MVT job control language
OS/MVT job management logic
OS/VS1 System management facilities /SMF/

Szerzők: Kovács György SZÁMKI
Vadócz László SZÁMKI
Márton Mátyás SZÁMKI

Kozma László

ABSZTRAKT ADATTÍPUSOK SPECIFIKÁCIÓJA PÁRHUZAMOS PROGRAMOZÁSI KÖRNYEZETBEN

A cikkben módszert közlünk absztrakt adattipusok egy lehetséges specifikálására párhuzamos programozási környezetben.
Bevezetjük a konzisztens specifikáció fogalmát, majd egy példán bemutatjuk a módszer alkalmazását.

Kulcsszavak: absztrakció, konzisztens specifikáció, osztott adattipusok, szinkronizáció

Bevezetés

Az adatabsztrakció jelentőségét soros programokra már régen felismerték a szakemberek. Az absztrakt adattipusok hasznos eszközei a megbizható szoftvert előállitó módszereknek. Napjainkban már szinte elképzelhetetlen, hogy az absztrakció eszközei nélkül sikeresen létrehozhatók legyenek nagy összetett programrendszerek.

Az ilyen programrendszerek ugyanis korunkban annyira bonyolultabbá váltak, - nem utolsó sorban a megoldandó feladat összetettsége, nagysága miatt - hogy áttekintésük, megértésük is igen komoly feladatot jelent. Az absztrakció lényege éppen abban áll, hogy eszközt ad a programok bonyolultságának csökkentésére, a programok különálló részekre való bontásával. És emellett lehetőséget ad egy adott objektum lényeges jellemzőinek szétválasztására azoktól, amelyek az adott környezetben lényegtelenek.

Az absztrakció még inkább hasznos eszköze lehet párhuzamos programok specifikálásának. Az adatok megosztása az egymással konkuráló folyamatok között ugyanis mind a programok áttekinthetőségét tovább rontja, mind a programok helyességének bizonyitását nagyon komplikálttá teszi.

Egy absztrakt adattipus absztrakt objektumok egy osztályát specifikálja az objektumokon végrehajtható müveletek halmazával együtt.

Egy absztrakt osztott adattipus absztrakt objektumok olyan osztályát specifikálja, amelyet az egymással konkurráló folyamatok közösen használhatnak az objektumok müveleteinek párhuzamos végrehajtása utján.

Az absztrakt osztott adattipus specifikációja igy az objektumok és a rajtuk értelmezett müveletek specifikációja mellett a szinkronizáció specifikációját is kell, hogy tartalmazza. A szinkronizációra mind az objektumok belső konzisztenciájának megőrzése, mind a magasabb szintü szinkronizációs döntések miatt szükség van.

Az eddigiek során jelentős erőfeszitések történtek az absztrakt adattipusok specifikálása terén /Guttag 75/, /Liskov és Zilles 75/, /Liskov és Berzsin 76/, /Parnas 72/, /Wulf és társai 76/. Ugyanakkor számos nyelvet fejlesztettek ki adatabsztrakciók kifejezésének támogatására, pl. CLU /Liskov és társai 77/, ALPHARD /Shaw és társai 77/. Dolgoztak ki az adatabsztrakcióra vonatkozó bizonyitási módszereket is /Hoare 72/, /Spitzen és Wegbreit 75/, /Varga 80/ és /Wulf és társai 77/.

Az absztrakt osztott adattipusok hasznos eszközei a párhuzamos programok jól-strukturált létrehozásának is /Owicki 76/. Laventhal kidolgozott egy módszert absztrakt adattipusok szinkronizációs tulajdonságainak specifikálására, amely lehetőséget ad arra, hogy a szinkronizációt függetlenül specifikáljuk egy absztrakt osztott adattipus esetén, a többi un. "adatkapcsolatu" tulajdonságtól /Laventhal 78/. A szinkronizáció-specifikáció ilyen megközelitési módja számos előnnyel jár, melyek közül talán a legfontosabb a modularitás biztositása, ami nemcsak a specifikáció, hanem a majdani implementálás során is megtartható. Ilyen megközelités mellett, ha egy soros /nem konkurrens/ specifikáció létezik egy absztrakt adattipusra vonatkozóan, akkor a specifikációhoz egyszerüen hozzávéve a szinkronizáció specifikációját nyerhetjük az osztott adattipus teljes specifikációját.

Tanulmányunkban az absztrakt osztott adattipus specifikációjában a Laventhal által kidolgozott módszert használjuk fel a szinkronizáció specifikálására.

A cikk szerkezete a következő. A második fejezet tartalmazza az absztrakt osztott adattipus egy specifikációs módszerét, s röviden ismerteti Laventhal szinkronizáció specidikációs eljárását. A harmadik fejezetben elemezzük az absztrakt osztott adattipus specifikációjának tulajdonságait, majd a negyedik fejezetben egy példán mutatjuk be a módszert. Az ötödik fejezet további problémák felvetését tartalazza.

2. Absztrakt osztott adattipus egy specifikációja

Egy osztott absztrakt adattipus specifikációja két jól elkülönithető részből állhat: az adatkapcsolatu részek ill. a szinkronizáció specifikációjából. A továbbiakban egy olyan specifikációs módszert adunk meg, amelyben az adatkapcsolatu részeket a soros programokra kidolgozott Hoare-féle axiomatikus módszer segitségével specifikáljuk /Hoare 72/, a szinkronizációt pedig a Laventhal által definiált nyelv jólformált formuláival adjuk meg.

2.1 A szinkronizáció -specifikáció eszköze

A szinkronizáció-specifikáció nyelve az elsőrendü egyenlőséges predikátumkalkuluson alapul. Ezt a kalkulust egészitjük ki azzal a lehetőséggel, hogy egyrészt hivatkozhassunk az absztrakt objektumok egyes müveleteinek aktivizálásaihoz rendelt eseményosztályokra, másrészt hogy definiálhassuk az idő szerinti rendezést az egyes eseményosztályok között.

Jelölje p/j/ egy objektum p müveleteinek j-dik aktiválását, ahol j az un. aktiválási sorszám az objektum teljes történetére vonatkozóan. Egy objektum minden p müveletének aktivizálásához három esemény tartoik egy request, egy enter és egy exit esemény. A p/j/ aktivizálásához tartozó request, enter és exit eseményeket jelölje rendre p/j/.request p/j/.enter ill. p/j/.exit. Az egy adott objektumosztályhoz tartozó összes ilyen esemény időben teljesen rendezett.

Jelölje ezt a rendezési relációt " $\longrightarrow$ ", melynek jelentése időben megelőzi. A szinkronizációs megszoritásokat a fenti eseményekre vonatkozóan tehát a $\longrightarrow$ reláció segitségével specifikálhatjuk. Például a p/j/.enter $\longrightarrow$ q/i/.exit reláció, rendezési klóz, azt a tényt fejezi ki, hogy a p művelet j-edik aktivizálásának enter eseménye időben megelőzi a q művelet i-dik aktivizálásának exit eseményét.

Egy absztrakt adattipusra vonatkozó szinkronizáció specifikáció a fenti alaku rendezési klózokból a normál predikátum-kalkulus operátoraival előállitott formula.

Feltesszük a továbbiakban, hogy az aktivizálási számok jelölésére használt i, j, .. változók a fenti klózokban minden lehetséges értékre vonatkozóan univerzálisan kvantáltak.

2.2. Egy specifikációs módszer

Az absztrakt osztott adattipusok egy lehetséges specifikációja a következő lehet:

type tipusnév/p/: a

elvárás: Elvárás /$\bar{p}$/

kezdeti feltétel: Kezd /a/

invariáns: I_a /a/

müveletek:

müveletnév /result $\bar{x}$; $\bar{y}$/

{pre_a/ a',$\bar{x}'\bar{y}'$/ }

/a,$\bar{x}$/ :=f_a/a',$\bar{x}'\bar{y}'$/

{$post_a$ /a,$\bar{x}$,$\bar{y}$/ }

szinkronizáció: S/a/

ahol

$\bar{p}$: az objektum paramétereinek listája

a: az absztrakt objektum

$\bar{x}$: a müvelet kimenő paraméterei

$\bar{y}$: a müvelet bemenő paraméterei.

a',$\bar{x}'$,$\bar{y}'$ jelöli rendre az absztrakt objektumot és a paraméterek értékeit a müvelet megkezdése előtt.

A specifikáció során első lépésként megadjuk az uj absztrakt tipus nevét és a tipusban specifikált absztrakt objektumot.

Az elvárás klóz - Elvárás - egy állitás, amely az absztrakt adattipus paramétereire ir elő korlátozásokat. A tipus egy ujonnan létrehozott példánya csak akkor használható korrektül, ha az uj példány paraméterei kielégitik az Elvárás állitást.

A kezdeti feltétel - Kezd - egy állitás, amely a tipus egy uj példányára megadja a kezdeti objektumot, amelyből az absztrakt müveletek alkalmazásával az absztrakt objektumok minden további példánya generálható.

Az invariáns klóz - I_a/a/ - egy konzisztens állitás, amely az absztrakt objektumok adott osztályát definiálja.

Egy müvelet specifikációja megadja a müvelet nevét formális paramétereit és azok tipusát.

Az előfeltétel - pre_a - egy állitás, amely megadja azt a feltételt, amelynek teljesülése esetén a müvelet végrehajtása korrekt.

Az utófeltétel - $post_a$ - egy állitás, amely leirja a müvelet hatását feltéve, hogy a müvelet megkezdése előtt a pre_a előfeltétel teljesült. A müvelet az absztrakt objektum egy uj példányát hozza létre és értéket ad az eredmény paraméternek.

A szinkronizáció - $S/a/$ - egy jól-formált formula a szinkronizáció-specifikáció nyelvén, azaz az $S/a/$ formula az egyes absztrakt müveletekhez tartozó eseményosztályokra vonatkozó idő szerinti rendezést specifikálja a teljes absztrakt adattipusra vonatkozóan.

3. Az absztrakt osztott adattipus specifikációjának elemzése

Az absztrakt osztott adattipus előző fejezetben megadott specifikációs módja lehetővé teszi, hogy szétválasszuk az adatokkal kapcsolatos részek specifikálását a szinkronizációs tulajdonságok specifikálásától. Ez azonban nem jelenti azt, hogy a két specifikációs rész teljesen független egymástól. Ellenkezőleg, az absztrakt objektumok belső konzisztenciájának megőrzése miatt a szinkronizáció-specifikáció eszközének olyan erősnek kell lennie, hogy valahányszor egy f absztrakt müvelet végrehajtása előtt teljesül az $S/a/$ szinkronizációs előirás, mindannyiszor teljesüljön az f müvelet pre_a előfeltétele is.

Definició: Azt mondjuk, hogy egy absztrakt osztott adattipus specifikációja konzisztens, ha minden f absztrakt müveletére teljesül a következő: valahányszor az f absztrakt müvelet végrehajtása előtt teljesül az $S/a/$ szinkronizációs előirás mindannyiszor teljesül az f müvelet pre_a előfeltétele.

A konzisztencia a specifikáció igen fontos tulajdonsága. Ha egy absztrakt osztott adattipus specifikációja nem konzisztens, akkor vagy a szinkronizációs előirásokat kell erősiteni - további rendezési klózokat kell megadni - vagy a müveletek specifikációját kell megváltoztatni ugy, hogy a

konzisztenciát biztositó gyengébb előfeltételek is elegendőek legyenek az utófeltételek teljesüléséhez.

Tekintsünk egy egyszerü példát, amely bemutatja konkurrens környezetben a konzisztencia fontosságát. Tegyük fel, hogy adott egy korlátos absztrakt puffer osztott specifikációja egy append, egy remove és egy empty müvelettel.

Az append müvelettel egy ujabb elemet tehetünk a pufferbe, ha nincs tele. A remove müvelettel egy elemet vehetünk ki a pufferből, ha nem üres. Az empty pedig egy olyan logikai függvény /müvelet/, amely megadja, hogy a puffer üres-e vagy sem.

Tegyük fel, hogy a specifikáció nem konzisztens, például a remove müvelet specifikációjára vonatkozóan; azaz valahányszor a remove végrehajtása előtt teljesül a szinkronizáció specifikáció ebből nem következik, hogy mindannyiszor a remove előfeltétele is teljesül.

Ilyen esetben egy folyamat az empty logikai függvénnyel lekérdezheti ugyan, hogy a puffer nem üres-e, de pozitiv válasz esetén sem biztos, hogy mire ugyanaz a folyamat végrehajtja a remove utasitást, a pufferben még mindig van elem; ugyanis közben más folyamatok kivehették az összes elemet a pufferből. Konzisztens specifikáció esetén ilyen nem fordulhat elő.

4. Korlátos absztrakt osztott stack specifikációja

Célunk egy olyan stack specifikálása, amelyen egyidőben több folyamat osztozhat. Mindegyik folyamat tetszőlegesen tehet be elemeket ill. vehet ki a stack-ből. A stack mérete véges.

```
type  stack /n: integer/:
      stack =sequence /a/
      elvárás  n >o
      kezdeti feltétel  nullsequence /a_o/
```

invariáns $0 \leq$ length /stack/ $\leq n$

műveletek

push /s: stack, y: elem/

pre $o \leq$ length /S/ $< n$ post $s = s' \sim y$

pop /s: stack, result x/

pre $o <$ length /S/ $\leq n$ post $y =$ last/s'/

$\wedge$ s=leader/s'/

Szinkronizáció:

/push /i/.exit $\rightarrow$ pop /i/.enter/ $\wedge$

/pop /i/.exit $\rightarrow$ push /i+n/.enter/ $\wedge$

/push /i/.exit $\rightarrow$ push /i+1/.enter/ $\wedge$

/pop /i/.exit $\rightarrow$ pop /i+1/.enter/ $\wedge$

//push /i/.exit $\rightarrow$ pop /k/.enter/ $\vee$

/pop /k/,exit $\rightarrow$ push /i/.enter//

Megjegyzések:

a./ sequence /a/ = $a_1, \ldots, a_k$ a specifikált elemek sorozata, speciálisan a $\langle\,\rangle$ jelöli az üres sorozatot, amelyet "nullsorozat"-nak nevezünk.

b./ s $\sim$ y jelöli azt a sorozatot, amelyet az s sorozat és az y elem konkatenációjával nyerünk.

c./ length /s/ jelöli az s sorozat hosszát.

d./ last /s/ jelöli az s sorozat utolsó, legjobboldalibb elemét.

e./ leader /s/ jelöli azt a sorozatot, amit ugy kapunk az s sorozatból, hogy töröljük az s utolsó elemét.

f./ A szinkronizáció-specifikáció első konjunkciós tagja biztositja, hogy a pop müvelet üres stacken nem dolgozhat. A második tag előirja, hogy a push művelet tele

stackbe nem tehet sohasem ujabb elemet. A következő két tag azt fejezi ki, hogy két push ill. két pop művelet egyszerre nem kerülhet végrehajtásra. Az utolsó tag biztositja, hogy egy push és egy pop művelet is kölcsönösen kizárják egymást.

Állitás

A korlátos absztrakt osztott stack specifikációja konzisztens.

Bizonyitás

Azt kell bizonyitanunk, hogy valahányszor egy push ill. pop művelet végrehajtása előtt teljesül a szinkronizáció előirása, mindannyiszor teljesül a push ill. pop művelet előfeltétele is.

A bizonyitást a push ill. pop műveletek aktivizálási számára vonatkozó un. párhuzamos indukcióval végezhetük el.

Először belátjuk, hogy i=0-ra teljesül állitásunk, majd feltesszük, hogy minden $0 \leq k \leq j < i$-re teljesül állitásunk és belátjuk, hogy akkor i-re is teljesül, ahol j a push müveletek aktiválási száma, k pedig a pop müveleteké.

A./ Tegyük fel, hogy i=0, azaz még nem történt meg egyetlen push ill. egyetlen pop művelet aktiválása sem.
Ekkor teljesül a nullarequence $/s_o/$ kezdeti feltétel, ebből és a length fgv definiciójából következik, hogy length $/s_o/=0$. Mivel az elvárás klóz alapján $n > 0$, igy $0 \leq$ length $/s_o/ < n$ teljesül az első push művelet aktivizálása előtt.

A pop művelet $o <$ length $/s_o/ \leq n$ előfeltétele nem teljesül, de ekkor a szinkronizáció specifikáció első tagja sem teljesül, mivel push /1/.exit →pop /1/.enter rendezési klóz csak akkor teljesülhet, ha az első push művelet végrehajtása már befejeződött. A push művelet

végrehajtása után viszont az utófeltételéből és a length fgv definíciójából következik, hogy
$n \geq$ length /s/ = length /$s_o \sim x$/=length /s_o/+1=0+1 $>$ 0
vagyis teljesül a pop müvelet előfeltétele is.

B./ Tegyük fel, hogy a tétel állitása minden olyan pop ill. push müveletre igaz, amelyeknek aktiválási száma k ill. j, ahol $o \leq k \leq j < i$. Belátjuk, hogy akkor az i-dik push ill. pop müveletre is igaz az állitás. Feltevésünk azt jelenti, hogy $\forall$j-re /$0 \leq j < i$/ a j-dik push müvelet végrehajtása után $0 \leq$ length /s/ = length /s'/+1$\leq n$ teljesül. Illetve k-ra /$0 < k \leq j < i$/ a k-dik pop müvelet végrehajtása után $0 \leq$ length /s/ = length /s'/-1$\leq n$ teljesül. Mivel a specifikáció szerint az egyes müveletek kölcsönösen kizárják egymást, igy $\forall$k-ra $\forall$j-re /$0 \leq k \leq j < i$/

length /s/ = j-k

A továbbiakban két esetet kell megkülönböztetnünk aszerint, hogy az i-dik push vagy az i-dik pop müveletet vizsgáljuk-e.

B.1. Tegyük fel, hogy j=i-1, azaz az i-1-dik push müvelet befejeződött. Az indukció feltevés szerint az i-1-dik push müvelet előfeltétele teljesült. Belátjuk, hogy valahányszor a szinkronizáció specifikáció teljesül az i-dik push müvelet előtt, mindannyiszor teljesül a push müvelet előfeltétele is.
Az i-1-dik push müvelet végrehajtása után
length /s/ = i-1-k, ahol
$i-1-n \leq k \leq i-1$, szükségképpen teljesül az indukciófeltevés alapján k-ra.
Tegyük fel, hogy k=i-1-n, azaz az i-1-n-dik pop müvelet fejeződött még csak be.
Ekkor length /s/=i-1-/i-1-n/=n vagyis a push müvelet e-

lőfeltétele nem teljesül, de ekkor nem teljesül a szinkronizáció specifikáció második tagja sem, amely előirja, hogy az i-dik push müvelet aktiválása csak akkor következhet be, ha az i-n-dik pop müvelet végrehajtása befejeződött, azaz

pop/i-n/.exit →push/i/.enter

Az i-n-dik pop müvelet a specifikáció szerint bekövetkezhet és rá érvényes az indukció feltevés.

Belátható továbbá, hogy $\forall$ k-ra i-n $\leq$ k $\leq$ i-1 esetén a push müvelet előfeltétele mindig teljesül, igy teljesül akkor is ha a szinkronizáció specifikáció teljesül. Ugyanis

0 $\leq$ length /s/=i-1-/i-n/=n-1 $<$ n, ha k=i-n ill.

0 $\leq$ length /s/=i-1-/i-1/=0 $<$ n, ha k=i-1.

B.2. Tegyük fel, hogy a k=i-1-dik pop müvelet befejeződött, akkor az eddigiekhez hasonlóan beláthatjuk, hogy ha az i-dik pop müvelet végrehajtása előtt teljesül a szinkronizáció specifikáció, akkor teljesül az i-dik pop müvelet előfeltétele is.

g.e.d.

5. Konkluzió

A tanulmányunkban közölt módszer alkalmas absztrakt adattipusok specifikálására párhuzamos programozási környezetben. Megvizsgáltuk a specifikáció legfontosabb tulajdonságát, de nem foglalkoztunk a specifikáció implementálásának kéréskörével, ami további kutatás tárgyát képezi.

Ugyancsak kutatás tárgyát képezheti az a probléma, hogy hogyan tudjuk egy absztrakt specifikáció egy megvalósitásának helyességét belátni.

Abstract

This paper presents a method for specifying shared abstract data types.
The concept of consistent specification is introduced and an example is given to show the proposed specification method.

Irodalomjegyzék

/Guttag 75/ J.V.Guttag
The specification and Application to Programming of Abstract Data Types.
Ph. D. Thesis, Computer Science, University of Toronto /1975/

/Hoare 72/ C.A.R.Hoare
Proof of Correctness of Data Representations
Acta Informatica I. pp. 271-281.
/1972/

/Laventhal 78/ M.S. Laventhal
Synthesis of synchronization code for data abstraktions
M.I.T.Laboratory for Computer Science
/1978/

/Liskov és Berzins 76/ B.H.Liskov and V.Berzins
An Appraisal of Program Specifications
/Computation Structures Group Memo 141 M.I.T.
/July 1976/

/Liskov és Zilles 75/ B.H.Liskov and S. Zilles
Specification Techniques for Data Abstractions
IEEE Trans. on Software Eng. SE-1, 1,pp 7-19.
/March 1975/

/Liskov és társai 77/ B.H.Liskov, A.Snyder, R.Atkinson and C. Schaffert
"Abstraction Mechanism in CLU."
Comm. of the ACM 20. 8; pp. 564-576.

/Owicki 76/ S.Owicki
An axiomatic prof technique for parallel programs II. Shared data abstractions
Stanford University
/1976/

/Shaw és társai 77/ M.Shaw, W.A.Wulf and R.L.London
/Abstraction and Verification in Alphard"
Comm. of the ACM 20. 8. pp. 553-564
/August 1977/

/Spitzen és Wegbreit 75/ J.Spitzen and B. Wegbreit
The Verification and Synthesis of Data Structures
Acta Informatica 4. pp. 127-144.
/1975/

/Varga 80/ Varga László
Specification of reliable software
Operációs Rendszerek Elmélete VI. Visegrádi Téli Iskola. MTA Számitástechnikai és Automatizálási Kutató Intézet. Tanulmányok 113/1980
309-325

/Wulf és társai 76/ W. Wulf, R.L.London and M.Shaw
"An Introduction to the Construction and Verification of Alphard Programs"
IEEE Transactions on Software Eng.
SE-2 pp. 253-264.
/1976/

/Wulf és társai 77/ W.A.Wulf, R.L.London and M.Shaw
Abstraction and Verification in Alphard:
A Symbol Table Example
In: Procedings of IFIP TC2 Working
Conference Novoszibirszk
/1977/

Kozma László: Számitógépalkalmazási Kutató Intézet
Budapest, I.
Csalogány u. 30-32.
1015.

Kőfalusi Viktor–Halmayné Szentirmay Edit

ÁLLAPOTTÉR SZEMLÉLETŰ MATEMATIKAI STRUKTÚRÁK

Kivonat A cikk két matematikai objektumot mutat be: az állapottér-halmaz struktúrát /SPSS/ és az állapottér-gráfot /SSG/, illetve definiál egy kifejezéstipust az n-prefix tipust. Ezek alkalmazási területei a szimbólikus matematikai kiszámitás és a formulamanipuláció.

Tárgyszavak szimbólikus matematikai kiszámitás, formulamanipuláció, vegyes-halmaz, állapottér-halmaz, állapottér-gráf, n-prefix kifejezés, általánositott struktúraosztás, tömegreláció.

0. BEVEZETÉS

Dolgozatunkban a szimbólikus matematikai kiszámitás, illetve a formulamanipuláció területén elért eredményeink egy részét mutatjuk be.

Mint az ismeretes, a számitógéptudomány alapnyelve a logika nyelve. Ennek megfelelően munkánk során nagyban támaszkodtunk Dávid Gábor logikai gép koncepciójára [Dá81], valamint R. Kowalski 'az elsőrendü predikátumkalkulus egy magaszintü programozási nyelv' tézisére [Ko79].

Háromféle megközelitést alkalmazunk: első szinten a szimbólikus matematikai kiszámitást és a formulamanipulációt, egy alsóbb szinten pedig a kifejezések tipizálását. A leirtaknak egységes, konstruktiv rendszerbe foglalását egy szimultán programozáselméleti és automataelméleti megközelités adja meg [Kő81/c, Ha81/d].

Az 1. rész - az itt bővebben ki nem fejtett általános állapotterek elméletének [Ha81/a] alapján bevezett - halmazelméleti megalapozástirja le [Kő80/a]. A 2. valamint a 3. részben leirtak ugyanannak az objektumnak az inkább szintaktikus, illetve inkább szemantikus aspektusu matematikai modelljei. A 4. rész ezen modellek implementálásakor alkalmazható segédeszközt, az n-prefix kifejezéseket mutatja be. Az 5. rész néhány eddig implementált program vázlatos ismertetésével illusztrálja az eddig leirtakat.

1. JELÖLÉSEK, DEFINICIÓK

Definicióinkban határozottan meg kívánjuk különböztetni a halmazelméleti "elem" és "tag" fogalmát, illetve a "halmaz eleme" és a "halmaz tagja" relációkat.

Definició 1.1. Ha S halmaz, akkor S-nek ezt a tulajdonságát s(S)-sel jelöljük, a tagadását pedig $\overline{s}(S)$-sal.

Definició 1.2. $(\forall B)(s(A) \wedge B \subset A \wedge s(B))$, akkor A-t halmazrendszernek nevezzük, és A-nak ezt a tulajdonságát ss(A)-val jelöljük. B-t a halmazrendszer tagjának nevezzük. Az A és B között fennálló relációt a "\" infix jellel jelöljük: $A \setminus B$. Ha U nem tagja S-nek, akkor ezt igy jelöljük: $S \not\ni U$.

Definició 1.3. Ha $(\exists x \exists B)(x \in A \wedge \overline{s}(x) \wedge A \setminus B)$, akkor A-t vegyeshalmaznak nevezzük, és A-nak ezt a tulajdonságát ms(A)-val jelöljük.

Definició 1.4. Legyen A egy halmazrendszer, X pedig egy tetszőleges halmaz. Ha létezik egy olyan $C_1, C_2, \ldots, C_n$ $(n \geq 0)$ halmazsorozat, hogy az $A \setminus C_1 \setminus C_2 \setminus \ldots \setminus C_n \setminus X$ kifejezés igaz, akkor a kifejezést X nyomának nevezzük A-ban és track(A,X)-szel jelöljük.

Definició 1.5. Legyen A egy halmazrendszer, B pedig egy tetszőleges halmaz. Ha track(A,B) létezik, akkor azt mondjuk, hogy B szerepel A definiciójában, és ezt a tényt in(A,B)-vel jelöljük.

2. ÁLLAPOTTÉR-HALMAZ STRUKTURÁK

Számos olyan gyakorlati probléma ismeretes, vagy kreálható, amely PROLOG-ban megoldható, ellentétben más programozási nyelvekkel [Sze77].

A PROLOG nyelv használata során azonban problémák merülnek fel a tárigényben és a futási időben való kombinatorikai robbanás örökös veszélyével kapcsolatban [Sá80]. Ezeket a nehézségeket a program és az adatok fastrukturáju ábrázolása okozza, limitálva a kiszámithatóság gyakorlati határát.

A PROLOG programok futási ideje jelentősen csökkenthető azáltal, ha a keresést az adatbázis megfelelő részére redukáljuk. A moduláris PROLOG (MPROLOG [Be78,Be79]) tartalmaz ilyen lehetőségeket, számottevően megnövelve ezzel a lehetséges PROLOG-alkalmazások körét.

Az adatok fastruktúráju ábrázolása különböző nehézségeket okoz. Tekintsük a legegyszerübbet! Tegyük fel, hogy egy kifejezés többször tartalmazza ugyanazt a részkifejezést. Ha egy ilyen kifejezést fával ábrázolunk, akkor a kifejezés feldolgozása több redundáns elemet tartalmaz. Eg észen egyszerüen szólva: meg szeretnénk teremteni a lehetőségét annak, hogy amit lehet, csak egyszer számitsunk ki, avagy megforditva: ne számitsuk ki kétszer ugyanazt a részeredményt.

A következőkben bemutatjuk az állapottér-halmaz struktúrát /SPSS/. Bizonyitható, hogy az SPSS a struktúraosztás [Bo72] általánositása.

Definiálunk egy olyan formulahalmazt, aminek semmiféle interpretációt sem tulajdonitunk, de tetszőlegesen interpretálható. A formulahalmazt a konstans, a változó, illetve a müveleti jel fogalmára épitjük.

Definició 2.1. Az F formulahalmazt a $\langle D,O\rangle$ halmazpárral irjuk le, ahol D az alaphalmaz, amelynek elemei tetszőleges tipusu konstansok és változók lehetnek. O a müveleti jelek halmaza, amelynek elemei tetszőleges szemantikáju szimbólumok. $D,O \sim N$, ahol N a természetes számok halmaza. Az üres formulát nil-nek nevezzük, $nil \in F$. Minden $h_1,\ldots,h_n$ n-esre, ha $h_i \in F$ $(i=1,\ldots,n)$, akkor $(\forall m \forall f^m)(f^m \in O \Rightarrow {}^1f^m(h_1,\ldots,h_m) \in F)$.

Definició 2.2. Legyen x és y F eleme. x az y részformulája, ha vagy x=y, azaz x és y leirható ugyanazzal a szimbólumfüzérrel, vagy x szimbólumfüzére valódi részfüzére y szimbólumfüzérének. Ebben az esetben x-et y valódi részformulájának nevezzük. Az x részformulája y-nak", illetve az "x valódi részformulája y-nak" relációkat az alábbi módon jelöljük: $x \sqsubseteq y$, illetve $x \sqsubset y$.

Definició 2.3. Definició 2.3. Állapottér-halmaz struktúra alatt az

$SPSS=\langle F',SPS,ID,\sqsubseteq,\setminus,<,\leftarrow,\nwarrow\rangle$ nyolcast értjük, ahol

- $F' \subseteq F$ (F egy formulahalmaz, $F' \sim N$, $nil \in F'$).
- ms(SPS), SPS-t állapottér-halmaznak nevezzük, SPS rögzitett.
- ID egy névhalmazt jelöl, $(ID \sim F')$.
- $\sqsubseteq$ a részformula reláció.
- $\setminus$ a tagja reláció.
- $<$ a rendszámokon értelmezett valódi rendezési reláció.
- $\leftarrow$ a felvétel müvelete (később definiáljuk).
- $\nwarrow$ az elémásolás müvelete (később definiáljuk). Az elémásolást balra kiemelésnek is nevezzük.

Az állapottér-halmaz felfogható egy olyan matematikai adatszerkezet gyanánt, amelyben ID' $(ID \sim ID')$ elemeivel felcimkézett H $(F' \subseteq H)$ a szögpontok halmaza; amelynek nevét SPS jelzi; ennek elemei között értelmezett a "részformulája", a "tagja" és a rendszámokon értelmezett valódi rendezési reláció; valamint kétfajta müvelet: a felvétel és az elémásolás müvelete végezhető el.

Az SPS által tartalmazott halmazokra a görög ABC kisbetüivel hivatkozunk. Ha $s(\alpha) \wedge in(SPS,\alpha)$, akkor α-t állapottérnek nevezzük, és ezt a tulajdonságát $sp(\alpha)$-val jelöljük. Ha $ss(\alpha) \wedge in(SPS,\alpha)$, akkor α-t állapot-hipertérnek nevezzük és ezt a tulajdonságát $shp(\alpha)$-val jelöljük. Ha $sp(\alpha) \wedge in(SPS,\alpha) \wedge SPS \setminus \alpha$, akkor α-t állapot-altérnek nevezzük és $ssp(\alpha)$-val jelöljük.

Ha $sp(\alpha) \wedge sp(\beta) \wedge in(SPS,\alpha) \wedge in(SPS,\beta) \wedge in(\alpha,\beta)$, akkor β-t α állapot-alterének nevezzük és $ssp(\alpha,\beta)$-val jelöljük.

Ha $sp(\alpha) \wedge sp(\beta) \wedge in(SPS,\alpha) \wedge in(SPS,\beta) \wedge \alpha \setminus \beta$, akkor β-t α direkt állapot alterének nevezzük és $dsp(\alpha,\beta)$-val jelöljük.

$\{\alpha\}$ az α állapottér azon elemeinek halmaza, amelyek nem halmazok. J_α illetve I_α jelzi α direkt altereinek, illetve $\{\alpha\}$ -nak az indexhalmazát. Minden állapottérnek neve van. Az α állapottér nevét $name(\alpha)$ -val jelöljük.

Hangsúlyozzuk, hogy $in(SPS,\alpha)$ azt jelenti, hogy a nyoma reláció érvényes SPS és α között (az 1.5. definició miatt).

Az állapottér-halmaz struktúra jellemzői 10 axiómával irhatók le.

A1. $(\forall\alpha\forall x)\,(in(SPS,\alpha) \wedge sp(\alpha) \wedge (x \in \{\alpha\} \Rightarrow x \in F'))$

A2. $(\forall\alpha\exists x)\,(in(SPS,\alpha) \wedge sp(\alpha) \Rightarrow (x \in \{\alpha\} \wedge x = nil))$

A3. $(\forall\alpha\forall j)\,(in(SPS,\alpha) \wedge i,j \in I_\alpha \wedge (x_i = nil \wedge x_j \neq x_i \Rightarrow i < j))$

A4. $(\forall\alpha\forall i\forall j)\,(in(SPS,\alpha) \wedge sp(\alpha) \wedge i,j \in I_\alpha \wedge (x_i = x_j \Leftrightarrow i = j))$

A5. $(\forall\alpha\forall i\forall j)\,(in(SPS,\alpha) \wedge sp(\alpha) \wedge i,j \in I_\alpha \wedge (x_i = x_j \Rightarrow i < j))$

A6. $(\forall\alpha\forall i\forall j\forall u\forall v\forall w\exists k)\,((in(SPS,\alpha) \wedge sp(\alpha) \wedge i,j \in I_\alpha \wedge x_i, x_j \in \{\alpha\} \wedge u,v,w \in F' \wedge v \neq x_i \wedge w \neq x_j \wedge u \neq v \wedge u \neq w \wedge (x_i = x_j \Rightarrow v \neq w \wedge w \neq v)) \Rightarrow (k \in I_\alpha \wedge x_k \in \{\alpha\} \wedge x_k = u))$

A7. $(\forall\alpha\forall i\forall j)\,(in(SPS,\alpha) \wedge i,j \in I_\alpha \wedge dsp(\alpha,\beta_i) \wedge dsp(\alpha,\beta_j) \wedge name(\beta_i) = name(\beta_j) \Leftrightarrow i = j))$

A8. $(\forall\alpha\forall i\forall j)\,(in(SPS,\alpha) \wedge i,j \in J_\alpha \wedge dsp(\alpha,\beta_i) \wedge dsp(\alpha,\beta_j) \wedge (ssp\,\beta_i,\beta_j \Rightarrow j < i))$

A9. $(\forall\alpha\forall i\forall j\forall\gamma\forall\vartheta\exists\varepsilon k)\,((in(SPS,\alpha) \wedge sp(\alpha) \wedge i,j \in J_\alpha \wedge dsp(\alpha,\beta_i) \wedge dsp(\alpha,\beta_j) \wedge ssp(\beta_i,\gamma) \wedge ssp(\beta_j,\vartheta) \wedge (\gamma \neq \vartheta \Rightarrow (ssp(\gamma,\varepsilon) \wedge ssp(\vartheta,\varepsilon))) \vee (\gamma = \vartheta \Rightarrow \varepsilon = \gamma)) \Rightarrow (k \in J_\alpha \wedge dsp(\alpha,\beta_k) \wedge \beta_k = \varepsilon))$

Az első axióma az F' formulahalmaz elemeit rögziti. A második axióma szerint az üres formulát (nil) az SPS definiciójában szereplő valamennyi állapottér elemként tartalmazza. A harmadik axióma alapján valamennyi SPS-beli állapottérben az üres formula megelőzi az összes többit a rendszámokon értelmezett valódi rendezési relációnak megfelelően. A negyedik, illetve a hetedik axióma miatt ugyanaz az SPS-beli állapottér nem tartalmazhat sem azonos formulát, sem azonos állapotteret; ezek az axiómák kizárják a redundanciát. Az ötödik, illetve a nyolcadik axióma az SPS-beli formulák, illetve állapotterek rendezettségét irja le a részformula, illetve az in reláció alapján.

Végül rátérünk arra, hogy hogyan generálható egy állapottér-halmaz. Egy elem a felvétel müveletével válik az állapottér elemévé. A felvétel müveletét $\leftarrow(\alpha_1 \setminus \ldots \setminus \alpha_n, x, id_x)$ -szel jelöljük, ahol $\alpha_1 \setminus \ldots \setminus \alpha_n$ egy nyom, x egy elem, $x \in F'$ és id_x az elem neve. id_x-et $id(x)$-szel jelöljük.

A10. $(\forall\alpha\forall\beta\forall x\forall y)\,(in(SPS,\alpha) \wedge in(SPS,\beta) \wedge x \in \{\alpha\} \wedge y \in \{\beta\} \wedge (x = y \Leftrightarrow id(x) = id(y)))$

A továbbiakban $SPS \setminus \alpha_1 \setminus \ldots \setminus \alpha_n$-et $\alpha_1 \setminus \ldots \setminus \alpha_n$-nel röviditjük. Meg kell jegyezzük, hogy $\leftarrow(\alpha_1 \setminus \ldots \setminus \alpha_n, x, id_x)$ akkor is érvényes, ha $ssp(SPS,\alpha_1)$ igaz.

Egy állapottér kétféleképpen jöhet létre: a felvétel müveletével felveszünk belé egy elemet, vagy pedig úgy, hogy a neve szerepel egy felvétel müvelet nyom kifejezésében.

A hatodik axióma α x_i elemeinek balra kiemelési szabályát irja le. Alkalmazása az elémásolási müvelet x_i-re, amelyet $\nwarrow(\alpha, x_i)$ -vel jelölünk. A kilencedik axióma α_k α_i állapotterének balra kiemelési szabálya. Alkalmazása az elémásolási müvelet α_i-re, amelyet $\nwarrow(\alpha_k, \alpha_i)$-vel jelölünk.

A felvétel müvelet definiciója a következő: a $\leftarrow(\alpha_1 \setminus \ldots \setminus \alpha_n, x, id_x)$ felvétel müveletét végrehajtva x az α_n elemévé válik úgy, hogy a következő kifejezés teljesül:

$$(\forall i \forall j \forall k)\ (1 \leq i \leq n \nwarrow(\alpha_i, x) \wedge \underline{\text{if}}\ n \geq 2\ \underline{\text{then}}\ 1 \leq j < k \leq n \nwarrow(\alpha_k, \alpha_j)) .$$

Egy állapottér létrehozása során az x_i-ket és az α_i-ket $\text{mult}(\alpha_i, x_i)$-vel, illetve $\text{mult}(\alpha_j, \alpha_i)$-vel jelöljük meg, ha x_i, illetve α_j balra kiemelendő. Felvétel esetén az x_i-ket illetve α_i-ket $\text{catch}(\alpha_j, x_i)$-vel, illetve $\text{catch}(\alpha_j, \alpha_i)$-vel jelöljük meg.

Az igy definiált SPSS egy körmentes irányitott gráffal jellemezhető [Kő80/a].

3. ÁLLAPOTTÉR GRÁFOK

A jólismert matematikai adatszerkezetek az $\langle A, R \rangle$ rendezett halmazpárral irhatók le, ahol A a szögpontok halmaza, R pedig az A-n értelmezett reláció. További információkat kapcsolva A-hoz, vagy R-hez, vagy mindkettőhöz, egy adott tipusu adatszerkezet finomabban irható le, mindez azonban nem változtat a szóbanforgó adatszerkezet lényegén.

Az R reláció valamilyenfajta megfeleltetést definiál A elemei között, és ebből a szempontból A elemei egyenrangúak. A valóságos rendszerek "épitőkövei" azonban nem egyenrangúak, azokat a matematikai absztrakció emancipálja, amely a rendszer matematikai adatszerkezettel való ábrázolásához szükséges. Másrészt a szögpontok jellegtelensége sem mindig megfelelő a rendszer valamilyen szempontból való tárgyalásához [Ha81/b].

A jellegtelenségből fakadó probléma önmagában még feloldható volna a rendszer cimkézett gráffal történő modellezésével; a szögpontok funkcionális hierarchiája azonban egyetlen hagyományos szemléletü matematikai adatszerkezettel sem ábrázolható maradéktalanul. A funkcionálisan nem egyenrangu szögpontokat tartalmazó matematikai adatszerkezetben a szögpontok tulajdonságai sem ábrázolhatók a cimkézett gráfoknál megszokott passziv módon, e tulajdonságok éppen a funkcionális hierarchia következtében aktivizálódnak.

A funkcionális hierarchia koncepciója öndefiniálható módon ábrázolható a vegyes-halmazok bázisán.

Az A-ban (szögpontok halmaza) definiált R reláció M-ben (szögpontok vegyes-halmaza) is értelmezhető. R-nek azonban most speciális jelentése van: olyan rendezett párok halmaza, amelynek koordinátái M tagjai is lehetnek. Más szavakkal M egy B tagjának elemei együttesen egy tömegként állnak relációban M más elemeivel, illetve tagjaival. Az ilyen relációt tömegrelációnak nevezzük.

Definició 3.1. Az M-en értelmezett R tömegreláció olyan $\langle m_1, m_2 \rangle$ rendezett párok halmaza, amelyek m_1, illetve m_2 koordinátái M tagjai vagy elemei.

Az n-áris tömegreláció a most definiált bináris tömegrelációhoz hasonlóan definiálható.

Ahogy azt a korábbiakban is kifejtettük, időnként lényegesek egy halmaz elemeinek a tulajdonságai is. Ha ez a halmaz valamilyen halmazrendszer, tagjainak is rendelkeznie kell tulajdonságokkal. Magától értetődik, hogy egy halmaz (illetve egy halmaz tagjának) a tulajdonságai elemeinek (vagy tagjainak) tulajdonságaiból kell következnie.

Az állapottér gráfot rekurzive definiáljuk a következőképpen.

Definició 3.2. Az állapottér gráf /SSG/ egy rendezett ötös:

$$SSG = \langle\langle\langle a_s, P_s\rangle, A_s, R_s, O_s\rangle, A, P, R, O\rangle \quad , \quad \text{ahol}$$

- A tetszőleges elemek (szögpontok) halmaza.
- P tulajdonságok halmaza.
- R az SSG alaphalmazán definiált tömegreláció.
- O P-n értelmezett müveletek halmaza.
- Az $\langle a_s, P_s\rangle$ rendezett pár SSG kezdő szögpontja (a_s) egy tulajdonsághalmazzal (P_s) összekötve. P_s elemei a_s tulajdonságait irják le. $P_s \subseteq P$ és $a_s \in A$.
- A_s egy halmaz, amelynek elemei az $\langle a_{s_i}, P_{s_i}\rangle$ ($i \in I$, ahol I A_s indexhalmaza) rendezett párok. a_{s_i} vagy egyedi elem, vagy a kezdő szögpontja az $\langle\langle\langle a_{s_i}, P_{s_i}\rangle, A_{s_i}, R_{s_i}, O_{s_i}\rangle, A, P, R, O\rangle$ állapottér részgráfnak. P_s elemei a_{s_i} tulajdonságait irják le. $P_{s_i} \subseteq P$ és $a_{s_i} \in A$.
- R_s $\langle a_s, P_s\rangle \cup A_s$ -en értelmezett reláció. $R_s \subseteq R$.
- $O_s \subseteq O$. O_s elemei P_{s_i}-ket (minden $i \in I$ -re) P_s-re képezik le.

Hangsúlyozzuk, hogy az állapottér gráfok definiciójában az állapottér részgráfok kezdő szögpontját reprezentáló szögpontok több szinten szerepelnek: egyszer a részgráfot tartalmazó gráf közönséges szögpontjaként, egyszer pedig a részgráf mint független gráf kezdő szögpontjaként. Mindez lehetővé teszi egyrészt az állapottér részgráfok egyetlen egység gyanánt való kezelését (a részgráfot tartalmazó részgráf szintjén), másrészt az állapottér részgráfok független kezelését (a részgráfok szintjén).

Egy állapottér gráf két irányból épithető fel. Felülről lefelé haladva egy állapottér részráf elemei a kezdő szögpont tulajdonságai és a gráfon definiált tömegreláció által a tulajdonsághalmazon értelmezett operációkon keresztül meghatározottak. Ellenkező irányban haladva a részgráf szögpontjai tulajdonságaikon keresztül lokalizálják magát a részgráfot, meghatározva a kezdő szögpont tulajdonságait.

Az állapottér gráfok alkalmazására a [Ha80/a,Ha80/b,Ha81/a]-ban láthatunk példát. A [Ha80/a]-ban vázolt programrendszert oktatási célokra használjuk. A programrendszer fő feladata valamilyen általános célu programozási nyelven irott program elemzése és minősitése jólstrukturáltság, kódolási stilus, komplexitás, stb. szempontjá ból. A felsorolt feladatok megvalósitása az elemzett input programok tervének rekonstruálását igényli. Ez a feladat egy állapottér gráf felépitésével valósul meg (részletes leirását ld. [Ha80/b,Ha81/a]).

4. N-PREFIX KIFEJEZÉSEK

Egy PROLOG kifejezés nem tartalmaz információt operátorainak attributumairól (tekintettel arra, hogy azok funktrok). Ha egy operátor attributumai lényegesek, akkor a felhasználónak kell gondoskodnia kezelésükről a programban. Mindez időnként rendkivül nehézkessé teszi a komplex kifejezések kezelését. Vizsgáljuk meg az alábbi egyszerü példát! +(A,v,D) egyesithető +(A,+(B,C),D)-vel, de nem egyesithető +(A,B,C,D)-vel (ahol v egy változó, A,B,C,D pedig konstansok). Ráadásul a három kifejezés háromfajta + operátort tartalmaz (egy két-, két három- és egy négyargumentumu + operátort). A jelenség okát a három kifejezés fareprezentációjában kell keresnünk. A + operátort ugyanakkor asszociativ operátorként használva, +(A,+(B,C),D) és +(A,B,C,D) egyenlőnek tekinthető (a jólismert algebrai tétel értelmében).

Ugy véljük, hogy a probléma feloldható, ha az asszociativ operátorokat struktúranevükkel jellemezzük, tekintet nélkül argumentumaik számára [Kő79/a].

Definició 4.1. Egy $o(a_1,\ldots,a_n)$ $n \geq 1$ kifejezést n-prefix kifejezésnek nevezünk, ha o egy asszociativ operátor és a kifejezést a struktúraneve jellemzi csupán, függetlenül argumentumainak számától.

Definició 4.2. Legyen p egy asszociativ operátor inverze. A $p(a_1,a_2,\ldots,a_n)$ n-prefix kifejezést igy értelmezzük:

$$p(a_1,a_2,\ldots,a_n) \stackrel{d}{=} (\ldots(a_1 p a_2) p \ldots).$$

Megjegyezzük, hogy az n-suffix kifejezések hasonlóan definiálhatók.

Az n-prefix kifejezések egyesitését [Kő80/b]-ben tárgyaljuk. Az n-prefix kifejezéseket általános állapotterekkel ábrázolva a kifejezés struktúraneve lesz az állapottér neve, elemei pedig a kifejezés argumentumai. Ilymódon két különböző argumentumszámu (de azonos struktúranevü) n-prefix kifejezés egyesitése egy egyszerü állapottérnek (a kevesebb argumentumu kifejezés) egy összetett állapottérbe

(a többargumentumu kifejezés) való leképezését jelenti, nem sérti az egyszerüsités szabályait.

5. AZ ÁLLAPOTTÉR-HALMAZ STRUKTURÁK ÉS AZ N-PREFIX KIFEJEZÉSEK ALKALMAZÁSA

Az állapottér-halmaz strukturák és az n-prefix kifejezések sikeresen alkalmazhatók sokféle probléma megoldásában, és felhasználási területük a tudományok fejlődésével párhuzamosan nő. Biztosan alkalmazhatók például az alábbi szakterületeken: matematikai nyelvészet, vegyészet, biokémia, rendszerelmélet, nyelvi elemzők, programozáselmélet, kódoptimalizálás, stb. Most néhány megvalósitott alkalmazást ismertetünk vázlatosan.

5.1. EGYSZERÜSITÉS MATEMATIKAI STRUKTURÁKBAN

A program a bináris fa gyanánt ábrázolt kifejezésekben alulról-felfelé, jobbról-balra haladva egyszintü előre- és többszintü hátratekintéssel végzi a feldolgozást. Asszociativ operátorlánc esetén a megfelelő sorrendben rendez, a müveleti szabályoknak megfelelően elvégzi az összevonásokat és az egyszerüsitéseket, majd n-prefix kifejezéssé alakitja, melyen további egyszerüsitéseket végez. A program a matematikai strukturák igen széles osztályában képes egyszerüsiteni, pl. csoportok, gyürük, testek, hálók, stb. A program fejlesztés alatt álló változata képes lesz képletrendezésre, és néhány reláció kiértékelésére [Kő79/a].

5.2. ADOTT VALÓS FÜGGVÉNY ELSŐ N FORMÁLIS DERIVÁLTJÁT ELŐÁLLITÓ PROGRAM

A PROLOG program [Kő79/b] egy olyan FORTRAN szubrutint generál, amely előállitja egy adott függvény első n formális deriváltját. A FORTRAN szubrutin a függvény első n formális deriváltjának a helyettesitési értékét számitja ki. Az adott függvény együthatóinak és változóinak helyettesitési értékét a szubrutin paraméterként kapja.

5.3. LIGAND-KÖTŐ RENDSZEREK NUMERIKUS ANALIZISE

A PROLOG program [Ba79] egy FORTRAN szubrutint generál, amely kiszámitja a megfelelő kezdőértékeket a ligandkötő rendszerek numerikus elemzéséhez. A PROLOG program müködése az 5.2.-ben leirt programgenerátoron alapul.

IRODALOMJEGYZÉK

[Ba79] Bartha Ferenc, Kőfalusi Viktor: Ligandkötő rendszerek numerikus analizise. SZÁMKI, SOFTTECH D42, 128-132 old. 1979.

[Be78] Bendl Judit, Varga Kálmán, Kósa Márton, Balogh Kálmán: Moduláris PROLOG interpreterének specifikációja. /Nagyvonalu rendszerterv, SZÁMKI, SOFTTECH D20,

/Nagyvonalu rendszerterv/, SZÁMKI, SOFTTECH D20, 1978.

[Be79] Bendl Judit, Boda Jánosné, Bogdánfy Géza, Kósa Márton, Naszvadi László, Visnyovszky József: Az MPROLOG rendszer felhasználói dokumentációja. NIM IGÜSZI tanulmány a SZÁMKI részére, 1979.

[Ber71] Berztiss, A.T.: Data Structure Theory and Practice. Academic Press, New York - London, 1971.

[Bo72] Boyer, R.S., Moore, J.S.: The sharing of structure in theorem proving programs. DAI memo No. 47, Univ. of Edinburgh, 1972.

[Ha80/a] Halmayné Szentirmay Edit, Gerő Péter: PROGART, a computerized assistant for the programming instructor. SZÁMOK tanulmány, 1980.

[Ha81/a] Halmayné Szentirmay Edit, Kőfalusi Viktor: Állapottér-szemléletü modellezés. SZÁMOK tanulmány, 1981.

[Ha81/b] Halmayné Szentirmay Edit: Problem-reduction approach of the designing process. SZÁMOK tanulmány, 1981.

[Ha81/c] Halmayné Szentirmay Edit, Gerő Péter: The PROGART program system. SZÁMOK tanulmány, 1981.

[Ko79] Kowalski, R.: Logic for problem solving. North Holland, New York, 1979.

[Kő79] Kőfalusi Viktor, Bartha Ferenc: A PROLOG alkalmazási és bővitési lehetőségeiről; állapottér-halmazokon alapuló PROLOG alkalmazásokról. tanulmánykötet. SZÁMKI, SOFTTECH D42, 1979.

[Kő79/a] Kőfalusi Viktor: Egyszerüsités matematikai strukturákban. SZÁMKI, SOFTTECH D42, 1979.

[Kő79/b] Kőfalusi Viktor: Adott, valós, nagybonyolultságu, többváltozós analitikus függvény első n formális deriváltját előállitó PROLOG program, ennek alkalmazásai, különös tekintettel a formális hatványsorba fejtésre. SZÁMKI, SOFTTECH D42, 104-127 old. 1979.

[Kő80/a] Kőfalusi Viktor: Állapottér-halmazok - matematikai megalapozás. Kézirat. 1980.

[Kő80/b] Kőfalusi Viktor: Egy kiterjesztett egyesitő. kézirat. 1980.

[Kő81/a] Kőfalusi, V., Halmay, E.: State-space sets, state-space graphs and n-prefix expressions. SIGSAM BULLETIN, No. 1. pp 33-36. 1981.

[Kő81/b] Kőfalusi, V.: The state-space set and its applicability in PROLOG language. megjelenés alatt a CL&CL-ben, 1981.

[Sá79] Sántáné-Tóth Edit: A hazai PROLOG alkalmazások helyzete 1979-ben. SZÁMKI, SOFTTECH D40. 1979.

[Sze77] Szeredi Péter, Futó Iván: PROLOG kézikönyv. Számológép, XIII. 3-4. szám, 1977.

[Ha81/d] Halmay, E.: Evolutional system model. res.rep. SZÁMOK. 1981.

[Dá81] Dávid, G.: Problemsolving = knowledge + strategy. Int. Conf. on AI and Inf. Control Systems of Robots. 1980. Czehoslovakia, in Computer and Artificial Intelligence. Journal in print. VEDA Publ. House, Czehoslovakia, 1981.

