

EGYETEMI DOKTORI ÉRTEKEZÉS

PROGRAMOK PÁRHUZAMOS VÉGREHAJTÁSÁNAK FORMÁLIS
LEÍRÁSA VDL SEGÍTSÉGÉVEL



FÜVESI ISTVÁN

JÓZSEF ATTILA TUDOMÁNYEGYETEM
KIBERNETIKAI LABORATÓRIUM

SZEGED, 1981.

TARTALOMJEGYZÉK

	oldal
BEVEZETÉS	1
1. PÁRHUZAMOS FOLYAMATOKBÓL ÁLLÓ RENDSZEREK . . .	3
1.1. Párhuzamos folyamatok	3
1.2. A vezérlés problémái	5
1.3. Párhuzamos utasítások	6
2. SZINKRONIZÁCIÓS ESZKÖZÖK	8
2.1. A szinkronizációs primitívek fejlődése	8
2.2. Párhuzamos folyamatok együttműködésének szervezése . . .	11
2.2.1. Memory Interlock	11
2.2.2. Test and Set	13
2.2.3. Lock és Unlock	14
2.2.4. Block és Wakeup	15
2.2.5. Szemaforok	15
2.2.6. Szemafor műveletek dekompozíciója (Up és Down) . . .	19
2.2.7. Critical region	21
2.2.8. Conditional critical region	22
2.2.9. Monitorok	24
2.2.10. Path kifejezések	27
3. PÁRHUZAMOS FOLYAMATOK GRÁF MODELLJE	31
3.1. Petri hálók	31
3.2. Petri hálók alkalmazása programokra	34
4. MAGAS SZINTŰ SZINKRONIZÁCIÓS ESZKÖZÖK MEGVALÓ- SÍTÁSA SZEMAFOROKKAL	37
4.1. A conditional critical region és a szemafor viszonya . . .	37
4.2. A monitor implementálása szemaforokkal	40
4.3. Path kifejezések implementálása szemaforokkal	42
5. FORMÁLIS LEÍRÁS	45
5.1. Az EPPL konkrét szintaxisa	47



	oldal
5.2. Konkrét programok fordítása absztrakt programmá	52
5.2.1. Konkrét programok absztrakt reprezentációja	53
5.2.2. Az EPPL absztrakt szintaxisa	54
5.2.3. A translátor	54
5.3. Az absztrakt interpreter	57
5.4. Az EPPL interpreter	61
6. AZ MPPL ÉS FORDITHATÓSÁGA EPPL-RE	71
6.1. A conditional critical region fordithatósága	75
6.2. A monitor fordithatósága	78
6.3. Path kifejezésekkel szinkronizált folyamatok fordithatósága .	80
IRODALOMJEGYZÉK	82
MELLÉKLET	85

BEVEZETÉS

A VDL (bécsi definíciós nyelv) egy formális programozási nyelv, amelyet az IBM Bécsi Laboratóriumában fejlesztettek ki a PL/I nyelv leírására. Azóta számos új területen alkalmazták a VDL-et, mint definíciós, illetve tervezési eszközt.

A VDL-lel foglalkozó tanulmányok közül [28] a VDL leírását, [17] , [19] és [15] programozási nyelvek formális definícióját tartalmazza. A [20] és [21] egy operációs rendszer formális definícióját adja, míg [16]-ban Lee a VDL segítségével definiál adatszerkezeteket, rendezési algoritmusokat, leír egy számítógép processzort és egy programozási nyelvet. A VDL érdekes és hasznos alkalmazását találjuk [26] és [22]-ben, ahol a szerzők egy linkage editor illetve egy loader modell megalkotásával a nagymértékben géptől függő software termékek általános jellemzőit definiálták.

Ezen dolgozat célja a VDL egy új alkalmazása volt. A párhuzamos folyamatokból álló rendszerek előtérbe kerülése adta az ötletet ahhoz, hogy ezeket a rendszereket válasszuk a leírás tárgyául.

A számítógépek sebességének nem csupán hardware útján történő növelésére irányuló erőfeszítés vezetett a párhuzamos rendszerek vizsgálatához. A programok elemi egységének az utasítás helyett a szegmenseket választva - amelyek mindegyike egymástól független vagy együttműködő soros program, egy-egy processzor saját programja - jutunk a párhuzamos program fogalmához.

Tekintsük az egyidejűleg működő, előrehaladásuk feltételeiben egymástól kölcsönösen függő folyamatokból (szegmensekből) álló rendszereket. Az egyes folyamatok számára olyan algoritmusokat kell készíteni, melyek felfüggesztik azokat a folyamatokat, amelyek továbbhaladási feltételei már nem teljesülnek, majd megfelelő időben (a feltételek teljesülésekor) újra aktivizálják azokat.

Az algoritmusok elkészítését különböző szinkronizációs eszközök használata teszi lehetővé.

A dolgozatban a párhuzamos folyamatokkal kapcsolatos fogalmak definiálása után a 2. fejezetben a szinkronizációs eszközök áttekintését adjuk, példákön szemléltetve alkalmazásuk lehetőségeit és korlátait. Megalapozzuk ezzel egy olyan szinkronizációs primitív (eszköz) kiválasztását, mely programozási nyelvek párhuzamos folyamatokból álló rendszerek leírására szolgáló kiterjesztésének alkalmas és célszerű eszközéül szolgálhat. A viszonylag könnyű implementálhatóság mellett a kiválasztandó primitívvel szemben azt a követelményt támasztjuk, hogy segítségével a nála magasabb szintű szinkronizációs eszközök is megvalósíthatóak legyenek.

Ilyen eszköznek a szemafor bizonyul, s utóbbi követelmény teljesülését a 4. fejezetben mutatjuk meg.