ABSTRACT

Two mathematical objects, namely the state-space set structure /SPSS/ and the state-space graph /SSG/, are defined. The so-called n-prefix expressions are also defined. SPSSs and SSGs and n-prefix expressions can be apllied in the field of symbolic mathematical computation and formula manipulation.

Köles Péter

DIGITÁLIS KÉPTRANSZFORMÁCIÓK SEJTPROCESSZORRAL

A képfeldolgozási müveletek az esetek tulnyomó részében nagyszámu, párhuzamos számitást tartalmaznak. Ez a hagyományos szekvenciális számitógépeken rendkivül időigényes. Célszerü ezeket a számitásokat párhuzamos müveletvégzésre alkalmas rendszerre bizni. Ilyen rendszer a sejtprocesszor. A dolgozat alapvetően kétféle tipusu képtranszformációval foglalkozik. Az első részben geometriai képtranszformációkról esik szó, konkrétan: nagyitásról és tengelyes tükrözésről. A második rész konvolueiós képtranszformációkkal foglalkozik, a Laplace és a Mérő-transzformáció kerül ismertetésre.
Az algoritmusok a Legendi Tamás által javasolt sejtprocesszor architektura /ld. [1]/ számára készültek.

Kulcsszavak: sejtprocesszor, nagyitás, tükrözés, Laplace-transzformáció, Mérő-operátor.

1. A javasolt sejtprocesszor architektura

A sejttér kétdimenziós, téglalap alaku. Körbe un. bábsejtek határolják, ezek biztositják a sejttérnek a külvilággal való kapcsolatát. A sejtek viszonylag kis állapotszámuak /max. 16/, Neumann-szomszédság szerint érintkeznek egymással, egy sejt a négyzetrács szerinti négy szomszédját érzékeli.

A sejtek változtatható átmenetfüggvénnyel vannak ellátva. A vezérlő központból /CCPU/ érkezett speciális parancsokat /mikroutasitásokat/ hajtják végre, a környezetük és a központi utasitások együttes hatására váltanak állapotot. Alkalmas mikroutasitás-sorozattal /mikroprogram/ tetszőleges átmenetfüggvény végrehajtása elérhető. Ezáltal az átmenetfüggvények tetszőlegesen cserélhetők, sőt az egymás után végrehajtott

lépésekben is különbözőek lehetnek /időbeli inhomogenitás/.

A sejtek belső állapottal is el vannak látva, ezeket a többi sejt nem érzékeli. A központból érkezett parancsok végrehajtásánál a sejtek figyelembe veszik saját belső állapotukat is. Igy elérhető, hogy a különböző belső állapotu sejtek másképp reagáljanak ugyanarra a parancsra, vagyis különböző átmenetfüggvényt hajtsanak végre. Ezáltal a sejttér térbeli inhomogenitással rendelkezik. A belső állapotok értékeit a sejttér inditása előtt beállitjuk, és azok a müködés során nem változnak.

2. Geometriai képtranszformációk

Hagyományos értelemben vett sejtautomatákra vonatkozóan ilyen jellegü transzformációkat találhatunk pl. Beyernél [4]. Az ismertetendő megoldások viszont a fenti konkrét sejtprocesszszorra készültek, kihasználva annak lehetőségeit. Minden esetben feltesszük, hogy egy képpont elfér egy sejtben, a képpontok denzitásértékei tehát 0 és 15 között lehetnek.

Az algoritmusok térben és időben inhomogén sejttérben valósithatók meg. A megoldásokat két csoportba sorolhatjuk aszerint, hogy a sejttér a teljes képet tartalmazza, vagy csak annak egy részét. A teljes egészében a sejttérben lévő kép előnye, hogy egymás után több különböző transzformáció elvégezhető rajta kis időigénnyel. Hátránya a nagy sejttérméret, és a kép betöltésével és kiolvasásával járó időveszteség. A másik csoport jellemzője, hogy a sejttéren való folyamatos áthaladás közben hajtjuk végre a transzformációt a képen.

Sejttérben a következőképpen értelmezhetjük pl. a kétszeres nagyitást: egy adott sejtnek a nagyitással négy db, négyzet alakban elhelyezkedő sejtet feleltetünk meg.

A leirt algoritmusok kétszeres nagyitást eredményeznek, de az átmenetfüggvényciklusok alkalmas megválasztásával tetsz.

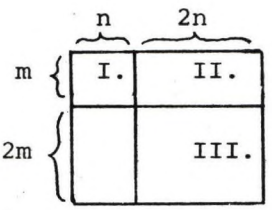
arányuvá tehetjük a nagyitást. 3/2, 5/2, 4/3 stb., ill. hosszanti irányban p-szeres, függőleges irányban q-szoros nagyitás is elérhető. Kicsinyitünk, ha a nagyitó algoritmusokat megforditva hajtjuk végre.

Kétféle nagyitó algoritmus kerül ismertetésre. Az egyik esetben a kép a sejttérben van, ott történik a nagyitás is. A másiknál a kép a sejttéren való folyamatos áthaladás közben nagyitódik fel. A függőleges tengelyre való tükrözésnél a teljes kép a sejttérben van.

2.1. Sejttérbeli kép nagyitása

A nagyitást két menetben végezzük. Az első menetben egy mxn-es képet mx2n-essé nagyitunk a második menetben ezt tovább nagyitjuk 2mx2n-esre. Az egyes menetek sorrendje felcserélhető.

I. menet: A sejtteret két részre osztjuk egy függőleges vonallal 1:2 arányban. A nagyitandó képet az I. mezőbe a sejttér bal felső sarkába tesszük /1. ábra/. Az I. mezőben minden második lépésben jobbra shiftelő /léptető/



1. ábra A sejttér bal alsó mezőjét nem használtuk ki.

átmenetfüggvényt alkalmazunk, a közbülső lépésekben a kép helyben marad. Az I. mezőn levő kép ily módon 2 billenés alatt egyet lép jobbra, mindaddig, amig teljesen ki nem halad az I. mezőről. A II. mezőn minden lépésnél jobbra shiftelő függvény van érvényben. A sejttér egy sorának

billenésenkénti müködése a következőképpen szemléltethető:

	I. mező	II. mező
kezdeti állapot	... ab	...
1. lépés	... ab	b
2. lépés	... a	bb
3. lépés	... a	abb
4. lépés	...	aabb stb.

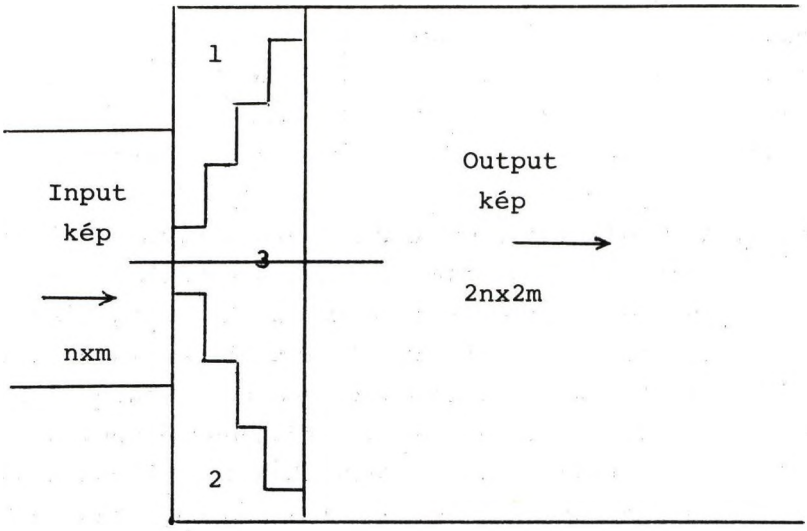
II. menet: A müködési elv ugyanaz, mint az első menetnél. A sejtteret most egy vizszintes vonallal osztjuk 2 részre, szintén 1:2 arányban. A "félig kész" kép a II. mezőn van /1. ábra/, itt minden második lépésben lefelé toló lépéseket hajtunk végre. A III. mezőn mindig lefelé toló függvény érvényes.
A két menet végrehajtása után kétszeresére nagyitott képet kaptunk. Mivel a kép nagyitás közben mozog a sejttérben, egy mxn-es kép kétszeresére nagyitásához 3mx3n méretü sejttérre van szükség. Az I. menethez 2n, a II-hoz 2m, tehát összesen 2(m+n) lépés kell.
A teljes folyamat az 1. ábrán látható inhomogenitást igényli /3 belső állapot szükséges/.

Az egyik irányba p-szeres, másik irányba q-szoros nagyitáshoz az átmenetfüggvényciklust ugy kell megválasztani, hogy az I. menetben a kép p lépésenként lép egyet jobbra az I. mezőn, a II. menetben q lépésenként lép egyet lefelé a II. mezőn. Az ilyen nagyitáshoz (p+1)mx(q+1)n méretü sejttér szükséges. A lépésszám pn+qm kell, hogy legyen.

2.2. Pipe-line nagyitás

A következőkben leirt algoritmus esetében a kép a sejttéren való folyamatos áthaladás közben nagyitódik fel, és ugy hagy-

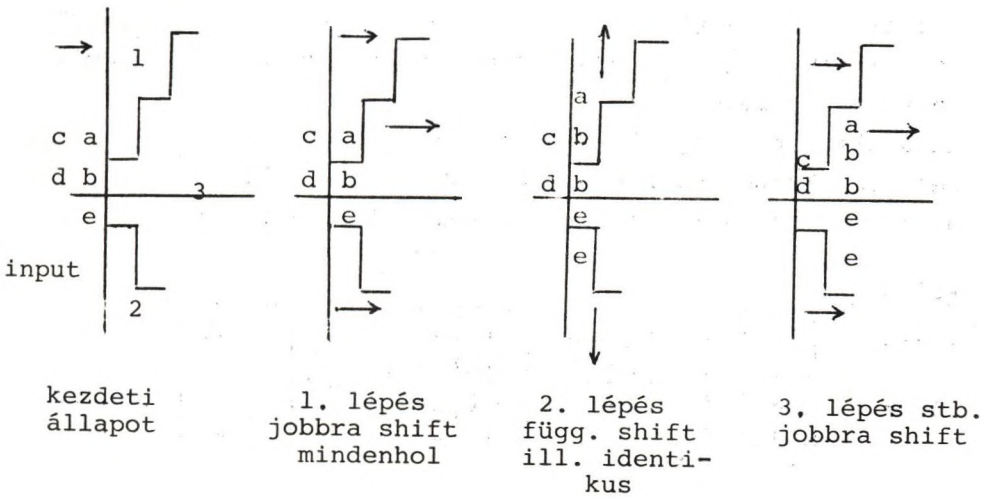
ja el a sejtteret. Az ismeretetésre kerülő szerkezettel kétszeres nagyitás érhető el. A 2. ábrán n=8. A lépcsősor magassága 2, szélessége 1 sejt.



2. ábra

Az input képpontok kétlépésenként érkezhetnek a sejttérbe balról oszloponként ugy, hogy a sejttér vizszintes középvonala 2 egyenlő részre osztja a képet. Az 1. részben felváltva jobbra illetve felfelé shiftelés ismétlődik. A 2. részben jobbra és lefelé shiftelés történik. A 2. részben kétlépésenként jobbra lép a kép. Az egyes részekben a jobbra tolás a bábsejtekre adással összehangolva, egyszerre történik. A szerkezeten való áthaladás után, a kép függőleges irányban kétszeresére lesz széthuzva /3. ábra/.
A másik irányu széthuzás ezután következik, ezt ugy érhetjük el, hogy a szerkezeten áthaladt képet lépésenként vesszük le a jobb szélső bábsejtoszlopról. Mivel kétszer egymás után ugyanazt a képpontot vesszük le, elértük a vizszintes irányu

széthuzást is.



kezdeti állapot

1. lépés jobbra shift mindenhol

2. lépés függ. shift ill. identikus

3. lépés stb. jobbra shift

3. ábra

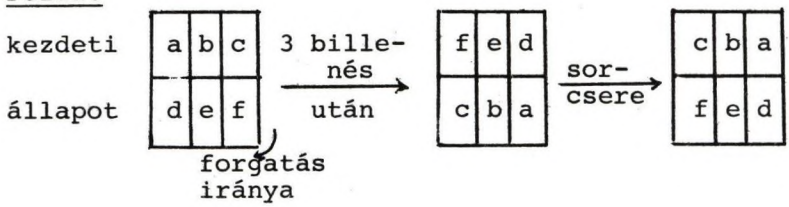
Egy mxn-es kép kétszeres nagyitásához $2nxn/2=n^2$ db sejt szükséges. Ahhoz, hogy a kép utolsó oszlopa is a sejttérbe érjen, 2m lépés kell, amig ez az oszlop áthalad a sejttéren az 2xn/2=n lépés, tehát összesen 2m+n lépés szükséges.

Alkalmasan megválasztva a "lépcsősor" magasságát ill. mélységét, és a szerkezeten belül az egyes különböző irányokba shiftelő lépésekből álló ciklust, tetszőleges méretü nagyitás érhető el. pl: 3 sejt magas, 1 sejt széles lépcső, és 2 függőleges irányu meg 1 jobbra shiftelő lépésből álló ciklus 3-szoros nagyitást eredményez. A bábsejtekre 3 lépésenként lehet az inputot ráadni, és lépésenként kell az outputot levenni.

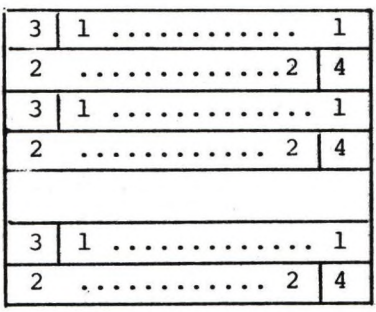
2.3. Függőleges tengelyre való tükrözés

Az algoritmus lényege a következő: A sejtteret két sejtsorból álló részekre osztjuk.Minden ilyen egység ugyanazt végzi, mégpedig a bennük tárolt adatok körbeáramoltatását. Ha a sejttér szélessége n, akkor n lépés után a két sor helyet cserél ugy, hogy ami eddig a baloldalon volt, az a jobboldalra kerül, s forditva.
Az alakzatok irányitása ezzel megváltozik, ahogy az a tengelyes tükrözésnél szükséges. Ezután egy billenés alatt a megfelelő két sor felcserélésével előáll a tükörkép.

Példa:



Ha n páros, a tengely az n/2 és n/2+1 sejtoszlop között helyezkedik el, ha n páratlan, akkor maga az n/2+1 sejtoszlop a tengely. A sejttér inhomogenitása a 4. ábrán látható.



1: jobbra shiftelő függvény
2: balra shiftelő függvény
3: felfelé shiftelő függvény
4: lefelé shiftelő függvény

4. ábra

Ez az elrendezés biztositja az adatok körbeáramlását. Az utolsó billenésnél /sorcsere/ az 1 és 3 belső állapotu sej-

tek fölfelé, a 2 és 4 belső állapotu sejtek lefelé shiftelést hajtanak végre.

Egy mxn-es kép tükrözéséhez ha m páros, akkor mxn-es, ha páratlan, (m+1)xn-es méretü sejttér szükséges, a tükrözés n+1 lépésben hajtódik végre.

A leirt módszer kétszer egymás utáni alkalmazásával, ahol a szimmetriatengelyek egymásra merőlegesek, 180°-os forgatást kapunk.

3. Konvolucióval végzett képtranszformációk

Adott egy A mxn-es mátrixkép. Legyen C adott 3x3-as maszkmátrix. A a transzformálandó kép mátrixa. Feladat a B mátrix számitása a következő formula alapján:

$$b_{u,v} = \sum_{i=1}^{3} \sum_{j=1}^{3} c_{ij}\, a_{u+i-2,v+j-2} \qquad \text{ahol} \quad \begin{matrix} u=2,\ldots,m-1 \\ v=2,\ldots,n-1 \end{matrix}$$

Ha megfelelő méretü sejttér rendelkezésünkre áll, a kép teljes egészében betölthető, és az egész képen egyszerre végezhető a transzformáció. /Pl. a Laplace-transzformáció 1 lépésben triviálisan megoldható./ Az esetek tulnyomó többségében a kép nem fér be a sejttérbe, ilyenkor célszerü egy alkalmas szerkezeten soronként vagy oszloponként keresztülhajtva folyamatosan végezni a transzformációt. Ezzel a beirással és kiolvasással járó időt is megtakaritjuk.

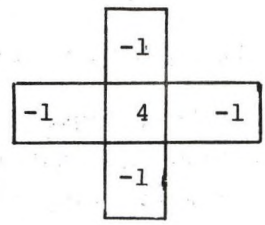
Feltesszük, hogy egy képpont elfér egy sejtben, azaz lehetséges értéke 0 és 15 között van. Emiatt a közbenső számitási lépéseknél bizonyos kerekitésre van szükség. Előfordulhat, hogy a kép szélesebb a rendelkezésre álló sejttérnél. Ilyenkor a sejttér méretének megfelelő vizszintes csikokra vághatjuk, amelyek két sor átfedéssel kapcsolódnak egymáshoz, és ezeket a csikokat egymás után keresztülhajtva végezhetjük a transzformációt. Igy bármilyen kis sejttérrel

tetszőlegesen nagy kép feldolgozható, természetesen megnövekedett müködési idő árán.

Jelölés: Legyen a továbbiakban A_i az A kép i-edik oszlopa, A_{i+1} az i+1-edik, stb..

3.1. Laplace transzformáció

A Laplace transzformáció alakzatok konturpontjainak kiemelésére szolgál, maszkja:



Az algoritmus két sejtoszlopból álló szerkezettel megvalósitható, a baloldali harmadik oszlop lehet a bábsejteken.

Kezdőállapot: (A_{i+1}) A_i A_{i-1}

1. lépés: (A_{i+1}) X A_i — X az adott maszk szerint számitódik, A_i jobbra lép.

2. lépés: (A_{i+1}) A_i X — A két oszlop helyet cserél.

3. lépés: (A_{i+2}) A_{i+1} A_1 — X kilép a bábsejtekre, uj input oszlop lép a baloldali bábsejtekre, a többi oszlop jobbra lép.

Az egyes oszlopok kiszámitásánál ez a 3 lépésből álló ciklus ismétlődik.

Az algoritmus megvalósitásához 2m, illetve ha a képet k db vizszintes csikra vágjuk:(m/k)x2 db sejt szükséges. A szükséges lépésszám 3n ill. 3kn.
A két oszlopban más-más belső állapot szükséges.

3.2. Mérő operátor számitása

Feladat ugyanarra a képpontra két különböző /X és Y/ maszk által kijelölt számitás elvégzése, a kettő hányadosa (X/Y) adja a képpont uj értékét. A transzformáció vonalak irányának jellemzésére szolgál /ld. [5]/.

X:

0	1	1
-1	0	1
-1	-1	0

Y:

1	1	0
1	0	-1
0	-1	-1

A Neumann-szomszédság miatt kissé bonyolultabb a számitás, a középső sejt nem érzékeli közvetlen a sarkokban lévő sejteket. Az algoritmus párhuzamosan számolja a két maszk szerinti értéket, a végén osztja el egyiket a másikkal. Ez 6 sejtoszlopból álló szerkezettel, 8 lépéses ciklussal valósitható meg.

Kezdőállapot: $A_{i+1}\ A_i\ A_{i-1}\ A_i\ A_{i+1}\ A_i$ A 2. oszlopban lévő A_i-vel számitjuk az első maszkot, a 4. oszlopban lévővel a másikat.

1. lépés: $A'_{i+1}\ X\ A'_{i-1}\ Y\ A''_{i+1}\ A_i$ X és Y maszkja egyaránt

	1	
-1	0	1
	-1	

Az 1. és 3. oszlop egy sorral feljebb, az 5. eggyel lejjebb lép.

2. lépés: $A_{i+1}\ X'\ A_{i-1}\ Y'\ A_{i+1}\ A_i$

Az eltolt oszlopok visszalépnek.

X' maszkja:

	0	
-1	1	0
	0	

Y' maszkja:

	0	
-1	1	1
	0	

3. lépés: $A_{i+1}\ X'\ A''_{i-1}\ Y'\ A_i\ A_{i+1}$ Az 5. és 6. oszlop helyet cserél, a 3. egy sorral lejjebb lép.

4. lépés: $A_{i+2}\ A_{i+1}\ X''\ A_i\ Y'\ A_{i+1}$ Input sejtoszlop belép,

X" maszkja:

	0	
1	1	0
	0	

5. lépés: $A_{i+1}\ A_{i+2}\ A_i\ X''\ A_{i+1}\ Y'$
6. lépés: $A_{i+1}\ A_i\ A_{i+2}\ A_{i+1}\ X''\ Y'$
} oszlopcserék

7. lépés: $A_{i+2}\ A_{i+1}\ A_i\ A_{i+2}\ A_{i+1}\ Z$ $Z = \frac{X''}{Y'}$, ugyanaz az input oszlop lép be mégegyszer.

8. lépés: $A_{i+2}\ A_{i+1}\ A_i\ A_{i+1}\ A_{i+2}\ A_{i+1}$ Az output oszlop levétele, az 5. oszlop balra és jobbra is lép, a 4. oszlop lép az 5. helyére.

Az oszlopok függőleges irányu mozgása miatt, hogy ne lépjenek ki a sejttérből (m+2)x6 ill. (m/k+2)x6-os sejttér szükséges. A lépésszám 8n ill. 8kn kell hogy legyen. 6 belső állapot szükséges, minden oszlopban más-más.
A Mérő-operátor 5x5-ös /vagy 7x7-es/ maszkok esetén pontosabb értéket ad, ehhez a leirtakhoz hasonló /némileg bonyolultabb/, több lépésből álló ciklus szükséges.

Abstract:
The picture-elaborating exercises contain in the overwhelming majority of cases a great number of paralell operations. It is advisable to entrust them to a system which is suitable for a great number of paralell operations. A system of that kind is the cell-processor.

The paper deals basically with two kinds of transformation: the first part is about geometrical picture- transformations, more exactly about enlargement and axial reflection, the second one is about convolational picture-transformation, the transformation of Laplace and Mérő is expounded. The algorithms are made for the cell-processor model suggested by Tamás Legendi./ see No. 1/

Irodalomjegyzék:

1 Legendi, T.: Cellprocessors in computer architecture. Computational Linguistics and Computer Languages 11 /1977/, 147-167.

2 Legendi, T.: Programming of cellular processors. Proceedings of the Braunschweig Cellular Meeting, June 2-3, 1977.

3 Katona, E.: Sejtalgoritmusok /Válogatás a Legendi Tamás által vezetett sejtprocesszor team munkáiból./ Neumann János Számitógéptudományi Társaság, 1981.

4 Beyer, W. T.: Recogmition of Topological Invariants by Iterative Arrays.
doktori disszertáció, MIT, Cambridge, Mass., 1969.

5 Mérő, L.: Edge extraction and line following using parallel processing /IEEE Workshop on Picture Data Description and Management, 1980/

Köles Péter
MTA Automataelméleti Tanszéki Kutató Csoport
6720 Szeged, Somogyi u. 7.

Függelék

Szimulációs példák

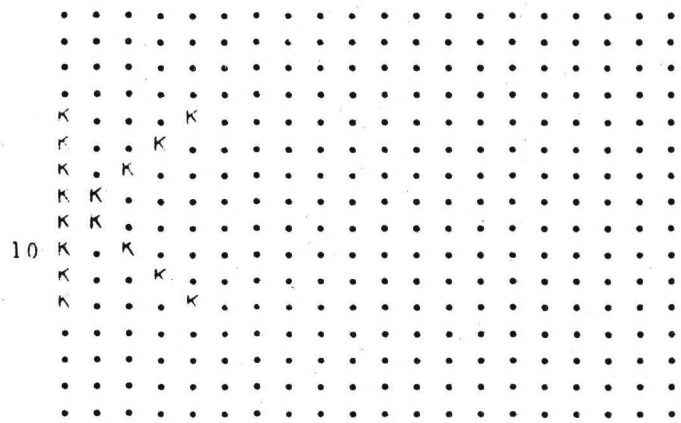
A példák a CELLAS sejttérszimulációs processzor segitségével készültek. A példákban kétállapotu sejtekkel dolgoztunk, az 1 állapot "K" betü karakterrel, a 0 állapot "." karakterrel van megjelenitve. A kezdőállapotban a sejtek belső állapotait is megjelenitettük.

1. szimulációs példa: pipe-line nagyitás
2. szimulációs példa: tengelyes tükrözés.

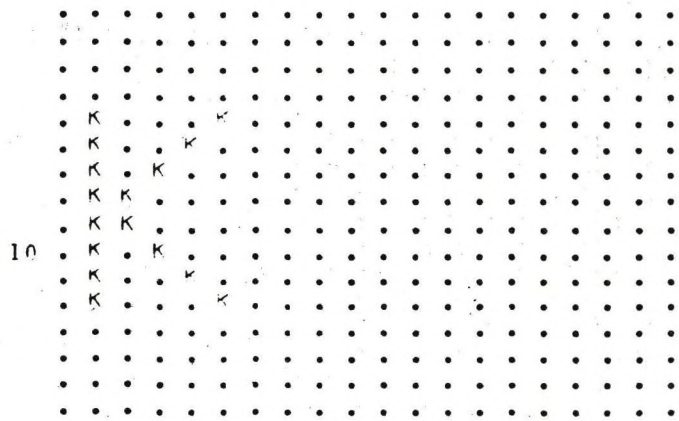
```
   LEPES=    0. MERET= 16 *  20, CSUCSOK=(  2,  2)( 17, 21),

   1  1  1  1  1  4  4  4  4  1  2  2  2  2  2  2  2  2  2  2      ***DUAL ***,
   1  1  1  1  1  4  4  4  1  1  2  2  2  2  2  2  2  2  2  2
   1  1  1  1  1  4  4  4  1  1  2  2  2  2  2  2  2  2  2  2
   1  1  1  1  1  4  4  1  1  1  2  2  2  2  2  2  2  2  2  2
   1  1  1  1  1  4  4  1  1  1  2  2  2  2  2  2  2  2  2  2
   1  1  1  1  1  4  1  1  1  1  2  2  2  2  2  2  2  2  2  2
   1  1  1  1  1  4  1  1  1  1  2  2  2  2  2  2  2  2  2  2
   1  1  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2  2
   1  1  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2  2
10 1  1  1  1  1  5  1  1  1  1  2  2  2  2  2  2  2  2  2  2
   1  1  1  1  1  5  1  1  1  1  2  2  2  2  2  2  2  2  2  2
   1  1  1  1  1  5  5  1  1  1  2  2  2  2  2  2  2  2  2  2
   1  1  1  1  1  5  5  1  1  1  2  2  2  2  2  2  2  2  2  2
   1  1  1  1  1  5  5  5  1  1  2  2  2  2  2  2  2  2  2  2
   1  1  1  1  1  5  5  5  1  1  2  2  2  2  2  2  2  2  2  2
   1  1  1  1  1  5  5  5  5  1  2  2  2  2  2  2  2  2  2  2
         PRIM
         CALLE 1  531
         CALLE 2  532
         CALLE 4  534
         CALLE 5  535
         DISP
```

```
   LEPES=    0. MERET= 16 *  20, CSUCSOK=(  2,  2)( 17, 21),

   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .      ***PRIM ***,
   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
   K  .  .  .  K  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
   K  .  .  K  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
   K  .  K  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
   K  K  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
   K  K  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
10 K  .  K  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
   K  .  .  K  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
   K  .  .  .  K  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
         DISP   0  1  6
         DO 20
```

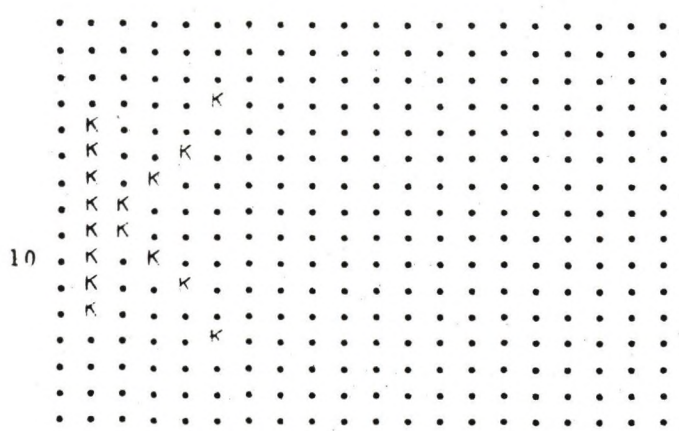
```
   LEPES=    1. MERET= 16 *  20. CSUCSOK=(  2,  2)( 17, 21),
                                                  ***PRIM ***,
```



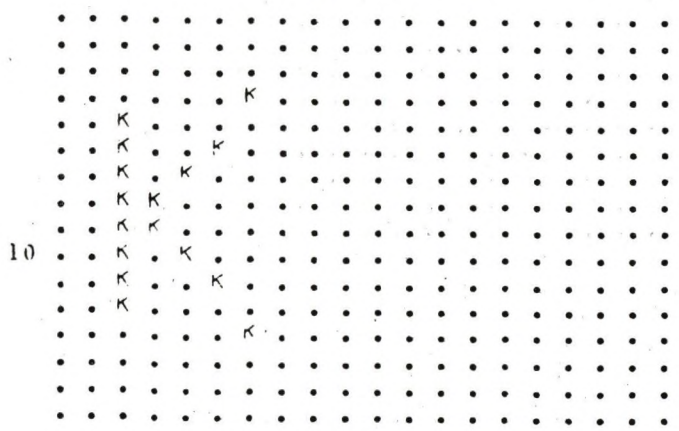
```
   LEPES=    2. MERET= 16 *  20. CSUCSOK=(  2,  2)( 17, 21),
                                                  ***PRIM ***,
```



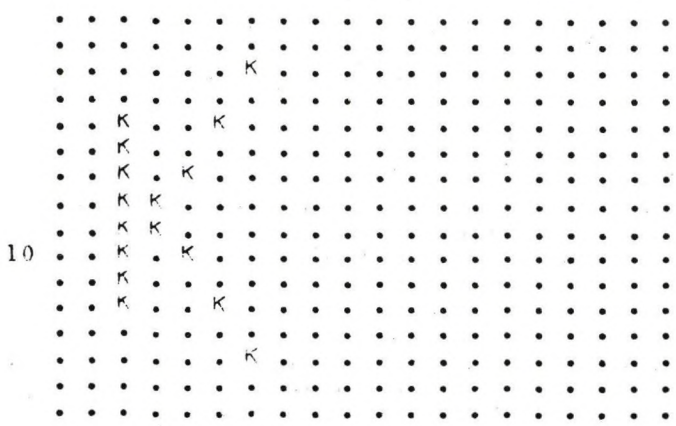
```
 LEPES=   3, MERET= 16 *  20, CSUCSOK=(  2,  2)( 17, 21),
                                                ***PRIM ***,
```



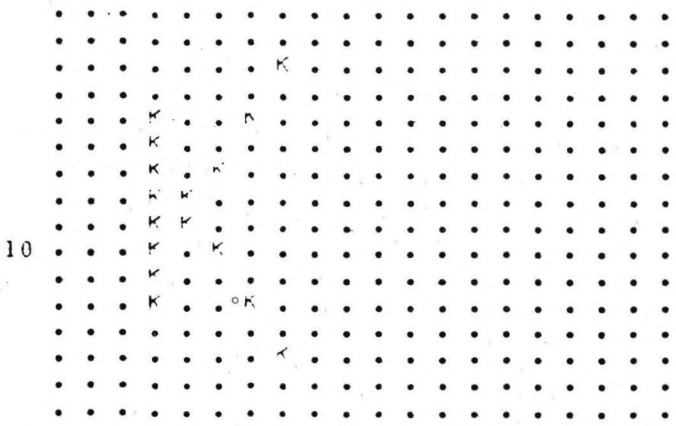
```
 LEPES=   4, MERET= 16 *  20, CSUCSOK=(  2,  2)( 17, 21),
                                                ***PRIM ***,
```



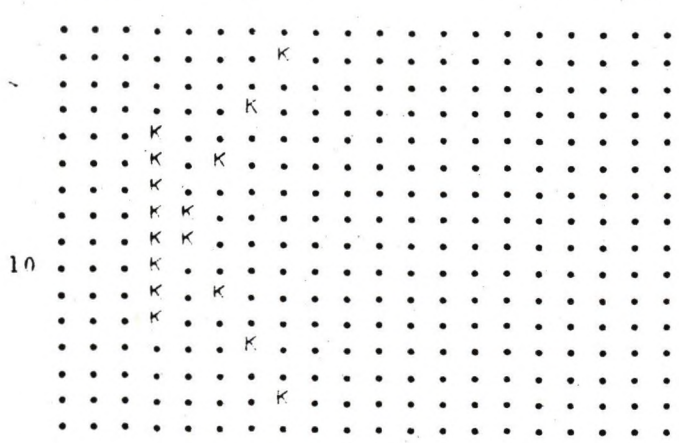
LEPES= 5, MERET= 16 * 20, CSUCSOK=(2, 2)(17, 21),*
***PRIM ***,



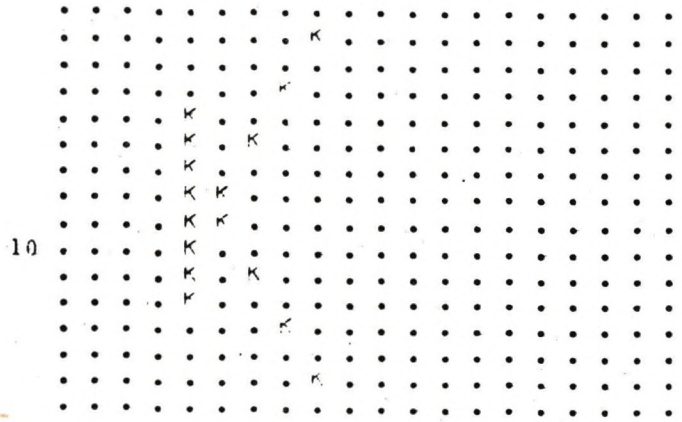
LEPES= 6, MERET= 16 * 20, CSUCSOK=(2, 2)(17, 21),*
***PRIM ***,



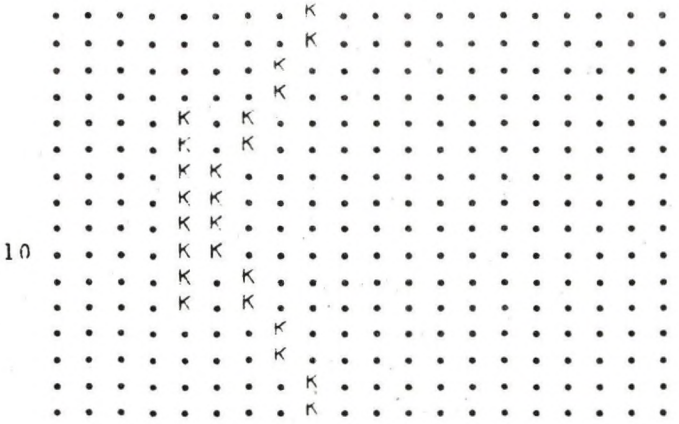
```
 LEPES=   7, MERET= 16 *  20, CSUCSOK=(  2,  2)( 17, 21),*
                                                ***PRIM ***,
```



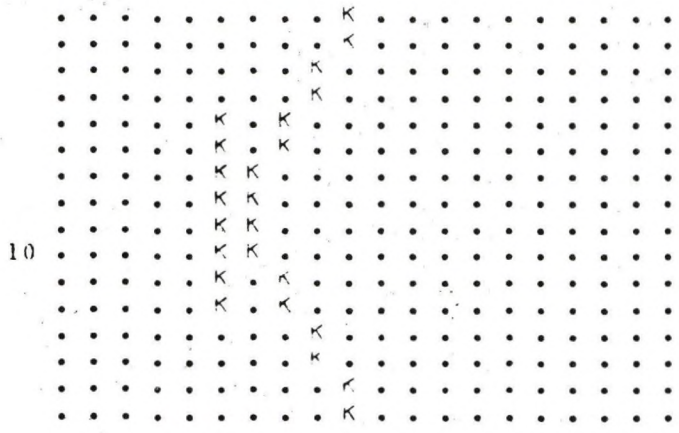
```
 LEPES=   8, MERET= 16 *  20, CSUCSOK=(  2,  2)( 17, 21),*
                                                ***PRIM ***,
```



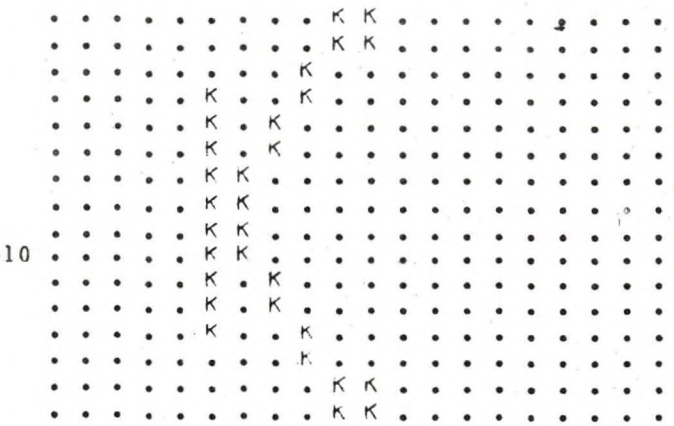
LEPES= 9, MERET= 16 * 20, CSUCSOK=(2, 2)(17, 21),*
***PRIM ***,



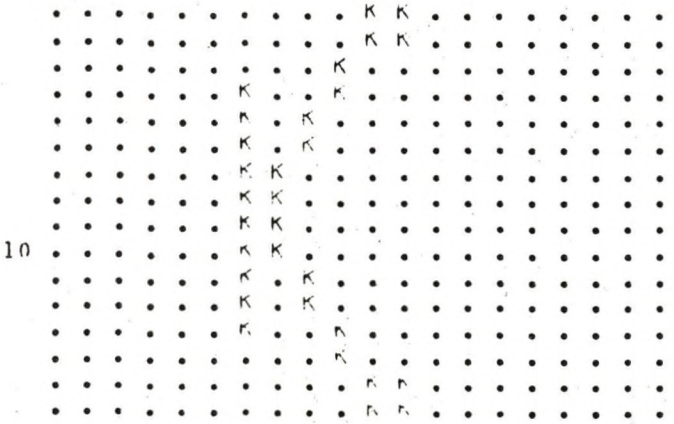
LEPES= 10, MERET= 16 * 20, CSUCSOK=(2, 2)(17, 21),
***PRIM ***,



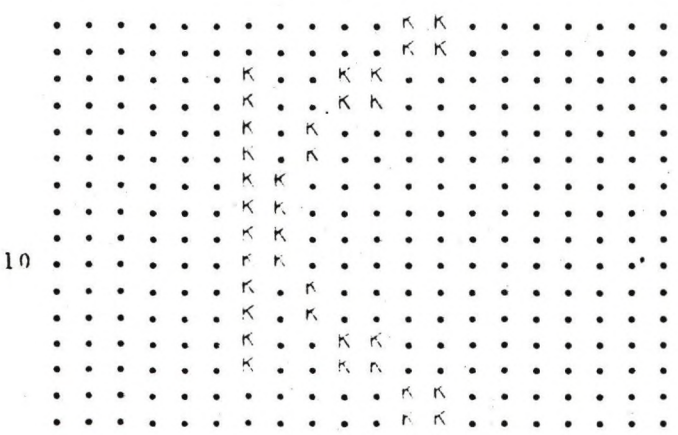
```
LEPES=  11, MERET= 16 *  20, CSUCSOK=(  2,  2)( 17, 21),*
                                                    ***PRIM ***,
```



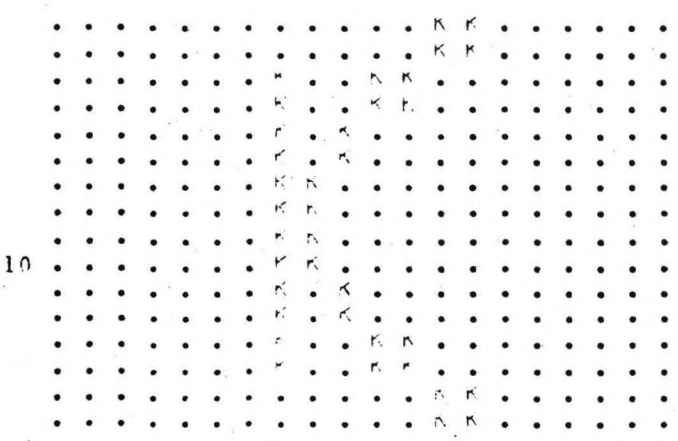
```
LEPES=  12, MERET= 16 *  20, CSUCSOK=(  2,  2)( 17, 21),*
                                                    ***PRIM ***,
```



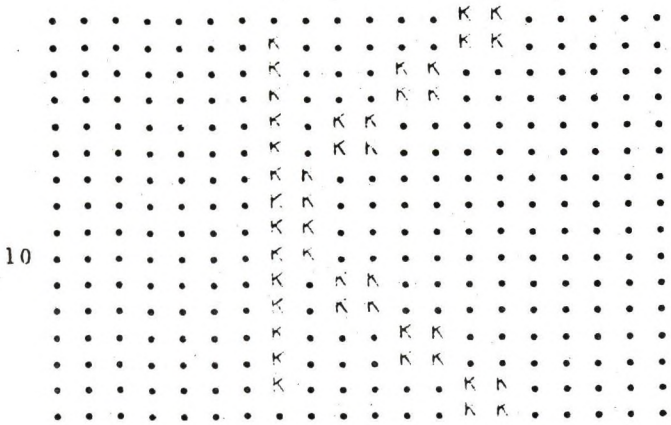
LEPES= 13, MERET= 16 * 20, CSUCSOK=(2, 2)(17, 21),*
PRIM *,



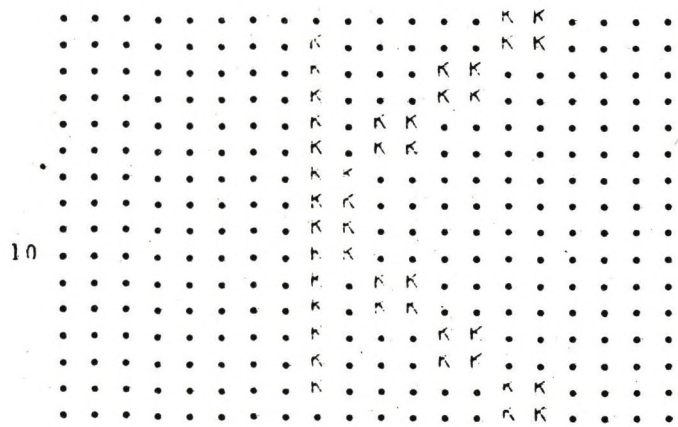
LEPES= 14, MERET= 16 * 20, CSUCSOK=(2, 2)(17, 21),*
PRIM *,



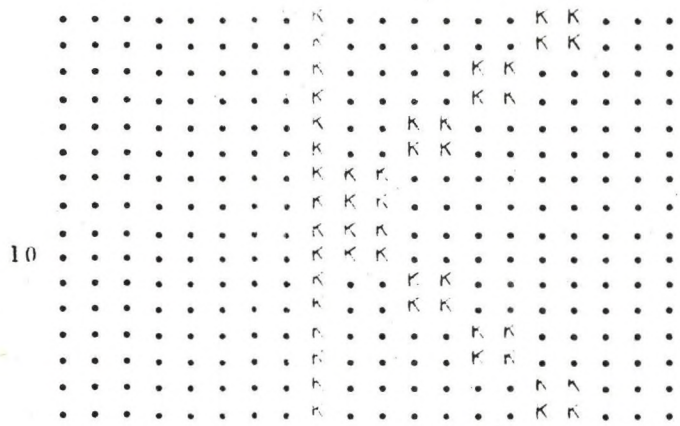
LEPES= 15, MERET= 16 * 20, CSUCSOK=(2, 2)(17, 21),'
'***PRIM ***,



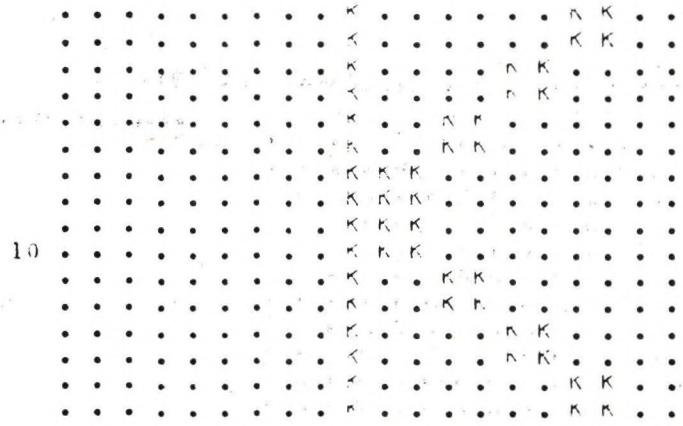
LEPES= 16, MERET= 16 * 20, CSUCSOK=(2, 2)(17, 21),
***PRIM ***,



```
LEPES=  17, MERET= 16 *  20, CSUCSOK=(  2,  2)( 17, 21),*
                                                  **PRIM ***,
```



```
LEPES=  18, MERET= 16 *  20, CSUCSOK=(  2,  2)( 17, 21),*
                                                  **PRIM ***,
```



```
 LEPES=  19, MERET= 15 *  20, CSUCSOK=(  2,  2)( 17, 21),*
                                                          **PRIM ***,
 . . . . . . . . . K K . . . . . . K K .
 . . . . . . . . . K K . . . . . . K K .
 . . . . . . . . . K K . . . . . K K . . .
 . . . . . . . . . K K . . . . . K K . . .
 . . . . . . . . . K K . . K K . . . . . .
 . . . . . . . . . K K . . K K . . . . . .
 . . . . . . . . . K K K K . . . . . . . .
 . . . . . . . . . K K K K . . . . . . . .
 . . . . . . . . . K K K K . . . . . . . .
10 . . . . . . . . . K K K K . . . . . . . .
 . . . . . . . . . K K . . K K . . . . . .
 . . . . . . . . . K K . . K K . . . . . .
 . . . . . . . . . K K . . . . K K . . . .
 . . . . . . . . . K K . . . . K K . . . .
 . . . . . . . . . K K . . . . . K K . .
 . . . . . . . . . K K . . . . . K K . .
 . . . . . . . . . K K . . . . . . K K .
 . . . . . . . . . K K . . . . . . K K .

 LEPES=  20, MERET= 15 *  20, CSUCSOK=(  2,  2)( 17, 21),*
                                                          **PRIM ***,
 . . . . . . . . . K K . . . . . . K K
 . . . . . . . . . K K . . . . . . K K
 . . . . . . . . . K K . . . . . K K . .
 . . . . . . . . . K K . . . . . K K . .
 . . . . . . . . . K K . . . K K . . . .
 . . . . . . . . . K K . . K K . . . . .
 . . . . . . . . . K K K K . . . . . . .
 . . . . . . . . . K K K K . . . . . . .
 . . . . . . . . . K K K K . . . . . . .
10 . . . . . . . . . K K K K . . . . . . .
 . . . . . . . . . K K . . K K . . . . .
 . . . . . . . . . K K . . K K . . . . .
 . . . . . . . . . K K . . . . K K . . .
 . . . . . . . . . K K . . . . K K . . .
 . . . . . . . . . K K . . . . . K K .
 . . . . . . . . . K K . . . . . K K .
 . . . . . . . . . K K . . . . . . K K
 . . . . . . . . . K K . . . . . . K K
      STOP
```

```
NPAGE
MAX 1
DIM 8 6
RDC 1 1 7 5
  1 1 0 1 1
  1 1 0 1 1
  1 1 1 1 0
  1 1 1 0 0
  1 1 1 1 0
  1 1 0 1 1
  1 1 0 1 1
KONV 2
A0 .  A1 K


RDFL 34 2 2
RDFL 34 3 3
RDFL 34 4 4
RDFL 34 5 5
DUAL
RDC 1 1 2 6
 3 4 4 4 4 4
 2 2 2 2 2 5
COPYR 1 1 2 6 0 3
DISP
```

```
LEPES=    0. MERET=  8 *    6, CSUCSOK=(  2,  2)(  9,  7),
                                                  ***DUAL ***.
3 4 4 4 4 4
2 2 2 2 2 5
3 4 4 4 4 4
2 2 2 2 2 5
3 4 4 4 4 4
2 2 2 2 2 5
3 4 4 4 4 4
2 2 2 2 2 5
     PRIM
     DISP
```

```
LEPES=   0, MERET=  8 *   6, CSUCSOK=(  2,  2)(  9,  7),
                                                ***PRIM ***,
K K . K K .
K K . K K .
K K K K . .
K K K . . .
K K K K . .
K K . K K .
K K . K K .
. . . . . .
     DISP 0 1 0
     DO 6
```

```
LEPES=   1, MERET=  8 *   6, CSUCSOK=(  2,  2)(  9,  7),
                                                ***PRIM ***,
K K K . K K
K . K K . .
K K K K K .
K K . . . .
K K K K K .
K . K K . .
. K K . K K
. . . . . .
```

```
LEPES=   2, MERET=  8 *   6, CSUCSOK=(  2,  2)(  9,  7),
                                                ***PRIM ***,
K K K K . K
. K K . . K
K K K K K K
K . . . . .
K K K K K K
. K K . . .
. . K K . K
. . . . . K
```

```
LEPES=   3, MERET=  8 *   6, CSUCSOK=(  2,  2)(  9,  7),
                                                ***PRIM ***,
. K K K K .
K K . . K K
K K K K K K
. . . . . K
. K K K K K
K K . . . K
. . . K K .
. . . . K K
```

```
LEPES=   4, MERET=  8 *   6, CSUCSOK=(  2,  2)(  9,  7),
                                                  ***PRIM ***,
K . K K K K
K . . K K .
. K K K K K
. . . . K K
K . K K K K
K . . . K K
. . . . K K
. . . K K .
```

```
LEPES=   5, MERET=  8 *   6, CSUCSOK=(  2,  2)(  9,  7),
                                                  ***PRIM ***,
K K . K K K
. . K K . K
. . K K K K
. . . K K K
K K . K K K
. . . K K K
. . . . . K
. . K K . K
```

```
LEPES=   6, MERET=  8 *   6, CSUCSOK=(  2,  2)(  9,  7),
                                                  ***PRIM ***,
. K K . K K
. K K . K K
. . . K K K
. . K K K K
. K K . K K
. . K K K K
. . . . . .
. K K . K K
     DUAL
     PDC 1 1 2 1
      3
      5
     COPY2 1 1 2 1 5 3
     DISP
```

```
LEPES=   6. MERET=  8 *   6. CSUCSOK=(  2.  2)(  9,  7),

3 3 3 3 3 3                                    ***DUAL ***.
5 5 5 5 5 5
3 3 3 3 3 3
5 5 5 5 5 5
3 3 3 3 3 3
5 5 5 5 5 5
3 3 3 3 3 3
5 5 5 5 5 5
      PRIM
      DO 1


LEPES=   7. MERET=  8 *   6. CSUCSOK=(  2,  2)(  9,  7),

. K K . K K                                    ***PRIM ***.
. K K . K K
. . K K K K
. . . K K K
. . K K K K
. K K . K K
. K K . K K
. . . . . .
      STOP
```

Krauth Péter–Koch Róbert–Nagy Mihály–Szlankó János

A REFLEX LOGIKAI ALAPÚ LEKÉRDEZŐ RENDSZER

Kivonat: A REFLEX logikai alapu lekérdező és jelentés készitő programcsomag, amely a PDP11/TPA11 tipusu számitógépekre lett kifejlesztve. RMS-11 tipusu fájlokat valamilyen séma szerint relációknak tekint, amelyekre vonatkozólag lehet kérdéseket feltenni. Röviden ismertetjük a nyelvet, helyét a lekérdező nyelvek családjában, eltérését az elsőrendü logika nyelvéhez képest. A fájlokban tárolt relációk mellett a kérdések virtuális relációkra is hivatkozhatnak.

Kulcsszavak: adatbázis, lekérdező nyelv, jelentés készitő, reláció, virtuális reláció, logika.

1. Bevezetés

Elsősorban ügyvitelgépesitési feladatoknál, de folyamatirányitási problémákkal kapcsolatban is egyre növekszik az igény a nagymennyiségü adat kezelését lehetővé tevő és megkönnyitő rendszerek, többek között az adatbázisok iránt is. Az adatbázisokkal szemben általában a következő főbb követelményeket támasztják [1]:

- adatfüggetlenség: Az adatok a felhasználói programoktól függetlenül deklarálhatók, azoktól függetlenül léteznek, az

ábrázolás fizikai strukturája a felhasználói programokból nem látszik, ugyanaz a fizikai reprezentáció esetleg más logikai adatként látszik a különböző felhasználói programokból.

- titkosság: minden felhasználó csak a hatáskörébe tartozó adatokhoz férhet hozzá.

- konzisztencia: az adatok módositásakor ill. ujabb adatok bevitelekor ellenőrizni kell, hogy bizonyos magától értetődő ésszerü vagy éppen a felhasználó által megkövetelt feltételeknek, korlátozásoknak eleget tesznek-e.

- visszaállitás: hálózatkimaradás, vagy más váratlan esemény után, ami esetleg adatok elvesztésével járhat, az adatbázis egy korábbi, nem tulságosan régi állapotának visszaállitására.

- dinamikus adatkezelés: az adatok bevitele, módositása folyamatosan történhet más adatbázisfunkciókkal /lekérdezés, jelentéskészités/ egyidejüleg.

- lekérdezhetőség: az adatbázis pillanatnyi tartalmáról a felhasználó könnyen információt kaphat.

- jelentéskészités: a lekérdezett adatokról formai és dokumentálási szempontok figyelembevételével áttekinthető jelentések összeállitása.

Mindezeket azért soroltuk fel, hogy kijelenthessük: habár a REFLEX mindezeket a követelményeket - különböző mértékben ugyan - megpróbálja kielégiteni, de nem adatbázis kezelő rendszer. A REFLEX lekérdező és reportgeneráló program a PDP11 sorozat gépein, ahol RSX 11M operációs rendszer alatt az RMS 11 programcsomag rendelkezésre áll. Az RMS 11 a fájlok indexelt, relativ és szekvenciális szervezését, hozzáférését biztositja.

A REFLEX /Relational Evaluator for Logical EXpressions/ rendszernél a fő hangsuly a lekérdezési funkció minél teljesebb megvalósitásán van, jóllehet a többi követelményt is biztositani kell egy müködőképes és használható rendszer létrehozásához. A lekérdezés teljességén azt értjük, hogy minden olyan kérdés, ami elsőrendü nyelven megformulázható, az feltehető és megválaszolható legyen a rendszerben is. Ugyanis a matematikai logika szempontjából az adatbázis tekinthető egy elsőrendü nyelv interpretációjának, ahol a lekérdezés azt jelenti, hogy meg kell határozni a kérdésnek megfelelő formula igazságértékét /zárt formula esetén/ ill. az adatbázis azon részét, ahol a formula igaz /nyilt formula esetén/.

Lényeges, hogy az adatbázisok és a logika kapcsolatának fenti felfogásában a logikai formulákat mindig konkrét interpretáción /az adatbázison/ vizsgáljuk. Az olyan tipusu kérdések, hogy egy formula minden interpretáción vagy egy interpretáció osztályon igaz-e, illetőleg melyek azok az interpretációk, ahol a formula igaz inkább a számitástechnika egy másik ágához, a tételbizonyitáshoz tartoznak.

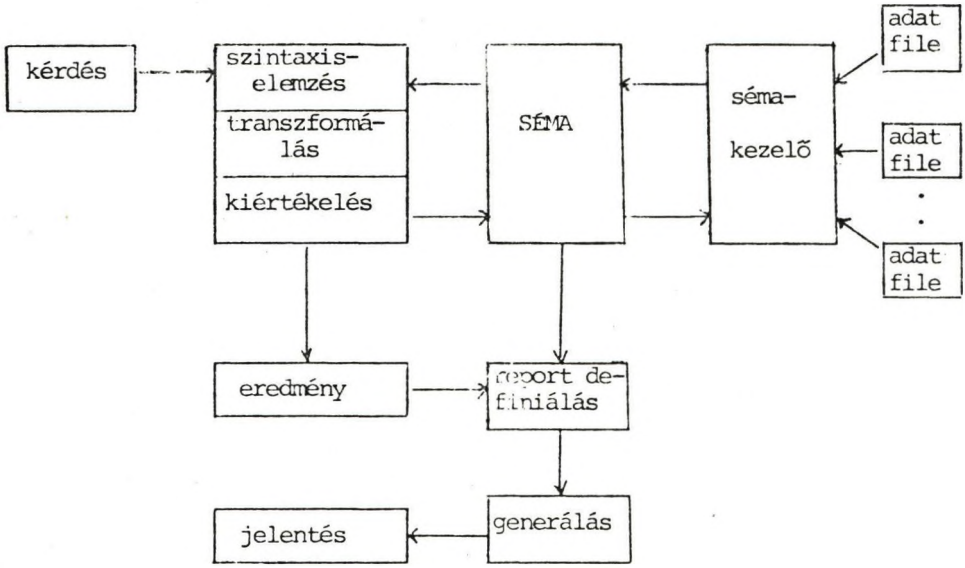
Adatbázis-terminológiában a REFLEX a relációs adatmodellhez illeszkedik. Ennek jellemzői az egyszerü adatszerkezet /két-dimenziós táblázatok, relációk/ és rugalmas lekérdezési lehetőség, ellentétben a hierarchikus és hálós rendszerekkel, ahol bonyolult adatstrukturák definiálhatók, de az adatokhoz csak kötött elérési utakon lehet hozzáférni.

Érdemes még néhány szóval érinteni a REFLEX és a többi magasszintü lekérdezõ nyelv viszonyát, hogy hol helyezkedik el a REFLEX a lekérdezõ nyelvek családjában. A magasszintü lekérdezõ nyelvek egy alapos összehasonlitó osztályozását adja [2]. A REFLEX leginkább a E. Codd vezette kutató csoportban C. Chang által kidolgozott DEDUCE 2 nyelvhez hasonlit. [3] adja a DEDUCE 2 egy informális definicióját. Mivel erõfeszitéseink forgalmazható programtermék elõállitására irányultak, a DEDUCE 2-höz képest bizonyos dolgokat a REFLEX-nél másképpen kellett definiálni, elhagyni, vagy bevezetni. [2] terminológiájában a REFLEX domain változókat és range predikátumokat használ tipusok helyett. A REFLEX pontos definiciója, használati, installációs leirása, performancia jellemzõk [4]-ben találhatók.

2. Szerkezeti felépités

A REFLEX 3 fõ részbõl áll:

- adatdefinició: adatok bevitele és sémadefinició
- adatlekérdezés: felhasználói kérdések elemzése és kiértékelése
- jelentéskészités: a kiértékelések végeredményeinek áttekinthetõ formátumba való összeállitása.



3. Adatdefiníció

A felhasználó adatait speciális adatbeviteli programokkal /DECFORM/ vagy általános szövegszerkesztővel irhatja fájlba. Ezután a sémakezelő program segitségével teremtheti meg a kapcsolatokat az adatok fizikai reprezentációja és logikai hozzáférése között. A sémadefiníció során minden adat fájlnak egy relációnevet feleltetünk meg, és feltüntetjük, hogy a fájl rekordjaiban milyen mezőkre, milyen névvel /attributum/ lehet hivatkozni a lekérdezés során, és hogy mi a tipusa a mezőknek /string, integer, real/. Szerepelniük kell a fájl pontos nevének, a használni kivánt kulcsoknak és a védelmi kódnak is.

A sémakezelő ezen adatok birtokában egy rekordot illeszt a sémába, amelyből a lekérdezéshez szükséges információk kinyerhetők, és az eredeti adat fájlt a megfelelő

alakba konvertálja. A sémakezelővel ezen kivül lehet törölni, módositani vagy kiiratni sémadefiniciókat.

Az eddig emlitett relációk alaprelációk voltak, mivel segitségükkel közvetlenül lehetett adatokra hivatkozni. Azonban a REFLEX megenged virtuális relációkat is, ami azt jelenti, hogy ezeknek a relációknak nem felel meg létező adat fájl, hanem csak az ut van leirva, ahogy az idetartozó adatokat meg lehet határozni. Egy virtuális relációt leiró sémadefinició lényegében egy olyan formula, amilyen a kérdésekben is szerepelhet. Ezen a módon alaprelációk közötti állandósult kapcsolatot lehet deklarálni, és mivel a séma definiálásának a kérdés feltevése előtt kell megtörténnie, a virtuális definicióban szereplő fájlokat elő lehet készíteni, és ezzel a kérdés megválaszolását gyorsitani lehet. A virtuális relációk definiciói nem lehetnek rekurzivak.

Példák:

```
1. APPEND BASE EMPLOYED (NAME/STRING/0:15 ,
                         COMPANY/STRING/15:15,
                         SALARY/INTEGER/30:5)
        IN EMP.DAT (0:45/0:15/15:15/0:15,30:5)
        BY GROUP;
```

E sémadefinició szerint az EMP.DAT nevü adat fájlon 4 darab kulcs van, azaz a fájl rekordjait 4 féle módon lehet elérni. Pl. az első kulcs a 0. pozición kezdődik és 45 hosszuságu és egy szegmensből áll. Az utolsó kulcs két szegmensből áll. A kulcsok száma, elrendeződése és a hozzájuk tartozó indextáblák jelentősen befolyásolják a kiértékelés sebességét.

A fájlhoz csak a létrehozóval egy csoportban /GROUP/ lévők férhetnek hozzá és a kérdésekben EMPLOYED relációként hivatkozhatnak rá. A fájl rekordjaiban 3 mezőt lehet elérni NAME,COMPANY,SALARY attributum névvel.

```
2. APPEND VIRT POORYOUNGS(NAME/X, SEX/Y)
        FROM (UNEMPLOYED(NAME=X) OR
              EMPLOYED(NAME=X,SALARY<'1000'))
              AND PERSON(NAME=X, SEX=Y)
        BY    WORLD;
```

A POORYOUNGS virtuális relációnak két attributuma van /NAME,SEX/, amelyek értékei kielégitik a FROM és BY kulcsszó közti formulát. A relációt mindenki lekérdezheti.

4. Adatlekérdezés

Egy REFLEX-kérdés az eddigiek alapján egy elsőrendü predikátum kalkulussal megfogalmazott formula, melynek relációi a sémában szerepelnek. Gyakorlati szempontok miatt azonban korlátozásokat ill. bővitéseket kellett bevezetni.

Egyrészt függvényekre a kérdésben nem lehet hivatkozni, és minden változónak valamilyen módon fel kell tüntetni egy tartományt, ahonnan az értékeit veszi. A tartománydefinició lehet explicit, ebben az esetben a változóra vonatkozó kvantifikációt közvetlenül követi, vagy pedig implicit, amikoris a formulából határozódik meg egy bizonyos természetes módon.

A megszokott kvantifikációkon tul /ALL,EXIST/ numerikus kvantort is lehet használni, ha a változó értékeinek számára akarunk kikötést tenni /pl.: legfeljebb 20 olyan ember létezik.../. A változókat és konstansokat összehasonlitó jelekkel kötjük a relációk attributumaihoz. Általában egyenlőséggel, de röviditeni is lehet a következőképpen:

EMPLOYED(NAME=X,SALARY=Z) AND Z<'1000' helyett
EMPLOYED(NAME=X,SALARY < '1000') irható.

A relációkat és a változó-összehasonlitásokat AND,OR és NOT logikai müveletekkel lehet összekötni.

A kérdések kiértékelése a következőképpen történik:

1. a kérdésben szereplő virtuális relációk helyére behelyettesitjük a sémadefiniciókat,
2. a már csak alaprelációkat tartalmazó kérdést diszjunktiv normálalakra hozzuk,
3. a szintaxiselemzéssel egyidejüleg felépül a kérdés belső reprezentációja, melynek lényege, hogy a változók kvantifikációjuk sorrendjében vannak feltüntetve és minden változóhoz a formulának csak az a része kapcsolódik, ahol ténylegesen szerepel,
4. a belső reprezentációt a kiértékelő algoritmus interpretálja, és az eredményt a kérdésben megadott fájlba teszi.

Példa:

```
RESULT.DAT(X) EXIST>1000 Y
(UNEMPLOYED(NAME=Y) OR EMPLOYED(NAME=Y,SALARY<'1000'))
AND PERSON(NAME=Y,CITY=X);
```

A RESULT.DAT fájl tartalmazza azokat a városokat, ahol több, mint 1000 ember, vagy munkanélküli, vagy 1000 fontnál kevesebbet keres.

5. Jelentéskészités

Az eredmény fájlokról és az alaprelációknak megfelelő adat fájlokról tetszetős formátumu jelentések generálhatók.

A fájlok rekordjai a kijelölt mezők szerint oszlopokba csoportositva iródnak ki. Az oszlopoknak nevet, magának a jelentésnek cimet lehet adni. Ujabb mezőket is definiálhatunk, amelyeket más mezőkből számitással kapunk. A for-

mátumot tördelni lehet, azaz az ismétlődő értékek nem iródnak ki, és bizonyos időközönként összesitéseket lehet elvégezni /hány különböző érték szerepelt, mi ezek összege stb./. A jelentések fájlba kerülnek, ahonnan igény szerint előállithatóak. A REFLEX reportgenerátora viszonylag egyszerü. Ha a felhasználó nagyobb követelményeket állit, használhatja a DATATRIEVE-11 nevü reportgenerátort is, amely szintén RMS-11 fájlokon dolgozik.

6. Gyakorlati megvalósitás

A REFLEX-rendszer a KFKI-ban került kifejlesztésre PDP 11 kompatibilis számitógépekre. Maga az implementáció OMSI-PASCAL és és MACRO-11 nyelven történt, mig az adat fájlok elérése, kezelése az RMS-11K V1.8 /Record Management Services/ segitségével valósul meg. A program lekérdező része több terminálról, vagy más programokból közvetlenül használható.

További terveink között szerepel, hogy a REFLEX ne csak RMS-11 fájlok egy halmazához legyen lekérdező rendszer, hanem az Intézetben kifejlesztés alatt lévő folyamatirányitási célokra használható adatbázis kezelő szolgáltatásaira épülve is használható legyen.

Abstract: REFLEX /Relational Evaluator for Logical EXpressions/ is a high level, logic-based query language and reportgenerator developed for PDP11/TPA 11 computers. It works on a set of RMS-11 files as relations whose structure is defined by the schema. Queries may refer to stored and virtual

relations. We briefly describe the language, its relation to the other high level query languages and its restriction to the first order predicate calculus.

Irodalom

[1] The ANSI/X3/SPARC DBMS Fromework Report of the Study Group on Database Management Systems. Information Systems Vol. 3 pp. 173-191

[2] A Pirotte, High Level Data Base Query Languages, In Logic and Databases, Plenum Press, 1978

[3] C.L. Chang, DEDUCE 2: Further Investigations of Deduction in Relational Databases, In Logic and Databases, Plenum Press, 1978

[4] REFLEX User's Guide, KFKI, 1981.

Krauth Péter, Koch Róbert, Nagy Mihály, Szlankó János
Központi Fizikai Kutató Intézet
1525 Budapest P.O.B. 49.

Laborczi Zoltán

AZ ADA RENDSZERTERV ÖSSZEFOGLALÓ ÁTTEKINTÉSE

Ebben a cikkben nagyvonalú, összefoglaló áttekintést adunk az SZKFT együttmüködés keretében kidolgozás alatt álló ADA rendszer tervéről. Először az ADA nyelv definiciójának pontatlanságából adódó nehézségekkel foglalkozunk, majd az ADA rendszer felépitését ismertetjük és bemutatjuk a rendszert alkotó komponenseket.

Kulcsszavak: ADA nyelv, nyelvi rendszer, forditóprogram

1. Az ADA nyelvről

Az ADA programozási nyelv [1] egyrészt általános célú, másrészt olyan eszközökkel rendelkezik, melyek a nagyméretü programok moduláris elkészitését támogatják, továbbá lehetővé teszik a folyamatszervezésü párhuzamos programozást. Figyelemreméltóak azok a tulajdonságok is, amelyekkel elérhető, hogy az ADA nyelven megirt programok a végrehajtó géptől, például annak szóhosszától függetlenek maradjanak. Ezáltal alkalmas keresztforditók segitségével alacsony átviteli költséggel lehet keresztfejlesztéseket végezni.
Az ADA kézikönyvnek programozói szemmel történő tanulmányozása során megállapithatjuk, hogy a fogalmak és az őket tükröző szintaktikus szerkezetek világosak, és alkalmasak arra, hogy egy-egy programrész kivánt hatását jól olvasható formába öntsük. Talán túl sok a kivétel, speciális eset, de a programozás során a gyanús konstrukciókat el lehet kerülni.

A forditóprogram készitői számára azonban nemcsak ezek a kivételek kikerülhetetlenek, hanem a fogalmak között lehetséges rejtett összefüggéseket is fel kell tárni.

A kézikönyv gondos tanulmányozása alapján nem mindent lehet egyértelmüen interpretálni. Vannak homályos, sőt egymásnak ellentmondó részek is. A tisztázás érdekében egy másik dokumentum is használható, melynek cime:
Utmutató ADA forditóprogramok hitelesitéséhez [2].
Ez a forditóprogram készitők számára is tartalmaz iránymutató magyarázatokat, amelyek a felmerülő kérdések egy részét megválaszolják.
A további problémákra is megoldást kellett találnunk.
A legérdekesebb és válaszra leginkább váró kérdéseket a függelékben irjuk le, megadva egyúttal az általunk valószinünek tartott megoldást is. Reméljük, hogy ezek összhangban vannak az ADA nyelv szellemével és az ADA szerzőinek szándékával, s nem fognak érvénytelenné válni az ADA kézikönyv későbbi, precizebb változatának megjelenésével sem.

2. Az ADA rendszerterv áttekintése

A cél egy könyvtárkezelőből, forditóprogramból és összeszerkesztőből álló hordozható rendszer kidolgozása, melynek segitségével különböző célgépekre lehet programot kifejleszteni és maga a fejlesztőrendszer is átvihető különböző tárgygépekre. Az alapváltozat a fejlesztőgépen áll össze és onnan kerül átvitelre a megfelelő tárgygépekre.
A forditóprogram három menetre bomlik. Az alábbiakban az egyes menetek és az egyéb rendszerkomponensek tevékenységének rövid ismertetését adjuk.

2.1 Első menet: lexikai és szintaktikai analizis

Az első menet feladata a lexikai és a szintaktikai elemzés elvégzése. Eredményképpen a program fastruktúrájú belső alakja áll elő /absztrakt szintaxisfa/, melynek pontos formáját LDM-beli [8] tartománydefiníciókkal adjuk meg. A szintaktikus elemzésről, a használt hibakezelési módszerről és a fa felépitésével részletesebben [3] foglalkozik.

A lexikai elemző feladatai:

- a forrásszöveget szimbólumok sorozatára bontja s az egyes szimbólumok kódját átadja a szintaktikai elemzőnek,
- forráslista készitése,
- a lexikai elemzéssel kapcsolatos pragmák kezelése.

2.2 Második menet: szemantikus analizis

A második menet fő feladata a környezetfüggetlen feltételek ellenőrzése, valamint az absztrakt szintaxisfa attributumokkal való kitöltése, amit összefoglalóan szemantikus analizisnek nevezünk. Ezen belül az egyik legfontosabb részfeladat az azonositás: minden felhasznált dologról meg kell találni definiciójának helyét (a deklarációját) . Egyéb, önálló részfeladatok:

- a generikus egységek kezelése,
- a könyvtári kapcsolatok felvétele és ellenőrzése.

Megjegyezzük még, hogy a szemantikus ellenőrzés nem végezhető el gépfüggetlenül, mert szükség van a célgéptől függő információkra is. (Például az a+b összeadás a célgépen megvalósitott ábrázolástól függően vonatkozhat
SHORT INTEGER-re, INTEGER-re vagy LONG INTEGER-re.)
A célgépfüggő részeket külön modulban gyüjtjük össze, ezáltal a forditóprogram első és második menete a szóbanforgó modul kivételével célgépfüggetlen lehet.
A második menet egyes kérdéseivel [4] foglalkozik.

2.3 Harmadik menet: kódgenerálás

A második mmenet által előállitott fa még "aránylag célgépfüggetlen", ezért nem alkalmas arra, hogy közvetlenül a kódgenerálás alapjául szolgáljon.
A hiányzó attributumokat a harmadik meneten belül egy önálló fázis helyezi el a fában. Egy későbbi, a rendszer hangolásával foglalkozó fejlesztési lépés során ez a fázis beolvadhat a második menetbe.

A kódgenerálás folyamatának vezérlését az absztrakt kódgenerátor irja le a denotációs szemantika jelölésmódjához hasonló egyenletek formájában. Az egyes nyelvi konstrukciók szemantikáját ezek az egyenletek hipotetikus, ADA orientált A-gép utasitásainak (A-utasitásoknak) a segitségével adják meg.
Az A gépi utasitások jelentését ADA-hoz hasonló formalizmussal adjuk meg. Ez az egyes célgépeken a futtatórendszer terveként is felhasználható. Itt önálló résztémaként megemlitjük a tárgygazdálkodással és a hulladékgyüjtéssel foglalkozó terveket [7].

Az A-utasitások megvalósitása célgépfüggő módon történik. A célgépek sajátosságaitól függően a továbbforditás vagy az interpretálás mellett lehet dönteni. A kódot mindkét esetben az absztrakt kódgenerátorhoz algoritmikus interface-n keresztül kapcsolódó konkrét kódgenerátor állitja elő. Ezzel a felbontással elérhető, hogy a kódgenerátor célgépfüggő részei a konkrét kódgenerátorba lokalizálódjanak. Részletesen lásd [5] .

2.4 Könyvtárkezelő

Az ADA nyelv megvalósitása nemcsak a hagyományos értelemben vett forditóprogram kifejlesztését jelenti. Szükség van ezenkivül egy minimális programfejlesztési környezetre is, amely első közelitésben csak könyvtárkezelőt tartalmaz.
A könyvtár az ADA forditási egységek tárolására alkalmas eszköz. A forditási egységek hierarchikus rendben helyezkednek el. A könyvtárkezelő információt tárol az egységek egymáshoz viszonyuló kapcsolatairól, valamint magukról az egységekről.
Az ADA által előirt különforditási rendszert olymódon valósitjuk meg, hogy a forditás után az adott egységről a könyvtárban elhelyezünk olyan információkat, amelyeket később, más egységek forditásakor felhasználhatunk. Módot ad a könyvtárkezelő arra is, hogy ha egy egység megváltozik, akkor a változás következményeképpen érvényüket vesztett forditások megismételhetők legyenek.

A könyvtárkezeléssel részletesebben [6] foglalkozik.

2.5 Összeszerkesztő program

A fordítási egységekről a könyvtár leforditott kódot is tárol. A futtatható célprogram általában több fordítási egység kódjából áll össze. Ennek az összeállitásnak a feladata hárul az összeszerkesztő programra. A gépen rendelkezésre álló szerkesztő (linkage editor) nem használható, mert egyrészt nem tudja az ADA könyvtárt kezelni, másrészt funkciói sem elegendőek az ADA kódrészek összefüzésére.

2.6 Kiegészitő komponensek

A forditóprogram tevékenységének nagy része fákon végzett manipulációkkal van megvalósitva. Ezek közös eszközeként egy fakezelő programcsomag kerül kidolgozásra, amely a fában végzett müveleteket realizálja CDL2-ben.
A forditóprogram memóriaigénye dinamikus, mert függ a leforditott program méretétől, s kisgépeken nem is elégithető ki a központi tárolóból. A memória dinamikus kiosztását és háttértárolóra történő lapozását külön memóriakezelő modul valósitja meg.

Köszönetnyilvánitás

Ezuton mondok köszönetet mindazoknak a munkatársaknak, akik az ADA projectben való részvételükkel közvetve vagy közvetlenül hozzájárultak ahhoz, hogy ez a dokumentum megszülethessék.

FÜGGELÉK: Az ADA nyelvvel kapcsolatos problémák

Csak azokat a kérdéseket soroljuk fel, amelyek nem abból fakadnak, hogy az ADA kézikönyv nem egészen pontos megfogalmazásait ahol lehet, félreértjük, hanem valódi problémákat takarnak, s a rájuk adott válaszok hozzájárulhatnak az ADA nyelv hibamentesebb definiciójának megszületéséhez.

1. A nyelv jelenlegi formájában megengedi az alábbi konstansdeklarációt:

```
SPACES: constant STRING:= SPACES'RANGE= ' ' ;
```

ahol a baloldal hosszát a jobboldalnak kellene definiálnia, de a jobboldal ehelyett visszahivatkozik a baloldali indexintervallumára. Egyszerü feltétellel meg lehetne tiltani az ilyen logikai ciklust.

2. A renaming deklaráció és az allokátorokkal megvalósitott dinamikus társzervezés olyan programok irását teszik lehetővé, amelyek futtatása során levegőben lógó hivatkozások jöhetnek létre. Ilyen az alábbi példa is:

```
declare
    type R is array (INTEGER range<>)  of INTEGER;
    type A is access R;
    ARRAY NAME:A:=new R(1..10);
    ELEM 10:INTEGER renames ARRAY NAME(10);
begin
    ARRAY NAME:=null;  --(1)
    ...
end;
```

Az (1) -es programponton az ARRAY NAME által mutatott tömb már nem létezik, mivel nincsen rá érvényes hivatkozás.
A 10-es indexü elem ennek ellenére az ELEM 10-en keresztül elérhető.
A levegőben lógó hivatkozások konstruálhatósága mindenképpen ellentmond a nyelv szellemének,ezért szükségesnek látszik

egy olyan szabály bevezetése, mely allokátorral létrehozott objektum komponensére megtiltja az átnevezést. Alternativ, de a hulladékgyüjtőtől többet követelő megoldás az lenne, hogy az objektumok élettartama a komponensekre érvényes hivatkozástól is függjön.

3. Nem világos, hogy egy egység lokális taskjainak aktivációja az egység aktivációjához vagy a törzsének futtatásához tartozik. A válasznak számos fontos szemantikai következménye van, például hogy mikor tekinthető allokátor futtatása befejezettnek vagy hogyan kell kivételek továbbgyürüzését értelmezni. Ugy véljük az aktivációk a törzs futtatásához tartoznak.

4. A rekord reprezentációspecifikáció a kézikönyvben megadott példa szerint a rekord egyik komponensére a komponens tipusa által meghatározott hossznál rövidebb hossz is megadható, ha ez a rövidebb hossz elég nagy ahhoz, hogy a komponens által felvehető értékeket tárolja. Ez ellentmond annak az elvnek, hogy az adatok ábrázolása azonos tipusú értékek esetén egyöntetü, ami a kézikönyv más helyén meg is van fogalmazva. Az általános elvet kellene iránymutatónak tartani, azzal hogy az emlitett példa kivánt hatását egyéb ADA eszközökkel is el lehet érni olymódon, hogy a komponens tipusát nem altipusként, hanem eltérő reprezentációval rendelkező derivált tipusként adjuk meg.

Abstract

An overview is given on the design specification of the ADA system being developed jointly by tive Hungarian research institutes. After dealing with the difficulties caused by the inexactness of the definition of ADA, information is given on the structure of the system and on the constituent parts.

IRODALOM:

[1] Reference Manual for the ADA Programming Language, Proposed Standard Document United States Department of Defense, 198o. július.

[2] ADA Compiler Validation Implementers' Guide Softtech, Inc. 198o. október.

[3] Laborczi Zoltán-Szeredi Péter: Szintaktikus elemzés az ADA forditóban. Programozási Rendszerek'81.

[4] Bach Iván: ADA programok szemantikus analizisének egyes kérdései. Programozási Rendszerek'81.

[5] Farkas Ernő-Groszmann Gusztáv: Az ADA kódgenerátor áttekintése. Programozási Rendszerek'81.

[6] Laborczi Zoltán-Soós Klára: Az ADA könyvtár megvalósitása. Programozási Rendszerek'81.

[7] Székely Judit-dr. Szőke Péter: Láncolt memóriájú ADA gép, szemétgyüjtés. Programozási Rendszerek'81.

[8] Szeredi-Balogh-Farkas-Sántáné-Tóth:
Az LDM tervezési nyelv specifikációja.
SZKI-NIMIGÜSZI tanulmány a SZÁMKI részére. 1979.

Szerző: Laborczi Zoltán
Számitógépalakalmazási Kutató Intézet
1536 Budapest, Pf.227.

Laborczi Zoltán–Soós Klára

AZ ADA KÖNYVTÁR MEGVALÓSÍTÁSA

Az ADA programozási nyelv lehetőségét ad a programok részekre bontására, és ennek megfelelően ezeknek a részeknek külön fordítására. A külön forditott részek közti kapcsolatot a fordítóprogram ellenőrzi. A nyelv definiciójában megadott lehetőségek megvalósitását az ADA forditóprogram az ADA könyvtár segitségével tudja elvégezni.

Az előadás az ADA programozási nyelv által a program részekre bontására nyújtott lehetőségeket és a részekre bontott program forditására előirt követelményeket ismerteti röviden, nem feltételezve az ADA nyelv ismeretét. Majd az ezen követelmények betartását segitő, ellenőrző, a magyar ADA projektben megvalósitott ADALIB könyvtárkezelő rendszerről és az általa nyújtott szolgáltatásokról ad rövid áttekintést.

Kulcsszavak: ADA nyelv, külön forditás, relációk programkönyvtárban.

1. Különforditás és könyvtárkezelés az ADA nyelvben

Programok részekre bontására nyújtott lehetőségek pontos leirását [1] tartalmazza, az elkövetkezőkben az ADA nyelvtől függetlenül adunk rövid áttekintést.

Az ADA programok forditási környezete az ADA könyvtár. A forditónak nem komplett programokat, hanem forditási egységeket kell forditásra átadni. A forditási egységekről a forditás során a forditóprogram a könyvtárba különféle információkat ir, elnagyolt fogalmazásban azt mondhatjuk, hogy a forditási egységet elhelyezi a könyvtárban.

A fordítási egységek alapján a könyvtárban erdő /azaz egy vagy több fa/ épül fel. Egy fa csúcspontjai egy, esetleg két fordítási egységnek felelnek meg. A gyökérpontban szereplő egységeket könyvtári egységeknek, a fák belső pontjaiban elhelyezkedő egységeket alegységeknek nevezzük.

Könyvtári egységként szerepelhet
- alprogram*, ami vagy a főprogramot, mint eljárást valósitja meg, vagy pedig olyan alprogram, amelyet a többi egység felhasználhat. A könyvtári egységként szereplő alprogramot lehet egyetlen fordítási egységként megadni, vagy pedig különválaszthatjuk a specifikációt, a törzs fordításának későbbre halasztásával.
- package, ami a többi egység által használható, általában logikailag együvé tartozó eszközöket /tipusokat, eljárásokat, konstansokat, esetleg változókat, vagy akár package-eket és taskokat/ ad meg. A könyvtári egységként szereplő package specifikációját és törzsét két külön fordítási egységként kell megadni.

Az alegység tulajdonképpen egy másik egység legkülső deklarációs szintjéről kihasitott olyan egység, amelyet külön fordítottunk le. Valójában csak a törzset lehet eredeti helyéről eltávolitani olymódon, hogy egy törzscsonkkal helyettesitjük, ezzel jelezve, hogy külön forditott törzsről van szó; a specifikációt meg kell hagyni eredeti formájában.

Alegységként
- alprogramtörzset
- package-törzset
- task-törzset lehet megadni.

---

* Az alprogram az eljárás és a függvény közös neve.

Amikor alegységet adunk át forditásra, /a context specification segitségével/ pontosan meg kell adni, hogy melyik már meglévő forditási egységben szerepel a törzscsonk, azaz melyik az alegység atyaegysége, ilymódon a fordító a könyvtárkezelő segitségével kialakithatja, illetve tovább bővitheti a forditási egységek erdőszerkezetét.

Érdemes megjegyezni, hogy a fentiekben vázolt különforditási séma mind az alulról felfelé, mind a fölülről lefelé történő programfejlesztést támogatja. Az "alulról felfelé" megközelitésben könyvtári egységeket definiálunk, hogy programozásunk alapfogalomkészletét, az ADA nyelv szintjét megemeljük fejlettebb, vagy speciális célú fogalomrendszer kialakitásával és megvalósitásával. Felülről lefelé haladva viszont a feladat egy-egy részproblémája megoldását /esetleg később megirandó/ alegységekre bizzuk, s segitségükkel "könnyen" meg tudjuk oldani az eredeti feladatot is. A két megközelités kevert alkalmazása is elképzelhető.

A forditási egységek közti felület ellenőrzése érdekében forditásuk sorrendjére bizonyos megkötéseket tesz a rendszer, mely lényegében "az előbb specifikálni /deklarálni/ és utána használni egy objektumot" szabály betartását követeli meg a programozótól, ha a forditási egység más egységben deklarált objektumra hivatkozik. A nyelv a következő szabályokat irja elő:

- könyvtári egység törzsét csak a specifikációjának forditása után lehet forditani;
- alegységet csak az atya egysége után lehet forditani;
- forditási egység forditása csak akkor végezhető el, ha a láthatósági listáján* előforduló könyvtári egységek specifikációjának forditása már megtörtént.

---

* A with-clause-ban

Ha egy objektumra más fordítási egységben is történik hivatkozás, és az objektum deklarációja /specifikációja/ megváltozik, akkor a rendszer a helyes használat ismételt ellenőrzését, azaz a fordítási egység újrafordítását követeli meg. Ezért, ha egy korábban már létezett fordítási egységet újrafordítunk, a korábbi változat érvényét veszti. Igy az összes ráhivatkozó, korábban leforditott egység forditása is érvényét veszti, és ezért ezeket újra kell forditani. Az újrafordítandó egységek:

- az egység összes alegysége,
- könyvtári egység specifikációja esetén a törzs, valamint az egységet használó más egységek,
- a fentiek szerint elért egységekből a fenti szabályok rekurziv alkalmazásával elérhető egységek.

Néhány speciális esetben a törzs megváltoztatása a specifikáció, ill. a ráhivatkozók újrafordítását is jelenti /pl. INLINE és generic esetén/.

Az ilyen újrafordítás nem jelenti szükségképpen azt a tevékenységet, amelyet teljes fordításkor el kell végezni. A szintaktikus elemzés utáni állapotra a környezet még semmi hatással nem volt, ezért ha a könyvtár tárolja ezt az állapotot, az újrafordítás ebből indulhat ki. Kevésbé igényes implementáció azonban a forrásszövegből is kiindulhat ilyenkor.

A programjavitás olymódon történhet tehát, hogy egy vagy több fordítási egység forrásszövegét megváltoztatjuk és lefordítjuk őket. A könyvtárkezelőtől ezután megkaphatjuk azon egységek listáját, melyek korábbi fordítása érvényét vesztette; ezeket újrafordítjuk.

Az ADA nyelv fentiekben leirt lehetőségeinek megvalósitása

érdekében az ADA könyvtár a forditási egységek kódján kivül tartalmaz olyan információkat az egységekről, melyekre részben a ráhivatkozó egységek forditásakor, részben a forditási egység állapotának vizsgálatakor és a program többi forditási egységével való kapcsolatának karbantartásához van szükség.

2. A kötetek kezelése

Az ADALIB könyvtárkezelőnek két szintje van.
Az alsó szint az ADA nyelvtől függetlenül kezelhető, a felső szint az ADA nyelv definiciójában előirt tulajdonságok megvalósitását szolgálja. Ebben a pontban az alsó szintet vázoljuk fel.

A könyvtárt alkotó elemi egység a kötet. Ezek a könyvtáron belül hierarchikus rendben helyezkednek el. Minden kötetnek van egy egyedi, legfeljebb egy gépi szót kitöltő belső azonositója, mely lehetővé teszi, hogy a kötetközi hivatkozások tömörek legyenek, s a programok is könnyen nyilvántarthassanak.ilyen hivatkozásokat. Ez az azonositó az installációban előforduló összes kötetre nézve egyedi.

A kötetek fastruktúrában helyezkednek el. A levélként szereplő csúcspontot könyvnek, a belső csúcspontot katalógusnak nevezzük. A katalógus nem tartalmaz más információt, mint ami a kötetek fájában a környezetet jelenti, azaz a katalógusból elérhetők a leszármazott kötetek, és a közvetlen ős /atya/ is. Könyvből kiindulva csak az atyához juthatunk el. A könyvben tárolt információk a kötetek szerkezete szempontjából érdektelenek.

A katalógusból a leszármazott kötetet karaktersorozatok, éspedig ADA azonositók választják ki.

A könyvtár az ilyen szintü kezeléshez az ADA forditóprogram készitőinek a következő jellegü tevékenységek elvégzéséhez nyújt segitséget:

- katalógus, könyv,létrehozása, törlése;
- kötet "bekatalogizálása" /azaz a kötet az adott katalógus közvetlen leszármazottja lesz/ és törlése katalógusból;
- kötet közvetlen ősének és leszármazottjainak /szelektornév és sorszám/ meghatározása;
- egy könyv tartalmának kitöltése, olvasása /egy könyv szekvenciális file-ként kezelhető/.

Egyes szelektorokat _ jellel kezdünk, annak ellenére, hogy ADA azonositó igy nem kezdődhet. Az ilyen szelektorok segitségével a forditóprogram tárol információkat a további forditáshoz.

A szelektorok ismerete nélkül is eljuthatunk az első, második ... stb. leszármazotthoz, de hogy pontosan melyik az első, második, ..., az nincs definiálva.

3. Forditási egységek kezelése

Ebben a pontban áttekinthetjük, hogyan valósitható meg az ADA nyelv által előirt könyvtárstruktúra az előző pontban ismertetett eszközök segitségével.

3.1. A forditási egységekhez tartozó könyvtárstruktúra

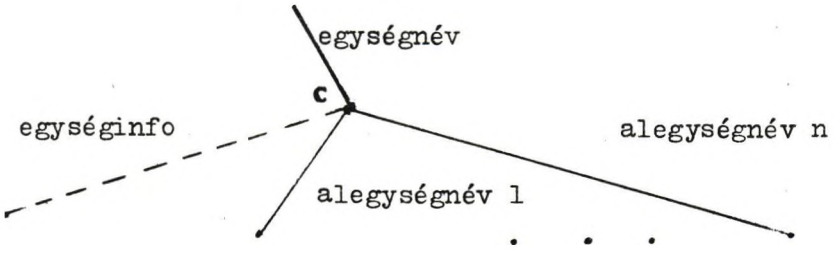
Az egy gépen létező könyvtárakat összefogjuk egy katalógus alá. Ezt a katalógust főkönyvtárnak nevezzük. A főkönyvtár az ADA rendszer installálásakor jön létre; az ADA fordító, illetve a könyvtárkezelő szolgáltatások segitségével nem lehet törölni. A főkönyvtár alá bekatalogizált fától várjuk el azt, hogy az ADA könyvtárfogalmának megfeleljen, illetve azt megvalósitsa.

A főkönyvtár alá /közvetlenül, vagy több áttétellel/ bekatalogizált kötetekre kétféleképpen lehet hivatkozni:

a/ az egyedi kötetazonositóval,
b/ a főkönyvtárból kiinduló szelektorsorozattal.

Az egyes ADA könyvtárakat ábrázoló katalógusok közvetlenül a főkönyvtár alatt helyezkednek el. A könyvtár neve nem más, mint a katalógusnak a b/ pont szerinti azonositója, azaz a főkönyvtárból induló egy elemü szelektor.

A forditási egységek könyvtárbeli állapotát az alábbi rajzzal illusztráljuk:



Az egységet a c-vel jelölt katalógus reprezentálja, amelyből az alegységek neveivel mint szelektorokkal érhetjük el az alegységeket reprezentáló katalógusokat. A szaggatott vonallal jelölt egységinfo komponens valójában több komponenst takar. Ezek a komponensek együttesen adják mindazokat az információkat, amiket az adott egységről tárolni kell. Ezek a komponensek valamennyien könyvek.

Az egységinfo részben lévő szelektorok _id alakúak, ezáltal megkülönböztethetők az alegységeket kiválasztó szelektoroktól. Az alegységeknél tárolt információk megvannak könyvtári egységek esetén is, ezért először az alegységekhez tartozó szelektorneveket adjuk meg az általuk kiválasztott komponens tartalmának leirásával:

_descr: a forditási egység leirója. A forditási egységre vonatkozóan tárol vegyes rendeltetésü információkat. Elsősorban a felhasználónak esetleg szükséges információkat tartalmazza. A komponens tartalmának kitöltése, módositása a könyvtárkezelőn keresztül történik. Részletesebben később ismertetjük.

_tree: a forditási egység elemzési fája. Leggyakrabban az első menet által előállitott formában /CF alak/ tárolódik. Ha későbbi állapotok /attributumos fa/ megőrzésére is szükség van, akkor ebből a CF alak visszaállitható. Erre azért van szükség, mert a CF alak szolgál az újrafordités kiindulásául.

_lext: az illető egység lexikai táblázata. A lexikai elemző épiti fel az első menet során, s a többi komponensek gyakran hivatkoznak rá.

_code: az adott egység kódja, amely a szerkesztőprogram /linkage editor/ inputjául szolgál. Bizonyos tipusú egységeknél ez hiányozhat.

_inf: ha az adott egység tartalmaz törzscsonkot, akkor ebben a komponensben őrizzük meg azokat az információkat, amelyek a törzscsonkhoz tartozó egységek majdani forditásakor kell felhasználni.

A könyvtári egységek nemcsak ilyen komponensekkel rendelkeznek, hanem olyan hasonló célú komponensekkel is, amelyek neve megkülönböztetésül a specifikációra utaló _sp jellel kezdődik.

Részletesen:

_spdescr: a könyvtári egység specifikációjának leirója. A könyvtári egységek specifikációja /az alegységek-

kel ellentétben/ általában külön fordítási egység. Felépitése, kezelése a _descr komponenshez hasonló.

_sptree: a specifikáció elemzési fája

_splext: a specifikációhoz tartozó lexikai táblázat

_spcode: a specifikációból generált kód

_spinf: a specifikáció kivonata azok számára, akik az illető egységet with mondatban megemlitve használni akarják.

3.2. Egységleiró információk a könyvtárban

A fordítási egységről két ponton tárolunk leiró jellegü információkat: a _descr / _spdescr/ komponensben és a közvetlenül a könyvtárkatalógus alá tartozó _RELS /Relations/ nevü könyvben. Először ez utóbbi tartalmával foglalkozunk.

A _RELS a könyvtárhoz tartozó összes fordítási egység állapotjelzését és az egységek egymás közötti relációit tartalmazza.

Egy fordítási egység állapota lehet:

COMPLETE az egység fordítása jó, teljes,

RECOMPILE újraforditandó, mert a külső környezet megváltozott,

INCOMPLETE az utolsó fordítás során csak a generikus egyediesitések nem sikerültek; a külső környezetben nem történt változás, de újraforditandó,

WRONG a fordítás nem sikerült, mert vagy a külső környezet nem megfelelő, vagy a forrásszöveg módositandó.

A fordítási egységek közti függőségi viszonyokat reláció formájában őrzi. Ez a reláció lehet

egységl WITH egység2
mely esetben az egységl láthatósági listájában szerepel egység2;

egységl INSTANTIATION egység2
mely esetben az egységl-ben az egység2-ben deklarált generikus objektum egyediesitése előfordul;

egységl INLINE egység2
mely esetben az egységl-ben az egység2-ben INLINE-nak definiált subprogram hivása előfordul;

egységl ELABORATE egység2
mely esetben az egység2 elaborálását az egységl elaborálása[1] előtt kell elvégezni.

A függőségi viszonyok között csak a közvetlen relációk szerepelnek, hisz megfelelő algoritmus segitségével a közvetett relációban álló egységek meghatározhatók. A függőségi viszonban csak COMPLETE és esetleg INCOMPLETE állapotú forditási egységek szerepelhetnek, azaz csak élő, tényleges kapcsolatok, mert ha egy egységet újra kell forditani, ez módositást és igy a hivatkozott egységek megváltozását is jelentheti.

A különböző relációk /melyek természetesen még bővithetők/ mutatják, hogy nemcsak a fordító állapithat meg relációkat és használhatja őket, hanem a rendszer egyéb elemei is /pl. linkage editor/.

A függőségi viszonyok felviteléről a könyvtárkezelőn keresztül a különböző programok /fordító, szerkesztő, stb./ gondoskodnak, törlésüket viszont bizonyos esetekben a könyvtárkezelő automatikusan végzi.

---

1 Az elaboráció az ADA definiciójában használt müszó, és a deklarációk kidolgozását, végrehajtását jelenti.

Az automatikus törlés a következő formákban történhet:

- egység összes függőségi viszonyát lehet törölni;
- egységtől közvetlenül és közvetve függő összes fordítási egység állapotát RECOMPLE-ra lehet állitani, ilyenkor természetesen a megfelelő függőségi viszonyok törlődnek. A közvetlenül függő egységek a relációk baloldalán szerepelnek, a közvetve függő egységek az 1. pontban az újrafordítandó egységeknél leirt módon határozódnak meg;
- fordítási egység könyvtárból való teljes törlésekor, nemcsak a fordítási egység és összes leszármazottja /teljes részfa/ törlődik, hanem a fent leirt módon a függő egységek állapota is megváltozik.

A fordítási egységhez tartozó _descr, ill. _spdescr, komponensek elemei a következők:

- az utolsó fordítás dátuma,
- a fordító verziója, mely az egységet definiálta, ill. újradefiniálta,
- az egység aktuális forrásszövegének helye;
- a komponens tipusa
  /vö. fordítási egységek, 1. pont/
- érvényes-e rá INLINE pragma, vagy közvetlenül, vagy egy részére, vagy egy bővebb INLINE eljárás részeként;
- generikus egység törzse-e vagy /nem feltétlenül közvetlen/ alegysége-e ;
- komment;
- az egység forrásszövegében előforduló csonka törzsek nevei;
- az egység láthatósági listájában szereplő könyvtári egységek nevei.

Az egységek állapota, függőségi viszonyok és a _descr, ill. az _spdescr komponens segitségével a könyvtárkezelő a felhasználót tájékoztatja programja állapotáról és szerkezetéről /fordítási egység szintjén/. Ezeket a lehetőségeket a következő pontban ismertetjük.

4. Operátori felület

A forditási egységek a fordítón keresztül kerülnek be a megfelelő könyvtárba, és igy erre közvetlenül nincs lehetősége /és szüksége/ a felhasználónak. A könyvtárkezelő operátori felülete lehetőséget ad

- a főkönyvtár különböző tipusú elemeinek /könyvtár, könyvtári egység, alegység/ kilistázására meghatározott tulajdonságaik alapján, illetve meghatározott tulajdonságaikkal együtt;
- egység /alegység/ és a hozzákapcsolódó egységek /alegységek/ állapotának kiiratására, lekérdezésére;
- főkönyvtári elem törlésére.

Az első csoportba tartozó tevékenységek lényegében a szokásos könyvtár-listázási funkciókat fedik le. Ezért ezekről nem szólunk részletesebben.
Törléskor nemcsak a könyvtári elem teljes egészében /részfa/ szünik meg, hanem a függőségi viszonyok is, és egyes elemek állapota megváltozik. /vö. függőségi viszonyok automatikus törlési lehetőségei - 3. pont/.

A szokásos könyvtárkezelői szolgáltatásokon kivül
1/ lekérdezhető
- egység /alegység/ állapota COMPLETE-e;
- program állapota COMPLETE-e /az adott egység és minden aktuális alegységének állapota COMPLETE-e/.

2/ kilistáztatható
- program komponensek neve állapotukkal együtt, vagy anélkül.
  A program komponensei az adott egységhez tartozó összes egység, azaz
  - ha alegység, az atya egysége,
  - összes alegysége,

- láthatósági listáján szereplő egységek,
- a fentiek során elért egységekből a fenti szabály rekurziv alkalmazásával elérhető egységek,

- egység forditásához szükséges egységek neve, esetleg az állapotukkal együtt.
  A szükséges egységek:
  - ha alegység, az atya egysége,
  - láthatósági listáján szereplő egységek,
  - a fentiek során a fenti szabályok rekurziv alkalmazásával elérhető egységek;
- egységtől /alegységtől/ függő egységek neve.
  A függő egységek az esetleges változtatás hatására újraforditandó egységek /lásd 1. pont/.

5. További lehetőségek

Az ADALIB könyvtárkezelő alapja lehet egy, a programfejlesztést minden fázisban segitő könyvtárkezelő rendszernek. Az ADALIB fejlesztése elsősorban két irányban látszik célszerünek. A szokásos szubrutinkönyvtárakhoz hasonlóan olyan ADA rendszerkönyvtár létrehozása, mely ún. befagyasztott forditási egységeket tartalmaz. Ezekre a forditási egységekre bármely vagy meghatározott könyvtárakban előforduló egységek hivatkozhatnak, és megváltoztatásukra csak igen indokolt esetben és a felhasználó előzetes figyelmeztetése után lenne lehetőség.

A másik irány pedig a forrásszöveg verziók tárolásának, karbantartásának lehetővé tételét jelenti. Ez a könyvtári egységek és alegységeik közti szoros kapcsolat miatt a szokásos, verziókat is kezelő könyvtáraknál nehezebben valósitható meg.

Már a jelenlegi könyvtár használatát is lényegesen gyorsitaná egy dedikált szerkesztő, mellyel a forrásszövegbeli változtatást is az ADALIB könyvtárban lehet elvégezni.

Abstract

The programming language ADA permits the decomposition of large programs into simpler parts and the separate compilation of those parts, where checks are done across the parts bounds at compile time. The separate compilation is based on the use of a library file. After outlining shortly the overall structure of programs and facilities for separate compilation in the language ADA an overview is given about the library management system and its services of the Hungarian ADA project.

Irodalomjegyzék

[1] Reference Manual for the ADA Programming Language
- Proposed Standard Document
July 1980.

Szerzők:

Laborczi Zoltán-Sós Klára
SZÁMKI Bp. I., Csalogány u. 30-32.
Postacim: H-1536 Bp. Pf. 227.

Laborczi Zoltán–Szeredi Péter

SZINTAKTIKUS ELEMZÉS AZ ADA FORDÍTÓBAN

Az alábbiakban az SZKFT együttmüködés keretében kifejlesztendő ADA forditóprogramban alkalmazott szintaktikai elemzésről, az elemzés során használt hibakezelésről és faépitési módszerről adunk ismertetést.

Kulcsszavak: ADA nyelv, LL(1) elemzés, szintaktikus hibák kezelése, absztrakt szintaxis.

1. A szintaktikus elemzés

A legkézenfekvőbb és a szakirodalom által leginkább ajánlott elemzési módszer a rekurziv leszállás technikája. Ez azt jelenti, hogy a szintaktikus szabályokkal azonos struktúrájú olyan eljárásokat konstruálunk, melyek elvégzik az elemzést. Ez akkor tehető meg további komplikációk visszalépés bevezetése nélkül, ha a nyelvtan speciális kikötéseknek eleget tesz, ún. LL(1) struktúrájú.* A nyelvtan és az elemző eljárások közötti transzformáció ekkor az alábbiak szerint végezhető el.

Ha a nyelvtan az alábbi szabályokat tartalmazza:

```
egymásutánirás ::= fej  törzs  farok
alternativa    ::= első
                  |második
                  |harmadik
sorozat        ::= elem  elválasztójel elem
```

* Ez durván fogalmazva azt jelenti, hogy egy jel előre olvasásával el lehessen dönteni, milyen úton kell az elemzésnek haladnia.

akkor ezekből a következő elemzést végző logikai függvényeljárások konstruálhatók /ADA jelölésben/:

```
function  EGYMÁSUTÁNIRÁS return BOOLEAN;
   begin  return FEJ and then TÖRZS and then FAROK;
     end  EGYMÁSUTÁNIRÁS;
function  ALTERNATIVA return BOOLEAN;
   begin  return ELSŐ or else MÁSODIK or else HARMADIK;
     end  ALTERNATIVA;
function  SOROZAT return BOOLEAN;
   begin  if ELEM then
                  while ELVÁLASZTÓJEL
                  loop ELEM;
                  endloop;
                  return TRUE;
          else    return FALSE;
          end if;
     end  SOROZAT;
```

Ugyanez CDL2 jelölésben:

```
'predicate'  EGYMÁSUTÁN IRÁS:
             FEJ, TÖRZS, FAROK.
'predicate'  ALTERNATIVA:
             ELSŐ;
             MÁSODIK;
             HARMADIK.
'predicate'  SOROZAT:
             ELEM,
               (ELVÁLASZTÓJEL,ELEM, *;
               ).
```

A bonyolultabb nyelvtani szabályok átirása a fenti alapesetek kombinációjával végezhető el.*
Az ADA szintaxis esetében az LL(1) követelmény okozott gondot. A nyelvtan nem LL(1) szabályaira is adható azonban olyan elemző eljárás, mely elkerüli a visszalépés beépitését olymódon, hogy a bizonytalan kimenetelü elemzési döntések közös részét kiemelve kezeljük. Alternativ megoldás az lehet, hogy az egyik ágon elemezve végezzük a feldolgozást, s ha kiderül, hogy rossz ágon vagyunk, akkor a felépitett elemzési fát utólag átalakitjuk.

2. A szintaktikus hibák kezelése

Szintaktikus hibára akkor derül fény, amikor az elemzés során alapjelet elemző tevékenység úgy találja, hogy nem az elvárt szimbólum következett, vagy akkor, ha egy tevékenység egyik alternativája sem adott igaz eredményt. Érdemes külön kezelni azt az esetet, amikor egy SOROZAT-szerü szerkezetben hibás, vagy hiányzik az elhatárolójel.

A hibajavitás alapgondolata az, hogy a hiba után a forrásszöveget egészen addig figyelmen kivül hagyjuk, amig az elvárt szerkezetre nem bukkanunk. Ha azonban ezt az ötletet finomitás nélkül alkalmazzuk, túlságosan nagy darab szövegek maradhatnak elemzés nélkül. Az elemzés során általában egy /esetleg rekurziv/ eljáráshivássorozat belsejében vagyunk,

---

* Célszerü kétféle elemző eljárást megkülönböztetni. A fenti tipusú predikátumok jelzik, hogy a szóbanforgó konstrukció következik-e . Tevékenységet akkor használunk, amikor biztosan tudjuk, hogy milyen szerkezetnek kell következnie. /Értékadás jobboldalán például csak kifejezés állhat./ A továbbiakban a kijelentés alakok neveit megkülönböztetésül IS prefixszel látjuk el.

ahol esetleg minden szinten más hiba történt, és ezért más szerkezetet óhajt az elemző látni. Hiba esetén ezért nemcsak az éppen futó, hanem a külső eljárások hibakezelésére is gondolni kell. Ez olymódon valósitható meg, hogy az elemző eljárások egy stop-stack nevü verembe elhelyezik azokat a szimbólumokat, amelyek szintaktikus hiba esetén sem maradhatnak ki a forrásprogramból. Ezeknél a jeleknél tud az elemzés a hiba észlelése után a megszokott mederbe visszatérni.