A szinkronizáció relációinak képi áttekintését adó Petri hálókka a 3. fejezet foglalkozik.

Egy párhuzamos programozási nyelv (EPPL) definiálását, s annak formális leírását az 5. fejezet tartalmazza. A szintaxis mellett a szemantika leírására is a VDL-et kívánjuk használni. Megvizsgáljuk milyen korlátai vannak ennek, s a VDL eszköztárát úgy bővítjük, hogy az alkalmassá váljon párhuzamos folyamatokból álló rendszerek definiálására is. Párhuzamos folyamatok szemantikájának leírását tehát értelmezős végrehajtásuk definiálásával adjuk meg. A definiált programozási nyelv egyszerű, bővíthető. Csak annak eszközéül szolgál (jelenlegi kiépítésében), hogy megmutassuk: a VDL alkalmassá tehető párhuzamos folyamatok formális leírására is.

A dolgozat végén definiáljuk az MPPL nyelvet, mint egy olyan párhuzamos programozási nyelvet, mely az EPPL-nél kényelmesebb eszköz programok írására. Vázoljuk EPPL-re történő fordíthatóságát. Csak annyit mutatunk meg, hogy az EPPL elég általános ahhoz, hogy az MPPL nyelvű programokat arra lefordítsuk.

Ezen a helyen szeretnék köszönetet mondani Dr. Varga László egyetemi tanárnak a problémák felvetéséért és a hasznos konzultációkért, valamint Dr. Makay Árpád egyetemi docensnek azokért az észrevételeiért, amelyeket a kézirat elolvasása után tett.



1. PÁRHUZAMOS FOLYAMATOKBÓL ÁLLÓ RENDSZEREK

A következőkben a párhuzamos folyamatok, s a velük kapcsolatos fogalmak definícióját adjuk meg.

1.1. Párhuzamos folyamatok

Egy szekvenciális processzor által végrehajtás alatt levő programot szekvenciális folyamatnak nevezzük. (Logikailag minden folyamat saját processzorral és programmal bír, a valóságban két különböző folyamat megoszthatja ugyanazt a programot vagy ugyanazt a processzort.) Így egy folyamatot a végrehajtásban levő $\langle \text{processzor, program} \rangle$ páros határozza meg.

A folyamatok működésük során a számítógép erőforrásait használják.

Az erőforrás a rendszer valamely alkotórésze (periféria, memóriaterület, stb.), amelyből csak véges mennyiségű áll rendelkezésre és több folyamat akarhatja azt egyidejűleg használni.

Egy erőforrás és az őt használó folyamat viszonya a következő lehet:

- a folyamat igényli az erőforrást (használni kívánja),
- a folyamat használja az erőforrást (lekötve tartja),
- a folyamat nem használja az erőforrást (felszabadította).

A folyamatok bizonyos erőforrások állapotát megváltoztathatják. Így a rendszerhez tartozó erőforrások állapotainak a halmaza és az erőforrásoknak a rendszerbeli folyamatokhoz való adott viszonya a rendszer egy állapotát határozza meg.

Megadjuk most a szekvenciális folyamatokból álló rendszerek formális definícióját.

D e f i n i c i ó : A szekvenciális folyamatokból álló rendszer egy (Σ, Π) kettős, ahol $\Sigma = \{s_0, s_1, \dots, s_i, \dots, s_k, \dots, s_l\}$ a rendszer állapotainak egy halmaza, $\Pi = \{p_1, p_2, \dots, p_i, \dots, p_n\}$ pedig a rendszert alkotó szekvenciális folyamatok egy halmaza.

D e f i n i c i ó : Egy p_i folyamat egy parciális függvény, mely a rendszer állapotait a rendszer állapotainak egy nem üres részhalmazába képezi

le:

$$p_i : \Sigma \rightarrow \{Z\}$$

Ha p_i definiálva van az S_i állapot esetén, akkor a p_i értékészletét $p_i(S_i)$ jelöli. Ha $S_k \in p_i(S_i)$, akkor azt mondjuk, hogy a p_i át tudja vinni az S_i állapotot S_k -ba egy művelettel és használjuk erre az $S_i \xrightarrow{i} S_k$ jelölést.

D e f i n i c i ó : Egy p_i folyamat blokkolt az S_i állapotban, ha nem létezik olyan S_k , melyre $S_i \xrightarrow{i} S_k$ teljesülne.

Ha egy folyamat nincs blokkolt állapotban, akkor azt mondjuk, hogy a folyamat aktív.

A rendszer kezdetben az S_0 állapotban van. A rendszer állapotait a folyamatok változtatják meg úgy, hogy a rendszer egy állapot változását egy folyamat eredményezi.

Ha egy folyamat első művelete elkezdődik egy másik folyamat utolsó művelete előtt, akkor a két folyamatot párhuzamosnak tekintjük. A párhuzamos folyamatok tehát időben átlapolt folyamatok, két csoportba oszthatók:

- diszjunkt (független): ha mindegyik folyamat csak privát adatra hivatkozik,
- interakciós (együtműködő): ha a folyamatok közös adatokra (erőforrásokra) hivatkoznak.

A továbbiakban együtműködő folyamatokat tekintünk, melyek előrehaladási sebességére vonatkozóan semmiféle feltételezéssel sem élünk.

A folyamatoknak azok az akciói, amelyek nem hivatkoznak a közös adatokra, párhuzamosan végrehajthatók, míg azok az akciói, amelyek a közös adatokon végeznek műveleteket, általában csak egymás után hajthatók végre.

A blokkolt (felfüggesztett) folyamatok külső jelzés segítségével kelthetők ismét életre.

Az ilyen tulajdonságú folyamatokból álló rendszereket párhuzamos

folyamatrendszereknek nevezzük.