A továbbiakban először azt mutatjuk meg, hogy épithető be az elemző eljárásokba a hibakezelés, majd a hibakezelő segédeljárásokkal foglalkozunk.

Az alternativára a következő hibajavitó elemző adható:

```
'action' ALTERNATIVA:
         IS ELSŐ;
         IS MÁSODIK;
         IS HARMADIK;
         SYNTAX ERROR+ROSSZ ALTERNATIVA+NYELEK,*;
```

A SYNTAX ERROR ágra akkor kerül a vezérlés, ha egyik korábbi ág sem adott igaz értéket. A ROSSZ ALTERNATIVA paraméter a hiba kijelzésére szolgál, mig a NYELEK az ELSŐ, MÁSODIK és HARMADIK lehetséges kezdőszimbólumainak halmazát ábrázoló paraméter. A SYNTAX ERROR kihagyja a forrásszöveget, amig szabad, azaz amig vagy a NYELEK paraméter által jelölt halmazba, vagy a stop-stackbe tartozó szimbólumot nem talál. Az előbbi esetben igazzal tér vissza, s ekkor az ALTERNATIVÁ-t újra megkiséreljük felismerni; stop-stackbeli szimbólumra történt megállás esetén lemondunk arról, hogy helyes ALTERNATIVÁ-t találjunk.

Ugyanez az EGYMÁSUTÁNIRÁS esetén:

```
'action' EGYMÁSUTÁNIRÁS:
         STOP AT + TÖRZSNYELEK,
         STOP AT + FAROKNYELEK,
         FEJ, FORGET STOPS, TÖRZS, FORGET STOPS, FAROK.
```

A STOP AT eljárás a paraméterében megadott szimbólumhalmazzal bővíti a stop stacket, a FORGET STOPS pedig az utoljára behelyezett és korábban még nem törölt adagot törli a stop stack tetejéről.

Hasonló, de kissé bonyolultabb szabály a SOROZAT-ra is adható.

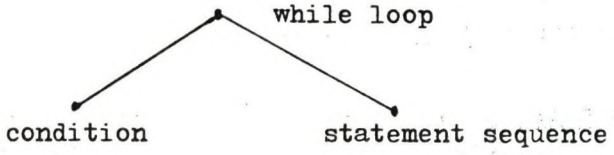
3. Az elemzés eredménye: a szintaxisfa

3.1 A szintaxisfa szerkezete

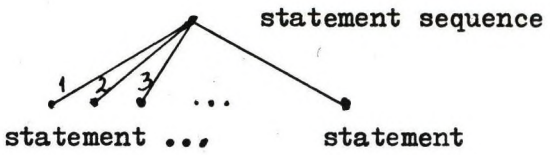
A szintaktikus elemzés eredményeképpen egy elemzési fának kell előállnia. Jelen pont a fa szerkezetével és megadásának módjával foglalkozik.

A szintaxisfa kétféle fajtájú csúcsot tartalmaz:

- az ún. struktúrafajtájú csúcsból rögzitett számú, általában különböző szintaktikus egységeknek megfelelő él indul ki, pl.



- az ún. listafajtájú csúcsból 0,1 vagy több él indul ki, amelyek azonos szintaktikus egységeknek felelnek meg, pl.



A szintaxisfa minden egyes csúcsát az adott szintaktikus fogalom nevével cimkézzük meg.

A szintaxisfa levelei egyes lexikális elemek /azonositók, számok, füzérek/ vagy pedig egyéb segédobjektumok /pl. for-ciklusnál a reverse alapszónak megfelelő irányjelzés/ lehetnek.

A szintaxisfa szerkezetét LDM nyelvü [1] tartománydefiniciókkal adjuk meg, pl.

```
loop statement:=
        for loop statement;
        while loop statement;
        endless loop statement.
for loop statement::= nstruct(
        loop parameter   : identifier,
        traversing order : (the normal; the reverse),
        loop range       : discrete range,
        for loop body    : statement sequence).
while loop statement::= nstruct(
        loop condition   : condition,
        while loop body  : statement sequence).
basic loop::= nstruct(
        basic loop body  : statement sequence).
statement sequence::= nlist statement.
```

A fentiekben az nstruct ... alakú definiciók a struktúrafajtájú, az nlist ... alakúak pedig a listafajtájú csúcsoknak felelnek meg; a segédobjektumok bevezetésére a the alapszót használjuk; a ";" pedig az alternativák elhatárolására szolgál.

A struktúrafajtájú csúcsok esetén minden egyes komponens mellett egy kiválasztó nevet is feltüntettünk. Ahol nem adunk meg kiválasztó azonositót a komponensekhez, ott automatikusan képződik egy "α of β" szerkezetü kiválasztó név, például a "loop range" helyett annak távolmaradása esetén a "discrete range of for loop statement" nevet kellene használni.

A formális leirásban megengedett még az "opt a" alakú szerkezet használata, ami az "a" objektum opcionális voltát jelzi; ez a szerkezet formálisan az "a; the null object" röviditése.

3.2 A fastruktúra megvalósitása

A fastruktúra CDL-ben való megvalósitása úgy történik, hogy minden egyes csúcsnak egy /néhány szóból álló/ blokkot feleltetünk meg, ezek a blokkok egy CDL-listában vagy más cimezhető memóriaterületen helyeződnek el. Az egyes blokkok tartalmaznak egy a csúcs cimkéjére /a szintaktikus egységre/ utaló jelzést, a csúcsból kiinduló élek leirását, valamint listafajtájú blokkok esetén ezek előtt a lista hosszát. Ha egy él levélhez vezet, akkor a blokkban közvetlenül szerepel a levelet, mint lexikai egységet vagy segédobjektumot azonositó kód; ha az él egy részfához vezet, akkor a részfa gyökerét leiró blokk cime szerepel az adott él leirásaként /a "szülő" blokkban/.

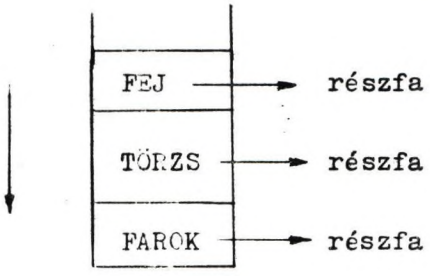
3.3 A szintaxisfa felépítése

A szintaxisfa épitése egy segédverem, az ún. "son-stack" felhasználásával történik. A verembe kell helyezni a felépitendő csúcs komponenseit, a faépitő eljárások a felhasznált komponenseket a veremből kiveszik, létrehoznak egy a csúcsnak megfelelő blokkot, és annak cimét helyezik a verembe. Ez a mechanizmus lehetővé teszi, hogy a faépitést az első menetben az elemzés során szinte teljesen automatikusan végezhessük. Elegendő ugyanis arról gondoskodni, hogy a levelek kielemzésekor a megfelelő kód a segédveremre kerüljön, valamint hogy az egyes szintaktikus egységek elemzésének végén az adott egységhez tartozó faépitő tevékenység meghivásra kerüljön. Igy mindig fennáll, hogy a segédverem tetején az adott egység leirása van.

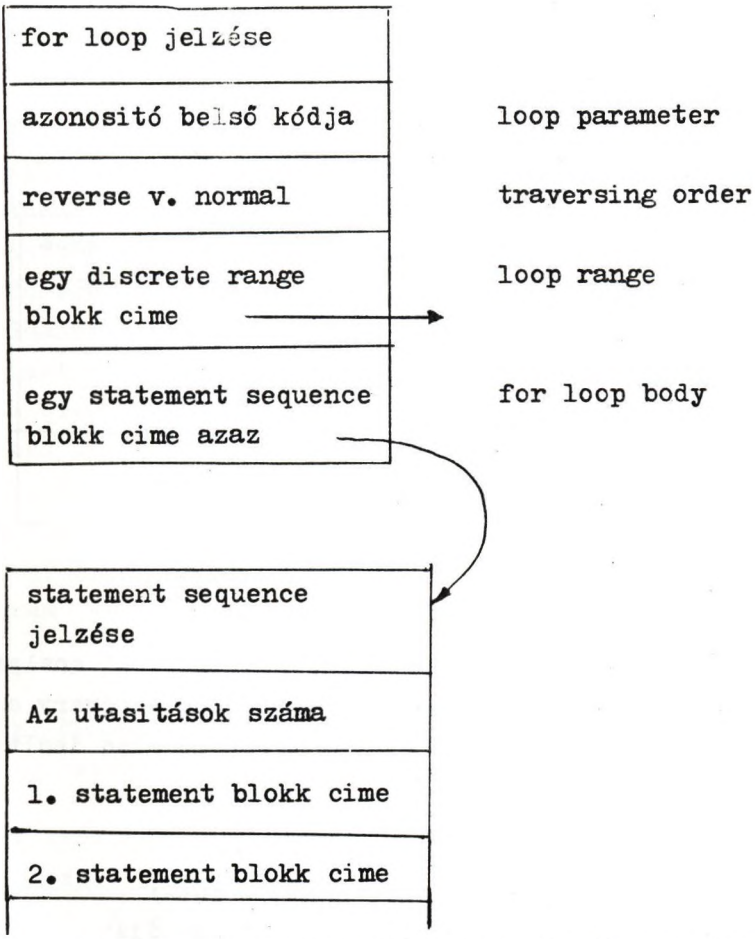
A fafelépitési mechanizmus a 2.pontbeli EGYMÁSUTÁNIRÁS szabály faépitést is végző alakjának bemutatásával illusztráljuk.

```
'action' EGYMÁSUTÁNIRÁS:
         FEJ, TÖRZS, FAROK, EGYMÁSUTÁNIRÁSÉPITÉS.
```

A FEJ, TÖRZS és FAROK szabályok sorra felépitik a felismert szerkezet fáját, a cimet elhelyezve a son stack tetején. A build catenation faépitő tevékenység meghivása előtt a son stack teteje:

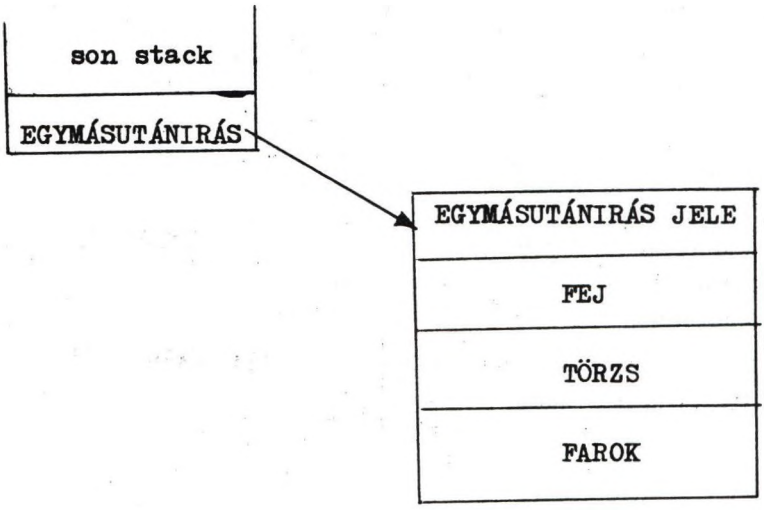


l:



A fastruktúra tárolása a fejlesztőgépen a /hardware virtuális/ memóriában történik.
A többi tárgygépen szoftver úton lapozott virtuális memória szervezést kell biztositani a fastruktúra számára.

Az EGYMÁSUTÁNIRÁS ÉPITÉS hivás hatására a verem tetején elő-állt csomópontleirás, kis mechanikus módositással, felkerül a három részfa fölé, mint új csúcspont:



Abstract

This paper presents the method of syntactical analysis used in the Hungarian ADA compiler. The error recovery and the construction of abstract syntax trees is also dealt with.

IRODALOM:

[1] Szeredi-Balogh-Farkas-Sántáné-Tóth:
Az LDM tervezési nyelv specifikációja
SZKI-NIMIGÜSZI tanulmány a SZÁMKI részére 1979.

Szerzők: Laborczi Zoltán
Számitógépalkalmazási Kutató Intézet
1536 Budapest, Pf.227.

Szeredi Péter
Számitástechnikai Koordinációs Intézet
1054.Budapest, Akadémia u.17.

Legendi Tamás–Szajbély György

DIALÓGUS PROGRAMOK KÉSZÍTÉSE A CHANGE PROGRAMOZÁSI NYELV KITERJESZTÉSÉVEL

Dolgozatunkban először általában beszélünk a dialógusokról, azok osztályozásáról és definiálunk néhány alapfogalmat. Ezután kiemeljük a CHANGE nyelv azon speciális tulajdonságait - elsődlegesen a kiterjeszthetőséget -, amelyek különösen alkalmassá teszik a nyelvet dialógus programok irására. A dolgozat második felében két dialógus programot mutatunk be. Az első egy "játék program",amely menüből való választást tesz lehetővé. Ezen a példán szemléltetjük a fő vonásokat: a dialógus-fa, a dialógus-iró nyelv megadását, valamint azt a leképezést, amelynek segitségével egy dialógus-fának egy dialógus programot feleltethetünk meg. A második program egy személyi nyilvántartást lekérdező rendszer.

Kulcsszavak: CHANGE, dialógus, dialógus-iró nyelv, kiterjeszthető nyelv, dinamikus programozás.

1. Dialógusokkal kapcsolatos alapfogalmak, dialógusprogramok elemzése

A számitógépek terjedésével párhuzamosan felmerült az igény az ember-gép párbeszédes kapcsolat megteremtésére. Amint a hardver és szoftver feltételek /perifériák, időosztásos rendszerek, stb./ kialakultak, rohamos fejlődésnek indultak az ember és a gép dialógusára épülő interaktiv rendszerek.

1.1. Néhány alapfogalom

1.1.1. Általában dialógusrendszeren a következőket szokás érteni: a párbeszédben résztvevő A-nak és B-nek mindazokat a szerveit és funkcióit,

amelyek közvetlenül vagy közvetve résztvesznek a párbeszéd kialakitásában és az információ áramot.

1.1.2. Dialógusnak nevezzük azt a folyamatot, amikor a párbeszédben résztvevő A és B között üzenetváltás, vagy üzenetváltások sorozata zajlik le, vagyis magát az információáramlást.

Ebben a dolgozatban olyan speciális dialógusrendszerrel foglalkozunk, amelyben az egyik résztvevő számitógép.

1.1.3. Dialógus-fa: a dialógus összes lehetséges lefutása.

1.1.4. Dialógus-iró nyelv: az a nyelv, amelyen a dialógusfa leirható.

1.1.5. Dialógusprogram: a dialógus-iró nyelven irt program.

1.1.6. Dialógusgép: a dialógusprogram futását biztositó hardver és szoftver környezet /futtató rendszer/.

1.2. A dialógusok osztályozása:

1.2.1. A dialógus statikus,ha a dialógus kezdetekor a folyamat összes lehetséges lefutása eleve adott és ezek közül egy realizálódik a válaszoktól függően. Tehát a dialógus-fa sem a dialógusok lezajlása alatt, sem azok után nem változik. /Pl. menüből való választás/.

1.2.2. A dialógus dinamikus, ha a program csak egy kezdeti dialógusvázat követel meg, amelynek a későbbi szögpontjai a dialógus során alakulnak ki. Tehát a dialógus-fának az egyes dialógusok lezajlása alatt is, annak eredményeként is uj szögpontjai jöhetnek létre, illetve szögpontjai szünhetnek meg. /Pl.: adaptiv, tanuló rendszerek/.

1.2.3. Pszeudodinamikus dialógusról beszélünk akkor, ha a dialógus lefutásai ugyan adottak, de a kérdések szövegei módosulhatnak a lefutás során. Tehát a

dialógus-fa szögpontjai statikusak, de az egyes szögpontokban levő akciók dinamikusak.

1.3. Mire való, mit kell "tudni" egy dialógusprogramnak

Az ember-gép között lezajló dialógus eszköz egy adott feladat erőteljes irányitás mellett történő elvégzéséhez. A feladatok igen sokfélék lehetnek. Bizonyos dialógusok esetén a dialógus célja csak a dialógus végén realizálódik. /Pl.: barkochba, információ visszakeresés/, más esetekben folyamatosan képződik a dialógus során /pl.: számitógépes tervezés, oktatás, programozás/. Egy adott feladathoz tartozó összes dialógus leirható egy állapothalmazzal és ezen állapotok közötti lehetséges átmenetekkel. Egy dialógus reprezentálható a lehetségesek közül egy állapotátmenet sorozattal. Egy általános dialógusprogramnak a következőket kell végrehajtani:

- állapot-kezdő akció
- információ adás /"kérdés"/
- válasz-beolvasás
- válaszelemzés
- akció
- új állapot meghatározása

2. A CHANGE-nyelv kiterjesztése dialógusprogramok készitéséhez

2.1. A dialógusprogramok leirása az általában használatos programozási nyelveken nehézkes, ugyanis speciális állapotrekordok kialakitását igényli, amelyek kezelése, mivel a nyelvek nem rendelkeznek ehhez megfelelő eszközökkel, bonyolult.

2.2. A CHANGE programozási nyelv, a hagyományos nyelvektől eltérően, két új lehetőséget biztosit:

kiterjeszthető és a CHANGE program futás közben

dinamikusan változtatható.[1]

Ezen adottságok különösen alkalmassá teszik a nyelvet dialógusprogramok irására, a következők miatt:

2.2.1. A CHANGE-nyelv kiterjesztő EXTEND utasitásának segitségével lehetőségünk van egy dialógus-iró nyelv definiálására. Ennek alkalmazásával elszakadunk a dialógus-fa eddigi adatjellegü leirásától, és helyette program formájában ábrázolhatjuk.

2.2.2. A CHANGE utasitással futás közben lehet a programot módosítani, vagyis a program utasitásai közé új utasitásokat beszúrni, illetve programsorokat törölni, igy a dinamikus dialógus dialógus-fáját is jól lehet ezzel a módszerrel alakítani.
Az előbbiekben ismertetett lehetőségeket két példán mutatjuk be.

3. A CDC 3300-as számitógépen CHANGE-nyelven megvalósított dialógusprogramok ismertetése

3.1. Egy statikus dialógusprogram ismertetése

A dialógus-fa egyes szögpontjaiban előre megadott menüből való választás lehetséges. Egy-egy menü két részből áll:

- egy tényleges kérdésből, ami egy karaktersorozat és arra utal, hogy miből, milyen céllal lehet választani. Ez esetenként el is maradhat.

---

[1] Megjegyezzük, hogy a nyelv a fentieken kivül is tartalmaz a megszokottól eltérő lehetőségeket /pl.: létrehozhatók processzorstrukturák/, de ezeket most nem kivánjuk kihasználni, ezért itt nem emeljük ki.

- A "menü-testből",amelyben 1-től N-ig sorszámozva fel vannak sorolva a lehetséges alternatívák. A menüből való választáson ezen sorszámok valamelyikének megadását értjük.

Példa egy menüre:

MILYEN PROGRAM ERDEKLI?
1: EGESZNAPOS
2: DELELŐTTI
3: DELUTANI

A dialógus gép müködése:
A dialógus gép ad egy menüt, majd választ kér. A választól függően újabb menüt ad vagy közli a dialógus eredményeként javaslatát. Ezek után ismét egy menüt ad annak eldöntésére, hogy a dialógus kezdődjön elölről vagy álljon le a program.

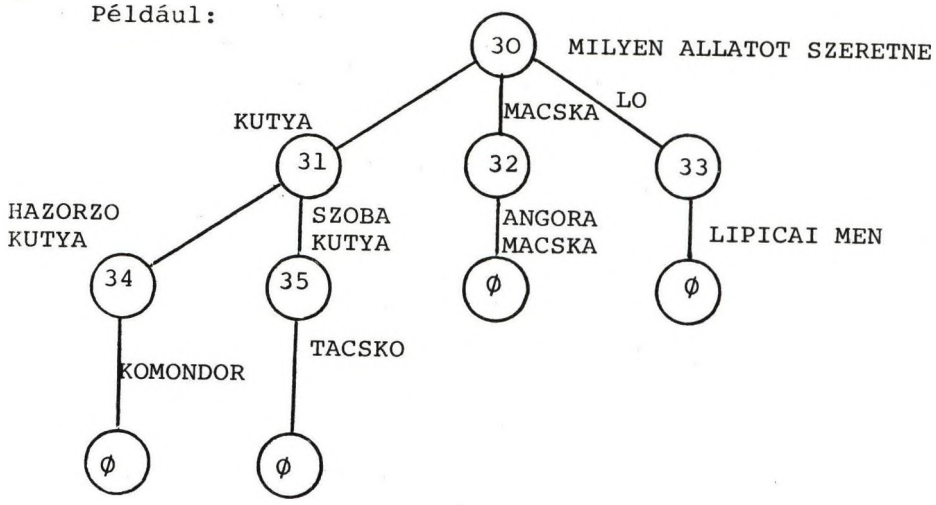
3.1.1. A dialógus-fa megadása

A dialógus-fát a következőképpen adjuk meg: minden szögpontját egy-egy sorszámmal látjuk el. Ez a sorszám megfelel a CHANGE-nyelvben használatos cimkének, tehát legfeljebb 5 jegyü szám, de nem kisebb 30-nál[2]. A sorszám mellett adjuk meg az esetleges menü-kezdő tényleges kérdést is. Az egyes élek azokat a szövegeket jelentik, amit abban a szögpontban kell kinyomtatni, ahonnan kiindulnak az élek és egy szöveg kiválasztása esetén /a menüből történő választáskor/ abban a szögpontban folytatódik a dialógus ahová az adott él befut. Ha egy dialógus végetér, akkor az utolsó nyomtatandó szöveget tartalmazó él egy 0 sorszámu szögpontba fut be.

---

[2] A kiterjesztett utasitások definiciójában, ill. az ezekhez kapcsolódó programkörnyezetben az 1-től 29-ig terjedő cimketartományt felhasználtuk.

Például:



3.1.2. A dialógus-iró nyelv

Olyan négy utasításból álló nyelvet definiálunk kiterjesztett CHANGE utasitásokként, amely lehetőséget biztosit dialógus-fa leíráshoz CHANGE program formájában.

A nyelv utasításai:

(A) TC

(B) TC THEN IC

(C) END

(D) OFFER:TC

Az utasításokban használt jelölések:

TC: szövegkonstans

IC: egész konstans

THEN
OFFER } Alapszavak
END

Az (A), (B) és (C) utasitások cimkézhetők, a (D) utasitás nem cimkézhető.

3.1.3. A dialógus-fa CHANGE programkénti megadása

A dialógusfa szögpontjait három csoportra osztjuk:

I. csop.: Egy szögpontból egynél több él indul ki

II. csop.: Egy szögpontból pontosan egy él indul ki (tehát csak 0 sorszámú szögpontba vezet él)

III. csop.: A szögpontból nem indul ki él (0 sorszámú szögpont)

3.1.3.1. A programozó tevékenysége a dialógus-fa CHANGE programkénti leirásakor

- A szögpont az I. csoportba tartozik:

(a) Ha a szögpont tartalmaz menü kezdő kérdést is, akkor a szögpont sorszámát, mint cimkét kell adni egy olyan (A) tipusu utasításnak, ahol a TC paraméter értéke a menü-kezdő kérdés. Ha a szögpontból n db él indul ki ($n \geq 1$), akkor ezeknek n db (B) tipusú utasitást kell megfeleltetni ugy, hogy az egyes TC paraméterek értéke a szögpontból kiinduló éleken levő szövegek, az IC paraméterek helyébe azon szögpontok sorszámát kell irni, ahová az egyes élek befutnak. Az utasítássorozat zárásaként END utasítást kell irni.

(b) Ha a szögpont nem tartalmaz menü-kezdő kérdést, akkor csak annyi az eltérés az (a) pontban megadottaktól, hogy az (A) tipusu utasítás elmarad és a cimke a sorrendben első (B) tipusu utasításra kerül.

- A szögpont a II. csoportba tartozik:

Egy (D) tipusu utasítást kell irni, ahol a TC paraméter értéke a szögpontból kiinduló élen levő szöveg.

- A szögpont a III. csoportba tartozik:

Nincs tevékenység.

Az ismertetett mechanizmust a 3.1.1. pontban szereplő dialógus-fa egy részletének leirásával szem-

léltetjük.

```
30  $MILYEN ALLATOT SZERETNE$
    $KUTYA$ THEN 31
    $MACSKA$ THEN 32
    $LO$ THEN 33
    END
31  $HAZORZO KUTYA$ THEN 34
    $SZOBA KUTYA$ THEN 35
    END
34  OFFER: $KOMONDOR$
    :
```

3.1.4. A dialógusgép definiciója - a kiterjesztett utasitások szemantikája

A dialógus-iró nyelv utasitásait "nyilt" kiterjesztett utasitásokként definiáltuk[3].

(A) TC

A dialógusgép kinyomtatja[4] a TC paraméterben megadott szöveget, vagyis a menü-kezdő kérdést.

(B) TC THEN IC

A dialógusgép az aktuális sorszámmal együtt kinyomtatja a TC paraméterben megadott szöveget, vagyis a menü egy sorát; megjegyzi a kinyomtatott sorszámhoz az IC paraméterben megadott konstans értéket; 1-gyel növeli az aktuális sorszámot tartalmazó változó értékét.

(C) END

A dialógusgép választ kér a felhasználótól, beolvassa az adott választ; /beállitja a megfelelő pointereket/;

---

[3] A CHANGE nyelv ismertetésére és igy ennek a speciális utasitásnak az ismertetésére nem térünk ki, a [3] felhasználói kézikönyvben megtalálható.

[4] A további elképzeléseinkben természetesen nem kártyaolvasó és sornyomtató szerepel a dialógusprogram I/O perifériáiként, hanem display; pillanatnyilag a dialógusgép és a dialógusprogram megvalósitására koncentráltunk.

végül a válaszként beolvasott sorszámhoz tartozó címkére adja a vezérlést.

(D) OFFER: TC

A dialógusgép kinyomtatja a TC szöveget; megkérdezi, hogy folytatódjon-e a dialógus; beolvassa a választ; a választól függően a dialógusprogram elejére vagy egy STOP utasitásra adja a vezérlést. Hangsúlyozni szeretnénk, hogy az egyes menük sorainak a száma, vagyis a dialógus-fa egy-egy szögpontjából kiinduló élek száma változhat és egy-egy dialógus tetszőleges számu kérdés-feleletből állhat, tetszőleges "mélységű" lehet.

3.2. Egy, a pszeudódinamikus osztályba tartozó program ismertetése

3.2.1. A program feladata

A VOLÁN 10. sz. Vállalat Munkaügyi Főosztályán hamarosan elhelyezésre kerül egy TPA-i kompatibilis, operációs rendszer nélkül speciális perifériákkal céleszközként müködő mikroszámitógép[5]. Ez a gép személyi adatok felvételére és nyilvántartására lesz fenntartva. A személyi nyilvántartási céleszköz háttértároló perifériája DCD-3 mágneskazetta, - I/O perifériája display. Egy-egy személyről kb. 100 adat van tárolva úgy, hogy külön kazettán a szegedi és külön a vidéki dolgozók adatai. A tárgyalt program feladata az, hogy különböző igazolások kiadásához a szükséges adatokat gyorsan biztosítsa úgy, hogy a céleszköz kezelése minimális szakértelmet igényeljen.

3.2.2. A megvalósitás

A feladat megoldásának vázát egy statikus dialógus alkotja. Ez könnyen megirható a 3.1. pontban definiált

---

[5] Lehetőség van CHANGE nyelven irt program mikroszámitógépen történő futtatására. Ennek megvalósitását ismerteti [2]

dialógus-iró nyelven. Az egyes szögpontokban vannak dinamikus feladatok, akciók is.

Ezek a következők:

- törzsszám beolvasása
- a megfelelő rekord kikeresése és beolvasása
- a beolvasott rekordokból a kért adatok kiválasztása és display-re kiirása.

Az akciók végrehajtására újabb kiterjesztett CHANGE utasitásokkal kell bőviteni a 3.2-ben definiált (A), (B), (C), (D) utasitásokból álló dialógus-iró nyelvet.

3.2.2.1. Az utasitások szintaktikája

(E) A TORZSSZAM BEOLVASASA

(F) TORZSSZAM SZERINTI KERESES

(G) NYOMTATAS: IC

IC: egész konstans

Minden más alapszó.

3.2.2.2. Az utasitások szemantikája

Az utasitások szemantikáját azok magyar nyelvü jelentése informálisan definiálja.

- A dialógusgép az (E) utasitás segitségével felszólitja a felhasználót a törzsszám begépelésére, majd annak beolvasása után numerikus ellenőrzést is végez. Ha szükséges, hibajelzést ad és újra kéri a törzsszámot.
- A dialógusgép az (F) utasitás hatására megkeresi és beolvassa a megfelelő rekordokat; ha nem találja, akkor hibajelzést ad és új törzsszámot kér.
- A dialógusgép a (G) utasitás hatására az IC-ben megadott sorszámú display képet kinyomtatja és új törzsszámot kér.

A kibővített dialógusiró nyelv konkrét alkalmazására bemutatjuk a "személyi lekérdező rendszer" egy programjának rövid részletét.

```
   :
   A TORZSSZAM BEGEPELESE
   $A DOLGOZO MUNKAHELYE:$
   $SZEGED$ THEN 72
   $VIDEK$ THEN 73
   $ISMERETLEN$ THEN 74
   END
72 $KEREM TEGYE BE A T1/SZEGED KAZETTAT$
   TORZSSZAM SZERINTI KERESES
   $MILYEN CELBOL KERI AZ INFORMACIOT$
   $RUHAKIADAS$ THEN 77
   $SZOKVANYOS IGAZOLAS$ THEN 78
   $LAKASKIUTALAS$ THEN 79
   $EGYEB$ THEN 8Ø
   END
77 TABLA NYOMTATAS:1
   :
```

Terveink szerint az egész programrendszert, beleértve az adatfelvevő programot is a fentiekben ismertetett nyelv segitségével irjuk meg, szükség esetén egy-két utasitással tovább bővitve a nyelvet.

Abstract:
In our paper first we give definitions on dialogues, their classification and introduce some fundamental conceptions. Then we emphasize the special properties of the CHANGE language - primarily the possibility of the language extension - making/the language especially suitable for writing dialogue programs. In the second part of the paper two dialogue programs are demonstrated. The first one is a "game program" making possible a choise from a menu. The main features are illustrated by this example: the dialogue-tree, the dialogue language specification just as the mapping of a dialogue-tree into the corresponding dialogue program.

The second program is an application dialogue program, a query system for personal data.

Irodalomjegyzék

/1./ Ambrózy Denise: Dialógus, dialógusrendszer /MTA AKI Közlemények, 7/1971./

/2./ Almási József, Legendi Tamás, Szajbély György, Szekeres Szilveszter, Tóth Károly: Mikroszoftver generálása felhasználói kiterjesztésekkel létrehozott célnyelveken irt és tesztelt programokból a CHANGE processzor felhasználásával /Programozási Rendszerek'81 konferencia, Szeged, 1981/

/3./ Pikler Gyula: Mini-számitógépes interaktiv alkatrészprogramíró rendszer NC szerszámgépek automatikus programozásához /MTA SZTAKI Tanulmányok, 18/1874./

/5./ Varasdy Imre: CHANGE programozás /Diplomamunka, JATE 1975./

Legendi Tamás: MTA Automataelméleti Kutató Csoport,
6720 Szeged, Somogyi u. 7.
Szajbély György: VOLÁN 10. sz. Vállalat,
6722 Szeged, Bakay N. u. 48.

Major Péter–Bidó Zsuzsa–Polgár Judit

ON–LINE ALKALMAZÁSI RENDSZER FEJLESZTÉSE IDMS KÖRNYEZETBEN

Kivonat

Az IDMS adatbázis-kezelő rendszer lehetőséget nyujt mind batch mind on-line rendszerek fejlesztésére. Ebben az előadásban egy kisérleti alkalmazási rendszer ismertetésén keresztül az IDMS on-line lehetőségeit szeretnénk bemutatni.
Először az IDMS on-line környezetének kiépitését, ezt követően pedig az alkalmazási rendszert mutatjuk be.
Az on-line funkciókat egy müködő batch rendszerrel párhuzamosan valósitottuk meg. A batch rendszer adatbázisát, az on-line környezet követelményeinek megfelelően, kisebb módositással tudtuk felhasználni.

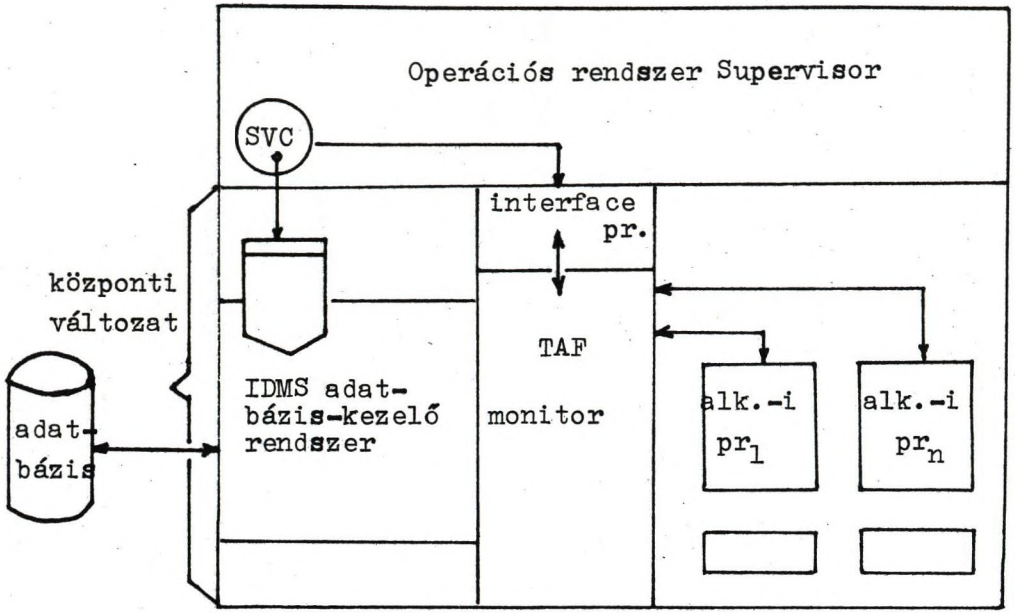
Az előadásban ismertetjük az on-line rendszer adatbázisát és a megvalósitott rendszer programjait /tranzakcióit/, végül a rendszer tervezése, tesztelése és müködtetése során nyert tapasztalatokat összegezzük.

1. A kisérleti on-line rendszer

1.1. Az IDMS on-line környezetének kiépitése

Az IDMS on-line alkalmazások előfeltétele az, hogy létezzen egy olyan adatátviteli monitor /TAF monitor/, amelyet az IDMS és a befogadó operációs rendszer [1] egyaránt támogat. Az alkalmazási programok az adatbázisban való keresési, kiválasztási, lekérdezési, módositási, törlési stb. funkciókat látják el, mig a terminálok kezelésére, az input/output müveletek lebonyolitására, az alkalmazási programok ütemezésére az adatátviteli monitor szolgál.

A távadatátviteli környezetben használandó alkalmazási programok és az adatbázis-kezelő rendszer, illetve az adatbázis kapcsolatát sematikusan az első ábra szemlélteti.



1. ábra: Az IDMS és az alkalmazási programok kapcsolata TAF környezetben

1.2. Az alkalmazási rendszer funkciói

Az IDMS on-line lehetőségeit egy kisérleti rendszer megvalósitásával próbáltuk ki. A rendszer alapját az Országos Takarmányozási és Állattenyésztési Felügyelőség /OTÁF/ részére készitett IDMS alapu batch rendszer képezte.

A rendszer hátterének megvilágitásához vázolnunk kell a takarmánygazdálkodás célját és funkcióit.

A takarmánygazdálkodás a népgazdasági termelésben egyre nagyobb jelentőségre tesz szert. Az alap és keveréktakarmányok minőségi szórásértékeiből adódó toleranciák még jelentős, ki nem aknázott tartalékokat rejtenek. A hazai takarmánybázis tartalékainak feltárása import fehérjetakarmányok kiváltását teszi lehetővé. Egy országos felméréshez vagy az országos takarmányhelyzet folyamatos ellenőrzéséhez nagyszámu laboratóriumi takarmányvizsgálati adathalmaz rendszer kiértékelése szükséges.

A batch rendszer feladata a takarmányminták vizsgálati értékeinek folyamatos nyilvántartása és a takarmányvizsgálati adathalmaz rendszeres kiértékelése. A mintavételi alap illetve a minta által reprezentált takarmány mennyiség kigyüjtésével a nagyszámu adat birtokában

- országos takarmánymérleg

a laboratóriumi vizsgálatok értékelésével pedig:

- országos táplálóanyag-mérleg készithető.

Ezek alapján viszont meghatározható országos szinten

- az import fehérje szükséglet adott időszakra, éves negyedéves, havi bontásban, vagy többéves időszakra előrejelzés is készithető.

Az összes termőterület összevetésével pedig:

- országos átlagtermés hektáronként
- összes országos termés - tonna
- országos, átlagos beltartalmi értékek
- országos, területi takarmányminősitések

oldhatók meg megyesoros, gazdaságsoros összesitésekben, átlagok kiszámitása területegységekre, valamint részletes statisztikai elemzés végezhető el.

A laboratóriumi vizsgálat hátterét részben az OTÁF Laboratóriumi Központja, részben a megyei szakszolgálati Laboratóriumok biztositják.

Az IDMS alapu batch rendszer a fenti feladatok ellátása érdekében

- létrehozza és feltölti az adatbázist
- rendszeresen karbantartja az adatbázist
- előállitja a kivánt táblákat az adatbázisból.

A batch rendszer leirását a [3], [4] és [5] tartalmazza. Több felhasználó egyidejü néhány másodpercen belüli válaszidejü kiszolgálása, egyes azonnal igények kielégitése azonban csak on-line rendszer segitségével [6] lehetséges.

Az OTÁF rendszernél ezek az on-line igények csak körvonalaikban fogalmazódtak meg, de az IDMS on-line lehetőségeinek valóságos körülmények közötti kipróbálására alkalmat nyujtottak./Megjegyezzük, hogy az IDMS installálását követően ez volt az első élő, IDMS alapú AB/TAF rendszer./

A kisérleti rendszerünkben felmerült real-time/on-line igények főbb tipusai a következők:

- import takarmányminták gyors értékelése
- gyártási helyek /gazdaságok/ értékelése
- megyei szintü értékelések
- adott takarmányminták gyártási helyének megállapitása
- kis mennyiségü karbantartások.

1.3. Az alkalmazási rendszer ismertetése

Az on-line alkalmazási rendszer megvalósitása során az adatbázis tervezésekor, az adatszótár létrehozásakor és a betöltésnél lényegében ugyanazokat a tevékenységeket kell elvégezni, mint batch IDMS adatbázisok létre-

hozásakor.

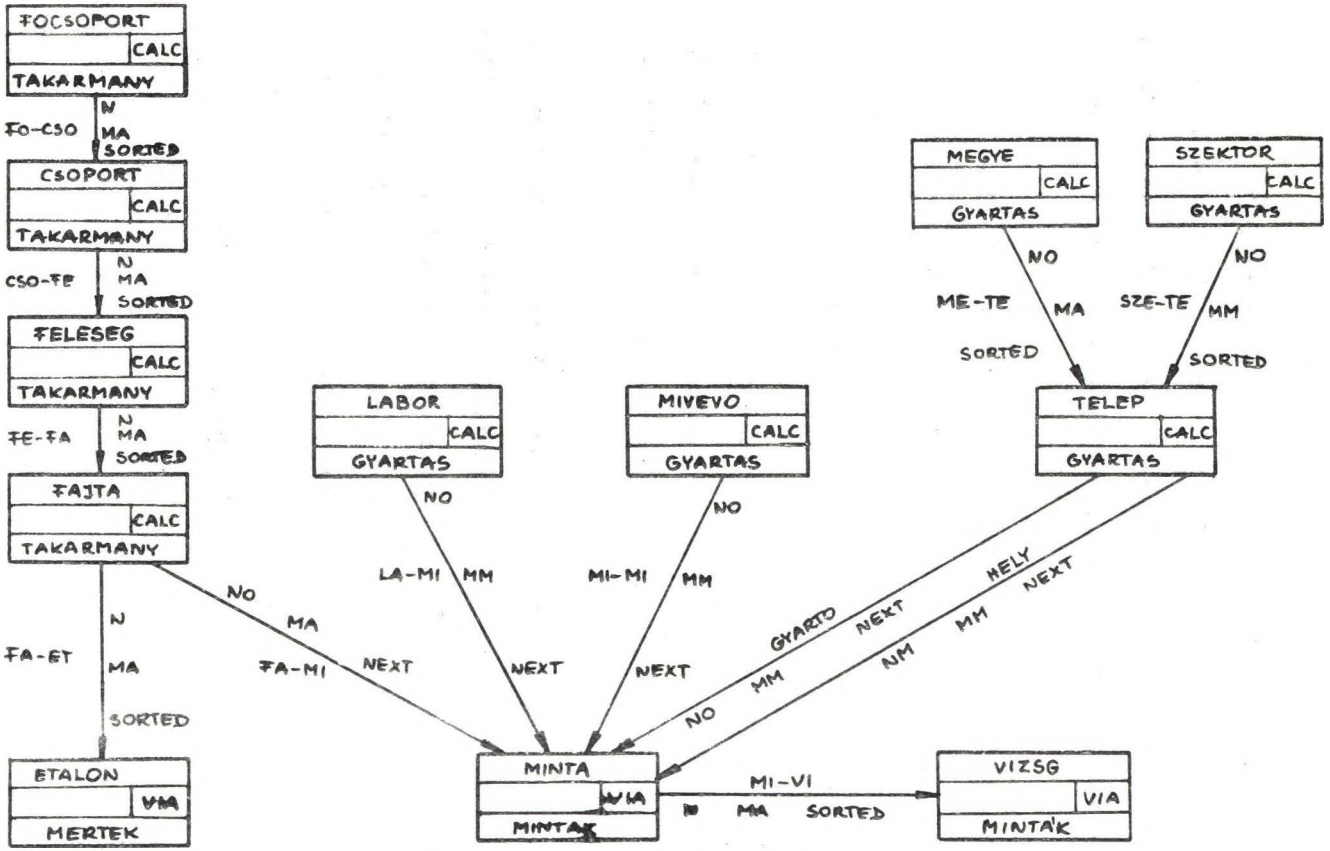
Esetünkben az on-line igények megjelenése nem tette szükségessé a batch rendszernél használt adatbázis ujratervezését és ujrabetöltését. /Az on-line adatbázisok betöltését általában batch üzemmódban célszerü elvégezni./

A felmerülő néhány módositás a már meglévő adatállományt nem befolyásolta. Ezekre, az on-line környezetben szükséges módositásokra a következő pontban térünk ki.

Az adatbázis fizikai tervét a 2. ábra mutatja.

Az adatbázis-alapu on-line alkalmazási rendszerek megvalósitásánál az adatbázis tervezését és betöltését követően az on-line alkalmazási programok - tranzakciók - fejlesztésére került sor.

Az on-line igények kielégitését megvalósitó tranzakciók olyan IDMS alkalmazási programok, amelyekbe a terminálmüveletek vezérlésére különböző makrók vannak beépitve. Kisérleti rendszerünkben négy lekérdező és egy karbantartó tranzakciót fejlesztettünk ki. A tranzakciókat olyan négyjegyü kóddal azonositottuk, amely a tranzakció funkciójára utal /VIZS, KERD, MEGY, GYAR/.



2.ábra: Az adatbázis fizikai terve

A VIZS tranzakció a takarmánymintákon végzett mérések eredményeit értékeli és megállapitja, hogy a takarmány elfogadható, fenntartással elfogadható vagy nem fogadható el. Az értékelés során a tranzakció a mért beltartalmi értékeket az adatbázisban tárolt etalonnal összehasonlitja, kiszámitja az eltéréseket és minősiti a takarmányt. Egy takarmánymintához több /max.10/ mérés tartozik.

A KERD tranzakció az adatbázisból azt kérdezi le, hogy egy takarmányfajtából egy adott időszakban egy gyártási helyen mennyi volt elfogadható, fenntartással elfogadható vagy nem elfogadható.

A MEGY tranzakció az adatbázisból azt kérdezi le, hogy egy megyében egy takarmányból milyen mennyiséget állitottak elő egy adott időszakban.

A GYAR tranzakció az adatbázisból azt kérdezi le, hogy egy gyártási helyen egy takarmányból egy adott időszakban milyen mennyiséget állitottak elő.

Az utolsó két tranzakció megadja még az értékelhető minták néhány fontos átlagos vizsgálati értékét /fehérje, keményitő tartalom/ is.

A karbantartási funkciókat ellátó KARB tranzakció segitségével lehetőség van a következő müveletek elvégzésére:

- uj minták beszurása;
- meglévő mintákhoz uj vizsgálat beszurása;
- meglévő minta módositása;
- meglévő vizsgálat módositása;
- minta törlése;
- vizsgálat törlése;

/A nagy volumenü karbantartásokat természetesen célszerü batch üzemmódban végezni./

A tranzakciók funkcióira, inputjára, outputjára vonatkozó leirást a rendszer un. menü tranzakciói /OTAF, OTAFVIZS, OTAFKERD, OTAFMEGY, OTAFGYAR és OTAFKARB/ tartalmazzák. Ezek segitségével a felhasználó viszonylag gyorsan megtanulhatja a lekérdező és karbantartó tranzakciók kezelését.

Az on-line alkalmazási programok tervezése és programozása során az IDMS programozók számára csak a speciális adatátviteli és terminálkezelő utasitások alkalmazása jelentett ujdonságot.

A programok kipróbálását és tesztelését jelentősen megkönnyitette az IDMS OLQ lehetősége.

Segitségével a lekérdező tranzakciók inputjának és eredményeinek tartalmi helyességét tudtuk ellenőrizni. A karbantartási funkciók végrehajtását az OLQ párhuzamos alkalmazásával tudtuk nyomonkövetni.

2. Tapasztalatok

Az IDMS alapú kisérleti on-line rendszer megvalósitásával számos tapasztalatot nyertünk; ezek egy része batch környezetre is érvényes.
Tapasztalataink egyrészt az adatbázis tervezésére és megvalósitására vonatkoznak, másrészt az on-line alkalmazói programokkal /tranzakciókkal/ kapcsolatosak.

Az on-line adatbázis tervezése során az általánosan ismert adatbázis tervezési szempontokon tulmenően figyelembe kell venni az on-line lekérdezési és karbantartási igényeket is.

Az adatbázis tervezésekor élni kell azokkal a lehetőségekkel, amelyek a hatékonyan müködő, optimális válaszidejü on-line alkalmazási programok /tranzakciók/ kialakitását teszik lehetővé; lehetőség szerint minél több belépési pontot határozzunk meg, az adatelérési idő csökkentése érdekében ne féljünk a redundanciától, lehetőleg CALC-osként definiáljuk azokat a rekord-tipusokat, amelyek nagy tömegben fordulnak elő és amelyekre gyakori elérés jellemző /ezáltal a rekordok viszonylag egyenletes szétszórását is biztosithatjuk/.

A CV alkalmazásánál az adatbázis fizikai tervezése során ügyelni kell arra, hogy a CALC kulcsos rekordok feltétlenül szóhatáron kezdődjenek.
Batch rendszer esetén erről az IDMS automatikusan gondoskodik.
Az alkalmazói programokban CV alkalmazásánál ugyancsak a felhasználónak kell gondoskodnia arról, hogy a Communication Block szóhatáron kezdődjön.

Az OLQ alkalmazásához a DMCL modulban speciális bejegyzés megadására van szükség, továbbá az adatbázis lapméretét az OLQ-hoz igazodva kell megválasztani /az OLQ csak 920 byte-os laméretet fogad el/.

Az alkalmazói programok tervezésénél alapelvnek tekinthetjük azt, hogy a szükséges funkciókat több kisebb méretü tranzakció lássa el. Az optimális válaszidő elérése érdekében törekedni kell arra is, hogy a tranzakciók minél egyszerübbek legyenek.
A tranzakciók funkcióját lehetőség szerint ugy kell meghatározni, hogy a szükséges adatok eléréshez az egyes tranzakcióknak ne kelljen az adatbázis összes szintjét bejárniuk és a visszanyert adatokkal ne kelljen hosszadalmas és bonyolult müveleteket végezniük.

Jelenleg még folynak azok a kisérletek, amelyek a kisérleti rendszer számszerü jellemzésére vonatkoznak /válaszidő, adatvolumen, stb./. Ezek eredményeit, valamint az üzemeltetés során felmerült egyéb tapasztalatokat az előadáson fogjuk részletesen ismertetni.

Abstract

The IDMS data base management system can be used for developing both batch and on-line systems. In this paper the on-line facilities of IDMS will be described by means of an experimental on-line application system.

First the on-line environment of IDMS will be shown, in the sequel the application system itself will be presented.

The functions of the on-line system have been implemented in parallel with a batch system already in operation. The data base of the batch system could be used with slight modifications according to the requirements of the on-line environment.

In the paper the data base and the programs /transactions/ of the on-line system are described. At the end of the paper experiences obtained in designing, testing and operating the system are summarised.

Irodalom

[1] Asztalos Domonkosné: IDMS On-Line Query ismertető és felhasználói kézikönyv
SZÁMKI, Budapest, 1981. március

[2] Bidó Zs.,Farkas A.,Major P.; Adatbázis-kezelő rendszerek használata távadatátviteli környezetben
SZÁMKI, Budapest, 1980. december

[3] Kertész E.,Polgár J.: OTAF Törzs
SZÁMKI, Budapest, 1980. október

[4] Polgár J.: OTAF Karb
SZÁMKI Budapest, 1980. december

[5] Major P.: OTAF Kérd
SZÁMKI, Budapest, 1981. február

[6] Bidó Zs.,Major P.: OTAF kisérleti on-line rendszer
SZÁMKI, Budapest, 1981. május

[7] IDMS Concepts & Facilities

[8] IDMS DB Design and Definition Guide
Release 4.5
1977. október

Major Péter, Bidó Zsuzsa, Polgár Judit
Számitógépalkalmazási Kutató Intézet Budapest, I. Csalogány u. 30-32.

Nagy Sándor

MAKROPROCESSZOR ALKALMAZÁSA ADATFELDOLGOZÁSI ÉS EGYÉB FELADATOKRA

Az előadás egy olyan makróprocesszort ismertet, mely PL/I és egyéb magasszintü programnyelven történő programozást támogat.
A processzor és a vele egybeépitett forráskönyvtárkezelő rendszer már célnyelvek definiálására és értelmezésére is képes. A célnyelv hordozója egy makrócsomag, mely az adott feladatot lehető legáltalánosabban végrehajtó programcsaládot definiálja. A makrócsomag alapján a processzor az aktuális paramétereknek megfelelően a konkrét feladatot megoldó programot, és annak jobkörnyezetét hozza létre.
Az aktuális feladat követelményeinek megfelelően a makrócsomag dinamikusan változtatható úgy, hogy más felhasználók számára e változtatások nem láthatók.
Az előadás végül ismerteti a miskolci Lenin Kohászati Müvekben bevezetett - makróprocesszoron alapuló - információ lekérdező rendszert.

Kulcsszavak:makró, információ visszakeresés, forráskönyvtár

1. Bevezetés

Bevezetésképpen néhány szó a makrókról:
" A makró szó a görög makrosz szóból származik, jelentése: nagy. Ezzel a szóval azt a leglényegesebb tulajdonságát akarjuk jelezni, hogy egy makró hivása a gyakorlatban több forrásnyelvi utasitás végrehajtását jelenti, a makró feladatának és aktuális paramétereinek megfelelően. Tehát egy forrásnyelvi utasitás helyét több utasitás foglalja le.

...A makró nyelvvel a gyakran előforduló, meghatározott utasitássorokat lehet megirni és forditáskor a programba egyszerüen beilleszteni."
A fenti sorokat /1/-ből idéztem.
Sajnos ezideig DOS operációs rendszer alatt csak az Assembler nyelvhez kifejlesztett makrógenerátor terjedt el széleskörüen, innen származott az indittatás: létrehozni egy makrónyelvet, s az azt realizáló makrógenerátort, egy tetszőleges magas szintü programnyelv használatának támogatására.
Az előadás három részre tagolódik, először a makrónyelvet, majd a forráskönyvtár kezelő rendszert ismertetem, végül egy gyakorlati példán látjuk a processzor használhatóságát.

A makrónyelvről

A makrónyelv konstruálásakor a következő szempontokat tartottuk szem előtt:

- a nyelv ne kötődjön egyetlen forrásnyelvhez se,
- a makrók tárolására könyvtárat lehessen használni.

Mindebből az alábbi következtetések vonhatók le:

- a processzor nem lesz összeépitve egyetlen forditóprogrammal sem, tehát preprocesszorként fog müködni,
- igy nem csak forrásnyelvi, hanem job-kontroll utasitások is generálhatók segitségével,
- a könyvtár felhasználásával a processzor alkalmassá tehető célnyelv definiálására, melyet a könyvtárba katalogizált makrók hordoznak,
- mindezt összefoglalva: egy-egy makróhivás segitségével komplett, feladatorientált feldolgozó job-ok hozhatók létre, igy az idézetben emlitett helyettesitést a lehető legmagasabb szinten valósithatjuk meg.