1.2. A vezérlés problémái

A rendszer folyamatai a számítógép konfiguráció erőforrásait használják. A közös erőforrások használata a folyamatok között elsőbbségi összefüggéseket teremt. Így a rendszer vezérlése a következő problémákat veti fel:

- holtponthelyzet probléma (deadlock)
- kölcsönös kizárás (mutual exclusion)
- szinkronizáció

Holtponthelyzet probléma. Olyan számítógép konfiguráción, amelyen kevesebb erőforrás áll rendelkezésre, mint amennyi a rendszer folyamatainak egy adott időpontban fellépő igénye, a folyamatok véglegesen blokkolt állapotba kerülhetnek, azaz a folyamatok holtponthelyzetre juthatnak. Ilyen rendszerekben a holtponthelyzet automatikus felismerése, illetve a holtponthelyzetmentes vezérlés megoldása a legfontosabb feladat. (Bankár algoritmus [3]). E kérdéssel a továbbiakban nem foglalkozunk.

Kölcsönös kizárás. Párhuzamosan futó folyamatok esetén fennáll a lehetősége annak, hogy egyidejűleg egynél több folyamat is a rendszernek ugyanazt az erőforrását kívánja használni. A folyamatok azonban megváltoztatják ezekben az erőforrásoknak a belső állapotát, ezért ezeket az erőforrásokat nem használhatja egynél több folyamat egyidejűleg. A különböző folyamatoknak vannak tehát olyan szakaszai, amelyek kölcsönösen kizárják egymást, ezeket kritikus szakaszoknak nevezzük.

A kölcsönös kizárási probléma tehát annak biztosítása, hogy legfeljebb egy folyamat hajthatja végre a kritikus szakaszt egy időben. A probléma megoldásának a következő igényeket kell kielégíteni:

- szimmetria (nincs statikus prioritás a folyamatok között),
- függetlenség a folyamatok relatív sebességétől,
- a folyamatok bármelyike megállhat a kritikus szakaszon kívül,
- amikor 2 vagy több folyamat szimultán próbálkozik belépni a kritikus

szakaszba, annak eldöntése, hogy melyik léphet be, véges időn belül eldől.

A szinkronizáció problémája a folyamatok együttműködésekor jelentkezik. Az együttműködés forrása rendszerint az, hogy a párhuzamosan futó folyamatok közös adatokon végeznek műveleteket. A folyamatok együttműködése esetén a folyamatok az erőforrások állapotának megváltoztatásával egymásnak információt adnak át. Követelmény, hogy egy folyamatnak mindaddig blokkolt állapotban kell maradnia, amíg egy másik folyamat nem állította elő számára a szükséges információt. Ilyenkor tehát a folyamatokat egymással szinkronban kell vezérelni, a rendszer folyamatainak bizonyos kritikus szakaszai egymás után, előre meghatározott sorrendben hajthatók csak végre.

A folyamatok szinkronizálása a párhuzamosan futó folyamatok olyan vezérlését jelenti, amely mellett a kritikus szakaszok az elsőbbségi összefüggéseknek megfelelő sorrendben kerülnek végrehajtásra.

A kölcsönös kizárás esetén is arról van szó, hogy egy folyamat a kritikus szakaszhoz érve mindaddig nem folytathatja az útját, amíg arra egy másik folyamat - a kritikus szakaszból kilépő folyamat-jelzésével engedélyt nem ad. A kölcsönös kizárás esetén azonban a kritikus szakaszok között általában nem állnak fenn elsőbbségi összefüggések. Kölcsönös kizárás esetén egy folyamat akárhányszor végrehajthatja a kritikus szakaszt, függetlenül attól, hogy egy másik folyamat hányszor hajtottta azt végre, csak az a lényeges, hogy a kritikus szakaszok végrehajtása nem történhet meg párhuzamosan.

Szinkronizáció esetén az, hogy egy folyamat végrehajthatja-e a kritikus szakaszt, attól is függ, hogy a rendszer többi folyamata hányszor hajtottta végre a kritikus szakaszt.

A kölcsönös kizárás és a szinkronizáció problémájának megoldására szolgáló eszközök áttekintését a 2. fejezet tartalmazza.

1.3. Párhuzamos utasítások

A párhuzamos programozás a programozási nyelvek területén is új eszközök létrehozását igényelte. Folyamatok párhuzamos végrehajtásának lehetőségét biz-

tosító, illetve azt explicit módon leíró nyelvi eszközök a párhuzamos utasítások.

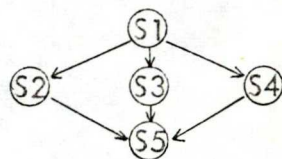
Dijkstra javasolta az ALGOL 60 nyelv kibővítését (1965-ben) a parbegin-parend utasítás zárójellel, hogy lehetővé váljon programok párhuzamos végrehajtásának leírása. Gyakran a cobegin-coend formával találkozhatunk, ami pontosan ugyanazt jelenti.

A parbegin-parend közötti programrészletet egy összetartozó egységnek lehet tekinteni, ami két dolgot jelent:

- az utasítás zárójelek között levő utasítások egyidejűleg végrehajthatók
- a parend utasítás zárójelet követő utasítás csak akkor hajtható végre, ha már a parbegin-parend között levő összes utasítás végrehajtása befejeződött.

Pl.: begin S1; parbegin S2; S3; S4 parend; S5 end

Itt S_i-k utasítások, s a konstrukció jelentése: S1 végrehajtása után egyidőben kezdődhet S2, S3 és S4 végrehajtása és amikor mindegyik befejeződött, akkor kezdődhet S5 végrehajtása (1.1. ábra).