A processzor által realizált nyelv a DOS MAKRÓ-ASSEMBLER nyelvének /1/ részhalmaza néhány formális eltéréstől elte-

kintve. Az eltérések lényeges oka az, hogy a processzort a - később ismertetendő - feladat megoldásának eszközéül hoztuk létre elsődlegesen, igy nem volt sem szükség, sem idő valamely meglevő makrónyelv felkutatására és implementálására, mert a MAKRÓ-ASSEMBLER-ből átvett alapvető funkciókkal a kitüzött eredeti feladat megoldható volt.

A makrónyelv szintaktikai egységei igy: formális paraméter, aktuális paraméter, makróváltozó /olyan, szimbolikus névvel meghatározott programozási eszköz, melynek helyére a generálás során mindig az ahhoz rendelt aktuális érték kerül/, makrócim /szimbolikus névvel jelzi a generálás elágazásait/, rendszerindex /a makrógenerálások általános sorszámát, indexét hordozó rögzitett nevü makróváltozó/, belső makró /makróhiváson belüli makróhivás/, valamint aritmetikai értékadó utasitás, feltételes és feltétel nélküli ugróutasitás.

Nézzük most mindazt, amit a makrógenerátor tud:

- aktuális-formális paraméter megfeleltetés
- a belső makrók 8 szintig egymásbaágyazhatók
- külső makróból a belső makró felé az információátadá biztositható, amennyiben a belső makró aktuális paramétere a külső makró formális paramétere, makróváltozója
- a belső makró neve lehet formális paraméter
- a generált output létrehozása vezérelhető ugróutasitások segitségével

A forráskönyvtárkezelő rendszer

A könyvtár szervezése több dologban is különbözik a DOS SL-től: több azonos nevü makró is található időben egyszerre a könyvtárban, melyekre nézve az LIFO stackként müködik, azaz mindig csak az időben utoljára beirt makrót lehet feldolgozni, törölni, vagy módositani. Némileg hasonlóan a CL müködéséhez, a felhasználói makrók csak a makrógenerátor

aktuális futásakor találhatók a könyvtárban, utána automatikusan törlődnek.
Mivel a felhasználói makrók mindig az időben utoljára beirt makrók, igy azonos nevü felhasználói és rendszermakró esetén mindig az előbbi az "érvényes".
Ha a makrógenerátorra mint valamely célnyelv definiálásának és megvalósitásának eszközére gondolunk, akkor mindez a következőt jelenti:

- Definiáljunk egy adott feladatcsalád megoldására célnyelvet rendszermakrók formájában. Igy tehát előáll egy egymás alá- és mellérendelt makrókból álló struktura, ahol minden makrónév egy standard tevékenységet ellátó programrészlet családot takar.
- Egy konkrét feladat megoldása esetén előfordulhat, hogy e célnyelv nem teljesen alkalmas, valamely makró által generált programrészlet nem felel meg az aktuális feladat kivánalmainak.
- Ekkor a megfelelő makrónéven felhasználói makrót kell irnunk, melyben már a feladathoz igazitott tevékenységet ellátó programrészlet szerepel. A processzor a makródefinicióval kiegészitett célnyelvi program "leforditásakor" a módositásnak megfelelő szöveget fog generálni.
- A felhasználói és rendszermakrók közötti különbségtételből adódóan igy csak az aktuális futás idejére módositottuk a célnyelv definicióját a saját igényeinknek megfelelően, és mindezt anélkül, hogy a célnyelv más felhasználó felé változott volna, vagy a változtatásról tudomást szerezne az illető felhasználó.

Igy a processzor+forráskönyvtár felhasználásával igen rugalmas célnyelv definiálási eszközt kaptunk a kezünkbe.
Térjünk vissza a könyvtárhoz. Mivel az forráskönyvtár is, igy megvannak az annak megfelelő szolgáltatásai, röviden:

- az egyszer már katalogizált makrót módosithatjuk a szokásos módon /rekordok törléše, beszurása, lecserélése/
- a rendszermakrók törölhetők: törlés után a lefoglalt

hely azonnal felszabadul, azaz a hagyományos értelemben vett könyvtársürítésre nincs szükség,
- a rendszermakrók opcionálisan védhetők az elrontás ellen, a védelmi jelszó dinamikusan változtatható.
- Külön utasitás végzi a könyvtár megformázását, kimentését, valamint a tulcsordulási területre került rekordok adatterületre fésülését. Ez utóbbit megelőzően automatikus kimentés történik.
- Egyéb szolgáltatások:
tartalomjegyzék listázás - ha kérjük a beépitett kommentek is megjeleníthetők,
forráslisták készitése - a makrónév megadásakor alkalmazhatunk maszk-karaktert, igy egy utasitással makrócsaládokról készül/het/nek forráslisták,
nyomkövetés - makróhivás esetén kérhető a makró által generált utasitások, a belső makrók, s az előfordult aktuális paraméterek áttekinthető formáju nyomtatása.

Most pár szót a program működéséről. A program IBM DOS 26.2 POWER alatt fut egy R-22-n. Inputja egy kártyafájl, mely makródefiníciókat, könyvtárvezérlő utasitásokat, valamint forrássorokat s köztük makróhivásokat tartalmazhat. Outputja egy lemezfájl, mely a forrássorokat - az esetleges előtét karakterek nélkül -, valamint a makróhivások helyén azok kifejtését tartalmazza. E fájlt SYSIN rendszerfájlnak kijelölve használhatjuk fel a generált forrássorokat.

A makróprocesszor gyakorlati alkalmazása, példa célnyelv definiálására

A kokrét feladat: a miskolci Lenin Kohászati Müvekben egy információszolgáltató rendszer létrehozása.
A rendszer a gyár személyi adattárán alapul, mely kb. 18 ooo rekordot, rekordonként 12oo byte-on 74-féle adatot tartalmaz. A probléma az, hogy az információigények tete-

mes része előre nem meghatározható, nagy gyakorisággal lép fel, s rövid időn belül kell kielégíteni.
Tekintsünk egy egyszerű példát: a személyzeti osztálynak össze kell gyűjteni az öt éven belül nyugdíjba menő dolgozókat, és a listát születési idő, nem, név szerint rendezve kell nyomtatni, de dolgozónként elég a név, nem, születési idő és a konkrét munkahely megjelölése.
A feladatot végrehajtó job a következőképpen épülhet fel:

FORDÍTÓ
    az érintett részfájlt leszedő forrásprogram
SZERKESZTŐ
    munkaerő adattárak
LEFORDÍTOTT PROGRAM
    az öt éven belül nyugdíjba menők kulcsai + rendezési adatok rekordonként
SORT
    az előző kulcsfájl rendezetten
    munkaerő adattárak
LISTÁZÓ PROGRAM
    a kívánt lista

Általában ilyen szerkezetű job jön létre, s az esetek többségében mindez két makróhívással generálható.

A listázási feladatokat egy interpreterrel valósítjuk meg, melyet a munkaerő törzstárakhoz hoztunk létre, de a processzorral tetszőleges törzstárra adaptálható.
A program standard formátumú listát állít elő egy input kulcsfájl, és vezérlőutasítások alapján. Lehetőségek:

- a kulcsfájl lehet lemez, kártya vagy mindkettő
- a listázandó kódok halmaza a kulcsfájl minden elemére külön-külön is megadható
- a kulcsfájl implicit módon is definiálható vezérlőkártyán, ahol megadható a vizsgálandó adat kódszáma vagy mnemonikja, valamint egy érték, mellyel történő egyezés esetén az adott rekordot listázásra kijelöljük.

Az érték tartalmazhat maszk-karaktert.
- A program interaktiv üzemmóddal is rendelkezik.

A célnyelvet definiáló makrócsomag ismertetése

Tekintsünk egy input adattárat szekvenciálisan feldolgozó PL/1 progam szerkezetét. /A célnyelv PL/1 bázisu./

```
job indító, fordítást vezérlő JC utasitások
program: deklarációk *
         fájl megnyitások
         egyéb inicializálások
         ciklus egy rekord feldolgozására
           egy rekord beolvasása
           feldolgozást megelőző tevékenységek *
           egy rekord feldolgozása *
         ciklus vége
         fájl lezárások
         a fájl feldolgozása utáni tevékenységek *
program vége
futáshoz szükséges JC utasitások
opcionális rendező job-lépés *
```

Ez az alapmakró standard felépitése, a *-gal jelölt részeket belső makrók generálják a kapott paraméterek függvényében. Az alapmakró hivása az aktuális paramétereikben definiált feladatu job generálását eredményezi.

A feladat tehát az, hogy valamely logikai feltételnek elegettevő kulcsfájlt, valamint a rendezéshez szükséges adatokat előállitó programot generáló makrócsomagot irjunk. Igy az egy rekord feldolgozását előállitó belső makrónak

```
IF a kulcsfájlt definiáló feltétel THEN ird a kulcs-
   fájlba a kulcsot és a rendezési adatokat;
```

utasitást kell generálnia.
Itt akadályba ütközünk. A konkrét adatfájl rekordjainak felépitése olyan, hogy bizonyos adatokból /iskolai végzettség, szakképzettség stb./ többet is tartalmazhat, s igy

ezeknél a fenti utasitást egy a tömbelemeken végiglépő ciklusba kell beépiteni.
E probléma megoldására vezettük be a támogatott és nem támogatott módu felhasználást.
Támogatott módban - azaz pusztán a rendszermakrók felhasználásával - csak olyan kulcsfájlt állithatunk elő, amelyet definiáló feltétel nem hivatkozik tömb tipusu adatra.
Nem támogatott módban általában egy makrót kell definiálni ujra, azaz a megfelelő névvel felhasználói makrót kell irni.
A makrócsomag nagy része ezen ujrairást támogatja olyannyira, hogy segitségével már tetszőleges logikai feltételnek eleget tevő kulcsfájl előállittatható.
A makrócsomag annyira széleskörü, hogy sok, támogatott módban nem realizálható feladat pusztán egyetlen makró egy soros ujradefiniálásával megoldható.
A nem támogatott módnak van egy speciális esete, amikor nem tételes listákat, hanem táblázatokat produkáló programokat kell generálni. Ennek elvi megoldása az, hogy a listázandó táblázat-mint mátrix-sorainak és oszlopainak megcimzéséhez bevezetünk egy-egy esetleg összetett függvényeljárást, melyek hivásával egy

```
GYÜJTŐ(sorindex,oszlopindex) = GYÜJTŐ(sorindex,oszlop-
        index)+ 1;
```

tipusu számlálás az adott rekordot a táblázat megfelelő helyére sorolja, ahol GYÜJTŐ tartalmazza a nyomtatandó táblázat elemeit. A makrócsomagban megtalálhatjuk az eddig alkalmazott összes bontást megvalósitó függvényeljárást, paraméterezhető makró formájában. Pl. az idő szerinti bontást megvalósitó makró esetén a makróparaméterek az időintervallumok határait adják meg.

Jogosan merülhet fel a kérdés: Miért jobb ez, mint pl. egy jól paraméterezhető adatfeldolgozási programcsomag?
Erre a következőket lehet válaszolni:

- a processzor a standard fájlkezelő rendszerre épül, igy az általa kezelt adatiárak más programból illesztés nélkül használhatók
- a legjobban paraméterezhető programcsomag is egy előre lerögzitett korlátok közötti feladatot képes végrehajtani, mig a makrócsomag dinamikusan és könnyen módositható.
- a makrócsomag által definiált célnyelv nem csak adatfeldolgozási feladatok megoldására lehet alkalmas
- végül, de nem utolsó sorban a processzor természetesen az alkalmi programozói munkában is hatékony munkaeszköz. Mivel az output kártyaformátumu, igy a könyvtár használata /DOS alatt/ mentesiti a programozót a programkártyákban történő javitástól, hatékony pontos és "visszacsinálható" javitási eszközt adva a kezébe.

Legvégül néhány szót a tapasztalatokról.
Az itt ismertetett rendszer 1980. februárja óta üzemel rendszeresen a Lenin Kohászati Müvekben, s a tapasztalatok egyértelmüen a rendszer alkalmazhatósága mellett szólnak.
Hagyományos módu igénykielégités esetén az átfutási idő 3-4 naptól 2 hétig terjedt, mig az uj rendszer alkalmazása mellett ez legfeljebb egy müszak.
A legbonyolultabb feladat egy vizszintesen és függőlegesen 5 bontási szintet tartalmazó táblázat elkészitése volt, amit az utoljára emlitett makrók segitségével kb. 3 óra alatt sikerült megoldani.
Támogatott üzemmód esetén a kivánt program generálása, futása valamint az interpreter futása mintegy 15 percet vesz igénybe + a lista nyomtatása POWER alatt.
Mivel a makrók előre letesztelt programrészleteket tartalmaznak, generálást csak paraméterezési hiba miatt kellett megismételni.
A processzor és az interpreter PL/1 nyelven iródott, igy más gépen történő üzemeltetésnek ez nem akadálya.

Abstract

The lecture makes you acquainted with such a macro processor which helps the programming in PL/1 and other high-level programming languages.
The processor and the source program library management system - which is built together the processor - are also capable of defining and interpreting purpose languages. The carrier of the purpose language is a macro package defining the program family which executes the given work most generally. On the basis of the macro package according to the current parameters the processor creates the program - which solves the concrete work - and the job environment of it.
According to the requirements of the current work the macro package may be dynamically extended so, that these extensions can not be seen for other users.
In the end the lecture makes you acquainted with the information interrogation system - based upon the macro processor - which is initiated in Lenin Metallurgical Works of Miskolc.

Irodalomjegyzék:

A makrónyelvhez és a generátorhoz:

/1/ Sepródi László: Makró. Műszaki Könyvkiadó, 1978.

/2/ John J. Donovan: Rendszerprogramozás. Műszaki Könyvkiadó, 1977.

/3/ Varga László: Rendszerprogramozás. Tankönyvkiadó Bp., 1975. kézirat

A forráskönyvtárkezelő rendszerhez:

/4/ SLICK forráskönyvtárkezelő rendszer. Felhasználói dokumentáció.

Szerző: Nagy Sándor okl. mat. Miskolc, Park u 17. 3529.

Páldi Vince

TSO FEJLESZTÉSEK A KSH SZÁMÍTÓKÖZPONTBAN

Rövid kivonat:

Az OS tipusu operációs rendszerek legelterjedtebb interaktiv alrendszere a TSO /Time Sharing Option/. Az előadás bemutatja azt a fejlesztési munkát, amelyet a KSH SZK-ban az IBM SVS TSO lehetőségeinek kibővitésében végeztünk.

Kulcsszavak:

SVS /Single Virtual Storage/, TSO /Time Sharing Option/, interaktiv munkavégzés, utasitás feldolgozó, előtér.

Bevezetés

A KSH SZK-ban 1979. szeptembere óta használjuk az IBM SVS TSO rendszerét. A nagyobb ESZR gépek elterjedésével várhatóan megnövekedik az ennek megfelelő ESZ operációs rendszer és TSO felhasználók száma, akik valószinüleg átmennek ugyanazokon a fejlődési- fejlesztési szakaszokon, amelyeken mi is átmentünk. Ezért az előadás a jelenlegi TSO felhasználókon kivül azokhoz is szól, akik csak később akarnak áttérni erre az interaktiv technikára.

A TSO - mint minden szoftver - legnagyobb hiányosságai csak használat közben derültek ki. Meg kell azonban jegyezni, hogy ezek nem müködésbeli hibák voltak, hanem olyan feladattipusok, amelyekre az IBM rendszere nem, vagy csak nehézkesen, több lépcsőben adott megoldást. A probléma eredete az, hogy az SVS alapvetően batch szemléletü operációs rendszer /különösen HASP-pal/, és ehhez a környezethez készült az interaktiv TSO. Munkánk egyik fő feladata ennek az ellentmondásnak a feloldása volt. Ezen kivül létrehoztunk számos, az IBM alaprendszeréből hiányzó, jellegzetesen interaktiv eszközt is a terminálóknál végzett munka megkönnyitésére, hatékonyabbá tételére.

Az előadás nem tartalmazza az egyes fejlesztések részletes elemzését, inkább a várható problémákra, a rendszerrel szemben támasztott felhasználói igényekre ad utalást, röviden bemutatja a feladatok egy lehetséges megoldását és az azokhoz felhasznált eszközöket.

1. Batch vagy TSO? Válaszuton a felhasználó

A rendszeres TSO üzem beinditásakor csak 6-8 felhasználó /fejlesztők és néhány vállalkozó szellemü programozó/ dolgozott az uj munkaeszközzel. A kezdeti idegenkedéstől ma eljutottunk odáig, hogy terminálparkunk /22 db/ és számító-

gépünk átbocsájtóképessége maximálja az egy időben dolgozó TSO felhasználók számát. A rendszeresen terminálnál dolgozók száma 50-re tehető, azaz átlagban minden második programozó és fejlesztő elsajátitotta az uj technikát.

A TSO ilyen mérvü elterjedése a korábban tisztán batch felhasználói környezetben nem magyarázható csak a szorosabb ember-gép kapcsolattal. Ez hiába adott, ha a programozó a batch feldolgozás nyujtotta előnyöket többre becsüli, és különösen igaza van akkor, ha a batch számos funkciója közül csak néhányat lát vissza a terminálok képernyőjén.

Viszonylag hamar, már a kisérleti üzem során felismertük, hogy alternativát csak ugy tudunk biztositani a hagyományos feldolgozási módszerrel szemben, ha
- annak rutinszerü feladatait TSO környezetben is könnyen elérhetővé, megoldhatóvá tesszük.
- kiemeljük az interaktiv munkavégzés előnyeit, biztositjuk a rövid válaszidőt, komplex feladatok megoldására is alkalmas szoftvert hozunk létre.
- olyan eszközöket állitunk üzembe, amelyek batch környezetből nem használhatóak.
- olyan üzemrendet alakitunk ki, amelyikben a nap egy meghatározott időtartama alatt a TSO munka az elsődleges.

A fejlesztés másik fő mozgatója az volt, hogy megszüntessük az akkor már kialakult batch feldolgozási rendszer és a TSO munkavégzés közötti ellentmondásokat, interface-t biztositsunk a batch erőforrásokhoz.
- Biztositani kellett a közös könyvtárak párhuzamos beirás elleni védelmét ügyelve arra, hogy a reteszelődés csak a minimálisan szükséges ideig tartson.
- Automatikussá kellett tenni a nyomtatott outputok készitését.

- Párhuzamossá tettük a terminálon folyó munkavégzést és a forditásokat.
- Meggyorsitottuk a job inditás folyamatát és biztositottuk a jobok sorsának állandó követését.

2. Házi fejlesztéseink

A TSO környezetben dolgozó felhasználó a rendszerrel utasitásfeldolgozókon /command processor/ keresztül tartja a kapcsolatot. A rendszerben meglévő lehetőségeket, funkciókat a felhasználó egy megadott szintaktikáju utasitás bevitelével veheti igénybe.

Házi fejlesztéseinket nyomonkövetve jól látható tendenciát vehetünk észre. Kezdetben csak egyszerü utasitásfeldolgozókat készitettünk, később bonyolultabb, az alapszoftver és a hardver lehetőségeit jobban kihasználó rendszereket hoztunk létre.

2.1. Egyszerü utasitásfeldolgozók

Ezeknek a programoknak a megirását két tényező motiválta. Egyrészt kaput akartunk nyitni a rendszerben meglévő, de a TSO által nem, vagy csak alig támogatott szolgáltatások felé, másrészt interface-t kellett biztositani a meglévő háziszabványokhoz. Ezeket a célokat részben uj programok irásával, részben a meglévő modulok kiegészitésével értük el. Az általunk megirt legfontosabb funkciók az alábbiak:

- Particionált állományok védelme a párhuzamos beirások ellen. Itt az évek óta a batch környezetben jól müködő ENQ/DEQ /sorbaállitási/ konvenciót használtuk fel a LINK utasitás kiegészitéséhez. A forrásprogramokat tartalmazó könyvtárak módositó utasitás /RESTORE/ a fenti konvención kivül a BÉTA rendszerhez is illeszkedik.

- Kevésnek, illetve nehézkesnek bizonyultak az IBM TSO listázó lehetőségei is. Ezért készültek el a rendszerkatalógus illetve a tetszőleges particionált állomány directory tartalmát maszk segitségével leválogató, listázó utasitások. Állományok tartalmának kiiratására egy sokrétüen paraméterezhető listázó utasitást készitettünk.
- Megoldottuk egy tetszés szerinti állomány OS SYSOUT /nyomtatott output/ állományba másolását, a megirt SYSOUT file elengedését.
- Lehetővé vált a TSO környezetben allokált állományok konkatenálása és a konkatenálás felbontása.
- Elkészült az OS batch utility programokat TSO alatt futtató prompter. Ez gondoskodik az állományok megfelelő allokálásáról, megteremti a választott utility program futtatásához szükséges környezetet és felhivja azt. A futás végeztével elengedi az állományokat és értesiti a felhasználót a visszatérési kódról.
- Lehetőséget biztositottunk a felhasználónak arra, hogy nyomon követhessék job-jaik sorsát, a várakozó sorok állapotát és a HASP initiatorok állását. Az SMF exitek segitségével a TSO-ból inditott batch jobok egyes lépéseinek visszatérési kódját is elküldjük a terminálnál ülő felhasználónak. Jelenleg a job inditás folyamatának meggyorsitásán dolgozunk.
- A TSO EDIT /szerkesztő/ utasitását kiegészitettük a szerkesztett állomány részeit mozgató funkciókkal.

2.2. A kifejlesztett rendszerek

A TSO fejlesztési munkák második csoportjába a lényegesen összetettebb, rendszermódositásokat és kiegészitéseket igénylő fejlesztéseket sorolhatjuk. Ezekkel a rendszerekkel is a hozzájuk csatlakozó utasitásfeldolgozókon keresztül lehet kommunikálni. Az eredeti rendszerben meg nem lévő, általunk létrehozott uj egységek a következők:

- Batch monitort készitettünk a fordítóprogramok számára. /Lásd Szigetvári Miklós előadását/
- Megvalósitottuk a terminál I/O müveletek rögzitését, igy a végzett munka bármikor reprodukálható. /Lásd Hámori István előadását/
- Elkészült egy interaktiv BASIC interpreter is. Az interpreter tudja a BASIC alaputasitásait, függvényeit és a közvetlen végrehajtásu utasitások legnagyobb részét is.

Külön kell szólni a nyomtatott outputok kezeléséről is. SVS HASP környezetben a lassu perifériákat, igy a nyomtatókat is a HASP kezeli. A TSO-nak sajnos csak egy interface-e van a HASP felé a megirt jobok átküldésére, ezért meg kellett irni azokat a programokat, amelyek a batch környezetben futva átmásolják a TSO munkavégzés során keletkezett nyomtatandó outputokat a HASP spoolra. Igy a felhasználónak a korábban emlitett utasitásfeldolgozók segitségével létrehozott nyomtatandó állományaival nem kell tovább törődnie, azok operátori beavatkozásra tetszés szerinti időben, vagy a TSO üzem végén automatikusan a printeren megjelennek.

2.3. A terminál hardver kihasználása

A KSH SZK-ban használt terminálok többsége IBM 3270 tipusu display, vagy azzal kompatibilis VDDS és VDT terminál. Ezeknek a berendezéseknek jellegzetessége a hagyományos klaviatura mellett létező 12, ugynevezett PFK /Program Function Key/ billentyü. Ezeknek használatát az IBM alap TSO rendszere nem támogatja.

A TSO környezetben végzett rendszeres munka felvetette az igényt a rutinmüveletek automatizálására, a terminálnál eltöltött holtidők csökkentésére. Ebben segitett a PFK billentyük programozhatóvá tétele. A rendszer átlátszó a felhasználók szempontjából, aki nem akarja használni, annak tudomást sem kell vennie róla.

Az egyes PFK billentyük lenyomásával egy TSO utasitássor begépelésével azonos értékü jelsorozat jön létre, ami a definició tipusától függően azonnal vagy módositás után hajtódik végre. A billentyük jelentése más és más lehet pl. TEST és EDIT környezetben. A PFK-k jelentése, azaz a lenyomásukkal létrehozni kivánt jelsorozat tartalma tetszés szerint változtatható a klaviaturáról. A szoftver lehetőséget ad arra is, hogy a billentyük definicióját a felhasználók egy lemezes állományból tölthessék be. Igy mindenki tetszés szerint definiálhatja a neki tetsző PFK jelsorozatokat és minden TSO munkája során állandó, a saját feladatainak legjobban megfelelő definició sorozattal dolgozhat.

3. A fejlesztés eszközei

Az IBM a TSO fejlesztésekor felismerte, hogy mik azok a közös, várhatóan saját laboratóriumaiban is előforduló feladatok, amelyeket célszerü az utasitásfeldolgozókon kivül megoldani. Ezek közül a legfontosabbak:

- állományok dinamikus allokálása
- szintaktikai elemzés
- terminál I/O kezelés, üzenetformázás.

A fentiekben emlitett funkciókat a megfelelő szabályok betartásával, adatterületek, paraméterlisták felépitése után a LINK folyamattal lehet meghivni. A szervizprogramok szolgáltatásainak könnyü elérését számos rendszermakró teszi lehetővé.

Munkánkat nagyban segitette, hogy a TSO szervizprogramjairól nagyon jó dokumentáció állt a rendelkezésünkre. Nehezitette a feladatot az, hogy a TSO magasabb szintjeiről és a TSO-TCAM /Telecommunication Access Method/ kapcsolatáról semmi anyagunk nem volt, és még ma sincs.

3.1. A dinamikus allokálás interface programja /IKJDAIR/

Segitségével lemezes állományokat lehet létrehozni, allokálni, elengedni. A dinamikus allokálás a terminál képernyőjét is állománynak tekinti. A képernyő egy időben több, eltérő attributumu input és output állományt is reprezentálhat. Az IKJDAIR gyakorlatilag az OS JCL DD utasitásának feladatait látja el. Sok funkciója TSO utasitásokon keresztül is elérhető.

3.2. Általános szintaktikai elemző program /IKJPARS/

Használatával rendkivül egyszerü a TSO utasitások különböző tipusu paramétereinek, operandusainak leválasztása. Az utasitás formátumát egy szintaxis leiró makrórendszerrel adhatjuk meg. Lehetőség van többszintü paraméterezés és előre ismeretlen számu elemből álló operanduslista elemzésére is.

3.3. Terminál I/O szerviz /IKJPTGT/

Ez a rendszerprogram a TSO modulok általános terminál I/O programja. A terminál adatforgalom lebonyolitásán kivül számos egyéb feladatot is elvégez, formázza az üzeneteket amelyek többszintüek is lehetnek, prompt üzenetet küld, tudósit a munkavégzés környezetéről /mode message/.

Másik fontos funkciója a terminál I/O szimulálása. Igy az egyes utasitásfeldolgozók meghivása, vezérlése nemcsak a terminálról történhet, hanem a memóriában lévő utasitássorozat segitségével is megvalósitható.

4. A legfontosabb: a rövid válaszidő

A terminálhoz a legvérmesebb reményekkel leülő felhasználót is elriasztja a TSO-tól az, ha a legegyszerübb feladat végrehajtására perceket kell várnia. Ennél is nagyobb hiba, ha

input módban kell sokat várakozni két sor bevitele között. Az interaktiv munkavégzés legfontosabb feltétele tehát a megfelelő válaszidő biztositása.

A válaszidőket nagymértékben befolyásolja a batch és a TSO terhelés aránya. Az SVS rendszer biztositja azt a lehetőséget, hogy fokozatosan "lehalkitsuk" a batch üzemet a TSO felhasználók bejelentkezésével arányban. Fel kell hivni a figyelmet a rendszer operátorának nagyon fontos szerepére, aki a terhelés nem megfelelő megválasztásával megkeseritheti a terminálnál folyó munkavégzést.

4.1. A modulelérés meggyorsitása

Az operációs rendszer és a TSO egymás mellett élő lapozási algoritmusainak /page és swap/ vizsgálatából kitünt, hogy megfelelő sebességü munkavégzésre csak ugy számithatunk, ha a két technikát összehangoljuk, a feladatokat megosztjuk közöttük. Méréseink alapján ugy döntöttünk, hogy a gyakran használt TSO modulokat /utasitásfeldolgozókat, szervizprogramokat és vezérrutinokat/ a rendszer LPA területén, a virtuális memóriában rezidens módon helyezzük el. Ennek az elrendezésnek számos előnye van:

- A programok csak az SVS lapozásának hatáskörébe tartoznak, a felhasználói region mozgatása azokat nem érinti.
- A modulok elérése rendkivül gyors.
- Az LPA-n lévő programok a rendszerben csak egy példányban vannak jelen. Futásuk során méretükkel nem csökkentik a felhasználóhoz rendelt memóriaterületet.
- A szervizprogramokat nem kell összeszerkeszteni az utasitásfeldolgozókkal.

Az SVS lehetőséget ad arra is, hogy az egyes modulok LPA-n való elhelyezkedését vezérelhessük. Igy azonos lapra helyeztük az egymásra hivatkozó programokat és a sürün egymás mellett használt rutinokat is.

Befejezés

Fejlesztési munkánk során, ugy érezzük, hogy a reális felhasználói igények kielégitését tartottuk szem előtt. Természetesen ennek megfelelően mindig ujratermelődnek az igények, uj elvárások alakulnak ki, a fejlesztési folyamat sohasem állhat meg, jelenleg is ujabb rendszereken dolgozunk.

A programok irása során törekedtünk arra, hogy azok lehetőleg rendszerfüggetlenek /a pillanatnyi installáció specifikumaitól függetlenek/ legyenek. Igy az eddigi release váltások során sem kellett módositani, és valószinüleg egy esetleges MVS átállás során sem kell moduljainkat teljesen átirni.

Érdekességként meg kell jegyezni, hogy szinte valamennyi általunk megirt utasitásfeldolgozó és TSO segédrendszer szerepel az IBM vagy valamelyik szoftverház bérelhető vagy megvásárolható termékeinek katalógusában.

Abstract

TSO /Time Sharing Option/, the most common interactive subsystem of the IBM OS-type operating systems was implemented in the Central Statistical Office Computing Centre in 1979. The development work to expand the facilities of SVS TSO is described in this paper.

Páldi Vince
KSH Számitóközpont
1024 Budapest
Budai László u. 1-3.

Pálvölgyi László

AZ INTERCELLAS SEJTTÉRSZIMULÁCIÓS NYELV

A sejtprocesszorok kutatásának mai szintjén szükségessé vált egy interaktiv sejttérszimulációs nyelv létrehozása. A korábbi szimulációs lehetőséget nagygépen batch módban futtatható /CELLAS [1]/ vagy speciális célu /[6]/ szimulációs nyelvek biztositották.

1976-ban specifikáltuk az INTERCELLAS nyelv TPA-i implementációjának utasitáskészletét és szintaktikáját /[2]/. Az INTERCELLAS általános /változtatható a tér mérete, az átmeneti függvény, a sejtállapotok száma/ és inhomogén sejtteret szimuláló, interaktiv módon kezelhető sejttérszimulációs nyelv.
A nyelv első 1.0 jelzetü változata 1977-ben készült el. Jelenleg az 1.1 változata kerül használatba, amelyhez önálló alrendszerként függvénydefiniáló nyelv /DEFFUNC 1.1/ is tartozik /ld. [2]/. A nyelv egy speciális változata az 1.16 jelölésü, amely a jelenleg fejlesztés alatt álló 16x16 sejt méretü sejtteret emuláló sejtprocesszor modell müködését szimulálja a célnak megfelelően módositott utasitások segitségével /[3], [12]/.

Kulcsszavak: Szimulációs nyelv, sejttér, sejtprocesszor, INTERCELLAS.

Általános ismertetés

A TPA-i számitógépre a kisgép következő előnyös tulajdonságai miatt esett a választás:

- Interaktiv lehetőség
- Gyakori hozzáférés a géphez /több helyen is rendelkezésre áll./

- Rugalmas fejlesztési lehetőségek.

Az INTERCELLAS rendszerek SLANG1 assembly nyelven iródtak. Erre elsősorban a kisgép jó memória kihasználása, és a sejtter billenési idejének minimálisra csökkentése érdekében volt szükség. A programok terjedelme egyenként* $\sim$ 5 kszó. /12 bites szóhossz./ A memóriaigény min. 16K. A forrásprogramok összesen mintegy 10000 assembly sor hosszuak.

Szimuláció

A nyelv célja a sejtprocesszorok sejtmezője müködésének szimulációja /1.16 esetében sejtprocesszor architektura szimuláció/.

A sejtteret alkotó sejtek un. Neumann szomszédsági kapcsolatban állnak egymással, vagyis ha a sejteket egy négyzetrács pontjaiba képzeljük, minden sejt a négy közvetlen szomszédjával áll összeköttetésben. /Átlós irányu szomszéd nincs./ A sejtek uj állapotukat önmaguk és négy szomszédjuk állapotának függvényében veszik fel egyszerre. Ezt nevezzük billenésnek. Az uj állapotot a lokális átmeneti függvény határozza meg.

A szimulált sejttér téglalap alaku, méreteit a program felhasználója szabadon választhatja, a sejt állapotok száma max. 32. A szélső sejtek szomszédságát un. bábsejtek alkotják, amelyek nem billennek és állapotuk beállitható.

A szimulált tér maximális méretét $(M+2)(N+2) \leq 4096(K-3)$ határozza meg, ahol MxN a sejttér mérete bábsejtek nélkül, K a memóriamodulok száma. /1 modul 4K szót tartalmaz./ Azaz minimális /16K/ operativ memóriáju gép esetén bábsejtekkel

---

* INTERCELLAS 1.1 szimulációs nyelv, DEFFUNC 1.1 függvénydefiniáló és INTERCELLAS 1.16.

együtt max. 64x64 méretü tér szimulálható.

A sejttér különböző mezőiben elhelyezkedő sejtek más-más lokális átmeneti függvény szerint billenhetnek, igy lehetőség van statikusan inhomogén sejttér szimulációjára. A szimuláció alapproblémája az, hogy párhuzamos müködésü rendszert kell soros müködésü rendszerrel szimulálni, mig a sejttérben valemennyi sejt müködése közös órajelre történik. Ezért a szimulált teret kétszer kell tárolni /régi és uj állapotok szerint/. A szimuláció során a sejtek billentése az inhomogenitások figyelembe vételével sorosan történik. Ha a szimuláció során egy sejt értéke biztosan nem változik, - ez akkor áll elő amikor a kérdéses sejt és szomszédai nem vettek fel uj állapotot - zárt jelzetet kap, egyéb esetben nyilt lesz. A sejttér billentése folyamán a zárt sejteket figyelmen kivül lehet hagyni, igy a szimuláció sebessége növelhető. 62x62 sejttérméret esetén az aktiv /nyilt/ sejtek arányától függően a tér billentésének ideje 0,5-3 sec. között változhat a tapasztalatok szerint.

Az átmeneti függvény leggyakrabban minimizált fastrukturáju táblázatban van tárolva. Igy kicsi a visszakeresési idő szórása /nem erősen környezetfüggő/. Az inhomogén függvény szerinti billentés szabta követelményeknek is leginkább a táblázatos tárolás felel meg.

Interaktiv szolgáltatások

A nyelv és processzora az interaktivitás követelményeinek maximális figyelembevételével készült, de batch futtatásra is alkalmas. Korlátozott lehetőségeket nyujt a sejttér müködésének a sejtállapotokon keresztüli automatikus figyelésére /On, SKIF utasitások /[2] /. Az előirt feltételek teljesülése esetére programelágaztatás vagy program input periféria váltás irható elő.

A függvénydefiniálás vagy szimuláció folyamán előforduló adatforgalmi hibák /pl. táblázat megtelése/, vagy beavatkozást kivánó feltételek teljesülése esetén a megfelelő információk kiirása után a vezérlés átadódik a program futtatójának. Interaktiv szimuláció esetén a szintaktikus hibák azonnal kijelzésre kerülnek, a hibás rész automatikusan törlődik, továbbá speciális törlő karakterek felhasználásával egyéb javitások is végezhetők. A batch futás során fellépő szintaktikus hibákat szintén azonnal kijelzi a processzor, és interaktiv beavatkozást vár. A batch futás csak a hibák javitása után folytatódhat.

A program futás közbeni módositására ad lehetőséget az interrupt rendszer. Speciális karakterek segitségével a batch futás vagy a sejttér billentési folyamata megszakitható, és a szükséges utasitások kiadása után /pl. megjelenitésre vonatkozó utasitás/ folytatható a szimuláció, illetve kijelölhető az eredeti input periféria.

Processzor-periféria kapcsolat

Az INTERCELLAS 1.1 programban négyféle processzor-periféria kapcsolatot különböztetünk meg.

A; Utasitások segitségével kijelölhető:

1; Inputperiféria: Az a periféria, melyről a processzor az utasitásokat és paramétereket várja.

2; Program-output periféria: Az a periféria, /azok a perifériák/ amely/ek/re a szintaktikusan helyes, elfogadott utasitások, paraméterek és paramétersorok kerülnek. Erre a közbenső outputperifériára lista készitéséhez vagy a megszerkesztett program rögzitéséhez lehet szükség.

3; Eredmény-output periféria: Az a periféria /azok a perifériák/ amely/ek/re a függvénydefiniálás vagy szimuláció során kapott eredmények kerülnek.

B; Felhasználói utasitásokkal nem változtatható, kijelölt:
   4; Fő periféria: Az a periféria, melyről lehetőség van programmegszakitásra, és amelynek a többi perifériánál magasabb a prioritása. /Erre kerülnek a hibaüzenetek; innen van lehetőség a szintaktikus hibák javitására; beavatkozást igénylő helyzeteknél innen vár inputot a processzor stb./ Inditáskor a program a fő-periférián jelentkezik.

Ugyanaz a periféria többféle funkciót is betölthet egyidejüleg /pl. lehet program- és eredmény-output is/.

INTERCELLAS 1.1° szimulációs nyelv utasitástipusai:

A processzor az utasitásokat a minimálisan szükséges utasitáskód alapján ismeri fel. Ha az input periféria interaktiv /TTY vagy display-klaviatura/, a processzor a teljes utasitásrészből még hátralevő karaktereket kiegészitésül visszairja.

Az utasitáskódot a következő csoportokra oszthatjuk:
1; Perifériakijelölő utasitások: A processzor-periféria kapcsolatokat definiáló végrehajtható, tehát nem deklarativ utasitások. Mindig az utolsó utasitás hatása érvényesül.
2; Sejttérdefiniáló utasitások: A szimulált sejttér méretét meghatározó, a sejttéren belül egyes mezőket kijelölő, bábsejt-szerkezet meghatározó, mező-függvény hozzárendelést leiró illetve sejttérbe közvetlenül konfigurációt beiró utasitások.
3; Sejttérszimulációs utasitás: Utasitás a szimulált sejttér billentésére.
4; Eredménymegjelenitő utasitások: Előirják hogy a tér mely része kerüljön megjelenitésre, milyen lépésszámok mellett történjen megjelenités, és milyen konverzióval történjen.

5; Vezérlő utasitások: Utasitások programelágaztatásra vagy feltételes inputperiféria váltásra a sejttér megfigyelt részeinek állapota szerint. A program ujrainditása ill. a szimuláció befejezésére kiadható utasitások is e csoportba tartoznak.

6; Egyéb utasitások /pl. komment /COMM/ utasitás/.

DEFFUNC 1.1

A DEFFUNC 1.1 az INTERCELLAS 1.1 sejttérszimulációs nyelv átmeneti függvény definiáló önálló alrendszere.

A DEFFUNC programok feladata a felhasználó által egyszerü, szemléletes formában megadott sejt-átmeneti függvényből a szimulációs nyelv függvényszámitó rutinjai számára használható táblázat épitése, ill. a felépitett táblázat elhelyezése valamely adathordozón a szimulációs nyelv inputjának megfelelő formátumban.

A függvénydefiniálás fejlesztésének iránya a lehetőségekhez mérten legegyszerübb és legáltalánosabb függvénymegadási módszerek alkalmazása mellett a lehető legkisebb méretü, gyorsan visszakereshető táblázatok épitése. A DEFFUNC 1.1-ben - a feladat jellegétől és a szimulálni kivánt sejtek tipusától függően - többféle tipusu táblázat épitésére van mód. A felépitendő táblázat tipusát és a definiálás módját alkalmanként a felhasználó határozhatja meg utasitások segitségével. Egyes táblázat tipusokra automatikus függvénytábla minimizáló eljárások vannak beépitve.

Kapcsolat a szimulációs nyelvvel

Az átmeneti függvény táblázatát - melyet a definiáló épitett és optimalizált - valamely adathordozó közvetitésével veszi át a szimulációs program. Ez a futtatásra rendelkezésre álló

gépkonfigurációk különbözősége miatt többnyire lyukszalag, de fejlesztési tervbe van véve a szimulációs és a függvénydefiniáló nyelv overlay rendszerbe kapcsolása.

DEFFUNC 1.1 utasitástipusai:

Az egyéb interaktiv szolgáltatásokhoz hasonlóan, a szimulációs nyelvvel megegyezően az utasitások azonositása a minimálisan szükséges karaktercsoport alapján történik.

Utasitástipusok:

1; Processzor-periféria kapcsolatot leiró /ld. INT. 1.1-nél/
2; Függvénydefiniáló utasitások: Meghatározhatják a felépitendő táblázat tipusát, a függvénymegadás módját, vagy mindkettüt.
3; Környezetdefiniáló: A mikrokonfigurációval történő függvénymegadásánál /ld. [1] / a mikrokonfiguráció mellett fel nem tüntetett környező sejtek állapotait határozza meg.
4; Szimbólumdefiniáló: Értékadó tipusu utasitások. A függvénydefiniálás kényelmesebbé tételére a felhasználó alkalmazhat szimbólumokat, melyeket maga definiál. Egy szimbólumhoz rendelhető sejtállapotok halmaza vagy sorozata.
5; Függvénytábla ellenőrző és minimizáló utasitások: A függvénymegadás befejeztével utasitásra automatikus ellenőrzés és - fastruktura esetén - végleges minimizálás történik.
6; Eredménymegjelenitő utasitások.
7; Állaptszám meghatározó utasitás: A definiálandó függvény alkalmazásakor szóbajöhető maximális sejtállapot számot határozza meg.
8; Egyéb.

INTERCELLAS 1.16

Az 1.16 nyelv az 1.1 specializált változata [3]. A program terjedelme az 1.1 szimulációs programmal megegyező.

Feladata a Budapesti Müszaki Egyetemen jelenleg fejlesztés alatt álló 16x16 vagy 8x32 sejt méretü sejtteret emuláló sejtprocesszor modell müködésének funkcionális szimulációja. /ld. [12]/.

A nyelv általános jellemzői - interaktivitás, perifériakezelés, hibajelzés-javitás stb. - teljes mértékben megfelelnek az 1.1 változatnak.

A módositás elsősorban az ujfajta, az inhomogenitást maszkok segitségével előállitó rendszer /ld. [12]/ szimulációjához volt szükséges. A processzor mikroprogramozhatóságának megfelelően definiálható időben változó átmeneti függvény is, a billenések sorozatához statikus átmeneti függvények sorozatát vagy ciklusát rendelve, igy kvázi-dinamikus inhomogén kétállapotu sejtmező szimulálására van lehetőség. További változtatásokat tett szükségessé a sejtmező mikroprogramozási és input-output rendszerének szimulációja. Az 1.16 a sejtprocesszor architekturájának /utasitás és mikroutasitás készletének/ részleges szimulációjára is képes.

Utószó

Az INTERCELLAS 1.1 szimulációs nyelvek fejlesztése jelenleg is folyik. A szimuláció fejlesztésének tervbe vett iránya egy - sejtprogram szempontjából - assembly szintü szimulációs nyelv kialakitása.

A programok jelenleg négy különböző gépkonfiguráción üzemeltethetők Szegeden és Budapesten. Felhasználói az MTA Automataelméleti Tanszéki Kutató Csoport tagjai és a téma iránt érdeklődő egyetemi hallgatók.

Abstract:

On the present level of research of cellprocessors it has become necessary the creation of an interactive cellular space simulation language. The earlier simulation possibilities ensured batch simulation on a large computer /CELLAS [1]/.

In 1976 we specified the instruction set and the syntax of the INTERCELLAS language implemented of TPA-i /[2]/. The INTERCELLAS is a cellspace simulation language which simulates universal /size of the space, the transition function and the number of states of cells can be changed/ and inhomogeneous cellspaces and can be used in an interactive manner.

The first /1.0/ version of the language was made in 1977. At present the 1.1 version is being used which contains a function definition language as an independent subsystem /DEFFUNC 1.1/.

A special version of the language is the 1.16 one which simulates the operation of a cellprocessor model emulating a 16x16 cellspace being under development. Its instruction set is reached from 1.1 through modifications according to the structure of the model.

Irodalomjegyzék:

[1] Legendi T.: A 2D transition function definition language for a subsystem of the CELLAS cellular processor simulation language /CL and CL XIII, 169-194 1979./

[2] Legendi T.; Hegedüs Gy.; Pálvölgyi L.: INTERCELLAS 1.1 felhasználói kézikönyv /1979/

[3] Legendi T.; Pálvölgyi L.: INTERCELLAS 1.16 kiegészités az 1.1 felhasználói kézikönyvhöz /1980 Szeged/

[4] Legendi T.: INTERCELLAS - an interactive cellular space simulation language /Acta Cybernetika, 3, 261-267, 1977./

[5] R. Vollmar: Über einen Interpretierer zur Simulation zellularen Automaten
/Angewandte Informatik 6/73.

[6] Nagy A.; Fazekas B.: Codd-ICRA sejttér szimulációja TPA-i számitógépen /kézirat 1973/

[7] Hegedüs Gy.; Pálvölgyi L.: Sejtautomaták szimulációja és programozása /OTDK 1978, II. helyezett dolgozat/

[8] Legendi T.: Cellprocessors in computer architecture /CL and CL XI, 147-168, 1976/

[9] Legendi T.: Programming of a cellular processor model /III. Magyar Számitástudományi Konferencia, Budapest, 1981/

[10] R. F. Brender: a programming system for the simulation of cellular spaces
/The University of Michigan, Ann Arbor 1970./

[11] R. Baker, G. T. Herman: CELIA - a cellular linear iterative array simulator
/Proceedings of the Fourth Conference on Applications of Simulation, pp. 64-73, 1970/

[12] Tóth J.: Egy inhomogén programozhatóságu sejtprocesszszor modell; A CCPU részletes hardware leirása /Homogén számitási rendszerek és alkalmazásuk V. SZKI részére készült tanulmány.
Szerk.: Legendi Tamás./

Pálvölgyi László

INHOMOGÉN, KÉTÁLLAPOTÚ SEJTPROCESSZOR PROGRAMOZÁSA

A cikkben ismertetésre kerülnek sejtalgoritmusok a 16x16 sejt méretü kvázisejt rendszerü sejtprocesszor modell alkalmazhatóságának illusztrálására.

A tárgyalt modell az [1]-ben javasolt sejtprocesszor architektura egy realizálása. A processzor fejlesztése jelenleg a Budapesti Müszaki Egyetemen folyik /[9]/. A 16x16-os modell elsősorban kutatási célzatu, a hozzá készülő sejtszoftverrel együtt a további fejlesztés lehetőségeit és irányát hivatott vizsgálni.

A modellhez mintegy 40 müvelet /asszociativ memóriák, összeadás, szorzás, vektorszorzás, mátrixkódolás stb./ algoritmikusan megoldott, többségük szimulációs nyelven /[8]/ is ellenőrzött. A cikk ezekből közöl válogatást, amelynek célja, hogy érzékeltesse a sejtprocesszor lehetőségeit és programozásának elveit.

Kulcsszavak: sejtalgoritmus, sejtprogram, sejtprocesszor.

A 16x16 sejtprocesszor modell

Az alkalmazott sejttér 16x16 sejt méretü, inhomogén, kétállapotu. Az inhomogenitás a következő módon valósul meg:

Tekintsük $(f_1, m_1),(f_2, m_2)\ldots(f_n, m_n)$ párok sorozatát, ahol f_i egy lokális sejt-átmeneti függvény 2-állapotu, Neumann szomszédságu sejtekre, m_i pedig egy a sejttér méreteivel megegyező méretü bitmátrix /un. maszk/. A tér egy billentése ugy

zajlik le, hogy a párok sorrendjében az m_i maszkok által jelölt sejtek az f_i függvény szerint veszik fel uj állapotukat. A maszkok lehetnek átfedésben, akkor a többszörösen kijelölt sejtekre a sorban előbb következő függvény hat. A taktusok során más-más függvény-maszk párok adhatók meg, igy a sejttér részben dinamikus inhomogén tulajdonságokat mutat, azonban nincs szó "belső" sejtállapotról /mint [3]-ban/, mert nincs lehetőség a maszkok sejtállapottól függő /adatfüggő/ váltogatására, a függvény-hozzárendelés előre meghatározott.

A sejttér inicializálását, input-output adatáramlását, a függvények és maszkok egymáshoz rendelését egy CCPU /központi sejt-vezérlő egység/ vezérli mikroprogram alapján. Bár a függvények és maszkok definiálása is a mikroprogram szerint történik, célszerü elméletben a sejtprogramot a mikroprogramtól megkülönböztetni, és az algoritmusokat csupán a sejttérben végbemenő folyamatok szintjén tárgyalni az érthetőség érdekében. A következőkben ily módon kerülnek tárgyalásra a sejtalgoritmusok.

Általános szempontok

Sejtprocesszorok alkalmazásakor a hatékonyan párhuzamositható müveleteket érdemes a sejtprocesszorral végeztetni. A tárgyalt modellnél azonban egyéb korlátozó tényezőkkel is számolni kell. A legfontosabbak a két állapot, a kis térméret és az input-output rendszer okozta megkötések. A processzorra tehát olyan feladatokat kell tervezni, amelyeknél a sejttér kihasználtsága száz százalék, vagy megközeliti azt. Az összetett szerkezeteket következésképp ugy kell összerakni, hogy jól illeszkedjenek egymáshoz, ne legyen közöttük nem müködő, és minél kevesebb közvetitő jellegü /shiftelő/ sejt legyen.

A sejtprocesszor modell programozásának szempontjai

A többállapotu sejtekre kidolgozott algoritmusok /ld. [5]/ nagy része átmenthető az adott térbe ugy, hogy egy többállapotu sejtet több kétállapotu sejttel helyettesitünk. Ez nem jelenti minden esetben az állapotbitek számának megfeleltetését. A szomszédsági kapcsolat jellege növelné az alkalmazandó sejtek számát, az időben változó átmeneti függvények és a maszkok ügyes alkalmazásával azonban elérhető e szám erős lecsökkentése.

El lehet érni olyan eredményt is, hogy egy 8-állapotu sejtekre kidolgozott algoritmus átirásakor egy 8-állapotu sejt funkcióját egy kétállapotu kvázisejt láthassa el. /ld. Asszociativ memória/. A következő igen hatékony módszereket alkalmazzuk:

- A maszkokat fel lehet használni statikus belső állapot tárolására.
- Az adatforgalom enyhe felritkitásával és az elvégzendő feladat által megkövetelt topológiai kapcsolatokhoz hangolásával /ferdeség- melyik adat előzze meg a másikat stb./ virtuálisan több, max. 4 bitcsatorna hozható létre.
- Az időben változó függvénybe információ /pl. konstans szorzó szám/ kódolható.

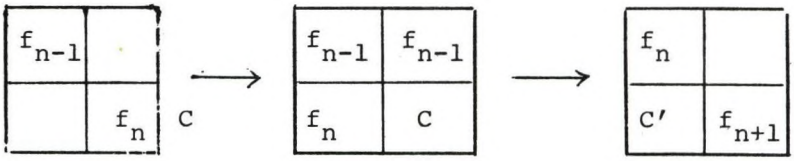
A feladatok igen nagy részének megoldásakor a térben periódikus szerkezetet hozunk létre, amelynek elemei funkcionálisan azonos müködésü sejtcsoportok - nevezzük őket tégláknak. A téglák azonos számu sejtet tartalmaznak, és a téglákon belül a helyzetük szerint egymásnak megfelelő sejtek azonos átmeneti függvény sorozatok szerint billennek. A téglára térbeli méretein kivül müködési ciklusának hossza és fázisa is jellemző. Igen jól lehet használni az adatforgalom egyszerüsitésére ill. az adott körülményekhez való jó illeszkedéshez a téglák aszinkron müködtetését. Ez természetesen növeli az alkalmazandó maszkok számát, de nem növeli a függvényekét.

Sejtalgoritmusok

1; Véletlenszámok generálása

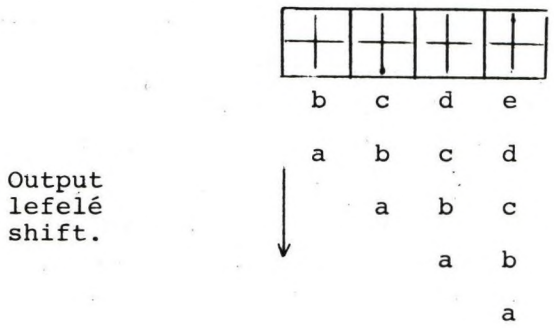
Ha a Fibonacci féle számsorozat elemeit modulo 2^k vesszük pszeudo véletlen számsorozatot kapunk, amelynek ciklushossza k-val exponenciálisan nő.

Tehát az $F_n=F_{n-1}+F_{n-2}$ /mod 2^k/ számsorozatot kell előállitani.
A megoldás 2x2-es téglákkal történhet.



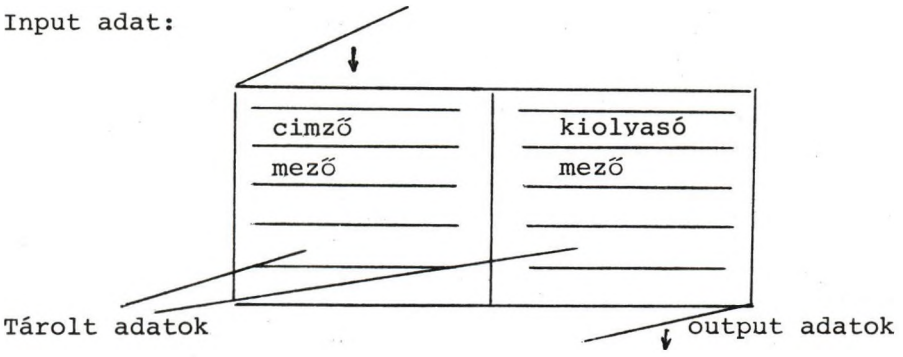
Ahol: f_n az F_n egyik bitjér jelöli.
Az átmenetfüggvény: $[C', f_{n+1}] = C+f_n+f_{n-1}$ /$[C', f_{n+1}]$ 2 bites szám/.
A jelölések [4]-nek megfelelőek.

A szerkezet sorban egymás mellé helyezett fenti tipusu téglákból áll. 2 lépésenként generálja a sorozat következő számát. Az outputot diagonális formában kapjuk:



2. Asszociativ memóriák

Asszociativ memóriának nevezzük a sejttérben a következő összetett szerkezeteket: /ld. [5]/.



Müködés: Az input adatot a tár bitenként összehasonlitja a cimző mezőben tárolt adatsorokkal. Azonosság esetén a kiolvasó mező megfelelő sorában tárolt adatot küldi az outputra. Az algoritmust a következőképpen lehet a 16x16-os sejtprocesszorra átültetni.

Egy bit tárolt adatnak egyetlen sejtet feleltetünk meg, igy 16x16-os méretü asszociativ tárat kapunk.

A belső állapotban - maszkban - külön tároljuk a memória 1 és 0 értékeinek megfelelő helyeket. /Vagyis rendelkezünk egy olyan maszkkal amely a tár egyeseire mutat, és egy másikkal amely a nullákra mutat./
Ebben az esetben egyetlen sejt is elég az adat és vezérlő bitek továbbitására egy kétlépéses ciklusban.
A müködés a bal oldalra: /cimző mező/
1. lépés: Adat lefelé shiftelődik:

2. lépés:

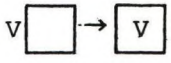
ahol

$$V' = \begin{cases} 1 \text{ ha } V=1, & A=1 \text{ és tárolt}=1 \\ 1 \text{ ha } V=1, & A=0 \text{ és tárolt}=0 \\ 0 \text{ egyébként} & \end{cases}$$

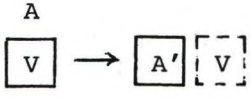
Ez egy kicsit paradoxnak tünik. Az ellentmondást azzal lehet feloldani, ha a teret sakktábla-szerüen felosztjuk /a belső állapottárul szolgáló 0 és 1 maszkokat is ennek megfelelően két-két maszkká alakitjuk/, és amig a "fehér" az 1. lépést végzi, addig a "fekete" a 2. lépést és megforditva.

Az összehasonlitandó adatok átlós formában, két lépésenként /2 sejt távolságra/ követhetik egymást. A müködés a jobb oldalra: /a ballal szinkronban számozva/. /Kiolvasó mező/

2. lépés: Vezérlőjel jobbra shiftelődik



1. lépés:



ahol

$$A' = \begin{cases} 0 \text{ ha } V=1 \text{ és tárolt}=0 \\ 1 \text{ ha } V=1 \text{ és tárolt}=1 \\ A \text{ ha } V=0 \end{cases}$$

Az egyes lépések itt is a fent leirt sakktábla felosztás szerint történnek.

Az output adat ferde formátumban 2 lépésenként lefelé jön ki

a térből. Megjegyzendő, hogy nem lesz egyvonalban a megfelelő input adattal, hanem attól egy hullámmal elmaradva.

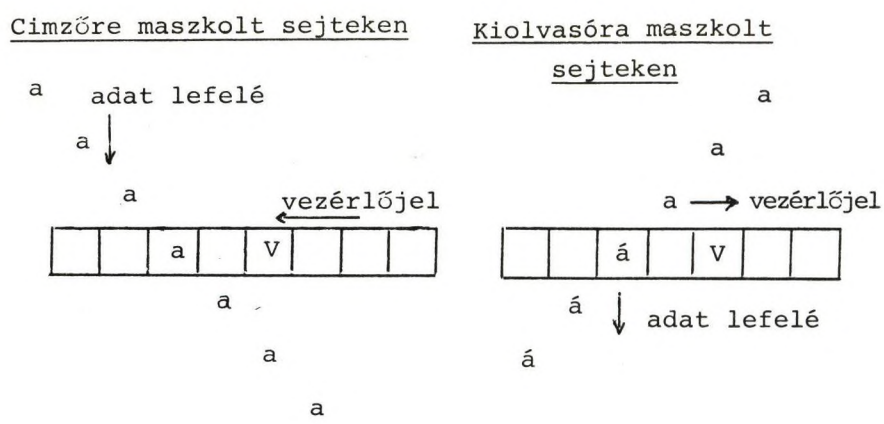
Az asszociativ tár maszk-igénye oldalanként 6, összesen 12 maszk, azonban egy billenési lépés során mindig csak 6 aktiv ezek közül. Meg lehet oldani 8 maszkkal is, de ekkor a billenések alatt mindegyik aktiv lesz, ez a megoldás igy lassubb.

A vezérlőjelek inditása történhet a bal szélre juttatott 1 inputtal, a baloldali sejtoszlop állandó 1-be állitásával, vagy a baloldali sejtoszlopra külön definiált maszk-függvény párokkal.

A tár mérete a sejttérének megfelelő. Változatlan maszkok és forditott szomszédságfüggő függvények mellett a tár ellenkező irányu keresésre is alkalmas.

Asszociativ memória /1x1/ dupla

A fenti példában látszott, hogy az adatok ritkitásával, és a sakktáblaszerüen inhomogén tér alkalmazásával virtuálisan 2 bitcsatornát lehet létrehozni a tárolt adatok mennyiségének csökkentése nélkül. Ha ennél is intenzivebb kihasználására törekszünk a térnek, az elvet tovább lehet fejleszteni. Az átmeneti függvények funkciójukban nem változnak, csak a mozgó adatok és vezérlőjelek ritkulnak tovább, és 2 belső állapotbitet 4 maszkkal szimulálunk.
Az input ill. output adatok 4 lépésenként diagonális formában követik egymást. Az azonositó vezérlőjel az input adat után két lépéssel, a kiolvasó vezérlőjel az output adat előtt két lépéssel fut. A tér jobb kihasználtsága annak a következménye, hogy az asszociativ memória két oldala egymáson van elhelyezve. A müködést az alábbi ábra szemlélteti:



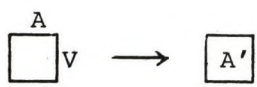
A megvalósitás ugy lehetséges, hogy az azonositás nem akkor történik mikor az input adat egy bitje az azonositandó sor egy sejtjén van, hanem mikor már egy lépéssel tuljutott rajta. Ugyanigy a kiolvasó vezérlőjel akkor hat egy sejtre, amikor már tuljutott rajta.

Átmeneti függvény mikrokonfigurációval:

Cimzőre maszkolt sejteken:

$$V' = \begin{cases} V \text{ ha } A=\text{tárolt} \\ 0 \text{ különben} \end{cases}$$

Kiolvasóra maszkolt sejteken:



$$A' = \begin{cases} A \text{ ha } V=0 \\ \text{tárolt ha } V=1 \end{cases}$$

A tárolt adatok a maszkokban vannak elhelyezve. Hogy egy sejt a cimző vagy kiolvasó oldalhoz tartozik, az sakktáblaszerüen, és időben 2 lépésenként változik.
Az inhomogenitás téglája:

1	7	2	8
5	3	6	4
2	8	1	7
6	4	5	3

Az inhomogenitást a számok a müködési ciklus szerint jelzik. Ezt tovább növeli, hogy a különbözőképpen müködnek a cimző oldal 0, cimző oldal 1, kiolvasó oldal 0, kiolvasó oldal 1 sejtjei.
Általános esetben tehát 32 maszk kell a kvázisejtprocesszorban történő megvalósitáshoz.

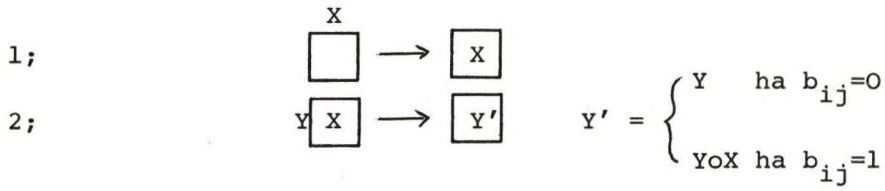
Bitmátrix-bitvektor szorzó

Az elvégzendő müvelet mátrixkódolás /ld. [6] és [7]./ Az Y bitvektort Y=Bx számitás utján állitjuk elő, ahol x az input bitvektor és B bitmátrix.

A szorzó megvalósitásánál az asszociativ memória /lxl/-nél megismert módszert alkalmazhatjuk. A tér inhomogenitása sakktábla szerü, igy virtuálisan két bitcsatorna jöhet létre. Az input adatok felülről lefelé az 1. bitcsatornán haladnak. A bitmátrix az inhomogenitásba - illetve az azt előállitó maszkba - van kódolva.

Ha az input vektor egy bitje a bitmátrix 1 értékével találközik az átteszi a másik bitcsatornára hol balról jobbra shiftelődik tovább. A 2 bitcsatornán ütközhet egy éppen áthelyezett bit, és egy balról érkező, ezt az átmeneti függvénybe kódolt bitmüvelettel /"vagy" vagy "KIvagy" müvelet/ lehet kapcsolni. A müveletek asszociativitása következtében a müködés helyes lesz.

Átmeneti függvény az i, j sejten:

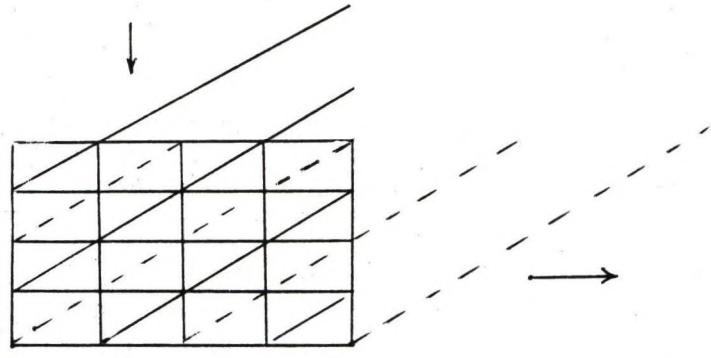


A müveleti jel:o

$b_{i,j}$ a bitmátrix eleme

A bitmátrix mérete: 16x16, a vektorok hossza ennek megfelelően 16 bit.

A leirt feltételek mellett ferde input esetén a szorzó pipe-line módon használható bitvektorok konstans bitmárixszal való szorzására.



Ha a vektorok egy bitmátrix oszlopvektorai, akkor a fenti szerkezetei konstans mátrixszal való mátrixszorzásra is használható.

Abstract

In this article cellular algorithms will be made known for the illustration of application of a cellprocessor model containing a 16x16 quasi-cellular field. The discussed model is a realization of the cellprocessor architecture suggested in [1].
At present this processor is under development in the Budapest Technical University /[9]/. The 16x16 model serves first of all for the purposes of research together with the cellsoftware made for it examines the possibilities and direction of future development.
About 40 operations for the model /associative memories, addition, multiplication, vector multiplication, matrix coding etc./ are solved algorithmically, most of them are checked on a simulator /[8]/. This article contains a selection from the above operations and its aim is to illustrate the possibilities of cellprocessors and principles of their programming.

Irodalomjegyzék:

[1] Legendi T.: Programming of a cellular processor mod 1 /III. Magyar Számitástudományi Konferencia 1981./

[2] Legendi T.: Párhuzamos müködésü homogén számitási rendszerek tervezése, programozása és alkalmazása /Programozási Rendszerek'78/

[3] Legendi T.: Sejtprocesszorok programozása /NJSZT Párhuzamos számitási rendszerek 79/3, 63-76/

[4] Legendi T.: A 2D Transition function definition language for a subsystem of the CELLAS cellular processor simulation language.
/CL and CL XIII, 169-194, 1979./

[5] Katona E.: Sejtalgoritmusok /válogatás az MTA Automataleméleti Kutató Csoportnál elért kutatási eredményekből. NJSZT 1981./

[6] Katona E.: Cellular algorithms for binary matrix operations /CONPAR'81 Konferencia kiadványa, Nürnberg 1981/

[7] Polner G.: Mátrixkódolás /kézirat/

[8] Legendi T., Pálvölgyi L.: INTERCELLAS 1.16 felhasználói kézikönyv /Szeged, 1980/

[9] Tóth J.: Egy inhomogén programozhatóságu sejtprocesszor modell; A CCPU részletes hardware leirása /Homogén számitási rendszerek és alkalmazásuk V. SZKI részére készült tanulmány, Szerk. Legendi Tamás/

Pálvölgyi László MTA Automataelméleti Tanszéki Kutató Csoport
6720 Szeged, Somogyi u. 7.

Siegler Andrásné–Szabó Rudolf–Varsányi István–Apor György

MÁGNESSZALAG NYILVÁNTARTÁSI ÉS BIZTONSÁGI RENDSZER (MNBR)

A cikk az ÁSZSZ-ben kifejlesztett és bevezetett MNBR programcsomag felépitését és az általa nyújtott szolgáltatásokat mutatja be. Tárgyalja azokat a szempontokat, amelyek szerint a rendszer megtervezésre került, bemutatja az ennek megfelelő adatbázis kialakitását, ill. szervezését. Részletezi az egyes mágnesszalagokról nyilvántartott adatokat. Röviden ismerteti a programcsomag főbb programjainak funkcióit. Összefoglalja az MNBR használatával járó előnyöket: a mágnesszalagok adminisztrativ kezelésének automatizálását, valamint a mágnesszalagokon őrzött file-ok védelmi lehetőségeit.

Kulcsszavak:
MNBR
Raktári nyilvántartás
Szalagmozgások
File védelem

1. Bevezetés

Nagy számitóközpontoknak, amelyek multiprogramozható számitógépeket üzemeltetnek és mágnesszalagos adathordozókat nagy számban tárolnak, igen nagy problémája a mágnesszalagok adminisztrativ és biztonsági kezelése. Ha erre kezelési utasitásokon és manuális módszereken kivül hardver és softver eszközök nem állnak rendelkezésre, akkor egy bizonyos mennyiségi határon túl megoldhatatlan feladat elé állitja az ügyfélszolgálatot és az operátorokat.

Nagy számú mágnesszalag raktári tárgyként való kézi nyilvántartása, fizikai jellemzőik, tulajdonosaik rögzitése, tárolási helyük, bérlésük, a szalagkopásra jellemző hozzáférések számának manuális nyomonkövetése igen nehézkes.

A legtöbb számitógép tipusnál meg lehet találni az operátori tévedések elleni védelem kidolgozott módszereit, de ugyanakkor a programozói véletlen vagy szándékos hibák miatti illetéktelen mágnesszalag-használat elleni védelemre szinte egyik gép sincs felkészülve.

A továbbiakban ismertetésre kerül az ÁSZSZ-ben készült Mágnesszalag Nyilvántartási és Biztonsági Rendszer /MNBR/ kialakitása, amely mind adminisztrativ, mind biztonsági szempontból kielégitő megoldást nyújt.

2. Az MNBR

Az MNBR programcsomag az ÁSZSZ-ben került kifejlesztésre Honeywell Bull 66/60 és 66/20-as sorozatú számitógépekre, melyek GCOS 4/J vagy 4/JS operációs rendszerrel rendelkeznek. Kialakitása során a következőket igyekeztünk megvalósitani:

Mágnesszalagok adminisztrativ kezelésének automatizálása

- a számitóközpont területén előforduló mágnesszalagok gépi nyilvántartása;
- a tárolóhelyenkénti leltározás megkönnyitése;
- a szalagmozgások azonnali követésének biztositása;
- a mágnesszalag bérleti dijak számlázásának automatizálása;
- a bérleti viszony lejáratának automatikus figyelése;
- a szalagok kopásának automatikus követése /tisztitás, selejtezés/;
- a mágnesszalag pillanatnyi tárolási helyének lekérdezési lehetősége konzolról;

A mágnesszalagon őrzött file-ok védelme:

- mágnesszalagos file-ok felhasználói csoportok közötti védelmének biztositása;
- mágnesszalagos file-ok felhasználói csoporton belüli egyedi védelme;
- annak lehetővé tétele, hogy a mágnesszalagos file tulajdonosa rendelkezhessen arról, hogy file-jához mely felhasználói csoportok milyen módon férhessenek hozzá /irási vagy olvasási engedély megadása/;
- a szalagos file tartalmának változtatása elleni védelem a megőrzési időtartam alatt;
- annak biztositása, hogy az adatvédelmi ellenőrzés minden esetben az egységen ténylegesen fennlévő mágnesszalagra vonatkozzék, ezzel kiküszöbölve mind az ope-

rátori, mind a programozói tévedések lehetőségét.

Az adatbázis védelme:

- Biztositani kellett, hogy az adatbázis a párhuzamos update-k esetén se rongálódjék meg.

2.1. Az adatbázis kialakitása

Meg kellett határozni, hogy a mágnesszalagok mely adatait akarjuk nyilvántartani, majd a definiált adatokat csoportositottuk a következők szerint:

Üzemeltetési adatok:

A mágnesszalagok azon adatai, amelyek megadására, módositására kizárólag a számitóközpont jogosult.

Pl.: raktári szám, szalagcimke, tulajdonos- vagy bérlőcsoport-azonositó, tárolás helye, nyilvántartásba vételi dátum, szalagtipus, szalagméret, számlázási mód, bérlet kezdeti dátum, bérlet lejárati dátum, bérleti bizonylat száma, stb.

Számlázási és leltári adatok:

Azon üzemeltetési adatok, amelyekre csak egy-egy számlázás vagy leltárkészités során van szükség, igy on-line nyilvántartásuk felesleges.

Pl.: bérleti bizonylat száma, a bérlő azonositója, nyilvántartási dátum, bérlet kezdeti és lejárati dátuma, szalagméret, szalagtipus.

Felhasználói adatok:

Azon adatok, amelyeket a felhasználók definiálhatnak és módosithatnak.

Pl.: a szalagon lévő file-jaik nevei, a file-ok védelmére szolgáló jelszavak, a szalagra vonatkozó általános I/O engedélyek, az egyes file-okhoz tartozó spe-

ciális I/O engedélyek.

A szalaghasználat adatai:

A szalag használatára vonatkozó, az operációs rendszer által automatikusan feljegyzett adatok.
Pl.: az utolsó hozzáférés és irás dátuma /file-onként/,
a szalag fizikai elhasználódását jellemző hozzáférések száma.

2.2. Az adatbázis szervezése:

Az adatbázis szervezésének kialakitása során a következő szempontokat kellett figyelembe venni:

- A mágnesszalagot adminisztrativ szempontból egyedül a raktári száma, ugyanakkor a szalagról nyerhető információk közül egyedül a cimkéje azonositja, ezért az adatbázis szervezésének alapja a raktári szám és cimke legyen.
- A nyilvántartásnak egyértelmünek és pontosnak kell lennie. Ugyanazon szalaghoz nem tartozhat több szalagleiró rekord.
- Az adatbázist ugy kell szervezni, hogy a biztonsági ellenőrzés a system overhead-et csak minimális mértékben növelje.
- Az on-line nyilvántartási file területének minimalizálása érdekében a számlázási és leltári adatokat off-line módon kell tárolni /mágnesszalagon/.

Ezek figyelembevételével az adatbázist két részre osztottuk:

- Szekvenciális hozzáférésü off-line nyilvántartási file, melyre a számlázási és leltári adatokat tartalmazó eredeti és módositott rekordok a raktári szám szerinti sorrendben és naplózva kerülnek fel.

- Random hozzáférésü 2 db szubfile-ra osztott on-line nyilvántartási file, melynek első szubfile-ja /szalagcimkén alapuló/ indexelt szekvenciális szervezésü és tartalmazza a biztonsági ellenőrzés miatt fontos üzemeltetési adatokból és a szalaghoz való hozzáférések számából álló szalagleiró rekordokat; a második szubfile tartalmazza a szalagon levő file-okra vonatkozó, a felhasználók által definiált adatokból és a file-használat adataiból álló file-leiró rekordokat. Az első és a második szubfile-ban lévő rekordok pointer-láncra vannak füzve.

Az adatbázis off-line és on-line rekordjai között a raktári szám teremt kapcsolatot. Az adatbázishoz az üzemeltetés a szalagleiró rekordokkal, a felhasználók a file-leiró rekordokkal kapcsolódnak.

2.3. Program-interface-k

Mivel több program egyidejüleg használja ugyanazt a nyilvántartási file-t, ezért biztositani kellett az adatbázist az egyidejü update-k ellen.

2.3.1.

Az Üzemeltetési Osztály és az MNBR adatbázisa között a következő programok teremtenek kapcsolatot:
/Ezen programok kizárólag operátori engedéllyel futtathatók./

- TDAT: az adatbázist inicializáló, felépitő és karbantartó program, amely a következő funkciók végrehajtására képes:
  1. Szalagok egyedi és tömeges raktárba vétele /raktári szám szerint/

2. Bérelt szalagok nyilvántartásba vétele /szalagcimke szerint/
3. A raktári és bérelt szalagok állománya közötti mozgás követése
4. A nyilvántartásban szereplő szalagok egyes adatainak /mezőinek/ megváltoztatása
5. Szalagok egyedi és tömeges törlése a nyilvántartásból /szalagcimke, raktári szám vagy felhasználói csoport azonositója szerint/.

- DCPP: A TDAT program input direktiváit szintaktikailag ellenőrző préprocesszor.

- TRC: Az adatbázis on-line file-jának szerkezetét karbantartó és utility program, mely a következő funkciók végrehajtására képes:

1. ORDER: A nyilvántartási file-ban a rekordmozgások /törlés, beszúrás/ során keletkezett üres helyeket a TDAT és TMS programok számára ujra elérhetővéteszi; ezzel az on-line file-terület optimális kihasználtságát biztositja.
2. FIRST: Az első szubfile méretét megváltoztatja a megadott értékre.
3. SECOND: A második szubfile méretét megváltoztatja a megadott méretre.
4. REST: A file régebbi /a TRC által korábban save-elt/ állapotát állitja vissza.
5. LIST: Listázza a nyilvántartásban szereplő szalagokat és adataikat.

A TRC minden futtatáskor save-eli a nyilvántartási file eredeti állapotát.

2.3.2.

A felhasználók és az adatbázis között a következő program teremt kapcsolatot:
/Ezen program futtatható batch-ben is, de Time Sharing alrendszereként terminálról is aktivizálható./

- TMS: a következő funkciók ellátására képes:

1. A szalagra vonatkozó általános I/O engedélyek megadása
2. A szalagon lévő file-okhoz file-leiró rekordok generálása /megadhatók: file-név, jelszó, megőrzési idő, speciális I/O engedélyek felhasználói csoportok számára/.
3. A file-leiró rekordok módositása
4. Listázás:
   - Felhasználói csoportok szerint:
     - teljes szalaglista
     - üres szalagok listája
   - Szalagok szerint
   - File-leiró rekordok külön-külön
5. A file-leiró rekord törlése

2.3.3.

Az operációs rendszer és az adatbázis között a következő program teremt kapcsolatot:

- XSA6: Az operációs rendszer XSA6-tal módositott változata figyeli a központi gépre kapcsolódó mágnesszalag-egységek állapotait és az egyes szalagokon végrehajatni kivánt I/O müveleteket. Ha az egységen ténylegesen fönn lévő szalag BOT-re kerül, akkor az XSA6 beolvassa a szalagon található cimkét. Amennyiben van rajta cimke és ez szerepel a nyilvántartásban, akkor a szalagleiró és

a file-leiró rekordok, valamint a szalagot igénylő program azonositója alapján az I/O müveletekre jogosultság-ellenőrzést hajt végre. Ha a szalagot igénylő program jogosulatlan müveleteket kezdeményezett, ugy a program futását felfüggeszti és abortálja.

Az XSA6 fel van készülve cimkézetlen szalagok biztonsági ellenőrzésére is, amikor az operátort utasitja, hogy a szalag külső cimkéjét gépelje be a konzolon, majd ezen cimke alapján végzi az ellenőrzést.

A nyilvántartásban nem szereplő cimkéjü szalagokra biztonsági ellenőrzést nem hajt végre.

2.3.4.

Az MNBR-hez tartozik még egy számlázó program és több különböző szempontok szerinti katalógus-listát készitő program.

3. Az MNBR által nyujtott szolgáltatások

Az MNBR a mágnesszalagok adminisztrativ és biztonsági kezelését igen nagy mértékben automatizálta.

3.1. Adminisztrativ szolgáltatások

Ha biztositjuk, hogy a mágnesszalagot csak az MNBR felelősön keresztül lehet kivinni-behozni a számitóközpont területére, akkor a bizonylati rendszer áz állóeszköz raktár - számitóközpont, valamint a bérlő - számitóközpont kapcsolatokra redukálódik.

Mivel lehetőség van a mágnesszalagok tárolóhelyenkénti listázására, igy a leltárkészités nagyon leegyszerüsödik.

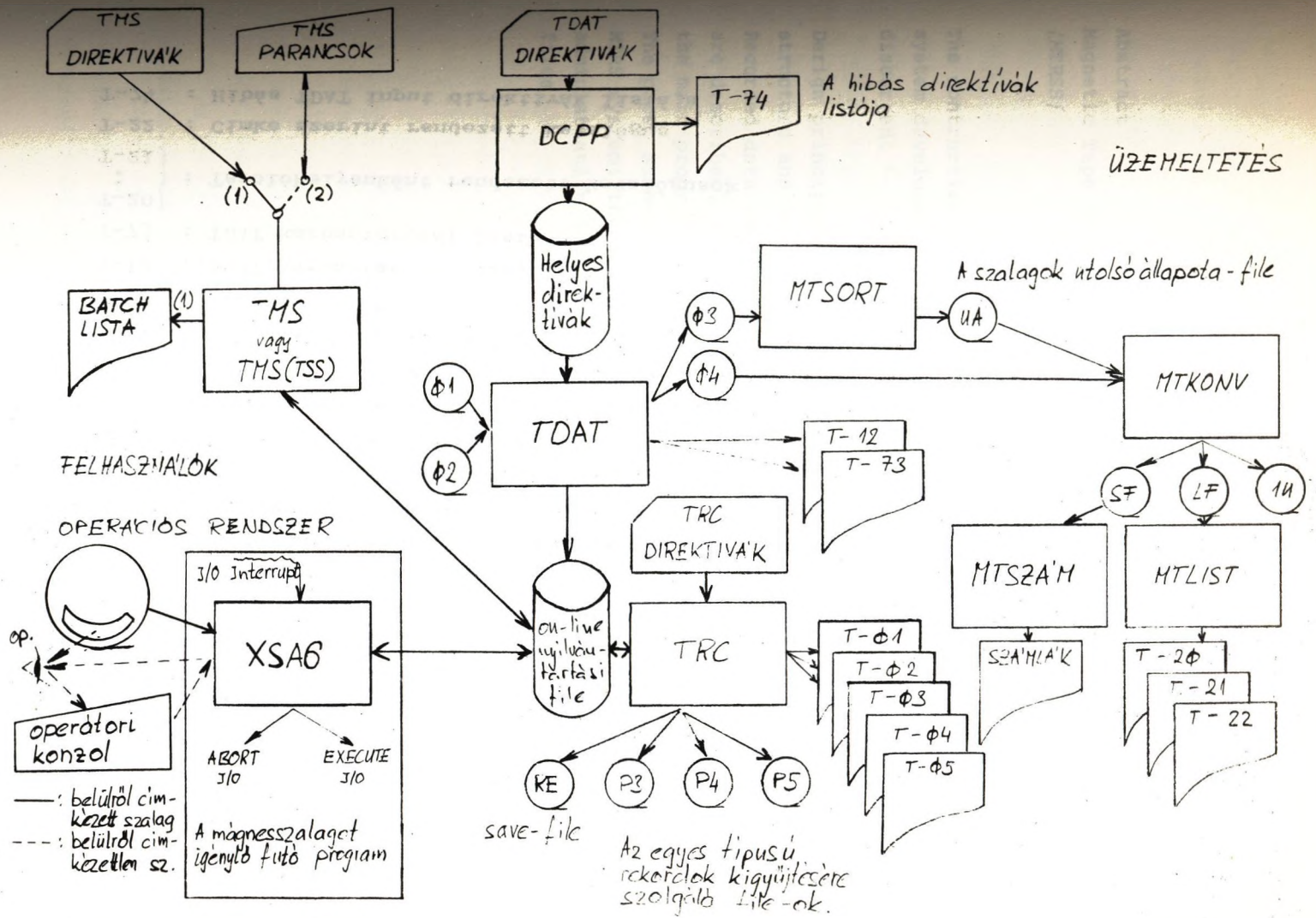
A mágnesszalagok bérletének számlázása teljesen automatikus.

Ha az operátor egy program által igényelt mágnesszalagot nem talál, akkor lehetősége van a szalag tárolási helyének lekérdezésére, igy biztonságosan elkerülhetők a tévedések miatti visszaszámlázások.

3.2. Adatvédelmi szolgáltatások

A szalagbérlőnek lehetősége van arra, hogy mágnesszalagos file-jait jelszóval és bizonyos felhasználói csoportok részére bizonyos I/O müveletek engedélyezésével megvédje az illetéktelen irási vagy olvasási kisérletek ellen.

Az MNBR tökéletes védelmet nyujt mind az operátori tévedések, mind a programozói véletlen vagy szándékos tévedések ellen, mivel ellenőrzést az egységen ténylegesen fönnlévő szalagra vonatkozóan végez és az ellenőrzés minden esetben ujra kezdődik, ha egy szalag BOT-re kerül.


THS DIREKTIVÁK
THS PARANCSOK
TDAT DIREKTIVÁK
DCPP
T-74
A hibás direktívák listája
(1)
(2)
Helyes direktívák
BATCH LISTA
(1)
THS vagy THS(TSS)
Φ3
MTSORT
UA
A szalagok utolsó állapota - file
Φ1
Φ2
TDAT
Φ4
MTKONV
T-12
T-73
FELHASZNÁLÓK
OPERÁCIÓS RENDSZER
SF
LF
IA
TRC DIREKTIVÁK
MTSZÁM
MTLIST
I/O Interrupt
on-line nyilvántartási file
TRC
T-Φ1
T-Φ2
T-Φ3
T-Φ4
T-Φ5
SZÁMLÁK
T-2Φ
T-21
T-22
op.
XSA6
operátori konzol
ABORT I/O
EXECUTE I/O
KE
P3
P4
P5
save-file
—— : belülről címkézett szalag
- - - : belülről címkézetlen sz.
A mágnesszalagot igénylő futó program
Az egyes típusú rekordok kigyűjtésére szolgáló file-ok.

01, 03: A bérelt szalagok off-line rekordjai

02, 04: A bérlésre kiadható, raktáron lévő szalagok off-line rekordjai

UA : A szalagok utolsó állapota-file

SF : Számlázási file

LF : Listázási file

1U : A bérelt szalagok egy leszámlázás utáni off-line rekordjai

RE : TRC save-restore-file

P3 : A bérelt és cimkézett szalagok on-line rekordjainak kigyüjtésére szolgáló file

P4 : A bérelt és cimkézetlen szalagok -"- -"-

P5 : A bérlésre kiadható, szabad szalagok -"- -"-

T-01 : TRC processzor üzenetek

T-02 : Nyilvántartási file karbantartási lista

T-03 : A bérelt és cimkézett szalagok on-line rekordok listája

T-04 : A bérelt és cimkézetlen -"- -"-

T-05 : Szabad szalagok -"- -"-

T-12 : TDAT karbantartási hibalista

T-73 : TDAT karbantartási lista

T-20
:
T-21 : Tárolóhelyenként rendezett katalógusok

T-22 : Cimke szerint rendezett katalógus

T-74 : Hibás TDAT input direktivák listája

Abstract
Magnetic Tape Registration and Security System
/MTRSS/

The contruction and services of the MTRSS program system developed and introduced at CSSA are discursed.

Derign principles of the system are shown, the structure and organization is also discussed. Recorded data about individual magnetic tapes are described. A short summary of the functions of the main programs in the package is provided. The paper summerizes the advantages of using the MTRS system: the automatization of magnetic tape administration and the ways to protect magnetic tape files.

Simon Endre–Gyimóthy Tibor–Zachar Zoltán

EGY ATTRIBUTUM NYELVTAN BÁZISÚ ÖNKITERJESZTŐ COMPILER GENERÁTOR+

A HLP/SZ compiler generáló rendszer, a Helsinki Language Processor továbbfejlesztése önkiterjesztő nyelvek fordító-programjainak automatikus előállitására. Programozási nyelvek lexikális és szintaktikus analiziséhez, rendszerünk reguláris kifejezéseket, LR(0), SLR(1), LALR(1), LR(1), vagy ELR(1) tipusu gramatikákat fogad el. A nyelvtanokat a szemantikus kiértékeléshez attributumokkal és szemantikus függvényekkel bővitjük ki, attributum nyelvtanos forditási sémát nyerve igy. A célkód generálás az attributiv levezetési fa végső bejárása során történik meg. Önkiterjesztő nyelvek fordítóinak specifikációja reguláris kifejezésekkel előirt szöveg transzformációkkal, vagy attributiv fatranszformációk segitségével adható meg. Az optimalizált elemző automaták mellé hiba javitó eljárások generálódnak. Az attributumok kiértékelését optimalizált, determinisztikus, alternáló vagy az attributumok rendezésén alapuló OAG eljárás végzi. Implementációnk bázis nyelve a SIMULA 67 programozási nyelv.

Kulcsszavak: forditó generálás, lexikális analizis, alulról felfelé elemzés, attributum nyelvtanok, nyelvkiterjesztés

1. Bevezetés

Dolgozatunk célja beszámolni egy attributum nyelvtan alapu, alulról felfelé elemző, két szinten is önkiterjesztő

+ Jelen munka az OMFB által koordinált hazai ESZR-3 kutatások részeként folyik, a Perspektivikus Számitási Rendszer bázis gépi nyelve témában.

compiler generátor rendszer tervezéséről, és az implementáció során nyert tapasztalatokról. Eredeti feladatunk egy olyan rendszer megvalósitása volt [Sim 80], amely hatékony eszközt biztosit magas szintü programnyelvek gyors implementálásához, és lehetőséget ad a fordító strukturált, lépésenkénti finomitással történő realizálására. Kiindulási rendszerként kritikai vizsgálat tárgyává tettük a HLP /Helsinki Language Processor/ [Räi 78] néven ismert rendszert. A HLP compiler a fordítást egymástól jól elkülönithető négy fázisban végzi el. A véges állapotu automata által végzett lexikális elemzés eredményeként kapott 'token' nevek sorozatából - ahol egy token név az azonos lexikális tulajdonságu jelsorozatokat helyettesiti -, az LALR(1) tipusu szintaktikus elemző egy attributumos levezetési fát állit elő. Az attributumok kiértékelése ASE /Alternate Semantic Evaluator/ [Jaz 75] stratégiával történik. Ezt követi a célkód generálás, amit a korábban kiszámitott-szemantikus-attributumok értékei vezérelnek. Ennek megfelelően a hibajelzés is négy különböző szinten történhet meg.

Vizsgálataink eredményeként az eredeti rendszerben több lényegi módositást vezettünk be, és az igy nyert compiler--compiler a HLP/SZ nevet kapta. Elsőként önkiterjesztővé tettük a HLP-t két szinten is. A lexikális elemzés fázisában lehetőség van /tetszőleges mélységben/ reguláris kifejezésekkel definiált kiterjesztő utasitások bevezetésére. A kiterjesztés másik szintjén, a szintaktikus elemzés során kialakult tetszőleges /attributumos/ részfa helyettesithető valamely más /attributumos/ fával, uj szintaktikát/szemantikát rendelve ezáltal az eredeti konstrukcióhoz. Az általunk ismert, és legjobbnak tartott kiterjesztő compiler generátor rendszer, az RCC [Nap 80] a kiterjesztésnek csak az első szintjét ismeri. Második bővitési irányunk, további szintaktikus elemzési módszerek bevezetése volt. Rendszerünkben a felhasználó öt elemző közül választhat /LR(0), SLR(1), LALR(1), LR(1)[Aho 74], valamint ELR(1) [Pur 81]/, a forditandó nyelv szintaktikus szerkezetének megfelelően. Az attributumok

kiszámitását vezérlő eredeti ASE módszer mellé bevezettük az OAG /Ordered Attribute Grammars/ [Kas 80] kiértékelést, amely az attributum nyelvtanok tágabb osztályára alkalmazható polinomiális komplexitással. A szemantikus feltételek rendszerünkbe történt integrálása, a hibajelzések információ tartalmának növelését és bizonyos attributumok értékei szükségtelen kiszámitásának elkerülését célozza.

A HLP/SZ rendszer tervezését, SIMULA 67 bázisu implementálása követte. Ehhez szükségünk volt a HLP/SZ rendszer HLP/SZ metanyelvü definiciójára. A programok tervezését és kivitelezését ugy végeztük, hogy a HLP/SZ és a mindenkori generált fordító közös részei egyszer kerüljenek megirásra. További terveink között szerepel az attributumok közötti olyan relációk bevezetése, melyek lehetőséget adnak konkurrens célkód előállitására is. Ez utóbbi téma számtalan, az attributum nyelvtanok körébe tartozó elméleti problémát vet fel. Ezekkel egy későbbi cikkben kivánunk foglalkozni.

Dolgozatunk további részében elsőként vázlatos ismertetést adunk a HLP rendszerről. Ezt követi a kétfajta kiterjesztés bemutatása, kiemelve a megvalósitásukhoz kidolgozott módszereket, majd az implementációról szólunk.

2. A Helsinki Language Processor

Egy magasszintü programozási nyelv lexikális, szintaktikus és szemantikus szerkezetének specifikálásához a HLP két metanyelvet alkalmaz. A lexikális metanyelv reguláris kifejezésekkel és string transzformációkkal definiálja a forrásnyelv szövegszerkezetét. A HLP fordító ennek alapján, Earley [Ear 70] módszerének egy továbbfejelsztésével generálja a lexikális elemzőt, mint egy optimalizált véges állapotu automatát. A szintaxis és szemantika megadás egységes koncepcióval került metanyelvi egyesitésre. A HLP fordító felhasználható LALR(1) [Aho 74] tipusu optimalizált szintaktikus elemző generálására a szemantika definició hiányában is. A szintaktikus hibák automatikus javitásához a rendszer [Sip 78] alapján generál eljárást. A szemantikát a CF /kör-

nyezet független/ gramatika nemterminálisaihoz rendelt attributumok értékei definiálják, a produkciókhoz tartozó szemantikus függvények kiértékelése után. A CF szabályokhoz rendeljük a kódgenerálási előirásokat is, melyeket az attributumok értékei vezérelnek. Az attributum előfordulások és függőségi halmazaik ismeretében a HLP fordító egy determinisztikus, ASE kiértékelőt állit elő, amely tetszőleges levezetési fára véges számu menetben meghatározza az attributum előfordulások értékeit.

2.1. Lexikális metanyelv

A lexikális leirás célja alapvetően kettős. Egyrészt a majdani forrásszövegben előforduló, önálló szintaktikai jelentéssel biró jelsorozat halmazok, un. token osztályok definiálása, másrészt a token osztályok elemeire vonatkozó szürési és helyettesitési eljárások előirása. A névvel ellátott token osztályok megadása reguláris kifejezésekkel történik, és a HLP fordító ezek alapján generálja, Earley módszerének [Ear 70] egy továbbfejlesztésével - az osztályok elemeit felismerő véges állapotu automatákat. A szürési és helyettesitési előirások az akció blokkokban találhatók. A lexikális elemzés minden pillanatában egyetlen akció blokk alapján történik a szürés. Egy akció blokk belsejében lehetőség van tetszőleges, más akció blokk aktivizálására. Ha egy akció blokk aktiv, kezdő állapotba kerül az összes olyan automata, amelyhez tartozó token nevekre létezik szürési előirás az adott blokkban. Ha valamely automata végállapotba kerül /felismerve a forrásszöveg egy részsorozatát/, az akció blokkbeli előirás alapján törlés, vagy két fajta helyettesités történhet. Lehetőség van az eredeti string tetszőleges másik szöveggel, vagy a szintaxisban használt token névvel történő helyettesitésére. Tehát a lexikális elemző outputja, ami egyuttal a szintaktikus elemző bemenete, terminális szövegek és /szintaxisbeli/ token nevek sorozata. Ez utóbbiak minden előfordulásához attributumként, feljegyzésre kerül a konkrét, forrásszövegbeli részsorozat.

A könnyebb érthetőség kedvéért, tekintsük az alábbi ALGOL-szerü, maximum 8 karakteres azonositókból álló nyelv lexikális elemzőjét megadó HLP programot.

```
lexical description alglex
character sets
   számjegy='0123456789';
   nemzero=számjegy-'0';
   betü='AB...YZ';
   szóköz='';
end of character sets
token classes
   azonosító=betü(betü|számjegy)* [8] ;
   egész     =nemzero számjegy* [10] | '0';
   szóközök =szóköz+;
   megjegyzés='C'any* endofline;
   specjel  =default tokens;
end of token classes
blokk_1: begin
   azonosító [blokk_2] ⇒ identifier|'IF'|'DO'|'GOTO'|
                          'BEGIN'|'END' ...;
   egész [blokk_2]     ⇒ unsigned_integer;
   szóközök            ⇒ ;
   megjegyzés          ⇒ ;
end of blokk_1
blokk_2: begin
   specjel [blokk_1] ⇒ keystrings;
   szóközök [blokk_1] ⇒ ;
end of blokk_2
end of lexical description alglex
```

Példánkban az alapszavakat aláhuzással, a terminális szövegeket felsővesszőkkel, a maximális jelsorozat hosszakat szögletes zárójelekkel, a reguláris müveleteket a megszokott módon jelöltük. Az any alapszó tetszőleges karaktert, mig az endofline sor végét jelöl. A default tokens azonositja a későbbi attributum gramatika kiirt terminálisainak halmazát,

az identifier pedig az ugyanott terminálisak helyén előforduló /szintaxisbeli/ token osztálynevet jelöl. A szóközök és megjegyzések az akció blokkokban kerülnek törlésre.

2.2. A szintaktikus metanyelv

Feladat a lexikális elemzés utáni, forditandó nyelv attributum nyelvtanos [Kas 80] forditási sémájának megadása. Csak a könnyebb érthetőség kedvéért jegyezzük meg röviden az alábbiakat.

Egy attributum nyelvtan definiciója a következőket jelenti. Egy CF nyelvtan nemterminálisaihoz speciális változókat un. attributumokat rendelünk. Az attributumok lehetnek öröklött vagy szintetizált tulajdonságuak. Egy levezetési fa tetszőleges nemterminálishoz rendelt csomópontjában előforduló attributum öröklött, ha az értékét meghatározó un. szemantikus függvény kiszámitásához csak a levezetési fában az ős, valamint a saját szintjén lévő attributum előfordulások értékeit használjuk fel. A HLP-ben használt globális /öröklött/szintetizált/ attributumok értékei hozzáférhetők a teljes részfában, amelynek gyökerében azok előfordulnak. Tetszőleges levezetési fára, az attributum előfordulások értékeinek meghatározására /a fa esetleg többszöri bejárásával/ különböző stratégiák léteznek [Boc 76], [Jaz 75], [Kas 80].

Egy attributum nyelvtanos forditási séma megadása - ahogyan ezt a HLP szintaktikus metanyelve is tükrözi - ezek után az alábbi módon történik. Elsőként szintetizált és öröklött, nem globális és globális tulajdonságu attributumokat deklaráljuk névvel és tipussal. Ezt követi a CF grammatika nemterminálisainak felsorolása az attributum hozzárendelésekkel. A nyelvtan kezdőszimbolumának specifikációját, a szemantikus függvények értékeit megadó eljárások deklarációi követik. /Az eljáráshivások során az aktuális paraméterek konstansok, vagy attributum előfordulások lehetnek./ Utolsóként adjuk meg a produkciókat, melyek három részből állnak. Magát a CF produkciót, a benne található nemterminálisokhoz tartozó attributum előfordulások értékeit specifikáló szemantikus egyenletek követik. Egy produkcióban /érthető

módon/ csak a jobboldal nemterminálisaihoz rendelt öröklött, valamint a baloldalhoz tartozó szintetizált attributumok szemantikus egyenletei szerepelhetnek. A szemantikus egyenleteket a kódgenerálási előirások követik. A kódgenerálást az attributumok értékei vezérlik, továbbá a jobboldal valamely nemterminálisának a kódgenerálási előirásban való előfordulása /tetszőleges levezetési fára/, a nemterminálishoz tartozó részfa gyökerében található kódgenerálási utasitások végrehajtását jelenti. Tehát a levezetési fa utolsó bejárását végső soron az attributumok vezérlik.

A produkciók jobboldalán előforduló azon szimbólumok, melyeket nem deklaráltunk nyelvtani jelnek, továbbá nem is terminális jelsorozatok, mint szintaxisbeli token osztály nevek egy speciális /value nevü/ kitüntetett szintetizált attributummal birnak. Ezen attributum értéke tetszőleges levezetési fára, a lexikális elemző által táblázatban rögzitett, azon jelsorozat melynek a forrásszövegbeli előfordulása helyettesitődött a szintaxisbeli token osztálynév tekintett előfordulásával.

A továbbiakban, az előző példa folytatásaként, egy blokkszerkezetü nyelv változóit kezelő forditási sémát közlünk HLP nyelven. Megadjuk továbbá egy egyszerü értékadó utasitás forditását is.

```
attribute grammar algszint
mnemonics are
   méret=1000;
end of menemonics
synthesized attributes are
   'text' 'array' uj[0:méret,1:2]; 'text' tipus, név,
                                   bal, jobb;
end of synthesized attributes
inherited attributes are
   'text' 'array' orig, használt[0:méret,1:2];
end of inherited attributes
nonterminals are
```

```
   program;
   blokk has használt;
   utlista has orig,használt,uj;
   utasitás has orig,használt,uj;
   dekl has tipus, név;
   végreh has használt;
   értékadás has használt, bal, jobb;
end of nonterminals
start symbol is program;
procedures are
   'comment' a && a szemantikus függvényeket és globalis
   változókat jelöli.
     && 'procedure' konkatenáció (uj,régi,tipus,név);
        'text''array' uj,régi; 'text' tipus, név;
        'begin'...'end';
     && 'boolean''procedure' névkeresés (használt,név,
         index);
        'text''array' használt; 'text' név; 'integer'
         index;
        'begin'...'end';
     && 'integer' index;
end of procedures
productions are
   program=blokk; out blokk end
   blokk  =utlista; do orig(utlista):=használt(blokk);
                       használt(utlista):=uj(utlista)end
  % az egyszerüség kedvéért a :=jelölje a mátrix érték-
  % adást is.
   utlista=utlista utasitás;
          do használt(utlista*)  :=használt(utlista);
             orig(1):=orig(0);
             használt(utasitás):=használt(0);
             orig(2):=uj(utlista*) ;
             uj(utlista):=uj(3)
          out utlista*; utasitás end
```

```
utlista=utasitás;
        do használt(1):=használt (0);
           orig(utasitás):=orig(utlista);
           uj(utlista):=uj(1)
        out 1 end
utasitás=deklaráció;
         do uj(utasitás)   konkatenáció(uj(0),uj(0),
                           tipus(1), név(1))
         end
deklaráció='integer' identifier;
         do tipus(deklaráció):="integer";
            név(0):=value (identifier) end
% a deklaráció többi változatát terjedelmi okokból
mellőzzük.
utasitás=végreh;
         do használt (végreh):=használt(utasitás);
            uj(utasitás):=orig(0) end
végreh=értékadás; do használt(1):=használt(0) end
értékadás=identifier':='unsigned_integer;
         do bal(0):=value(identifier);
            jobb(0):=value(unsigned_integer)
         out if névkeresés(használt(0),bal(0),index)<0
                then
             begin
             write.outtext("deklarálatlan változó:");
             write.outtext(bal(0));
             end else begin
             code.outtext("LDA#"); code.outtext(jobb(0));
             code.outtext("STA"); code.outtext(bal(0))
                end
             end
utasitás='begin' blokk 'end';
         do használt(blokk):=használt(utasitás);
            uj(utasitás):=orig(utasitás)
         end
end of productions
end of attribute grammars algszint
```

A mnemonikok a HLP programok változó részeinek kijelölésére szolgálnak. Az idéző jeleket tartalmazó, általában deklarációkat kijelölő sorok, beleértve az eljárás deklarációkat is, változtatás nélkül kerülnek át a HLP fordító által generált utkódba. Az egyes attributumok jelentése [Boc 76] alapján nyilvánvaló. Az attributum előfordulások azonositására, alkalmaztuk a HLP mindhárom jelölését. A % kezdetü sorok megjegyzések. A szintaxisbeli két token osztálynév value attributumát nem szükséges deklarálni. Az értékadás forditásának megadása során feltételeztük a tipus összeférhetőséget, és terjedelmi okokból eltekinthettünk a futtató rendszer és a célkód kapcsolatától.

3. Kiterjesztő mechanizmusok

A kiterjesztő mechanizmusok közös célja, hogy az un. kiterjesztő szöveget [Sol 71] a forditás különböző fázisaiban, a felhasználó által előirt módon vezessék vissza bázis nyelvi konstrukciókra. A felhasználó minden esetben egy speciális un. definiciós nyelven, hozzárendelést ir elő a kiterjesztő és a definiciós szöveg között. A definiciós szöveg maga is tartalmazhat kiterjesztő részszöveget. A makroprocesszorokat a kiterjesztő mechanizmusok legegyszerübb speciális esetének szokás tekinteni [Sim 76]. Amennyiben a definiciós nyelv része a bázisnyelvnek, önkiterjesztő nyelvről beszélünk. Ha a kiterjesztés /a kiterjesztő szöveg definiciós szöveggel való helyettesitése/ a szintaktikus/szemantikus analizis során/után történik meg, a kiterjesztő ill. definiciós szöveg természetesen nem jelsorozat, hanem annak valamely belső - pl. fa - reprezentációja. A szakirodalom megjegyzi, hogy a kiterjesztő mechanizmusok bonyolultsága miatt nem várható kiterjesztő nyelvek forditóit generáló hatékony, áttekinthetően használható compiler-compiler rendszerek létrejötte. Jó példa erre az RCC rendszer [Nap 80].

A kiterjesztő nyelvek létjogosultságát az indokolja, hogy nem sikerült minden felhasználót egyformán kielégitő nyelvek bevezetése. A kiterjesztés, alkalmasan választott bázis nyelv birtokában, - az adattipus/utasitás valamint a

szintaktikus/szemantikus kiterjesztés révén - bármely felhasználó számára lehetőséget ad a problémáját adekvát módon tükröző nyelv definiálására. A kiterjesztés különböző tipusainak bevezetése indokolt. A szöveg-szöveg, vagy A-tipusu kiterjesztés [Sol 71] ugyanis nem old meg minden problémát. Gondoljunk egy olyan bázis nyelvre, amely alaptipusként nem ismeri a tömböt. Ha a felhasználónak mátrix aritmetikára van szüksége, ezt A-tipusu kiterjesztéssel nem képes bevezetni. A fordítás kódgeneráláshoz közeli szintjén viszont, a célkód ismeretében, egy más tipusu kiterjesztéssel /esetleg az A-tipussal kombinálva/ könnyedén megteheti.

A bázisnyelv compiler-compiler rendszerrel történő implementálása esetén a kiterjesztések specifikációjára két lehetőség kinálkozik. Ha a definiciós nyelv nem része a bázis nyelvnek -ND eset-, a felirt fordítót lehet, annak teljes ismerete nélkül bőviteni a kiterjesztések definicióival. Hátránya az ujraforditás szükségessége, előnye a kiterjesztett nyelv fordítójának gyorsasága. Ha egy bázis nyelvet ugy akarunk kiterjeszteni, hogy a definiciós nyelv annak része legyen -D eset-, a fordító specifikációt kell bővitenünk a definiciós nyelv megadásával. A teljes fordítót ez esetben sem szükséges ismerni. Az ujraforditás után olyan fordító programot nyerünk, amely a futás során felismeri a kiterjesztések definicióit valamint a későbbi kiterjesztő szöveget is, és elvégzi a megfelelő szintü helyettesitést. Előnye nyilvánvaló, hátránya hogy a definiciókat felismerő, a fordítót önmódositó, valamint a kiterjesztő modulok a mindenkori fordító részét képzik.

A HLP rendszerben történt fejlesztések közül éppen a kiterjesztések bevezetését tartjuk minőségileg ujnak. A HLP/SZ compiler-compiler rendszer, a fentieknek megfelelően a kiterjesztés két tipusát ismeri és tipusonként mindkét - D és ND - esetet.