1.1. ábra

A párhuzamos utasítások egymásba ágyazása is megengedett. Meg kell jegyezni, hogy az ALGOL 60 nyelv ilyen értelmű bővítése széles körben nem vált be, de ezt a jelölési formát azóta is elterjedten használják a párhuzamosság leírására.

A parbegin-parend páros nem ír elő explicit végrehajtási sorrendet (szekvenciálisat vagy párhuzamosokat), csak biztosítja a párhuzamosság lehetőségét.

A párhuzamos programfutás lineáris leírására Conway a fork, join és quit primitiveket vezette be [23].

Ezek explicit előírják a végrehajtó processzorok párhuzamosságait; folyamatokat létrehozhatnak, megszüntetnek, bizonyos mértékig szinkronizálnak, s a parbegin-parend programok implementálásának eszközei.

2. SZINKRONIZÁCIÓS ESZKÖZÖK

E fejezetben a párhuzamos folyamatok szinkronizációs eszközeit vizsgáljuk. Áttekintjük a szinkronizációs primitivek fejlődését a Dijkstra által bevezetett szemafor előtt és után, majd definiáljuk a fontosabb - s a dolgozat 5. fejezetében alkalmazott szinkronizációs eszköz kiválasztását is befolyásoló - primitiveket, példákkal illusztrálva azok használatát.

2.1. A szinkronizációs primitivek fejlődése

A kölcsönös kizárási probléma első megoldásai a gépi megszakítás rendszer használva realizálódtak, az I/O eszközök kommunikáltak a központi processzorral. Megszakítás előfordulásakor a folyamatok között a multiprogramozott operációs rendszerek "kapcsoltak". A felhasználói folyamatok logikailag nem osztoztak az erőforrásokon (pl. memórián), az operációs rendszer védelmezte a hardware-t megosztó folyamatokat.

Amint szükségessé vált a szinkronizáció alkalmazása a felhasználói programokban, a felfüggesztéses megszakítás vázolt módszere nem volt elegendő.

A szinkronizációs primitivek kifejlesztésének három szakaszáról beszélhetünk:

- szemafor előtt
 - Load/Store
 - Test and Set
 - Lock/Unlock
- szemafor szintű szinkronizációs eszközök
 - Block/Wakeup
 - P/V
 - Wait/Signal
 - Up/Down



- szemafor utáni magas szintű szinkronizációs eszközök
 - critical region
 - conditional critical region
 - monitorok
 - path kifejezések.

(A dolgozatban ezen eszközök gépi implementációjának kérdésével nem foglalkozunk, csak mint nyelvi eszközök kerülnek tárgyalásra.)

A legtöbb számítógépen a Load és Store utasítás oszthatatlan. Hosszu ideig azt hitték, hogy kritikus szakaszok kölcsönös kizárásának problémája nem oldható meg csupán a Load és Store utasítások használatával, míg végül is Dekker és Dijkstra talált megoldást.

A probléma további megoldását jelenti a Test and Set utasítás illetve a lock w - unlock w primitív pár, ahol w egy 1 bites lock változó. (A Petri hálók a szinkronizáció relációinak képi áttekintését adják és lehetővé teszik a folyamatok halmaza különböző tulajdonságai igazolásának leírását is. Részletes vizsgálatukkal a 3. fejezetben foglalkozunk.)

Az eddigiekben vázolt műveletek hardware eszközökkel, lényegében nyelvi segédeszközök nélkül valósítják meg a kölcsönös kizárást, kevésbé hatékony módon, s nagyon komplikáltan. Ha egy folyamat nem léphet be a kritikus szakaszba, hurokban várakozik (busy waiting), ami processzor idővesztést jelent. Az aktív várakozási mód - bizonyos értelemben fiktív utasítások ismételt végrehajtogatása - nagyon költséges.

Alapvetően új az a megoldás, hogy minden olyan folyamat, amely kritikus szakaszba akar lépni, de nem tud, mert egy folyamat már kritikus szakaszban van, blokkolódik, míg lehetősége nem lesz a kritikus szakaszban folytatódni.

A kezdeti lépést a Block és Wakeup primitív pár jelentette. A Dijkstra féle szemafor, s az ezen műveleteket végző P és V primitivek megjelenése emeli a szinkronizáció problémáját a gépi szint fölé. A programozó képessé válik a párhuzamos folyamatok programozására. A kölcsönös kizárás problémáját elegáns

módon oldja meg, a busy waiting-et implicit folyamat sorral helyettesítve. A P és V műveletek a szinkronizációt a program tetszőleges pontján írják le. További szemafor szintű szinkronizációs eszközök az IBM primitívek illetve P.B.Hansen primitívjei [23] .

A P és V műveletek dekompozíciójaként konstruált műveletek az up és down, melyek összetettebb problémák megoldásához használhatók sikerrel. A wait és signal műveleteket pedig a program bizonyos tulajdonságai igazolásához célszerű használni.

Az eddigiekben említett eszközök assembly nyelvi szintűek. A műveleteket könnyen fel lehet cserélni, vagy rossz helyre tenni és a fordítónak nincs módja ellenőrizni azok használatát. A szemaforok hátránya tehát, hogy helytelen használatuk esetén (rossz sorrend) idő függő hibákat véthetünk és ezek fordítás idején nehezen ellenőrizhetők.

Még hatékonyabb, kevesebb hibalehetőséggel rendelkező megoldást jelentenek az 1970. után kifejlesztett ún. magas szintű szinkronizációs eszközök.

A vázolt problémát oldja meg a critical region oly módon, hogy az előírt nyelvi konstrukció használatával képessé teszi a fordítót annak ellenőrzésére, hogy a közös változók csak kritikus szakaszokban legyenek használva. A critical region a szinkronizáció egyik oldalát oldja meg, a kölcsönös kizárást, egy folyamatot egy speciális feltétel teljesüléséig késleltet.