3.1. Kiterjesztés a lexikális analizis során

A továbbiakban A-tipusu kiterjesztésnek hivott eszköz

feladata tetszőleges, reguláris kifejezéssel leirható kiterjesztő szövegek bevezetése ugy, hogy a definiciós szöveg tartalmazhasson bázis nyelvi és kiterjesztő jelsorozatokat egyaránt. A kiterjesztő szöveg bázis nyelvi jelsorozatokra történő visszavezetése a lexikális analizis során valósul meg, és rendszerünk gondoskodik a kapott szövegek szüréséről valamint a helyettesitési eljárások végrehajtásáról is. A kiterjesztő mechanizmust, figyelembe véve a HLP lexikális metanyelvét, igyekeztünk a lehető legáltalánosabbá tenni ugy, hogy a szöveg-szöveg helyettesitések egyezzenek meg a véges állapotu automaták által indukált leképezésekkel. Ahogyan ez a későbbiekből látható, mind a kiterjesztő szövegek, mind a definiciós jelsorozatok osztálya bővebb mint a Chomsky féle L_3 nyelvosztály. Ez más szóval azt jelenti, hogy kiterjesztő mechanizmusunk a szokásos makro kiterjesztések lényeges általánositása.

A továbbiakban elsőként az A-tipusu kiterjesztés azon esetével foglalkozunk, ahol a definiciós nyelv nem része a bázis nyelvnek /ND eset/.

3.1.1. Az A-tipusu kiterjesztés ND esete

A korábbiak szerint, a kiterjesztést a bázis nyelv HLP nyelvü definiciójának bővitésével definiáljuk. Vezessük be elsőként a generátor kifejezés fogalmát.

Tekintsük jelsorozat változóknak és konstansoknak egy S halmazát. Az S halmaz valamely s_i elemére történő hivatkozáskor egy stringet jelölünk ki /ha konstans saját magát, ha változó annak aktuális értékét/. Tekintsük ugyanebben az értelemben egész tipusu változók, konstansok és kifejezések egy E halmazát. Jelölje P az S illetve E elemeiből képzett relációk halmazát. Ezek után a generátor kifejezéseket e három halmaz elemeiből az alábbi módon nyerjük:

(1) S minden eleme generátor kifejezés;

(2) ha g_1 és g_2 generátor kifejezés, akkor a g_1g_2 konkatenációjuk is az;

(3) ha $p_1,\ldots,p_n \in P$ és $g_1,\ldots,g_n$ generátor kifejezések,

akkor a $(p_1 \rightarrow g_1 \ldots p_n \rightarrow g_n)$ is generátor kifejezés és jelentése megegyezik a szokásos McCarthy kifejezésekével;

(4) ha g generátor kifejezés és $e \in E$ akkor $g*[e]$ is az, és jelenti a g generátor kifejezés által kapott jelsorozat e számu iterációját;

(5) a müveletek prioritása a reguláris kifejezéseknél megszokott.

Ahogyan ezt korábban láttuk, egy reguláris kifejezés terminális szimbólumból, karakter halmaz névből, token osztály névből, részkifejezésekből valamint az any és endofline szimbólumokból épül fel. Konstruáljuk meg a hozzá tartozó véges állapotu automatát ugy, hogy

1 a reguláris kifejezésbeli minden szimbólumhoz jegyezze fel a felismerés során hozzá tartozó résszöveget; ezekre az S halmazban rendre a szimbolum nevével és balról jobbra az előfordulás számával hivatkozhatunk

szimbólum név [előfordulási szám] [1]

formában; a (4)-beli g kifejezésben előforduló

szimbólum név [%]

részkifejezésben a % az iteráció ciklusváltozóját jelöli

2 iteráció esetén őrizze meg a végrehajtott iterációk számát; az E halmazbeli változók értékei éppen az iteráció számok; a hivatkozás

*[előfordulási szám] vagy + [előfordulási szám] [2]

formában történik.

Ezek után az A-tipusu kiterjesztéseket a HLP lexikális metanyelvi programjában az akció blokkokban definiáljuk

<token osztály név> <goto> == <generátor kifejezés> <goto 1> [3]

alakban, a következő szemantikával. Ha egy akcióblokk aktiv kezdő állapotba kerül a token osztály definiciójában konstruált automata is. Ha végállapotba kerülve felismert egy szöveget, rendelkezésre állnak a részsorozatok és az iteráció

számok. Ezt a fázist tekintjük a kiterjesztő szöveg felismerésének, a részsorozatok pedig az aktuális paraméterek. Ezek után a generátor kifejezésből kiszámitjuk az egyetlen /esetenként különböző/ definiciós szöveget. Kezdő állapotba hozzuk a ⟨goto 1⟩nevü akcióblokkot és átadjuk lexikális elemzésre a definiciós szöveget. Az elemzés befejezése után a ⟨goto⟩nevü blokkot aktiváljuk.

Kiterjesztő rendszerünknek számtalan előnye van az eddig ismertekhez képest. Csak a legfontosabbakat soroljuk fel röviden.

/1/ A kiterjesztés definiálása nem kiván lényeges uj ismereteket. Bevezethetők további és felhasználhatók a régebbi token osztály definiciók.

/2/ Az implementáció természetesen adódik és nem növeli számottevően a HLP fordító méreteit.

/3/ A kiterjesztés során nyert bázis nyelvi szöveg tetszőleges akció blokk alapján kerül lexikális elemzésre.

/4/ Kiterjesztő szövegként akár végtelen sokat is definiálhatunk egyszerre.

/5/ Ugyanaz a token osztálynév tetszőleges számu akció blokkban szerepelhet az eredeti, és a [3] értelemben is.

/6/ A kétfajta akció blokk előirással figyelembe vehetünk kontextus feltételeket is a kiterjesztés során.

Végül bemutatunk egy terjedelmi okokból egyszerü példát az A-tipusu kiterjesztésre, figyelembe véve a korábbi HLP programot. Feladatunk a vektor értékadás, szorzás és osztás kiterjesztéssel való bevezetése. Kiterjesztő utasitásaink az alábbi szövegszerkezetüek:

A[n]:=B[n] ; értékadás

$A[n]:=B_1[n]@\ldots@B_m[n]$; szorzás

$A[n]:=B_1[n]\%\ldots\%B_m[n]$; osztás,

ahol az n tetszőleges azonositó vagy egész konstans lehet. Az értékadás jobb oldalán eltérő müveleti jelek is előfordulhatnak. A szóközt nem tekintjük szignifikáns jelnek.

Feladatunk megoldása a következő. Definiáljunk egy karakter halmazt a müveleti jelek számára.

```
op='@%';
```

Vezessünk be további token osztályokat.

```
sp='';
ind=egész|azonosító;
azon_1=azonosító;    ind_1=ind;
azon = azonosító;
```

A kiterjesztő utasitásokat a következő token osztály definiálja.

```
vekt=sp azonl sp'['sp indl sp']'sp':='sp
     azonl sp'['sp indl sp']'sp
     (|op sp azon sp'['sp ind sp']'sp)*';';
```

Ezek után megadhatjuk a tulajdonképpeni kiterjesztés specifikációt a blokkl nevü akció blokk részeként.

```
vekt[blokk_1] ==>'begin' 'integer' 'i' ';'
'for' 'i' ':=1' 'step' '1' 'until' ind_1[1] 'do'
azonl[1]'[i]' ':='azonl[2]'[i]'
(*[1]=0 →';'|1=1 →
((op[%]='@' →'*'|op[%]='%' →'/')azon[%]'[i]')
*[*[1]]';') 'end' ';' [blokk3];
```

A generátor kifejezés által előállitott /egyetlen/ szöveg a blokk3 nevü akció blokkban kerül lexikális elemzésre, ahol természetesen lehetőség van további, pl. a for utasitásra vonatkozó kiterjesztés bevezetésére vagy a tényleges output /token nevek és terminálisok sorozatának/ generálására.

3.1.2. Az A-tipusu kiterjesztés D esete

A HLP lexikális metanyelvét kibővithetjük ugy, hogy lehetővé váljon a kiterjesztéseket előiró definiciós nyelv specifikációja. A definiciós nyelvet nem célszerü rögziteni, hiszen nem független a forrásnyelv utasitásaitól, valamint érdemes vezérelni a kiterjesztő definiciók hatáskörét is.

Ezek után egy A-tipusu kiterjesztés definiciós nyelvének specifikációja, a definiciós konstrukciók reguláris kifejezésekkel leirt szintaxisának megadását és a felismerésüket

végző akcióblokkok kijelölését jelenti. Maga a definiciós konstrukció kell hogy tartalmazzon minden információt a [3] előállitásához. Vezessünk be egy % speciális jelet a reguláris kifejezésekben és számitsuk ki a hozzátartozó automatát ugy, hogy felismerés során a forrásszövegben, a neki megfelelő helyen tetszőleges string állhat un. paraméter jelek között. A paraméter jel bevezetését az automata állapot tábla kiszámitása során fellépő shift-reduce tipusu konfliktusok elkerülése indokolja. Automatáink az elemzés során természetes módon feljegyzik a %-hoz rendelt részsorozatot is. Nevezzük paraméteres reguláris kifejezésnek azt, amelyben % az előbbi jelentéssel fordul elő.

A HLP lexikális metanyelvét ilyen feltételek mellett az alábbi módon bővitettük ki a definiciós nyelv előiráshoz.

<type-A extension>=type-A extension specification
parameter specifier=<character>
pattern definition=<par reg expr>
body definition=<par reg expr> [4]
definitions will be isolated at <action block name list>
patterns will be isolated at <action block name list>
bodies will be isolated at <action block name>
end of type-A extension specification.

A parameteres reguláris kifejezések helyett állhatnak azokat azonositó token osztálynevek is. Izolált első paraméterük rendre a kiterjesztő utasitást illetve a törzset, a második pedig az akcióblokkok nevét irja elő. Ez utóbbi nem kötelező, de felülbirálja a [4] utolsó két sorában adott alapfeltételezést. A HLP rendszer az előzőek hatására automatikusan generálja a definiciós nyelv utasitásait felismerő automatát, továbbá a megfelelő /automata ill. törzs generáló stb./ eljárásokat, a felirt forditóba. Mivel a definiciós nyelv maga is kiterjesztő, rendszerünket önkiterjesztőnek nevezhetjük. Megjegyezzük, hogy a megfelelő irodalom áttanulmányozása után sincs tudomásunk /hasonlóan jól használható/ önkiterjesztő compiler generátor rendszerről. Csak terjedelmi okokból te-

kintünk el e helyen, az ilyen kiterjesztés hasznosságát illusz
ráló példák bemutatásától.

3.2. Kiterjesztés a szintaktikus analizis során

Az A-tipusu kiterjesztés során a kiterjesztés bázis nyelvre történik. Ennek következménye, hogy nem vezethetünk be pl. tetszőleges adattipusokat és müveleteket, hiszen ezeknek bázisnyelvi konstrukciókra történő leképezése sokszor nehézkes. Feltételezve, hogy a célkód az architekturát jobban tükröző, alacsonyabb szintü nyelv része, természetes módon adódik a következő. Halasszuk el a kiterjesztést a szintaktikus elemzés fázisáig. Ez azonban azt jelenti, hogy a "kiterjesztő szöveg" valamely pontosan definiált attributumos részfa lesz és a kiterjesztés nem más, mint egy attributumos fatranszformáció. Ennek megfelelően a felhasználó számára ugy jelenik meg mint egy a szintaktikát/szemantikát tetszőleges ujra definiáló eszköz. Nevezzük B-tipusu kiterjesztésnek azt a mechanizmust, amely a fatranszformációt a szintaktikus elemzés fázisában hajtja végre.

A B-tipusu kiterjesztést is igyekeztünk ugy megtervezni, hogy maximálisan figyelembe vegye a HLP filozófiáját. Ezáltal könnyen elsajátitható a felhasználó számára, valamint egyszerüen implementálható is. Ahogyan az előző fejezetben láttuk, egy attributumos produkció három részből áll: a CF szabály, a szemantikus egyenletek és az attributumok által vezérelt kódgenerálási előirások. A kiterjesztés igy vonatkozhat /egymástól nem függetlenül/ mindhárom részre. Egy produkció tehát

⟨production rule⟩=⟨production⟩;⟨semantic part⟩⟨type-B extension part⟩ end.

formában irható. A kiterjesztő rész szerkezete a következő

⟨type-B extension part⟩=type-B extend⟨conditional tree transf list⟩.

⟨conditional tree transf⟩=⟨condition⟩⟨tree transf⟩⟨semantic part⟩.

A fatranszformációt előiró kifejezés itt is a McCarthy kifejezés megfelelője. A feltétel az elemzés fázisában kiértékelt szintetizált attributumokból képzett boole kifejezés, mig az uj szemantikus egyenleteket és kódgenerálást megadó szemantika rész változatlan. A fatranszformációt előiró kifejezésben előfordulhatnak metaszimbólum azonositók /ld. előző fejezetben/ bizonyos részfák kijelölésére, terminálisok, tokenek valamint olyan nemterminálisok, melyek egyértelmüen határoznak meg egy részfát. Pl.:

A=ABdA; do <semantic rule list>;out<translation rule list>;
type-B extend s(A)=0 → A_2cDA_1; do...; out...end

ahol D olyan nemterminális, amely egyértelmüen határoz meg egy részfát, s pedig a helyettesitendő részfa gyökerének szintetizált egy menetben alulról kiszámitható attributuma. Az indexek a nemterminális előfordulásait azonositják.

Amikor a szintaktikus elemző az A produkció szerint redukálna eldönti a feltétel teljesülését, és ha szükséges az uj részfát helyettesiti a régi helyére, majd elvégzi a redukciót. Tekintettel arra, hogy az attributum kiszámtási algoritmus /ASE vagy OAG/ determinisztikus, a B-tipusu kiterjesztések az eredeti HLP program részét kell, hogy képezzék. Ilyen esetben a generált forditóban nincs szükség az attributum kiértékelő stratégia ujbóli meghatározására. Hátránya viszont, hogy minden uj kiterjesztés bevezetésekor ujra kell generálni a forditót.

Járjunk el a továbbiakban ugy, mint az A-tipusu kiterjesztés esetében. Adjunk lehetőséget a felhasználónak, hogy bevezessen egy tetszőleges B-tipusu definiciós nyelvet HLP szinten. Ezen a nyelven deklarálhatja a forrásszövegben az uj attributumokat, nemterminálisokat, és a feltételes fatranszformációs listát, ez utóbbit a bázisnyelvi produkciókhoz egyértelmüen hozzárendelve. A generált forditó a definiciók ismeretében meghatározza tetszőleges levezetési fára az uj attributum kiértékelési stratégiát és elvégzi a kiterjesztésekhez a produkciók hozzárendelését /ha tudja!/. A definiciós nyelv HLP szintü leirásának bemutatásától terjedelmi okokból

eltekintünk. Természetes módon, nincs lehetőség a definiciós nyelv kiterjesztésére ezen a szinten, viszont egy korábban kiterjesztéssel kapott részfa maga is kiterjeszthető.

További terveink között szerepel a kiterjesztés egy harmadik tipusának bevezetése is. A C-tipusu kiterjesztés esetén a fatranszformáció az attributumok kiszámitása után történik meg feltételesen. A feltételek itt is az attributumokból nyert logikai kifejezések. A fatranszformációt ezen a szinten a VDL operátorához hasonlóan irhatjuk elő a levezetési fára vonatkozóan. Ezt követi megszokott módon a kód generálás.

4. Implementáció

A HLP első, Helsinkiben készült implementációja Extended Algol bázissal, az irodalom szerint mintegy 12 kutató év munkával készült el 1978-ban. Tekinttel arra, hogy nem állt rendelkezésünkre a finn implementáció leirása, egy informális kézikönyv [Räi 78] alapján fogtunk az implementáció tervezéséhez. Bázis nyelvként a SIMULA 67 nyelvet választottuk, messzemenően kihasználva az osztály fogalom által biztositott lehetőségeket. Elsőként az öt tipusu /LR(0), SLR(1), LALR(1), LR(1), ELR(1)/, szintaktikus elemző átmenet tábla generátorát, valamint a célkódbeli attributumos levezetési fát generáló elemzőt készitettük el. A starter és follower mátrixok [Gri 76] számitása során szükséges $O(n^3)$ komplexitásu tranzitiv lezárási eljárás helyett a bitmátrixokon, itt nem részletezett egyszerübb eljárást dolgoztunk ki. Az állapot táblákat optimalizált egyszerü programrészekkel realizáltuk helytakarékossági okokból.

Ezt követte a lexikális elemző implementálása, már a tervezés során figyelembe véve az A-tipusu kiterjesztést is. A HLP fordító bizonyos eljárásait párhuzamosan szintetizáltuk standard részekként a célkódba. A HLP lexikális elemzőjének állapotait a lexikális metanyelv szintaxisa alapján a már elkészült szintaktikus elemző állapot generátorával állitottuk elő és kézzel optimalizáltuk. A véges állapotu automata generáló eljárása része a célkódnak is, ha előfordul A-tipusu

kiterjesztés.

Következő lépésként megvalósitottuk az ASE és OAG kiértékelési strogiákat az attributumok függőségi halmazai alapján meghatározó két eljárást, valamint a generált fordítóba kerülő attributum kiértékelő algoritmust. Munkánk jelenlegi stádiumában a HLP HLP nyelvü definiciójának felhasználásával, a szemantikus egyenleteknek és kódgeneráló utasitásoknak a produkciókat SIMULA objektumokkal leiró részfák csomópontjaihoz történő hozzárendelését végezzük.

5. Befejezés

Dolgozatunk célja kettős volt. Egyrészt igyekeztünk röviden bemutatni a HLP metnyelvét és az irodalomra támaszkodva, az implementáláshoz szükséges algoritmusok lényegét. Másodsorban, és ezt tekintjük munkánk érdemi részének, terjedelmi okokból tömören ismertettük a HLP/SZ rendszerhez vezető lényegi módositásokat. Az öt szintaktikus elemzőt generáló eljárás az LR(0) állapotokat csak egyszer számitja ki, igy nem szükséges az sem, hogy a felhasználó maga tippelje meg gramatikájának tipusát. A rendszer a nyelvtan alapján a legszükebb a nyelvet tartalmazó osztály elemzőjét állitja elő. Az [Sip 78] alapján generált automatikus hibajavitónál megbizhatóbb nem várható a [Wet 78]-beli elméleti számitástudományi eldönthetetlenségi tételek miatt. Rendszerünk legerősebb részének a négy fajta kiterjesztést érezzük. Az utóbbi kettő a napjainkban folyamatosan születő számitástudományi tételek alkalmazásaként némileg változhat.

Abstract

HLP/SZ is a compiler writing system developed on the basis of Helsinki Language Processor to generate compilers of self-extending languages. The description machinery employed by the system for lexical and syntactic analysis consists of regular expressions, LR(0), SLR(1), LALR(1), LR(1) or ELR(1) grammars with semantic attributes. Code generation is performed during the final traversal of the attributed syntax tree. In

the description of the compilers of self-extending languages one can use string-transformations defined by regular and generator expressions. At the second level of extension attributed tree transformations can be applied. The system includes an algorithm for generating efficient lexical scanners, automatic generation of error recovery routines for an optimized LR parser. Evaluating of attributes is carried out by an Alternate Semantic Evaluator or an algorithm based on the ordering of the attribute occurences. Our implementation was developed on the base programming language SIMULA 67.

Keywords: compiler writing, lexical analysis, LR parsing, attribute grammars, language extension

Irodalom

[Aho 74] Aho, A.V. and Jonson, S.C.: LR parsing, Computing Surveys, 6,2. pp. 99-124.

[Boc 76] Bochmann, G.V.: Semantic evaluation from left toright, CACM, 19,2. pp. 55-62.

[Ear 70] Earley, J.: An efficient context-free parsing algorithm, CACM, 13,2. pp. 94-102.

[Gri 76] Griffiths, M.: LL(1) grammars and analysers, Lecture Notes in Computer Science, 21, Springer Verlag, pp. 57-83.

[Jaz 75] Jazayeri, M. and Walter, K.G.: Alternate Semantic Evaluator, Proceedings of the ACM Annual Conference, pp. 230-234.

[Kas 80] Kastens, U.: Ordered attributed grammars, Acta Informatica, 13,3. pp. 229-256

[Nap 80] Napper, R.B. and Fisher, R.N.: RCC - A user - extensible systems implementation language, The Computer Journal, 23,3. pp. 207-212.

[Räi 78] Räihä, Kaii-Jonko et al.: The compiler writing system HLP, Department of Computer Science, University of Helsinki, Report A-1978-2

[Sim 76] Simon, E.: Software eszközök programozási nyelvek és rendszerek definiálására, Egyetemi Doktori Értekezés, Szeged.

[Sim 80] Simon, E., Zachar, Z. és Gyimóthy, T.: Előkészületek a Perspektivikus Számitási Rendszer bázis gépi nyelvének hatékony tervezéséhez, OMFB Tanulmány, Szeged.

[Sip 78] Sippu, S.S. and Soisalon-Soininen.: On defining error recovery is context-free parsing, Proceedings of Fifth International Colloquium an Automata Languages and Programming, Lecture Notes in Computer Science, 62, Springer Verlag

[Sol 71] Solntseff, N. and Yezerski, A.: A survey of extensible programming languages, McMaster University, Hamilton, Technical Report No. 71/7.

[Wet 78] Wetherell, C.: Why automatic error correctors fail, Computer Languages, 2, pp. 179-186.

Simon Endre, Gyimóthy Tibor, Zachar Zoltán
MTA Automataelméleti Tanszéki Kutató Csoport
H-6720 Szeged, Somogyi u. 7.
Akadémia Székház

Székely Judit–dr.Szőke Péter

LÁNCOLT MEMÓRIÁJÚ ADA–GÉP, SZEMÉTGYŰJTÉS

Előadásunkban az ADA nyelvi rendszer futtatórendszerének a dinamikus tárgazdálkodást és ezen belül a szemétgyüjtést megvalósitó alrendszerével foglalkozunk. Az alrendszer célja a futó program dinamikus tárigényeinek kielégitése /ilyenek pl. egy task, subprogram, blokk, access-változó vagy dinamikus attributumú tömb adatterületei/.

Az előadás első részében
- az A-gép memóriájának az ADA nyelv tulajdonságainak megfelelően kialakitott szerkezetét, és
- a dinamikus tárgazdálkodás A-gépi utasitásait

ismertetjük.

Az előadás második részében az ADA futtatórendszer szemétgyüjtési algoritmusát irjuk le nagy vonalakban. Ez elsősorban
- az első részben leirt memóriaszerkezethez,
- az ADA nyelv által megengedett adattipusokhoz, és
- az A-gép adatábrázolási módszeréhez

alkalmazkodik.

Az előadásban tárgyalt problémákhoz hasonlóakkal foglalkoztak az ALGOL-68 nyelv kapcsán [2], [3], [4] és [5].

Kulcsszavak: ADA, dinamikus tárgazdálkodás, szemétgyüjtés, fastruktúra, láncolt memória, hosszspecifikáció, task.

1. Az ADA-gép memóriaszerkezete

1.1 Az ADA forditóprogram - legalábbis elvben - egy közbülső

kódra, az általunk A-kódnak nevezett nyelvre forditja le a forrásprogramokat. Az ADA nyelvi rendszerének egy konkrét megvalósitása történhet akár ennek a kódnak az elkerülésével /gépközeli alakot előállitó algoritmikus interface használatával, mint pl. [7]/, akár az A-kód továbbforditásával valamely gépközeli nyelvi alakra, de akár az A-kód közvetlen /esetleg mikroprogramozott/ interpretálásával.

Az A-kód értelmezhető úgy, mint egy hipotetikus A-gép által értelmezett kód. Előadásunk szempontjából ennek a gépnek az érdekessége a nem lineárisan szervezett memória. Pontosabban, mint később látni fogják, a gép memóriája láncolt szervezésü, lineáris darabokból álló tár, ahol esetenként egyes lineáris darabok nagyobb, egybefüggő darabokká olvaszthatók egybe, vagy kisebb, egymástól független láncszemekké eshetnek szét.

A tárgazdálkodást szolgálják a /futtatórendszerhez tartozó/ tárfoglaló /allokáló/ és felszabaditó /deallokáló/ A-gépi utasitások. Eme utasitások a rendelkezésre álló korlátos - virtuális vagy operativ - tár szabad helyeiből elégitik ki a tárigényeket, és a felszabaditott területeket oda teszik vissza. A dinamikus tárgazdálkodás utasitásait segiti az A-gép szemétgyüjtési utasitása, mely felderiti és felszabaditja a futó program által lefoglalt, de hivatkozásukat vesztett memóriaterületeket, ill. a rendelkezésre álló több kisméretü adatterületből létrehoz egy egybefüggő nagyobb területet.

1.2 Az ADA nyelvnek /lásd [1] / a tárgazdálkodást leginkább befolyásoló sajátosságai

- Párhuzamos folyamatok irásának lehetősége /ezért a tár nem lehet szekvenciális/;
- blokk-struktúra /ezért a statikus adatterületek használata verem jellegü/;
- dinamikus indexhatárú tömbök /a statikus adatterületek mellett szükség van dinamikus veremre is/;

- dinamikus helyfoglalás hivatkozási változók /access-változók/ számára /nem verem jellegü, úgynevezett "heap"/ memória/;
- hosszspecifikációk access tipusokra, illetve taskokra /ún. "pool" használatával/.

1.3 Az A-gép memóriája

Az A-gép adatok tárolására szolgáló memóriája láncstruktúrájú memória. Ezen azt értjük, hogy mind a foglalt, mind pedig a szabad tárterületek egymáshoz pointerek segitségével csatlakozó, lineárisan cimezhető darabkákból, "láncszemekből" épülnek fel.

1.3.1 A tár szabad területei egy kétirányban kapcsolt, változó elemméretü láncolt listát alkotnak, mely az A-gép "szabadhely"-regiszterén keresztül érhető el.

1.3.2 A foglalt területek egy fát alkotnak. A fa csúcsaiban találhatók az önálló adatterülettel rendelkező programegységek /az ún. "unitok", vagy "egységek", amelyek lehetnek taskok, blokkok vagy subprogramok/ adatterületei. Egy-egy ilyen terület maga is láncolt struktúrájú, azaz őket több lánc alkothatja. A láncszem szerkezete /kapcsolatainak mennyisége/ és mérete flexibilis, alkalmazkodik a láncszem felhasználásához. Erre majd később látunk példát. Mind a helyfoglalás, mind a felszabaditás ebben az esetben láncszemeknek az egyik listából a másikba való átfüzésével történik. Ennek során a láncszemek szükség szerint szeletelődnek, illetve a szabadhely láncban lehetőség szerint összeolvadnak.

A fa gyökere egy fiktiv task adatterülete. Ez a task biztositja az ADA program számára azt a minimális standard környezetet, amibe a program beágyazódik.

A fában egy csúcs "fiai":

- a, a csúcshoz tartozó unit által inditott taskok adatterületei, és
- b, egy általa hivott subprogram adatterülete.

Megjegyzés: 1. Egyszerre egy csúcsnak csak egy b, tipusú fia lehet.

2. Ha egy program nem indit taskokat, akkor a fa egy lineáris lista, mely a hagyományos, blokkstruktúrát megvalósitó veremnek felel meg.

A fa csúcsai közötti kapcsolat olyan pointereken keresztül valósul meg, amelyeket a program egyébként is használ, tehát e pointerek további tárterületet nem foglalnak le.

Egy egység adatterülete három részből áll:

1. Statikus terület - magában foglalja az egység adminisztrációs területét és az egység azon lokális /deklarált és ideiglenes/ adatainak tárterületeit, melyek mérete forditási időben kiszámitható. A statikus terület egy összefüggő terület, szerkezete forditási időben ismert, azaz a bennük elhelyezkedő értékeknek a terület kezdetéhez képest relativ cimei a forditóprogram által generált tárgyutasitásokba elhelyezhetőek.

2. Dinamikus verem - az egység dinamikus attributumú /deklarált vagy ideiglenes/ adatainak, mint például dinamikus indextartományú tömböknek, a memóriaterületei. A dinamikus verem egy egyszeresen láncolt lista, melyben a mutatók a verem legújabb elemétől a legrégebbi eleme felé mutatnak. Az egység statikus adatterülete tartalmazza az ún. "verempointert", mely a verem legújabb elemére mutat.

3. Access-lánc /másnéven heap-lánc/ - az unitban deklarált

access tipusokhoz tartozó objektumok adatterületei. Az ilyen objektumokat az ADA nyelv new utasitása hozza létre. Az access-lánc kétirányú lánc. Az egység statikus adatterülete tartalmazza az ún. "heappointert", mely a lánc fejéül szolgál.

1.3.3 Hosszspecifikált objektumok számára foglalt adatterületek

Az ADA nyelvben taskokra és access tipusokra elő lehet irni a hozzájuk tartozó objektumok maximális összméretét; ezt nevezzük hossz-specifikációnak.

Hossz-specifikált taskhoz illetve access tipushoz tartozó összes objektum számára egy egybefüggő területet jelölünk ki. Ezt pool-nak nevezzük. Mivel a task, illetve a hossz-specifikált access tipussal megadott new utasitások végrehajtása során adatterületeket kell foglalni, mégpedig a task illetve az access tipus számára már lefoglalt pool-ból, ezért egy-egy pool-ban a tárfoglalást szintén a dinamikus tárgazdálkodó alrendszer végzi, úgy mintha a teljes rendelkezésére álló terület csak a pool lenne.
Taskok és access tipusok számára különböző pool-okat használunk.

1.3.3.1 Task pool - a task végrehajtása során felmerülő minden tárigényt ebből a pool-ból elégitünk ki. Belső felépitése azonos a teljes tár felépitésével, azaz tartalmaz egy szabadhely-láncot és a foglalt területek fáját.

Maga a task pool a program /ill. az aktivizáló task/ foglalt területeinek fájára nincs felfüzve; ezzel szemben az első belőle leválasztott terület az adott task statikus területe, ez pedig fel van füzve a taskot aktivizáló egység ún. "függőségi lánc"-ára /ez az adott task által aktivizált taskok lánca/, igy közvetve fel van füzve a teljes pool is.

1.3.3.2 Access pool - egy hossz-specifikált access tipushoz rendelt tárterület. Ha egy olyan access változó számára kell helyet foglalni, mely az adott tipussal rendelkezik, akkor ezt a helyet a tipus pool-jából választjuk le.

Belső szervezését tekintve az access pool két láncot tartalmaz: a tipus objektumai számára lefoglalt területeket, mint egymással össze nem füzött láncszemeket, és egy szabadhely láncot.
A pool maga az access tipus deklarációját tartalmazó unit access-láncán helyezkedik el.

2. A dinamikus tárgazdálkodás A-gépi utasitásai

2.1 A gépnek 6 helyfoglaló utasitása van:

2.1.1 Allocate pool to task - hossz-specifikált task számára foglal területet

2.1.2 Allocate value - dinamikus attributumú értékek számára foglal helyet, és azt elhelyezi az egység dinamikus vermében.

2.1.3 Allocate access - access-változó által mutatott objektum számára foglal helyet, és felfüzi azt a tipus deklarációja által meghatározott access-láncba.

2.1.4 Allocate location - programegységek statikus adatterülete számára foglal helyet.

2.1.5 Allocate pool to access - hossz-specifikált access tipus számára foglal helyet.

2.1.6 Allocate in pool - access változó számára helyet foglal egy access pool-ban.

Ezek az utasitások a különböző adatterület tipusoknak megfelelően különböző adminisztrációs mezőkkel ellátott láncszemeket alakitanak ki, és az ebben levő pointerek állitásával létrehozzák a láncszemek közötti kapcsolatokat.

/A láncszemek különböző tipusait az előadáson fólián bemutatjuk./

2.2 A tárfelszabaditó utasitások száma szintén hat:

2.2.1 Deallocate location - egy egység számára lefoglalt területet szabadit fel, az összes hozzáfüzött területtel együtt /dinamikus verem és access-lánc, esetleg az egység statikus területét tartalmazó access pool/.

2.2.2 Deallocate dynamic value - a dinamikus verem legfelső elemének felszabaditása.

2.2.3 Deallocate dynamic value last but one - a dinamikus verem legfelső alatti elemének felszabaditása. /Akkor kell használni, ha egy dinamikus értéket adó müvelet operandusait az érték dinamikus veremre tétele előtt onnan nem lehet eltávolitani./

2.2.4 Deallocate on heap - egy access változó területének illetve egy hossz-specifikált access tipus pooljának felszabaditása.

2.2.5 Deallocate in pool - hossz-specifikált access tipusú objektum adatterületét szabaditja fel a tipushoz tartozó pool-ban.

2.2.6 Deallocate location and return value - egy érték visszaadó eljárás adatterületeit szabaditja fel, hasonlóan a deallocate location utasitáshoz; akkor kell ezt az eljárást hivni, amikor a visszatérési érték a felszabaditandó területen található. Az értékből egy láncszemet alakitunk ki, és azt a hivó egység dinamikus vermére felfüzzük.

A felszabaditott láncszemek adatterületeit nullázza az alrendszer, és belőlük szabad láncszemet alakitva felfüzi a szabadláncba, majd elvégzi a lehetséges összeolvasztásokat.

3. Szemétgyüjtés

Előfordulhat, hogy egy helyfoglaló utasitás nem talál megfelelő méretü szabad helyet, az adott tárigény azonban mégis kielégithető. Ez akkor áll fenn, ha a szabad terület elaprózódott, vagy az access láncokban olyan értékek találhatók, amelyekhez a program már nem tud hozzáférni, mivel a korábban reájuk mutató összes access változók értékeit valamilyen más értékkel felülirták.

A szemétgyüjtés feladata, hogy ilyen helyzetben a tárat úgy átszervezze, hogy az igényelt tárolóterületet biztositani lehessen. Ehhez

- felderiti és összegyüjti a hivatkozásukat vesztett adatterületeket, illetve
- a szabadlánc apró láncszemeit egyetlen nagyméretü láncszemmé olvasztja össze.

3.1 A szemétgyüjtéssel szemben támasztott követelmények

1. Szemétgyüjtés kezdeményezésekor általában nem áll rendelkezésre szabad adatterület, ezért a szemétgyüjtő algoritmusnak kerülnie kell dinamikus méretü munkaterületek használatát.

2. A csak a szemétgyüjtéssel kapcsolatos programfutás alatti tevékenységeket és adattárolást minimalizálni kell, hogy a szemétgyüjtést nem igénylő programok ezért a lehetőségért minél kisebb árat fizessenek.

3. A fordítóprogramban jól el kell különiteni a szemétgyüjtéssel kapcsolatos tevékenységeket végző részeket, hogy könnyen lehessen szemétgyüjtés nélküli tárgykódot generálni.

3.2 Az A-gép adatábrázolásának a szemétgyüjtést alapvetően befolyásoló tulajdonságai

- Minden érték /összetett értékek is/ a tárban összefüggő területen helyezkedik el;
- az egy objektumon belüli pointerek önrelativak, azaz relativak annak a tárolóelemnek a ciméhez képest, melyekben maguk találhatók. Ennek következtében belső pointerekkel rendelkező bonyolult értékek tárban történő mozgatása egyszerüen eltolás, mivel a belső pointerek változatlanok maradnak;
- az ADA gép memóriája az előző fejezetekben leirt láncstruktúrájú memória. A láncszerkezet olyan, hogy a szemétgyüjtésnek is alapegysége a láncszem /mind a bejárás, mind a szemét kigyüjtése, mind pedig a szabadlánc összeolvasztása szempontjából/.

3.3 A szemétgyüjtés algoritmusa nagy vonalakban

Az eddigiekben a dinamikus allokálás és deallokálás szempontjából úgy tekintettünk a memóriára, mint szabad és különböző célokra lekötött területek halmazára, amelyek között a helyfoglalás céljának megfelelően valamilyen a terület konkrét tartalmától független kapcsolat van. A memóriaterületek konkrét tartalmára semmilyen más szempontból sem volt szükségünk, igy az eddigi fa-, illetve láncstruktúrájú memóriaképünket is ennek megfelelően alakitottuk ki.

A szemétgyüjtés szempontjából az előzőeken túlmenően részben a lefoglalt memóriaterületek tartalma is érdekel minket, nevezetesen az, hogy a lefoglalt területeken elhelyezett változók hogyan hivatkoznak egymásra, és van-e közöttük olyan, amelyre a programban egyáltalán nincsen semmilyen hivatkozás.

A szemétgyüjtés szempontjából az A-gép tárolójában szabad helyek és értékek egy halmaza található. Ezek az értékek össze

lehetnek kapcsolva access változók /pointerek/ segitségével, és az első részben ismertetett láncszemekben találhatóak. Minden érték, mely a program számára még egyáltalán elérhető, elérhető a pointerek "láncán" valamely olyan értékből kiindulva, mely valamely egység statikus területén helyezkedik el.

Ebből a szempontból a tároló tartalmát egy általános irányitott gráfnak és szabad helyek egy láncának tekinthetjük. A gráfban a csúcsok a program által kezelt egyes objektumok /értékek/, az élek az ezeket összekapcsoló pointerek, a csúcsokba irt értékek mérete különböző, és a gráfban vannak körök is. Ez a gráf nem összefüggő, de a komponenseit a foglalt helyek fája összekapcsolja.

A szemétgyüjtés három fázisban valósul meg:

1. A programból elérhető értékek gráfjának a bejárása a statikus adatterületeken elhelyezkedő értékekből kiindulva. Közben az összes meglátogatott access-lánc-beli szemet megjelöljük /minden érték, mely nem komponense egy összetett értéknek, külön láncszemben található/. Egy egység t rületének bejárása után végig kell járni az egység access láncát, miközben a jelölés nélkül maradt láncszemeket át kell füzni a szabadhely láncba.

2. Összeolvasztás: a szabadhelyek lánca láncszemeinek egy nagy láncszemmé alakitása a memória átszervezésével. Lineáris, összefüggő memóriában a szabadhelyek lánca mindig cimek szerint növekvő sorrendbe van rendezve; itt a foglalt területeket a memória egyik végéhez eltoljuk. Közben táblázatot készitünk a foglalt terület egyes, eltolás előtt összefüggő darabjainak elmozdulásának mértékéről. Egy táblaelem a darab régi cimét és az eltolás mértékét tartalmazza.

3. Pointerek értékének módositása az eltolásnak megfelelően. Ismételten be kell járni az értékek gráfját, mint az első

fázisban, továbbá a foglalt területek fája láncszemeinek adminisztrativ területeit, közben az összes bennük talált pointertértékéből kivonjuk az elmozdulás nagyságát.

Megjegyzés: a három fázis közül nem minden esetben végezzük el mindet. A szemétgyüjtő müködését az ADA program irója pragmákkal és a forditóprogram opcióival szabályozhatja:

- Kérheti a szemétgyüjtő részleges használatát:
  - csak a hivatkozásukat vesztett objektumok kigyüjtése a szabadhely-láncba a memória átszervezése nélkül;
  - csak a memória átszervezése; hivatkozásukat vesztett objektumok kigyüjtése nélkül;
- A szemétgyüjtés ideiglenesen letiltható /ellenkező jelentésü pragma észleléséig/;
- Szemétgyüjtés nélküli programkód generálása;
- Szemétgyüjtés inditása a programozó által adott pragmával.

3.3.1 Értékek gráfjának a bejárása

Ahhoz, hogy a program számára hozzáférhető minden értéket meglátogassunk, sorra kell venni minden egyes egység statikus adatterületét és az onnan induló összes pointer-láncot be kell járni. Az egységek fájának bejárása a fiktiv gyökértasktól indul. Sorravesszük a task-láncokra felfüzött taskok statikus adatterületeinek és /ha van/ az aktuális hivott subprogramnak az adatterületét. Ezekből, mint programegységek adatterületéből ismételten task- és subprogram-láncok indulnak ki. Közismert fabejárási algoritmusok biztositják, hogy minden statikus adatterület sorra kerüljön. Az egyes statikus adatterületeken belül a területen az adott pillanatban található deklarált illetve munkaváltozókból kiinduló pointer-láncokat kell bejárni. Ezek, mint már emlitettük, irányitott gráfot alkotnak, bejárásuk a fabejáráshoz hasonló, azzal a kiegészitéssel, hogy ha a bejárás során egy más uton egy korábban már megje-

lölt láncszemhez jutunk, akkor ezen az ágon nem folytatjuk tovább a bejárást. A gráf bejárására nem használunk vermet, ehelyett [6] "pointer-forgatásos" módszerét alkalmazzuk.

3.3.1.1 A szemétgyüjtéssel kapcsolatos forditási és futási tevékenységek

A szemétgyüjtő csak akkor tud müködni, ha számára mind a futóprogram /szemétgyüjtésen kivül/, mind a forditóprogram bizonyos előkészitő tevékenységeket végez /információt gyüjtenek/. A szemétgyüjtést sok program egyáltalán nem fogja hivni, ezért célszerü az előkészitő tevékenységekkel a lehető legkevésbé terhelni a futóprogramot. Ennek megfelelően az A-gép szemétgyüjtéséhez az ADA forditóprogramnak elég bonyolult információgyüjtést kell végeznie.

3.3.1.2 A forditóprogram előkészitő tevékenységei

Mind a statikus adatmezők tartalma /deklarált és ideiglenes objektumok/, mind pedig az, hogy az ADA nyelvben lehetséges végtelen sok tipus közül a konkrét ADA programban, mely tipusok /véges sok/ fordul elő, magától a konkrét ADA programtól függ, igy az ezekre vonatkozó információkat a forditóprogram a kódgenerálás során feljegyzi a szemétgyüjtő program számára. A programban előforduló adattipusok szerkezeti tulajdonságait egy ún. tipustáblába jegyzi fel, a statikus adatmezők a program futása során előforduló összes lehetséges állapotáról pedig unitonként egy-egy térképet készit /mindkettő részletesebb leirását lásd később/. A kódgenerátor feladata fentieken kivül az, hogy a leforditott kódba elhelyezze azokat az utasitásokat, amelyek a szemétgyüjtőt a program pillanatnyi állásáról tájékoztatják /GCP állitón utasitások - lásd később/.

3.3.1.3 A térkép felépitése

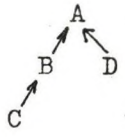
A térkép fastruktúrát alkot. A fa csúcsai egy-egy konkrét ér-

ték attributumait tartalmazzák /érték statikus területen belüli cime, tipusa/, és egy pointert. A fának egy csúcsától a gyökérig vezető útja mentén elhelyezkedő bejegyzések a statikus adatterület egy pillanatnyi állapotát tükrözik.

Példa: változzon a verem állapota az alábbiak szerint:

```
A  A  A  A  A
   B  B     D
      C
```

Akkor a térkép igy néz ki:



A térképben a pointerek a levelektől a gyökér felé mutatnak. Igy ha ismerjük a statikus adatterületen a legmagasabb cimen elhelyezkedő értékhez tartozó térképelemet, akkor a térképben levő pointerek mentén a gyökérig találhatók a statikus adatterületen alacsonyabb cimeken található értékeket leiró térképelemek.
Az olyan értékek számára, melyekből nem indul pointerlánc a heap-be, térképelemet készitени nem kell.

3.3.1.4 A tipustábla felépitése

A tipustáblából minden tipusról megállapitható, hogy milyen komponensekből áll, és hol vannak benne pointerek. A térkép egyes elemei a tipustábla elemeire hivatkoznak. A tábla egy gárf, melynek csúcsai az egyes tipusokra vonatkoznak, élei pedig a tipusok közötti összefüggéseket tükrözik. A térképet és a tipustáblát a forditóprogram minden programhoz elkésziti, és a programfutás idejére eltárolja /külső adathordozón/. A generált kód ezeket a táblázatokat addig nem használja, mig a szemétgyüjtés meg nem indul.

3.3.2 A futóprogram egyetlen szemétgyüjtés előkészitő tevékenysége az ún. szemétgyüjtő-pointer /GCP/ rendszeres állitása. Ennek a pointernek mindig a statikus adatterület tetején levő legfelső, a szemétgyüjtő szempontjából szignifikáns elem térképbeli azonositóját kell tartalmaznia. A pointer értéke minden olyan alkalomkor változik, amikor a statikus területre egy új értéket /vagy értékre mutató pointert/ teszünk, amely komponensként access változót tartalmaz. Ilyen értéknek a veremből történő levételekor is változik a GCP tartalma.

A forditóprogramnak tehát a térkép és tipustábla épitésén túlmenően GCP állitó utasitásokat kell generálnia a fent jelzett pontokban. Mivel a térképelem térkép-relativ cime /térképbeli azonositója/ ilyenkor már rendelkezésre áll, az utasitás generálása nem okoz nehézséget.

A forditóprogram értelmezi továbbá a szemétgyüjtéssel kapcsolatos pragmákat, és ha szükséges, generálja a pragmával kért közvetlen szemétgyüjtő hivást.

Abstract

We outline in our note the dynamic storage management and garbage collector subsystem of the run-time system in the hungarian ADA implementation. The goal of the subsystem is to meet the dynamic memory request of the running program /e.g. allocating data areas for a task, a subprogram, a blokk, an access-variable, or an array with dynamic attributes/.

In the first part of this note we
- draft the structure of the A-machine developed according to the properties of the ADA-language, and
- review the commands of the dynamic storage management of the A-machine.

In the second part of this note we draft the garbage collector algorithm. This algorithm is in keeping with the memory structure described in the first part of this note, with the data types promided by the ADA language and with data representation of the A-machine.

Irodalom

1. Reference Manual for the ADA Programming Language, US DoD, 198o. július

2. ALGOL 68 Implementation Ed. by J.E.L. Peck, North-Holland, 1971.

3. Dinamicseszkoe raszpredelenija pamjati, Szőke Péter, Kandidátusi Disszertáció. Leningrádi Állami Egyetem

4. Compiler Construction. Ed. by K.L. Bauer and J. Eickel, Lecture Notes in Compiler Science 21., Springer-Verlag, 1974.

5. P. Branquart, J.P. Cardinael, J. Lewi, J.P. Delescaille, M. Vanbegin: An Optimized Translation Process and its Application to ALGOL 68
Lecture Notes in Computer Science 38.
Springer-Verlag, 1976.

6. G. Schorr and W. Waite: An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures, CACM, august 1967.

7. Per Holager: Description of an Interface to the Machine-Dependent Code Generator of the CDL2 compiler
Informatical Computer Graphics Facultaty of Science, Nijmegen University; Netherland 1979.

8. B.K. Haddon and W.M. Waite: A Compaction Procedure for Variable-Length Storage Elements
The Computer Journal; vol. 1o. 1967.-68. pp. 161.

Székely Judit
Számitógépalkalmazási Kutató Intézet, 1536. Pf. 227.
dr. Szőke Péter
OMKDK KF-IIR csoport 1428. Pf. 12.

Szigetvári Miklós

BATCH MONITOR RENDSZER AZ SVS/TSO KÖRNYEZETBEN

Rövid kivonat:

Az előadás egy saját fejlesztésü batch monitor rendszer ismertet, amely a KSH-SZK-ban készült az SVS/TSO operációs rendszerhez. A rendszer segitségével hatékonyan használhatók azok a programok /elsősorban forditóprogramok/, amelyekre szüksége van az előtéren dolgozó felhasználónak, de erőforrásigényük miatt nem futtathatók előtéren. A felhasználó egy előtér utasitásfeldolgozónak fogalmazza meg a feladatait, amely egy SVC rutinon keresztül sorbaállitja azt a monitorprogram számára.

Kulcsszavak:

SVS /Single Virtual Storage/, TSO /Time Sharing Option/, forditóprogram, nem interaktiv végrehajtás, előtér, háttér.

Bevezetés

Interaktiv környezetben a legáltalánosabb feladatsorozat az, hogy a felhasználók javitják programjaikat; felhivnak valamilyen forditóprogramot; összeszerkesztik a programjukat majd tesztfutásokat végeznek. Erre a feladatsorozatra megfelelő programok állnak rendelkezésre az IBM SVS /Single Virtual Storage/ operációs rendszerben.

Számunkra a legnagyobb problémát az jelentette, hogy a TSO rendszerben nem állnak rendelkezésre megfelelő eszközök a nem interaktiv munkavégzésre /pl. programforditás/. Ezt az IBM ujabban kifejlesztett szoftver termékei több kevesebb sikerrel megoldották /gondolunk itt a VM/CMS batch gépre, a VSPC hasonló szolgáltatására vagy akár az SPF megosztott képernyőjére/ másrészt azt is világosan kell látni, hogy a számitógépek teljesitményének növelésével a probléma lényegtelenné válik /nincs erőforráskorlátozás/.

Tisztában vagyunk azzal, hogy az általunk megoldott feladat csak egy "korszerütlen" operációs rendszer lehetőségeit növeli meg egy aránylag kis teljesitményü számitógépen, azonban azt mindenkinek tudomásul kell vennie, hogy még hosszu ideig Magyarországon hasonló teljesitményü gépeken és "elavult" operációs rendszerekkel kell megoldani a számitástechnikai feladatok legnagyobb részét.

1. Lehetőségek a forditóprogramok használatára TSO környezetben.

A TSO /Time Sharing Option/ rendszer legfontosabb jellemzője az ,hogy az előtérben futó felhasználók váltakozva használják ugyanazt a memóriaterületet. Általában négy-öt terminál dolgozik egy memóriaterületen /egy region-ben/ és a rendszer az egymás mellett futás régiok között osztja szét a központi egység idejét. Ha egy felhasználó elhasználta a rendelkezésre álló időszeletet vagy valamilyen oknál fogva várakozik, akkor az általa használt memóriatartalom mágneslemezre kerül /swap out/ és a következőnek ütemezett felhasználó

A másik lehetőség az lenne a forditóprogramok futtatására, hogy a felhasználók előtérben job-okat állitanak össze és a SUBMIT TSO utasitás segitségével átküldik azokat batch feldolgozásra.
Ennek az eljárásnak is lényeges hátrányai vannak:

- Minden esetben job-ot kell összeállitani a TSO szerkesztőprogram segitségével.
- Az elkészült job-ot át kell küldeni a háttérre batch feldolgozásra. Az átküldés folyamata időigényes, a jelenlegi SVS/HASP rendszerben a job-ot tartalmazó állomány többszöri átmásolását jelenti.
- A fenti problémákon saját fejlesztéssel lehetne segiteni, azon azonban már nehezen, hogy az operációs rendszer egyik legnagyobb erőforrásokat lekötő tevékenysége a jobok és job-lépések inicializálása, terminálása. Minden inicializálás és terminálás a rendszer legjobban igénybevett adatállományához a job sorhoz /SYS1.SYSJOBQE/ való többszöri hozzányulást jelent, ezenkivül az induló vagy leálló joboknak többször sorba kell állniuk a csak sorosan használható erőforrásokért /ENQ/DEQ/.
- A TSO renszer egyik legnagyobb előnye az, hogy dinamikusan szabályozható az előtéren futó interaktiv munkák és a háttéren futó batch jobok aránya. Ha az előtéren a felhasználók száma magas /nálunk kb. 20 terminál/ akkor az elfogadható válaszidő elérése érdekében csökkenteni kell a háttéren a multiprogramozás szintjét. A megmaradó háttér kapacitást célszerü olyan munkák részére fenntartani, amelyek vagy nagyon sürgősek vagy nagyon fontosak valamelyik TSO felhasználó számára, de előtérben nem végezhetőek el.

A fenti problémák felismerése után született meg egy batch monitor rendszer a forditóprogramok futtatására, amelynek a KSHBATCH nevet adtuk.

kerül a memóriába /swap in/.

A müködés ilyen rövid ismertetéséből is kiderül az, hogy a rendszer hatékonysága szempontjából jelentős tényező az, hogy egy felhasználó milyen nagyságu virtuális memóriát használ /region-méret/ és az is, hogy a virtuális memória lapjai közül hány darabra történik hivatkozás.

Ebben a helyzetben a forditóprogramok előtérben való használata több problémát vet fel:

- A forditóprogramok memóriaigénye igen nagy. Gépünk IBM 370/155 1 Mbyte valós memóriával ezen az "aránylag" kis gépen nem engedhettük meg azt, hogy a TSO felhasználók 128 Kbyte-nál nagyobb memóriát használjanak. Ebben a memóriában azonban a forditóprogramok nagy része egyáltalán nem vagy csak nagyon rossz hatékonysággal üzemelhet.
- A forditóprogramok által használt aktiv alpok száma /swap load/ igen magas. TSO nyomkövetési adatok alapján mig az EDIT szerkesztő utasitás kb. 40-44 Kbyte aktiv memóriát használ addig ez az érték néhány forditóprogramnál:

  COBOL 140-156 K
  Assembler 104 K
  FORTRAN 100-136 K

- A forditóprogramok futásához igen sok adatállomány szükséges. Ezeknek az állományoknak a dinamikus allokálása a TSO DAIR /Dynamic Allocation Interface/ lehetőségével megoldható, azonban a dinamikus allokálás elvégzése elég jelentős többletmunkát jelent az operációs rendszer számára.
- Az eddig felsoroltakon kivül, véleményünk szerint nagyon lényeges az, hogy a programok forditása tulajdonképpen nem interaktiv munka. A feladat megfogalmazása után, ha a forditás elindult, akkor nincs párbeszéd a rendszer és a felhasználó között, a képernyőn megjelenő forditási lista nem használható fel programjavitásra. A forditás idejére a felhasználó ki van zárva a rendszerből, nem végezhet más munkát.