Kiterjesztése a conditional critical region, mely egy folyamatot egy tetszőleges feltétel teljesüléséig képes várakoztatni. Egy eseményre várakozás idejére a folyamat a közös változóhoz tartozó sorra fűződik fel. Ha a feltétel már teljesül, a folyamat kilép a sorból. A szinkronizációt a közös adat objektum használatának helyén írják elő.

Hangsúlyozni kell, hogy a critical region illetve a conditional critical region csak nyelvi forma (miként a parbegin-parend), s a fordító feladata a szinkronizált végrehajtást megvalósító utasítások generálása (szemben pl. a szemaforral, mely P és V műveleteivel egy tényleges szinkronizációs eszköz).

A párhuzamos programok helyessége bizonyításának szükségessége és az adat absztrakt fogalmának megjelenése a szinkronizáció leírásának új megközelítéséhez vezetett. A szinkronizációt a közös adat (objektum) definíciójában írják elő a monitorok és a path kifejezések.

A monitorok esetében az összes definiált művelet (a monitor eljárások) implicit kölcsönös kizárásáról van szó, és a szinkronizáció összes egyéb típusa a műveletek belsejében van leírva a signal és wait alacsony szintű szinkronizációs primitívek felhasználásával.

A path kifejezések esetén a szinkronizáció egy kifejezésként van formalizálva, meghatározva azt a sorrendet, ahogy a műveletek hozzá tudnak fordulni az osztott adat objektumokhoz.

A programozó számára kényelmes magas szintű szinkronizációs eszközök a szinkronizáció megvalósítása terén is komolyan támaszkodnak a fordító tevékenységére.

A terület újabb eredménye pl. [7], [13], [27] ill. [29]-ben tekinthető át. Mivel az ezekben szereplő eszközökre a dolgozatban nem kívánunk támaszkodni, ismertetésüktől is eltekintünk.

Egyes eszközökkel megvalósított szinkronizáció helyességének igazolásával [5] illetve [12] foglalkozik.

2.2. Párhuzamos folyamatok együttműködésének szervezése

Vizsgáljuk, hogyan lehet különböző nyelvi eszközökkel biztosítani párhuzamos folyamatok helyes együttműködését, szinkronizációját.

2.2.1. Memory Interlock

A kölcsönös kizárás megvalósítható akkor is, ha csak azt tételezzük fel, hogy a számítógép Load és Store utasítása oszthatatlan. Ezt nevezzük memory interlock-nak. Ez annyit jelent, hogy a számítógép hardware biztosítja, hogy

begin

C1,C2,t : INT;

C1:=1; C2:=1 ; t:=1 ;

cobegin

P1: begin A1 : C1 := 0;

L1 : if C2 = 0 then

begin if t=1 then goto L1 ;

C1:=1 ;

B1 : if t=2 then goto B1 ;

goto A1

end ;

1. kritikus szakasz;

t:=2 ; C1:=1 ;

1.ciklusmag további része ; goto A1

end ;

P2:begin A2 : C2:=0;

L2 : if C1 = 0 then

begin if t=2 then goto L2;

C2:=1 ;

B2: if t=1 then goto B2 ;

goto A2

end;

2. kritikus szakasz;

t:=1 ; C2:=1 ;

2. ciklusmag többi része; goto A2

end

coend

end

egy időben csak egy folyamat férhet hozzá a memória egy szavához, ha egy másik is ugyanezt a szót akarja felülni vagy vizsgálni, várnia kell.

Legyen P1 és P2 két szekvenciális folyamat, melyeket 1-1 ciklus valósít meg. Mindkettő tartalmazzon egy kritikus szakaszt. Ezek kölcsönös kizárását biztosítja a 2.1. algoritmus.

A kezdeti próbálkozások, s a Dekker által talált előzőekben ismertetett algoritmus N folyamat esetére való általánosítása Dijkstra [3] cikkében található részletesen.

2.2.2. Test and Set

A kölcsönös kizárás megoldásához a hardware a memory interlock-nál több támogatást is nyújthat. Egyik ilyen lehetőség a Test and Set utasítás [23].

Az utasításnak 2 paramétere van, L és C. Az utasítás C értékét L-hez rendeli, majd C értéke 1 lesz. ($L:=C$; $C:=1$).

E két művelet egy oszthatatlan utasítás alatt játszódik le. A C változó a szinkronizálandó folyamatok közös változója. A kölcsönös kizárás problémájának megoldását a Test and Set utasítás segítségével a 2.2. algoritmus mutatja. (Természetesen a memory interlock létezése itt is előfeltétel).

```
begin C:INT;  
      C:=0;  
cobegin  
      P1: begin L1:INT;  
            while true do  
              begin L1:=1;  
                while L1=1 do Test and Set (L1,C);  
                1. krit. szakasz ;  
                C:=0;  
                :  
                :  
              end  
            end  
end
```

/ folyt. /


```
P2: begin L2 : INT;  
      while true do  
        begin L2 :=1;  
          while L2=1 do Test and Set (L2,C) ;  
            2.krit.szakasz;  
            C:=0;  
            ;  
          end  
        end  
      end  
    coend  
end
```

2.2. algoritmus

2.2.3. Lock és unlock

E megoldás esetén - melyet Dennis és Van Horn javasolt - a kritikus szekció egy lock w - unlock w utasítás pár között van, ahol w lehet egy 1 bites változó [23]. A primitivok definíciója:

```
lock w ::= L : if w = 1 then goto L  
                  else w:=1  
unlock w ::= w := 0
```

A kölcsönös kizárás problémájának megoldását ez alapján a 2.3. algoritmus mutatja. A folyamatok indulásakor a w bit értéke 0.

```
begin m : INT;  
      m :=0 ;  
    cobegin  
      P1 : begin  
            ;  
            lock m ;  
            1. kritikus szakasz ;  
            unlock m;  
            ;  
          end
```

/ folyt./



```

P2 : begin
      :
      : lock m ;
      : 2. kritikus szakasz;
      : unlock m
      :
      : end
coend
end

```

2.3. algoritmus

2.2.4. Block és Wakeup

Saltzer és Lampson definiálta a Block és Wakeup primitív párt, melynek jelentése a következő. Minden folyamathoz tartozik egy wakeup waiting switch (WWS).

Block(p) ::= ha a p folyamathoz tartozó WWS egyenlő 1-gyel, akkor WWS a nulla értéket veszi fel, és a folyamat tovább fut ; ha WWS értéke nulla, akkor a folyamat blokkolódik.

Wakeup(p) ::= ha a p folyamat blokkolt, akkor aktív válik, míg ha aktív volt, akkor WWS az 1 értéket veszi fel.

A primitívek paramétere egy folyamat azonosító. Hátránya, hogy ha több folyamat vár ugyanazon feltétel teljesülésére egy egész folyamat listát kell nyilvántartani valahol, vagyis a folyamatoknak túl sokat kell tudniuk környezetükről.

2.2.5. Szemaforok

A Dijkstra féle szemafor [3] a folyamatok egy speciális célú, nem negatív egész értékű közös változója. Deklarációját követően meg kell adni kezdeti értékét is. Pl:

s : SEM ; s:=1

Az s szemaforon a folyamatok két - oszthatatlan végrehajtásu - primitív utasítással végezhetnek műveletet.

Csökkentési kísérlet : $P(s)$

1. ha a szemafor értéke nulla, akkor a hívó (a $P(s)$ primitív utasítást végrehajtó) folyamat blokkolódik a szemaforon (ebből csak egy erre a szemaforra kiadott V művelet veheti ki);
2. egyébként a szemafor értéke eggyel csökken és a hívó folyamat blokkolás nélkül folytatódik.

Növelés : $V(s)$

1. ha a szemafor értéke nulla és (több) folyamat (is) blokkolódott a szemaforon, akkor a hívóhoz való visszatérés előtt a(z egyik) folyamat fel lesz ébresztve és visszatérhet (befejezettnek minősülő) P műveletéből;
2. egyébként a szemafor értéke eggyel növekszik, majd visszatérés lesz a hívóhoz.

A V művelet során a blokkolódott folyamatok közötti választásnak olyanak kell lenni, hogy egyetlen folyamatnak sem szabad meghatározatlanul hosszú ideig blokkoltan maradnia.

(A fenti primitív utasításokat rendszerint úgy valósítják meg, hogy azok az 1. pontban megadott feltételek teljesülése esetén megszakítást eredményeznek és a megszakítás hatására az operációs rendszer kapja meg a vezérlést intézkedés végett. Ilyenkor az operációs rendszer a $P(s)$ utasítás esetében gondoskodik a megszakított folyamat állapotának a megőrzéséről és rögzíti a megszakított folyamat adatait az s szemaforhoz rendelt várakozó sor adatai között. A $V(s)$ utasítás esetében viszont az operációs rendszer határozza meg egy kiválasztási algoritmus szerint azt, hogy az s szemaforhoz rendelt várakozó sorból melyik folyamat folytathatja a futását és gondoskodik annak indításáról.)

Szemafort alkalmazva a kölcsönös kizárási mechanizmus megvalósítására a 2.1. algoritmusnál jóval egyszerűbb megoldást kapunk (2.4. algoritmus):

```

begin   f : SEM ;
        f:=1;
        cobegin
            P1 : begin ... end ;
            P2 : begin ... end ;
        coend
end

    ahol:
    Pi : begin
        Li : P(f) ; i. Kritikus szakasz ; V(f) ;
        i. ciklusmag többi része ; goto Li
    end

```

2.4. algoritmus

Látható, hogy a 2.4. algoritmus N folyamatból álló konstrukció esetén is egyszerű és könnyen áttekinthető marad. Ha nincs folyamat a kritikus szakaszban, akkor $f = 1$, míg ha a folyamatok egyike már belépett a kritikus szakaszába, akkor $f = 0$.

A szemaforokat számos szinkronizációs probléma megoldására is sikerrel alkalmazták.

Egy gyakorlati szinkronizációs feladatként tekintsük az ún. üzenet-buffer problémát. Adott egy kommunikációs puffer, mely egy n elemű ciklikusan láncolt sor,

- a P_i ($1 \leq i \leq k$) termelő folyamatok elemeket állítanak elő és elhelyezik azokat a pufferbe (<betesz>),
- a Q_j ($1 \leq j \leq l$) fogyasztó folyamatok a betétel sorrendjében kiveszik az elemeket (<kivesz>) és feldolgozzák azokat.

A termelők és fogyasztók közötti szinkronizációt a következő invariáns fejezi ki:

$$(\forall t \text{ in } to..T) \quad (0 \leq \sum_{t=to}^T \langle \text{betesz} \rangle - \sum_{t=to}^T \langle \text{kivesz} \rangle \leq n)$$

ahol $\sum_{t=t_0}^T$ <művelet> az adott művelet sikeres végrehajtási számát adja meg a to.. T tartományban, s kezdetben a puffer üres. Vagyis a betételek száma mindig legalább annyi, de legfeljebb <pufferméret> -tel több, mint a kivételek száma.

A kizérési feltétel: az adatszerkezeten egyidejűleg csak egy folyamat végezhet műveleteket.