2. A KSHBATCH rendszer

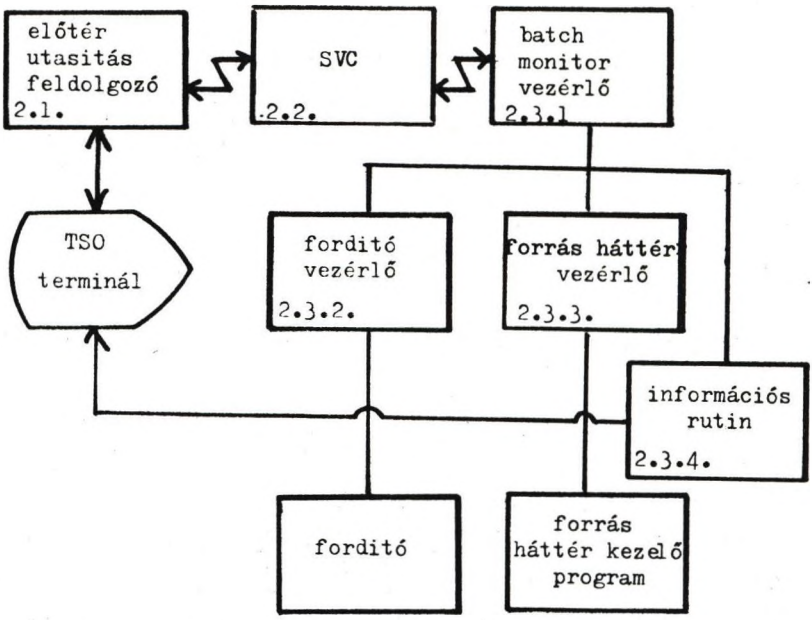
A rendszer lényege az, hogy a felhasználók előtéren megfogalmazott forditási igényei egy a háttéren futó magas prioritásu monitorprogramokhoz kerülnek és ez a monitorprogram végzi el a forditóprogramok futtatását. Ez a koncepció hasonló a VM/370 operációs rendszer CMS batch gépéhez, azonban hasonló apparátus az SVS/TSO rendszerben nem állt rendelkezésre. A rendszer használata közben derült ki, hogy lehetőségeit más feladatok elvégzésére is fel lehet használni:

- futtathatók a segitségével olyan általános célu programok /táblagenerátorok, file-kezelő rendszer/ amelyek előtér futtatására gondolni sem lehetne.
- szétoszthatő a felhasználók között olyan erőforrás is, amelyet rövid ideig használnak ugyan, de használata nagyon lényeges /lásd 2.3.3. Forrás háttér rendszer/

A KSHBATCH rendszer három részből áll:

- előtér utasitás-feldolgozók a feladatok megfogalmazására
- saját fejlesztésü SVC /supervisor call/ rutin a feladatok sorbaállitására a batch monitor számára
- batch monitorprogram a forditóprogramok futtatására.

Az egyes részek kapcsolatát és a batch monitor felépitését /task strukturáját/ az 1-es ábra szemlélteti.



2.1. Az előtér utasitásfeldolgozók /command processors/
Az utasitásfeldolgozók segitségével határozza meg a felhasználó azt, hogy milyen forditóprogrammal milyen programot akar leforditani. Az utasitásfeldolgozó ellenőrzi azt, hogy a szükséges állományok léteznek-e /input, lista, object stb./. Ha valamelyik output állomány nem létezik, akkor létrehozza ezt az állományt. Az ellenőrzések és allokálások befejezése után az utasitásfeldolgozó egy kb. 100-200 byte nagyságu vezérlő blokkot állit össze, feltüntetve itt a felhasználó azonositóját a terminál számát és a feladat tipusát /a feladat tipusáról lásd 2.3.3./

A különböző forditóprogramok futtatásához szükséges utasitásfeldolgozókat egy assembler makró segitségével generáljuk. A makró a szükséges paraméterek megadása után előállitja a kivánt utasitásfeldolgozót.

A makró hivásakor meg kell adni:
- a forditóprogram nevét

- a forditóprogram alapértelmezés szerinti paramétereit
- a szükséges állományok rögzitett neveit /DD név/
- az esetleg allokálandó output állományok méret paramétereit /OS job control SPACE/ és az állományok jellemző attributumait /DCB/

Egy utasitásfeldolgozó elkészitése 6-8 sor kódolását jelenti. Jelenleg a következő forditóprogramokat, általános célu programokat futtatjuk a KSHBATCH rendszer felügyelete alatt:

ASSEMBLER F és G
VS/COBOL
PL/1 Optimizer
FORTRAN G és H
CDL 2
PASCAL
TPL /az USA Munkaügyi Hivatal táblagenerátora/
TAB68 /a Svéd Statisztikai Hivatal táblagenerátora/
MARK IV /az Informatics cég file-kezelő rendszere/

2.2. Az SVC rutin

Egy saját készitésü SVC rutin végzi a különböző felhasználók igényeinek sorbaállitását. Megvizsgálja, hogy az adott felhasználó számára hozzáférhető-e a batch rendszer várakozósora /ENQ/. Ha igen, akkor a felhasználó igényét tartalmazó vezérlő blokkot a batch monitor memóriában lévő várakozósorának a végére illeszti és értesiti a monitort egy ujabb feladat érkezéséről /POST/, majd elengedi a memória queue-t /DEQ/. Az igények sorbaállitásánál igyekeztünk igazságosak lenni, beérkezési sorrend szerint állnak sorba /FIFO queue/.

2.3. A batch monitor program

A batch monitor program strukturája, az egymás mellett illetve alatt futó taskok szerkezete az 1. ábrán látható. A monitor program a TSO inditásával egyidőben indul egy magas prioritásu batch particióban. Az egyidőben aktiv taskok száma 3 és 5 között változik.

2.3.1. A batch monitor vezérlő

Indításkor a vezérlő task elvégzi a szükséges közös táblázatok inicializálását, elinditja az alatta futó három taskot: - fordítóprogram vezérlő
- forráskönyvtár háttérrendszer vezérlő
- információs rutin.

Ha az operátor leállitja egy STOP utasitás segitségével a KSHBATCH rendszert akkor a vezérlőtask leállitja a subtask-jait és befejezi a munkát.
A rendszer inditásához szükséges job kártyák között található minden olyan előre definiált könyvtár és munkaállomány allokálása, amelyekre szükség lehet valamelyik fordítóprogram futtatásához.

2.3.2. Fordító vezérlő

A fordító vezérlő az SVC rutin által felépitett sorból előveszi a következő feladatot. A fordítóprogram inditása előtt elvégzi a fordításhoz szükséges felhasználói állományok dinamikus allokálását. A dinamikus allokáláson most azt kell érteni, hogy átirja a program a job vezérlő blokkjait /TIOT, JFCB/ olyan formára, hogy a fordítóprogram számára ugy tünik, hogy a felhasználói állományok már a job indulásakor allokálódtak.

A fordítóprogramok a fordító vezérlő alárendelt subtaskjaként futnak. Erre azért van szükség mert bármilyen hiba előfordulása esetén biztositani kell a folyamatos müködést. Az alárendelt subtask abnormális befejezésekor az operációs rendszer végzi el a lefoglalt állományok és a memória felszabaditását. A fordítások befejezése után a vezérlő elszámolja a fordítás során felhasznált erőforrásokat /CPU idő, input/output müveletek száma stb./. Összeállit egy saját rekordot az elszámoló alrendszer /SMF/ számára és a felhasználó számlaszámával ellátva mágneslemezre irja. A fordítás befejezéséről üzenetet kap a felhasználó. Az üzenet tartalmazza a visszatérési kódot és a felhasznált erőforrások alap-

ján számitott pontszámot. A forditási listák átnézését, programjavitást előtérben interkativ uton végzi a felhasználó. /A lehetőségekről lásd Páldi Vince: TSO fejlesztések a KSH SZK-ban cimü előadását./

2.3.3. Forrás háttér kezelés vezérlő

Számitóközpontunkban hosszu ideje müködik a széles körben ismert BÉTA forráskönyvtárkezelő rendszer. A rendszer lényeges eleme az, hogy az aránylag régen használt programok nem a lemezes könyvtárban hanem egy több kötetes háttérszalagrendszeren helyezkednek el. Ha a felhasználó egy olyan programot kiván javitani vagy forditani, amely háttérszalagon van, akkor le kell futtatni egy programot amely visszahozza a kivánt programokat mágneslemezes könyvtárba.

Rendszerünkben a TSO felhasználók részéről nagyon gyakori az ilyen igény, ezért célszerünek látszott biztositani azt, hogy ilyen szolgáltatások a forditóprogramokkal párhuzamosan is futtassanak.

A forráskönyvtárral kapcsolatos munkák egy különálló feladattipust jelentenek, ezekhez egy önálló várakozó sor tartozik.

2.3.4. Információs rutin

Tapasztalatunk az, hogy mig 20 felhasználó esetén is igen ritka az, hogy két-három elemnél hosszabb várakozó sor alakuljon ki, ennek ellenére célszerünek látszott egy információs rutin elkészitése, amely kérésre kiirja a KSHBATCH várakozó sorában álló feladatok jellemzőit.

A KSHBATCH rendszer 1980 első félévétől kezdve üzemel elfogadható válaszidőkkel és kielégitő megbizhatósággal. Ugy érezzük, hogy rendszerünk megoldja a közepes méretü gépeken a TSO használata során jelentkező problémák egy csoportját.

Abstract

This paper describes an own developed batch monitor system wich was made in the Hungarian Central Statistical Office Computing Center for the SVS/TSO operating system. With this product the users effectivly can use those programs /mainly compilers/ wich are needed for the foreground users but because of their resource requirements they can not run in foreground regions. In foreground the user defines his task to a command processor wich queues the request to the batch monitor through an SVC routine.

Szigetvári Miklós
KSH Számitóközpont
1024 Budapest
Budai László u. 1-3.

Sztrókay Kálmán

INTERAKTÍV ADATBÁZIS-LEKÉRDEZŐ RENDSZER

Az Állami Népességnyilvántartó Hivatal céljaira kifejlesztett adatbázis lekérdező rendszer a terminálok kezelésének módja, valamint az ÁSZSZ számitógépének szük keresztmetszetéből adódó lemezkezelési és memóriagazdálkodási problémák megoldása folytán tarthat számot hasonló kérdésekkel foglalkozó más HWB felhasználók érdeklődésére.

Az Interaktiv adatbázis-lekérdező rendszert, amelyet az Állami Népességnyilvántartó Hivatal adatállományának az ÁSZSZ HWB 66/20 számitógépén történő interaktiv lekérdezésének céljából, az adatállomány elérésére jogosult szervek egyedi lekérdezési igényeinek kielégitésére, valamint az ÁNH belső használatára fejlesztettünk ki, batch üzemmódban futó felhasználói programként terveztük meg, kihasználva a GCOS operációs rendszernek azt a lehetőségét, hogy a felhasználó termináljairól kapcsolatot létesithet és párbeszédet folytathat programjával. Ennek megvalósitását a GMAP assembler nyelv szintjén az operációs rendszer a MME GEROUT /Master Mode Entry, Remote Output Record/ utasitással biztositja.

A lekérdező rendszert az adatállomány index szekvenciális felépitésével összhangban elsősorban kulcs /személyi szám/ szerinti lekérdezés céljaira fejlesztettük ki, de lehetőséget biztositottunk más szempontok szerinti lekérdezésre is /például: adott korosztályhoz tartozó adott vezetéknevü, adott környéken lakó, adott helyen, napon született

személyek stb./. Lehetőség lesz az adatállomány update-olására is, ilymódon távlatilag a jelenlegi központi feldolgozásu változásjelentési rendszert bizonyos esetekben felválthatja a változásoknak az Interaktiv adatbázis-lekérdező rendszer utján az adatállományba történő közvetlen átvezetésének lehetősége.

A személyi rekordok lekérdezésekor a terminálra a felhasználó kivánsága szerint dekódolt formában a fontosabb adatok, vagy dekódolás nélkül a teljes rekordkép iratható ki. Soros lekérdezés esetén a lekérési szempontoknak eleget tevő rekordok egymás után jelennek meg a képernyőn, a lekérdezés bármely rekord kiiratása után megszakitható. Printer lista készithető a teljes rekordokról eredeti és dekódolt formában - kivánság szerint a munkamenet során lekérdezett összes rekordról, vagy a terminálra éppen kiirt rekordról. A felhasználó közvetlenül is meghivhatja az adatállomány szegmenseit felkérő, lekérő szubrutint, ez azonban általában felesleges, mert ez a szubrutin az adatállomány nem online részére irányuló lekérdezési igény esetén automatikusan meghivódik, és kiadja az operátornak a szükséges lemez felrakására a parancsot. Szabad lemezegység hiányában a rendszer az online lemezek közül keres olyat, amelyről meghatározott idő óta nem történt lekérdezés, és annak helyére rakatja fel a kért lemezt. A lekérdezési igényt az Interaktiv adatbázis-lekérdező rendszer csak akkor utasitja vissza, ha ilyen lemezt nem talál.

Feladatunk néhány szempontból alapvetően különbözött a szokásos felhasználói problémáktól:

1. Az Állami Népességnyilvántartó Hivatal adatállománya lényegesen több lemezen helyezkedik el, mint ahány lemezegység egyidejü használatára az ÁSZSZ számitógépén lehetőség van. Ezért a lemezekkel és lemezegységekkel történő dinamikus gazdálkodás érdekében meg kellett valósitanunk a lemezigényeknek a lekérdező rendszerből történő figyelését, a lemezhasználat nyomon követését, ki kellett alakitanunk

a lemezek felrakásának és levételének céljára szolgáló operátori kapcsolat eszközeit. Ezenkivül a file-ok logikai leirására szolgáló file-kódokkal is olyan takarékosan kellett gazdálkodnunk, hogy ne lehessen egyszerre több filekódunk, mint amennyit az operációs rendszer megenged.

2. Az Interaktiv adatbázis-lekérdező rendszernek egyszerre számos terminállal kell kapcsolatot tartania. Ezért ahhoz, hogy a felhasználóknak ne kelljen várniuk egymás input-output müveleteinek befejeződésére, az input-output müveleteket még a GMAP assembler programozásban általánosan használt File and Record Control szintnél is mélyebben, az Input-Output Supervisor szintjén kellett megvalósitanunk. Az irógép tipusu terminálok viszonylagos lassusága miatt különös jelentősége van annak, hogy lekérdező rendszer vezérlőmodulja a várakozási idő csökkentése érdekében az egyes terminálóknak egyszerre csak egy input-output müvelet kiadásának idejére, illetve az input-output müvelet befejeződése után a válasz feldolgozásának idejére adja át a vezérlést. Ennek érdekében a lekérdező rendszert legfeljebb egy input-output müveletet tartalmazó modulokból épitettük fel. A vezérlésátadás megvalósitása, a megfelelő modulok meghivása a terminálok pillanatnyi állapotát tükrözö terminál táblázat segitségével történik.

3. Az ÁSZSZ gépén a felhasználói programok számára rendelkezésre álló viszonylag kis memória takarékosságra ösztönzött. Ezért az input-output müveletekhez szükséges adatterületeket, buffereket dinamikusan kezeljük, csak az input-output müvelet idejére rendeljük hozzá egy adott terminálhoz, a müvelet befejeződése után azonnal ujból felhasználhatóvá tesszük, a file-ok kezeléséhez /file allokálás, elengedés, státusz lekérdezés, stb./ egyetlen többcélu buffert alkalmazunk. A lekérdező rendszer által kezdeményezett mágneslemezes output müveletekhez /hardcopy készités, elszámolási információk gyüjtése, stb./ a system standard formátumhoz szükséges 320 szavas bufferek helyett 64 szavas

blokkokat használunk, hardcopy készitésekor azonban még ezt a memóriaterületet is megtakaritjuk azzal, hogy a rekordot közvetlenül a már lefoglalt rekordterületről visszük ki annak elengedése előtt. Jelentős memóriát takaritunk meg azáltal is, hogy az utónevek dekódolását a korábbiakban általánosan használt nagy memóriaigényü COBOL program helyett az utónév szótár átstrukturálásával mindössze 400 szóban oldottuk meg.

4. Kiemelt fontosságot kellett tulajdonitanunk az adatvédelemnek, az adatállományhoz történő illetéktelen hozzáférés megakadályozásának. Mivel azt interaktiv felhasználói programokkal a program azonositójának ismeretében bárki kapcsolatot létesithet, ezért a bejelentkezéskor a GCOS operációs rendszer userid-password ellenőrzéséhez hasonlóan kettős ellenőrzést végzünk. A megengedett azonositókat és jelszók t saját területünkön, megfelelően védett file-on tartjuk. Ugyanitt tároljuk azokat az engedély kódokat is, amelyek meghatározzák, hogy felhasználónk milyen tevékenységeket hajthat végre és az adatbázis rekordjai közül melyekhez férhet hozzá. Az azonositót és a jelszót a bejelentkezés után a képernyőről azonnal töröljük. Az illetéktelen hozzáférési kisérleteket elszámolási file-unkon feljegyezzük. Ugyanitt rögzitjük a jogos felhasználók tevékenységének fontosabb lépéseit is /munkamenet kezdete, vége, nem engedélyezett tevékenység végrehajtásának kisérlete, lemezek felrakása, levétele, stb./. Ezen tulmenően a lekérdező rendszer üzemeltetője, mint egy kiválasztott, több jelszóval védett privilégizált felhasználó, ellenőrizheti a lekérdező rendszer többi felhasználóját, információkat kérhet róluk, nyomon követheti tevékenységüket.

Abstract: The Interactive Database Query System has specially been designed for purposes

of State Population Registration Office. The main software problems to be solved were concurrent handling of several terminals, and an operator interface for mounting and dismounting database disk packs since their number basically exceeds the number of available disk units.

Sztrókay Kálmán
Állami Népességnyilvantartó Hivatal

Vass Éva

AZ R40 SZÁMÍTÓGÉP MUNKÁJÁT DOKUMENTÁLÓ PROGRAMCSOMAG

Számitógépek hatékony üzemeltetéséhez elengedhetetlen az operációs rendszer müködését leiró adatok előállitása és feldolgozása.

Cikkünk a KFKI R4Ø-es számitógépén üzemelő adatgyüjtést és az azt feldolgozó programcsomagot ismerteti.

Az OS operációs rendszer nyujtotta adatok SMF rekordok mellett további adatok gyüjtését (felhasználói SMF rekordok) is bevezettük.

A feldolgozó programcsomag táblázatokat és hisztogramokat készit, dokumentálja a lefuttatott jobokat, az operátori munkát, valamint statisztikai adatokat szolgáltat a rendszer hangolásához.

Kulcsszavak:
SMF rekord, programcsomag, hatékonyságelemzés, statisztikai táblák.

Minden számitóközpontban felmerül az igénye annak, hogy valamilyen módon képet tudjon adni munkájáról, mérni tudja az üzemeltetés hatékonyságát. A KFKI Számitóközpontjában programcsomag készült az üzemeltetési munka hatékonyságának mérésére.

1. Üzemeltetési környezet

A KFKI-ban 1977 óta egy ESZ 1040-es számitógép üzemel IBM OS/360-as operációs rendszerrel MVT üzemmódban.

1978-ban bevezetésre került a CEDRUS /Conversational Editor and Remote User's Support/ interaktiv szövegszerkesztő program [4], amely a gép átbocsátóképességét ugrásszerüen megnövelte. Ma már jobjaink több, mint fele /kb. 60-80%-a/ a CEDRUS rendszeren keresztül kerül futtatásra.

A Számitóközpontban futtatott munkák legnagyobb része tudományos-müszaki számitás, és csak egészen kis része adatfeldolgozás. A jobok nagy része nagy számitásigényü.

2. Az OS SMF lehetőségének általános ismertetése

A feladatunk az volt, hogy létrehozzunk egy olyan programcsomagot, amely a számitógép munkáját különböző szempontok szerint dokumentálja.

Ennek alapjául az operációs rendszer SMF /System Management Facilities/ funkciója által gyüjtött adatok szolgáltak.

Az SMF
- adatokat gyüjt a gépen futtatott jobokról
- kilépési lehetőségeket biztosit a job futásának különböző pontjain. Ezeken a pontokon a rendszerprogramozó saját ellenőrző vagy egyéb funkciókat végző rutinjait csatlakoztathatja.

Az SMF bővitési lehetőségeket nyujt, amelyeket az üzemeltetési környezetünk által felvetett problémák megoldása során ki is használtunk.

Ilyen bővitési lehetőség, hogy
a/ felhasználói rekordot irhatunk az SMF adathalmazba,
b/ kilépési rutinokat csatlakoztathatunk az SMF-hez.

a/ Az SMF adatállományba egy 0-ás védelmi kulcsu programból kiadott makro - az SMFWTM makro - segitségével irhatunk. Ilyen program megirása külön feladatot jelentett.

Felhasználói SMF rekord irására több ok miatt is szükségünk volt.

Egyrészt felmerült a szükségessége annak, hogy az operátori műszakok kezdetét és végét az SMF adatai segitségével meghatározhassuk, valamint annak, hogy az operátorok a müszak során előfordult különleges eseményekről információt jegyezhessenek be az SMF adatállományba.

Másrészt az SMF hiányossága, hogy START paranccsal indítható operátori procedurák programjairól nem jegyez SMF információkat. Ezek közé tartozik a CEDRUS is, amely egy, szinte az egész üzemidő alatt futó program, s igy nem hagyható figyelmen kivül egyetlen gépteljesitményt, számlát, stb. készitő értékelés során sem. A CEDRUS rendszerről gyüjtött információkat különböző felhasználói SMF rekordokban gyüjtjük.

Felhasználói SMF rekord iródik az SMF adathalmazba például

i/ terminálon keresztül a CEDRUS rendszerbe történő ki- és belépésről /LOGIN, LOGOUT/;

ii/ a CEDRUS-on keresztül történő - azaz a writer használatát megkerülő sornyomtató használatról.

b/ Kilépési rutint is épitettünk a rendszerbe. Ez az SMF IEFUJV kilépési pontjához csatlakozó JCL ellenőrző program. A program feladata, hogy az ütemezés megkönnyitése érdekében bevezetett üzemeltetési rend jobvezérlő kártyákra tett kikötéseit ellenőrizze, másrészt automatikusan prioritást rendeljen a jobokhoz.

Az üzemeltetési rend szigoru szabályok szerinti jobosztályokat, valamint egy ezeken belüli, a jobok erőforrásigényétől függő prioritási rendet jelent. Az automatikus prioritás hozzárendeléssel a célunk az volt, hogy a prioritás a job erőforrásigényét tükrözze.

A JCL ellenőrző bevezetésére azért került sor, mert egyrészt erőforrásaink korlátozottak, másrészt a CEDRUS rendszeren keresztül érkező nagy számu job miatt a terhelés dinamikusan változik, ami miatt a munkát előre ütemezni csak kevéssé lehetne.

3. A programcsomag

A bemutatandó programcsomag elemei a rendszer, valamint az általunk gyüjtött SMF információk feldolgozását végzik. Különféle szempontok szerint összesitőket, táblázatokat készitenek a rendszer és az üzemeltetés munkájáról.

3.1 Az SMF adatállomány mentését végző eljárás

A rendszer az SMF információkat egy speciális, erre a célra fenntartott rendszerfile-ba irja, amely nálunk mágneslemezen van.

Létrehoztunk egy olyan eljárást, amelynek segitségével minden nap az utolsó operátori müszak végén az SMF adatállomány tartalmát mágnesszalagra átmásoljuk, és az SMF adatállományt az átmásolás után kiüritjük.

A mentést végző procedura bevezetésével, valamint az SMF adatállomány naponta történő mentésével sikerült lecsökkenteni a lemez és szalag paritáshibák vagy rendszerhibák miatt elveszett SMF adatok számát.

Az eljárás
- az SMF adatok mentésén kivül
- mágnesszalaghiba statisztikát és
- SMF REPORT-ot készit.

3.2 SMF REPORT program

A feldolgozott adatok:

A REPORT program a rendszer által gyüjtött 1, 4, 5, 6, 7-es tipusu SMF rekordok információit dolgozza fel. Az üzemidőt üzemkezdettől napfordulóig, illetve napfordulótól üzemzárásig terjedő időintervallumokra bontja, és ezen időintervallumokból összesitő listát készit.

A program által nyujtott információk:

Az összesitő lista tulajdonképpen egy időrend szerint rendezett kép a rendszer előző napi munkájáról.

A jobok befejezésének sorrendjében a következő információkat szolgáltatja a lefutott jobokról:

- a job reader által történt beolvasásának ideje,
- a jobnak az inditó-befejező program általi kiválasztási ideje,
- memóriába történő betöltés ideje,
- a job futásának vége,
- writer-output elkészülési ideje,
- a job CPU ideje,
- a job futási ideje /a jobfutás vége és az inditó-befejező program által történt kiválasztási idő különbsége/,
- a job gépben töltött ideje /a writer-output elkészültének, illetve ennek nem léte esetén a jobfutás vége és a reader általi beolvasási idő különbsége/,
- a job által kért memória nagysága,
- a job input osztálya,
- a jobfutás completion code-ja és abend indikátorának értéke.

A program a müszak során előfordult IPL-eket, valamint az

SMF adatállomány betelése miatt nem regisztrált rekordok számát is feltünteti.

Az egyes időintervallumokról készült lista végén a következő összesitő adatokat szolgáltatja az ezen idő alatti müszakról

- müszak kezdetének és végének dátuma /év, nap, óra/
- a müszak alatt lefutott jobok száma,
- CEDRUS rendszeren keresztül futtatott jobok száma,
- a müszak alatt a CPU idő,
- a müszak során az üzemidőből az IPL-ek miatt kiesett idő.

A REPORT program végén lista készül azokról a jobokról. amelyeknek csak töredékei készültek el /van lefutott joblépése, de nem fejeződött be a futása, vagy befejeződött a futása, de nem készült el a writer-outputja/.

A program célja, feladata:

- A diszpécser munkájának megkönnyitése;
- A felhasználók informálása a job-ok futási adatairól, és ezáltal a felhasználó
  - optimálisabbá teheti a job futási feltételeit /pl. a kért és használt memória különbség és az ezzel összefüggő input osztály preciz megadásával/, valamint
  - ezzel javithatja az MVT rendszer áteresztőképességét is.

3.3 Müszaknapló

A feldolgozott adatok

A rendszer által gyüjtött 1, 4, 5, 6-os tipusu SMF rekordok és egy általunk irt felhasználói rekord információit dol-

gozza fel. A program a napot operátori müszakokra bontva készit összesitő értékelést.

A felhasználói rekord beirása az SMF file-ba

Elkészitettünk egy eljárást, amely a paramétereként kapott szöveget SMF rekordba helyezi, és az SMF adathalmazba irja.

Használatával lehetővé válik

- az egyes operátori müszakok kezdetének és végének meghatározása,
- a müszak során előforduló különleges hardware vagy operációs rendszerbeli eseményekről információ jegyezhető fel az SMF adatállományba.

A program által közölt összesitők

A program az egyes operátori müszakok munkájáról a következő adatokat szolgáltatja:

- a müszak alatt - óránkénti bontásban - a futtatott jobok száma /a job-számmal mérve tehát a gép áteresztőképességét/,
- az input job-osztályok szerinti bontásban az adott jobosztályból hány darab job tartózkodott a gépben 0-1, 1-2, 2-3 stb. órát, és az egyes job-osztályok jobjainak milyen az átlagos fordulási ideje.
  Ugyanez a táblázat az egyes job-osztályok jobjainak a számát 100%-nak véve, %-os bontásban is elkészül.
- az input job-osztályok szerinti bontásban táblázat készül a jobokról az általuk lekötött memória nagysága alapján,
- ugyancsak input job-osztályonként összesitést készit a jobok CPU idejéről,
- a tul nagy erőforrásigényü jobokat tételesen felsorolja jellemző adataikkal együtt /beolvasási idő, inditó befejező program által történt kiválasztási idő, input jobosztály, CPU idő, kért memória/,

- feldolgozza a felhasználói rekordok adatait, amelyekből megállapitható, kik dolgoztak az adott operátor csoportban, milyen tényezők gátolták a csoport hatékony munkáját, stb.,
- összesitő adatokat szolgáltat
  - a müszak során a lefutott jobokra irható CPU időről
  - a müszak üzemidejéből az IPL-ek miatt kiesett időről.
- mikor és hány darab IPL volt a müszak során,
- listát közöl azokról a jobokról, jellemző adataikkal együtt, amelyek operátori hiba miatt estek ki, illetve amelyeket az operátor törölt a rendszerből.

A program szerepe az operátori munka hatékonyságának mérésében

Igy lehetővé vált, hogy a program-összesitők alapján képet kaphassunk az egyes operátor csoportok munkájáról.

Ez fontos lehet a vezetők számára az operátori munka irányitásában és értékelésében, az operátorok pedig az összesitők nyujtotta információk segitségével javithatnak teljesitményükön.

3.4 Számlaprogram

A feldolgozott adatok

A rendszer által szolgáltatott, valamint az általunk irt felhasználói SMF rekordok adatai. Havonta készülő program.

A program feladata

A program hasonló jellegü erőforrás használatot számláz, mint más számitóközpontok számlaprogramjai.

Amiben különbözhet:

Nálunk néhány speciális szempont is közrejátszott az egységnyi erőforrás használatért kért összegek meghatározásában. Például:

- a job-prioritás nem jelenik meg a jobárban, hiszen a job-prioritást az üzemeltetési rend szabályai határozzák meg.

Speciális problémá volt a CEDRUS rendszer használatának figyelembevétele a számlában, mivel a rendszer a CEDRUS-ról eredetileg nem gyüjtött SMF információkat. Ezt a hiányosságot felhasználói SMF rekordok segitségével küszöböltük ki. Másrészt számitásokat végeztünk arra vonatkozóan, hogy a CEDRUS mekkora erőforráshasználat növekedést jelent a CEDRUS nélküli rendszerhez viszonyitva. Ez az előzetes statisztikák adatai alapján mintegy 20%-os növekedést jelent a jobárban kifejezve.

Számlaformulánkba ezeket az adatokat direkt módon beépitettük.

3.5 Gépidő statisztika

A feldolgozott adatok

A gépidő statisztikai program két programból áll. Az első program az SMF adatállomány rekordjaiból kiválogatja azokat, amelyek a paraméterlistán megadott időintervallumban kerültek feljegyzásre. Az időintervallumot a kezdő és záró nap dátumának megadásával jelöljük ki.

Ezen rekordok közül a 0, 4, 5, 6, 14 és 15-ös tipusu SMF rekordok adatait dolgozza fel a statisztikai program.

A program által szolgáltatott táblázatok és összesitők

Az egyes táblázatok a programtól opcionálisan, program paraméterek segitségével kérhetők. A program a következő információkat szolgáltatja az adott időintervallumra vonatkozóan:

Az időszak alatt futtatott összes jobok száma, ezekből a CEDRUS-on keresztül leadott jobok száma.

Az időszak alatt naponta futtatott jobok száma.

A teljes időszakra vonatkoztatva 0 és 24 óra között, óránkénti bontásban, a futtatott, illetve a leadott (beolvasott) jobok átlagos száma.

Hisztogramot készit a
- job prioritásokról
- job befejezési code-okról
- abend indikatorokról.

/Ezek x koordinátái például jobprioritás esetén a lehetséges prioritásértékeket vehetik fel, a hozzájuk tartozó y koordináta értéke pedig az adott prioritással rendelkező jobok darabszáma./

Adatokat szolgáltat arról, hogy egy felhasználónak hány jobja futott az adott időszak alatt /a felhasználókat jobnevük kötött első négy betüjével azonositja/.

Adott programot hányszor hivtak /hány //EXEC PGM=programnév kártyán szerepelt a program neve/.

Hány allokációs kérés érkezett az egyes háttértárakra, perifériákra.

Hány csatornaátvitel történt az egyes perifériákra.

Adott memóriát hány job kért, illetve használt /itt adott memória alatt egy memória-intervallumot értünk pl. 100-110K közötT stb./.

Hisztogram készül a
- SYSOUT osztályokról
- input jobosztályokról

azaz, hogy hány job futott az adott osztályokban.

A jobok számáról CPU idejük perces bontásában.

Az egyes step-prioritásokhoz tartozó jobok számáról.

Az időintervallum egyes napjaiban az IPL-ek számáról.

Adatállományok használatáról azok inputra, illetve outputra történő megnyitása alapján.

A program célja

Havi összesitő képet ad a rendszer munkájáról, amely segitséget jelenthet mind a hardver-es, mind a szoftver-es szakemberek számára, hogy a gép hatékonyságát növelhessék.

Például:
- hatékonyabb üzemeltetési rend kialakitásában /jobosztályok definiálásában/,
- REGION, TIME default paramétereinek meghatározásában.

De segitséget jelent olyan jellegü munkákban is, amelyek során a modulforgalom vizsgálatával és ésszerü szervezésével próbáljuk a rendszer hatékonyságát növelni.

Abstract

To the efficient management of computers it is essential to produce and process data describing the work of the operating system.

Such a program package for collecting and processing data is described. The package is running on the ES 1040 computer of the Central Research Institute for Physics, Budapest.

In addition to the SMF /System Management Facilities/ data provided by the OS operating system user-written SMF records are collected and processed, too.

The programs print statistical tables and draw up charts and histograms on jobs and programs run on the computer. The results are used in better organization of the operator's activities and increasing overall system performance.

Irodalomjegyzék

1 OS SMF IBM Systems Reference Library GC28-6712-7

2 OS Data Management Services Guide GC26-3746-1

3 IBM System/360 Operating System: System Control Blocks

4 A.Arató - J.Sarkadi-Nagy - F.Telbisz: A Local Network for the Support of Software Development. COMNET'77 1977. Vol.1. p.227 /1977/.

5 Arató A. - Sarkadi-Nagy I. - Telbisz F.: Feladatmegosztás ESZR gép és front-end processzor között a CEDRUS terminálhálózatban. Programozási Rendszerek '78, Szeged, Vol.1. 14.old.1978.

Zsombok Zoltán

EGY EGYSZERŰ ÉS HAJLÉKONY ADATKEZELŐ RENDSZER

Az előadás egy bináris keresési fán alapuló, szekvenciális és közvetlen elérést lehetővé tevő file-kezelő rendszer működését mutatja be. Összehasonlítja más, pl. a hashing technikát és az indexelt szekvenciális módszert alkalmazó rendszerekkel. Bemutat egy - Pascal nyelvü - algoritmust eléggé jól kiegyensúlyozott, blokkokba rendezett fa előállítására. A javasolt technika legfőbb erénye a rugalmasság: az egyszerü alkalmazásokat, azaz az elérési mutatókra nem kényes file-okat egyszerüen, a nagyméretüeket a hozzáférési paraméterek szerint jól alakíthatóan valósíthatjuk meg.

Kulcsszavak: adatszerkezetek, fa, bináris keresés, Pascal.

Köztudott, hogy a leggazdaságosabb adatkezelési technika kiválasztása egy-egy probléma megoldására korántsem egyszerü feladat. A nehézség abban van, hogy nem ismerjük - vagy nem vesszük a fáradságot, hogy megismerjük - az alkalmazandó eszköz müködési paramétereit, ill. azok meghatározási módját. Ugyanezt mondhatjuk a feladat várható adatairól is. Az eszközök pedig - ráadásul - érzékenyen reagálnak a tervezési paraméterek megválasztására. Minél bonyolultabbak, annál inkább. Igy nem ritka eset, hogy korszerü eszköz használata ellenére rossz eredményt érünk el.

Tervezési paraméterek pedig már egy indexelt szekvenciális file-nál is szép számmal találhatók. Már a file első feltöltésekor meg kell adnunk az index táblák számát, méretét, az adatblokkok nagyságát, az elsődleges és másodlagos túlcsordulási területeket, az első feltöltésnél alkalmazandó kitöltöttségi tényezőt stb.

Mindez nem kis terhet ró a tervezőre, aki nehezen tud va-

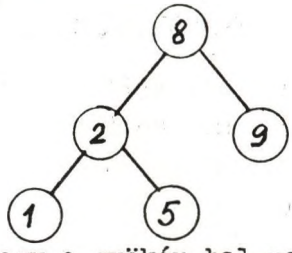
lamennyi követelményének eleget tenni, s számos vonatkozásban - pl. biztonsági okokból - kényszerül pazarlásra. És ami talán a legfontosabb, nehezen tudja file-szerkezetét folyamatosan hozzáalakítani a file élete során egyre pontosabban látszódó elvárásokhoz.

A következőkben egy, az irodalomból jól ismert technika és néhány ötlet segítségével olyan file-szerkezetet mutatunk be, amely közvetlen /kulcs szerinti/ és szekvenciális elérést biztosít változó hosszúságú rekordokkal, s rugalmasan alkalmazható különféle méretü és elérési paraméterü feladatokhoz.

1. Bináris keresési fa mint file-szerkezet

Bináris keresési fának /binary search tree/ az irodalom az alábbi ábra szerinti bináris fát nevezi.

A fa minden csúcspontja egy-egy rekord, amely az adatokon kívül kulcsot, bal- és jobb-pointert tartalmaz a rekord megtalálására és azonosítására.



A fára és minden részfájára igaz, hogy a gyökér bal-pointere a /rész/fában lévő valamennyi nála kisebb kulcsú rekordból álló részfa gyökerére mutat, ha ilyen rekordok egyáltalán vannak, különben a "nil" /üres/ értéket veszi fel. Hasonló állítás igaz a jobb-pointerre is, csak "kisebb" helyett "nagyobb" értendő.

Knuth enciklopédia-számba menő könyvében [1] természetesen foglalkozik e szerkezettel is /3. kötet, Search and Sorting/ megemlítve, hogy a technika inkább az operatív memórián belüli adatszerkezetek képzésére valók, a külső tárolókon gyorsabbak a hashing-en alapuló módszerek.

Az állítás második felével mi sem szállunk vitába, hiszen a

tárolási cím meghatározására kétségtelenül a hashing nyújtja a - lényegesen - gyorsabb módszert. Nekünk azonban más szempontjaink is vannak a blokk-kiválasztás gyorsasága mellett:

- Az egyik az, hogy a hashing lényegét alkotó ú.n. randomizáló függvény erősen feladatfüggő.
- Másodszor, külön meg kell oldani a blokkon belüli keresést, ami túlzsúfolt blokkok esetén gyakran a következő blokk/ok/ átnézésével is együtt jár.
- Harmadszor, az ilyen szerkezetü file-t nem lehet szekvenciálisan kezelni.

Mindezek miatt egyáltalán nem kell csodálkozni azon, hogy a hashing nem terjed eléggé, a programozók idegenkednek tőle, szabványosítottsága kismértékü. Mi ezért inkább a - gyakorlatban széles körben elterjedt - indexelt szekvenciális file-kezelést tekintjük mércének.

2. Elérhetőségi mutatók

Ha a kulcsa szerint rendezett file n rekordot tartalmaz, akkor egy megadott kulcsú rekord megtalálásához legalább $\log_2 n$ igen/nem választ kell kapnunk. Egy rekord kulcsának összevetése egy értékkel három kimenetelü, habár az "egyenlő" esetnek jóval kisebb a valószínüsége a többinél. Ezért $\log_2 n$ -nél valamivel kevesebb rekordösszehasonlítással tudunk egy rekordot megtalálni optimális esetben.

Ha bináris keresési fánk teljesen szabályos /kiegyensúlyozott/, akkor elég jól meg tudjuk közelíteni az optimumot. A bináris keresési fa szabályossá tétele azonban nem egyszerü és nem is olcsó feladat. Egy rekordot ugyanis a fára /annak átrendezése nélkül/ csak egy meghatározott helyre tehetünk, így a fa alakja kizárólag a rekordok sorrendjétől függ. Kimutatható, hogy rendezetlenül /egymástól függetlenül/ érkező rekordokból előállt véletlen fa esetén kb. $2 \ln n$ összehasonlítás szükséges átlagosan egy-egy rekord kereséséhez, ami az előbbi optimumnak csak kb. 1.4-szerese!

/lásd [2] alatt./ Ez a 40% többletidő még mindig olcsóbb megoldás, mint a fa folyamatos kiegyensúlyozása, ezért mi csak szükség esetén egyensúlyozzuk ki a fát, alapvetően a véletlen módszert alkalmazzuk.

Figyeljük meg, hogy a bináris keresési fa szekvenciálisan is olvasható, mégpedig rekordonként átlagosan két rekordérintés mellett. Ez nem rossz érték.

3. Blokkok

A bináris keresési fán alapuló file-szerkezetek háttértárolón való alkalmazása ellen szól az alábbi két érv:

- Igaz, hogy a véletlen fa elérési ideje átlagosan csak 40%-kal rosszabb a kiegyensúlyozott fáénál, de ez nem gátolja meg, hogy kis valószínüséggel nagyon rossz elérésü file keletkezzék. Pl. ha egy milliós file rekordjai teljesen rendezetten érkeznek, akkor a "véletlen" file lineáris lista lesz, $1.4 \log_2 n = 28$ átlagos elérési hossz helyett $n/2 = 500$ ezerrel. Ez pedig nagy baj.
- Ugyancsak szélsőségesen rossz hatásfokot okozhat egy rosszul sikerült blokkszerkezet. 100 rekordot feltételezve blokkonként, 3 blokkérintéssel el lehetne érni minden rekordot, ha optimálisan építjük fel a file-t. Szerencsétlen esetben azonban elképzelhető, hogy még kiegyensúlyozott fa esetén is 20 blokkérintés szükséges, ha t.i. minden rekord más blokkba került a keresett rekordhoz vezető úton.

Az irodalomban számos példát találhatunk e problémák megoldására, s eközben nemritkán találkozunk bináris keresési fákkal is.

Mindenképpen figyelmet érdemelnek a Bayer-féle B-fák [2], amelyek átmenetet jelentenek a bináris keresési fák és az indexelt szekvenciális szerkezet között: egy-egy blokkban legalább n, de legfeljebb 2n rekord van, s a rekordok in-

dexszerüen mutatnak az alacsonyabb szintü blokkokra. Egyszerü algoritmusokkal lehet biztosítani az ilyen, n-edrendü fa kiegyensúlyozottságát, de azon az áron, hogy a blokkok átlagosan csak 75%-ban vannak kihasználva, változó rekordhossznál még annyira sem.

Némely indexelt szekvenciális file-kezelő rendszer az indexkeresést binárisan végzi, sőt, változó indextábla esetén bináris keresési fával [3].

Érdekes, hogy nem szokás bináris keresési fát használni a blokk megtalálására és a blokkon belüli keresésre egyaránt, ahogyan mi tervezzük. Meg kell ezért mondanunk, hogy miképpen oldjuk meg a fenti két problémát.

A fa költséges, folyamatos kiegyensúlyozása helyett szükség esetén /pl. a kezdeti feltöltéskor vagy újraszerkesztés esetén/ egy, a következő fejezetben bemutatandó algoritmust adunk a felhasználó kezébe, amellyel a fát a hagyományos értelemben is és a blokkszerkezetet tekintve is kiegyensúlyozottra építheti fel.

Másrészt, eszközt adunk a kezébe, hogy menetközben észrevehesse a fa degenerálódásának a veszélyét. Emellett alapértelmezésként olyan blokk-kiválasztási algoritmust adunk, amely véletlen fa esetén igyekszik a blokkmozgatások számát alacsony szinten tartani. Ennek alapelve a következő:

Tegyük az első blokkba az első rekordot. Ezután helyezzünk minden rekordot lehetőleg ugyanabba a blokkba, amelyben az őse van. Ha erre már nincs hely, kezdjünk új blokkot mindaddig, amíg lehet. Ha nincs már új blokk, a felhasználónak kell útmutatást adnia a blokk kiválasztásához.

Figyeljük meg, hogy szerencsés esetben, t.i. ha a blokkok nagyjából egyszerre telnek be a fa ugyanazon szintjén, k blokk érintése szükséges az n^k méretü file egy-egy rekordjának a kiválasztásához, ha egy blokkban n rekord van.

Észre is vehetjük az analógiát egy k-1 szintü indextáblával.

Más blokk-szelektáló algoritmust a programozó az előbbi újradeklarásásával írhat elő, s eközben használhat néhány, a file pillanatnyi állapotára jellemző változót, pl. az ősrekord helyének adatait, a blokkok telítettségét, a pillanatnyi keresési úthosszat stb. Ezek konkrét felhasználási módját nem részletezzük.

4. Algoritmusok

Bináris keresési fák kezelésére az irodalom bőven szolgáltat algoritmusokat. /Lásd pl. [1], [2] és [4] alatt./ A legegyszerübbek, a véletlen fákat kezelő algoritmusok alkotják rendszerünk alapjait. A pointerek kialakításánál gondolni kell a blokkszám és a blokkon belüli cím megadhatóságára. Az elemi utasítások helyett három problémakör megoldására térünk ki.

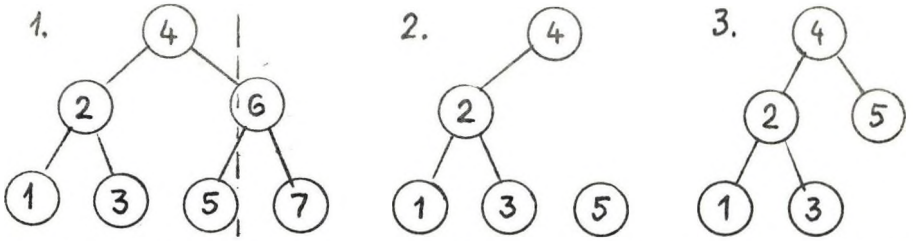
Az egyik a pufferezés kérdése. Mivel a fa gyökerének blokkja szinte mindig nélkülözhetetlen, a levelek felé haladva a blokkok felhasználási gyakorisága egyre csökken, olyan többpufferes technika kívánatos, amely valamilyen LFU vagy LRU algoritmussal cserélgeti a puffereket. Mi az utóbbi egy változatát használjuk. / [4]/

A másik probléma a blokkon belüli helygazdálkodás. Erre a célra egyfajta Garbage-collectort használunk. T.i. számíthatunk arra, hogy a blokkon belüli rekordok többségére nem mutat külső pointer. Ezeket szükség esetén, összefüggő hely kialakítása céljából odébb tolhatjuk. A mozgatható rekordokat egy bittel megjelöljük. A tologatásra sajnos szükség van, mivel a változó rekordhossz miatt sok töredék hely marad.

Harmadikként foglalkozunk a kiegyensúlyozás időszakos elvégzésével. Bemutatunk egy blokkonként majdnem kiegyensúlyozott

fát előállító eredeti algoritmust, amely egy szekvenciális rekordforrást feltételez. E célból előbb bevezetünk néhány fogalmat:

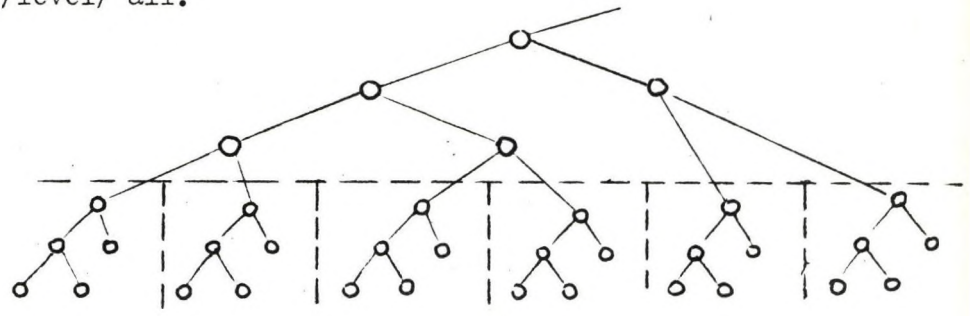
Majdnem kiegyensúlyozott a fa, ha vagy üres, vagy bal részfája teljesen szabályos és jobb részfája a balnál nem több elemből álló majdnem kiegyensúlyozott fa. Ilyen fát szemléletesen úgy nyerhetünk, hogy egy teljesen szabályos fából levágjuk az egy megadott értéknél nagyobb kulcsú rekordokat, s a csonkán maradt részeket új ágakkal összekötjük. Így:



Blokkonként majdnem kiegyensúlyozott a fa, ha minden blokkjában egyetlen majdnem kiegyensúlyozott részfát találunk.

Figyeljük meg, hogy a majdnem kiegyensúlyozott fa átlagos keresési ideje alig több a kiegyensúlyozott fa idejénél.

Algoritmusunk balról-jobbra, ill. -felfelé növeszti a fát, amelyet két függvény-eljárás rekurzív hívásával ér el. Megértésükhöz tekintsük az itt következő ábra szerinti szerkezetet, amelyen megfigyelhetjük a blokkok fa-struktúráját is. A legalsó szintü blokkokat 1. rétegnek /layer/ nevezzük, a felettük levőket 2. rétegnek stb. A blokkok belsejében található egy-egy részfa hasonlóan számozva szintekből /level/ áll.



A load(i) hívás felépít egy i rétegből álló struktúrát. A hívás értéke a gyökérre mutató pointer lesz. A build hívás i,j,q paraméterekkel egy, az i-edik rétegben lévő, maximálisan j szintü részfát készít, amely legbaloldalibb végéről a q pointer által mutatott részfához csatlakozik. Többi levele egy-egy i-1 rétegből álló részfát tart, amelyet az eljárás load hívásokkal állít elő.

Pl. a build(1,10,nil) hívás egy maximálisan 0 szintü fát épít fel a legalsó réteg egy új blokkjában.

Az algoritmus Pascal nyelvü leírását s környezetének specifikációját a függelékben mutatjuk be. A környezet a következőkből áll:

- "maxconst" valamely nagy állandó érték, mely nagyobb az egy blokkon belül lehetséges legnagyobb szintszámnál;
- "key" és "data" a rekordok kulcs és adatmezei, melyek formáját most nem részletezzük;
- "exhausted" jelzi a szekvenciális rekordforrás /pl. egy másik fa olvasása vagy egy szekvenciális file/ kimerülését;
- "newpage" egy új, üres blokkot előállító függvényeljárás;
- "nextrec" függvényeljárás, mely egyrészt egy következő rekordot ad, másrészt be is illeszti a specifikált blokkba. A rétegszámot sem árt tudnia, mert elképzelhető, hogy pl. a legalsó réteget nem akarjuk teljesen megtölteni. A "currec" globális változó mutat az előállított rekordra. Ha a specifikált blokkban már nincs hely, egy következő hívás ismét ugyanazt a rekordot fogja szolgáltatni.

5. A rugalmasság illusztrációja

Végezetül elemezzük rendszerünket egy nem ritka példán. Tekintsünk egy olyan file-t, amelynek élete során előre nem látható pontokon a kulcssürüség ugrásszerüen megnő.

Az indexelt szekvenciális file feltöltésekor nem áll módunkban erre a jelenségre felkészülni, ezért a nagy kulcssürüsé-

gü adatok zöme a túlcsordulás területekre kerül, ahol már csak a lassabb, esetleg csak a soros keresési módszerek alkalmazhatók.

Mi pedig a file teljes területén bináris keresést tudunk megvalósítani, s ha kell, a file módosításakor ráaggatott részfákat kiegyensúlyozottá is tehetjük. Ezzel váltjuk be a bevezetőben tett igéretünket: alkalmazkodni tudunk a szélsőséges követelményekhez is.

A vázolt file-kezelő rendszert jelenleg a Honeywell 66-os gépen, a Pascal nyelvhez illeszkedően implementáljuk. Az elmult évben megvizsgáltuk a szóbajöhető algoritmusokat, s a gyakorlati jelentőségüek egy részét modellszerüen megírtuk Pascalban. Ezúton köszönöm meg munkatársam, Szádeczky K. Gedeon tevékenységét vizsgálódásaiért és algoritmusaiért. Sajnos, ezen előadás keretei nem a legalkalmasabbak további eredmények ismertetésére.

Függelék: Egy blokkonként majdnem kiegyensúlyozott bináris keresési fát készítő algoritmus

```
const maxlevel=...;
type  ref = ↑rec;
      rec = record left, right: ref;
                   key: ...; data: ... end;
var   exhausted: Boolean;
      currec:    ref;

function newpage: integer; extern;
{   newpage:= <page number of an empty page>   }

function nextrec (pagenumber,layer: integer) : Boolean;
         extern;
{   1. It presents a record of type "rec" with higher key
       than the previous one, but: if the previous call
       returned with "false"  then the same record will be
```

presented again. The record can be referenced by currec↑.

2. exhausted:= <record is not presented (=eof)>.
3. If not exhausted and <the page number "pagenumber" on the layer "layer" has enough free space for the record> then
   - the record will be inserted;
   - nextrec:= true

   else nextrec:= false. }

```
function load(layer: integer): ref;
var i, pageno: integer; p: ref;

function build(layer, level: integer; q: ref) : ref;
var j: integer;
begin if level=0 then build:= nil else begin j:=1; q:=nil;
        while (j<=level) and nextrec(pageno,layer) do begin
           currec↑.left:= q; q:= currec;
           q↑.right:= build(layer,j-1,load(layer-1));
           j:= j+1 end ;
        build:= q end
end ;
begin if layer=0 then load:= nil else begin i:=1; p:=nil;
        while (i<=layer) and not exhausted do begin
           pageno:= newpage;
           p:= build(i,maxlevel,p) end ;
        load:= p end
end ;
```

Abstract: This paper presents a data management system based on the well-known binary search trees. Variable length records can be accessed randomly and sequentially. An algorithm is given in Pascal to build "paged, almost balanced trees" from sequentially ordered input.
The main advantage of the system is its flexibility: Small files can be used in a simple way, but with some additional work a good performance can be achieved for the big ones.

Irodalom:

[1] Knuth, D. E.: The Art of Computer Programming, Addison-Wesley, 1973.

[2] Wirth, N.: Algorithms + Data Structures = Programs, Prentice-Hall, 1976.

[3] Martin, J.: Computer Data-Base Organization, Prentice-Hall, 1977.

[4] Mérey, A.: Adatszerkezetek, KSH SZÁMOK, 1979.

Zsombok Zoltán, Államigazgatási Számítógépes Szolgálat
1536 Budapest, Pf 232.