Szemaforok alkalmazásával a probléma megoldását a 2.5. algoritmus adja:

```

begin  s,r,w : SEM ;
      s := 0 ; r := n ; w := 1 ;
      cobegin
        Pi : begin A1 : <köv. elem előállítás> ;
                P(r) ;
                P(w) ;
                <betesz> ; (1) (3)
                V(w) ;
                V(s) ;
                goto A1
              end ;
        Qi : begin
                A2 : P(s) ;
                    P(w) ;
                    <kivesz> ; (2) (4)
                    V(w) ;
                    V(r) ;
                    <kiemelt elem feldolgozása> ;
                    goto A2
              end
            end
          coend
        end

```

Itt 3 szemaforot használtunk : egy szemafor (w) biztosítja a puffer műveletek kölcsönös kizárását, míg két másik szemafor (r és s) a termelők és fogyasztók közötti szinkronizációt. r mindenkor a pufferbeli üres helyek számát, míg s a pufferben levő (feldolgozásra váró) adatok számát tartalmazza. ((1) - (4) -re a későbbiekben fogunk hivatkozni.)

Mint láttuk, a szemafor eszközt adott passzív várakozás megvalósítására. Bonyolultabb folyamatrendszerek programozásakor szemaforok alkalmazása nehézségekkel jár. Több nevezetes feladat ismeretes, melyekre csak hosszabb idő után sikerült megoldást találni. Ilyenek például a "Dining philosophers" feladat [4] és a "Readers, writers" feladat [2].

2.2.6. Szemafor műveletek dekompozíciója

Wodon a P-hez és V-hez hasonló, de a létező szinkronizációs problémák megoldására alkalmasabb primitiveket konstruált, melyeket Belpaire és Wilmotte fejlesztett tovább.

A P és V műveletek viszonylagos komplikáltságát az okozza, hogy a szinkronizáló hatás nincs elválasztva a feltételes várakoztatástól. E probléma enyhítésére a folyamatokbeli események függésének definiálására 3 d-művelet szolgál. Ezek a műveletek a P és V-nél alacsonyabb szintűek. A szemafor változón értelmezett 3 elemi d-művelet:

1. Szemafor folyosó: s :
akkor hajtható végre, ha $s > 0$; s -re nincs hatása
2. Szemafor zárás: down s
bármikor végrehajtható; hatása : $s \leftarrow s - 1$
3. Szemafor nyitás: up s
bármikor végrehajtható; hatása: $s \leftarrow s + 1$

Ezek a d-műveletek oszthatatlanok és felhasználásukkal további oszthatatlan végrehajtható műveletek definiálhatók, pl. az "s:down s" kombináció a Dijkstra féle P műveletnek felel meg.

Egy komplexebb kompozícióra szolgáljon példaként az " $s_a \cdot s_b : \underline{\text{down}} s_a, s_b$ " konjunkció. Az $s_a \cdot s_b$ végrehajtására akkor kerüljön sor, ha $s_a > 0$ és $s_b > 0$, míg $\underline{\text{down}} s_a, s_b$ hatását a $\underline{\text{down}} s_a$ és $\underline{\text{down}} s_b$ egyesített hatásával definiáljuk. ($\underline{\text{down}} s_a$ és $\underline{\text{down}} s_b$ oszthatatlanságából nem következik a belőlük konstruált $\underline{\text{down}} s_a, s_b$ oszthatatlansága. Utóbbi csak akkor lesz oszthatatlan végrehajtásu művelet, ha annak definiáljuk, vagyis ennek megfelelően implementáljuk.)

A 2.5. algoritmus (3) illetve (4) által jelölt részét írjuk át új eszközünk felhasználásával két különböző módon is, a w, r, s szemaforoknak rendre s_w, s_r, s_s szemaforokat megfeleltetve (2.6. algoritmus).

$$P'_i : \begin{array}{l} : \\ s_r : \underline{\text{down}} s_r ; \\ s_w : \underline{\text{down}} s_w ; \\ \quad \langle \text{betesz} \rangle ; \\ \quad \underline{\text{up}} s_w ; \\ \quad \underline{\text{up}} s_s ; \\ \quad : \end{array}$$

$$P''_i : \begin{array}{l} : \\ s_w \cdot s_r : \underline{\text{down}} s_w, s_r ; \\ \quad \langle \text{betesz} \rangle ; \\ \quad \underline{\text{up}} s_w, s_s ; \\ \quad : \end{array}$$

$$Q'_i : \begin{array}{l} : \\ s_s : \underline{\text{down}} s_s ; \\ s_w : \underline{\text{down}} s_w ; \\ \quad \langle \text{kivesz} \rangle ; \\ \quad \underline{\text{up}} s_w ; \\ \quad \underline{\text{up}} s_r ; \\ \quad : \end{array}$$

$$Q''_i : \begin{array}{l} : \\ s_w \cdot s_s : \underline{\text{down}} s_w, s_s ; \\ \quad \langle \text{kivesz} \rangle ; \\ \quad \underline{\text{up}} s_w, s_r ; \\ \quad : \end{array}$$

2.6. algoritmus

Látható, hogy az up és down a legegyszerűbb formában használva semmit sem változtat a P és V segítségével konstruált megoldás formáján (P'_i, Q'_i). A d-műveletekben konjunkciót használva, a megoldás formáját (P''_i, Q''_i) tartalmazza.

A d-műveleteket összetettebb feladatok megoldására célszerű használni (pl. "Readers , writers" feladat).

2.2.7. Critical region

P.B.Hansen [8] a strukturált programozás szempontját is figyelembe véve a kölcsönös kizárás megoldására a (T típusu) shared (közös) változó használatát javasolta, melynek deklarációja:

var V: shared T

Ehhez a változóhoz a folyamatok csak egy strukturált utasítás belsejében férhetnek hozzá. A strukturált utasítás formája:

region V do S illetve
region V do begin S1 ; S2 ; ... ; Sn end

Az S_i utasításokat egy közös V változóhoz rendeli, s ezeknek az utasításoknak kizárólagos hozzáférési joguk lesz V-hez. A kritikus szakaszok ugyanarra a V változóra csak egymást időben kizárva hivatkozhatnak, tehát az erőforrást (V - t) legfeljebb 1 folyamat használhatja egyidejűleg. A többi folyamat egy sorban várakozik. A folyamatok kritikus szakaszukban véges ideig tartózkodhatnak, s ha egy folyamat be akar lépni kritikus szakaszába, erre véges időn belül lehetősége lesz.

E megoldás legfőbb előnye, hogy képessé teszi a fordítót annak ellenőrzésére, hogy a közös változók csak kritikus szakaszokon belül legyenek használva, és helyes kódot képes generálni. A kód generálás szintjén alkalmazhatók például a szemafor műveletek.

Az üzenet-buffer problémát leíró program szekciók törzse (2.5. algoritmusban (1) ill. (2)) a következőképp írható:

P_i : ... region PUFFER do begin <pufferbe töltés> end ; ... (1')

Q_i : ... region PUFFER do begin <olvasás a pufferből> end ; ... (2')

ahol a deklarációk között

var PUFFER : shared T

is szerepel. (Egy komplett megoldás található [8] -ban).

2.2.8. Conditional critical region

Nemcsak a kölcsönös kizárást valósítja meg region-ok között, hanem region-ok végrehajtási sorrendjét is tudja vezérelni.

Egy folyamatot egy tetszőleges feltétel teljesüléséig képes várakoztatni a conditional critical region (Hoare [11] , P.B. Hansen [8]),

formája:

region V when B do S illetve
region V when B do begin S1; ... ; Sn end

ahol V T típusu közös változó a

var V: shared T

deklarációval. Jelentése: a kritikus szakasz akkor hajtható végre, amikor a B feltétel érvényes. A B logikai kifejezés operandusai a V közös adatstruktúra komponensei lehetnek.

A folyamatok kritikus szakaszukba lépésük előtt egy a V közös változóhoz tartozó Q_V fő sorba kerülnek. Ebből a sorból lép egyidőben egy folyamat kritikus szakaszába (kölcsönös kizárás), s kiértékelésre kerül a B feltétel. Ha B igaz, a folyamat az S utasítás végrehajtásával folytatódik. Ha B hamis, a folyamat ideiglenesen elhagyja a kritikus szakaszt, bekerül egy ugyan csak a V közös változóhoz tartozó Q_e esemény sorba és mindaddig késleltetve lesz, amíg egy másik folyamat ugyanazon a V változón egy kritikus szakaszt teljesen végre nem hajt. Ekkor a Q_e -beli folyamatok átkerülnek Q_V -be, s a felfüggesztett folyamat számára megengedett, hogy ismételten belépjen kritikus szakaszába és kiértékelje B-t. Ez a ciklus mindaddig folytatódik, míg B igaz lesz és az S utasítás végrehajtásra kerül.

Az előbbi forma egy kézenfekvő kiterjesztéseként vezessük be a követke-

ző szimmetrikus konstrukciót [6] :

region V do S await B

mely az S utasítás végrehajtását követően késlelteti a folyamatot a B feltétel teljesüléséig. Egy további konstrukció:

region V do begin ... await B; S end

melynek jelentése: ha B igaz, akkor az S utasítás végrehajtásra, míg egyébként a folyamat felfüggesztésre kerül.

var v: shared record s: array [max] of T; full: INT end ;

m, n : T;

:

full := 0 ;

:

P_i: ...

region v when full < max do begin

put (m,s); full:=full+1

end ;

Q_i: ...

region v when full > 0 do begin

get (n,s); full:= full-1

end;

2.8. algoritmus

Az üzenet-buffer problémát conditional critical region-ok felhasználásával megoldva az általános szemaforok helyére számlálók kerülnek (2.8. algoritmus). A 2.5. algoritmushoz képest a deklarációs rész változik, s (3) és (4) tömörebbé válik. E megoldás esetén csak egy termelő vagy egy fogyasztó tudja egyidőben elérni a puffert, mivel mindkét kritikus szakaszban ugyanazt a változót használtuk (kölc.s.kizárás). A szinkronizáció egyéb formáit a full számláló segítségével egy mechanizmus oldja meg. Ha egy termelő megpróbál adatot tenni a tele pufferbe, a kritikus szakaszba lépéskor le fog állni és egy várakozási listára kerül.

Valahányszor egy másik folyamat elhagyja a v-hez tartozó region-t, a feltétel újra kiértékelésre kerül. Ez esetben egy termelő képessé válik tevékenysége folytatására, mihelyst egy fogyasztó adatot vesz ki a pufferből és elhagyja kritikus szakaszát.

(A vizsgált feladat komplett megoldását [8] , míg a "Readers, writers" feladat megoldását [9] tartalmazza.)

2.2.9. Monitorok

A szinkronizáció előírásának az osztott adat objektum definíciójába és a rajta végzett műveletekbe helyezésére az első lépés a monitor fogalmának bevezetése volt.

A számítógépek operációs rendszereinek fő feladata a számítógép erőforrásainak kiosztása az együtt futó programok előre jelezhetetlen igényeivel összhangban. A munka megkönnyítése érdekében célszerű típusokba sorolni az egyforma berendezésekhez tartozó ütemezési eljárásokat. Egy "ütemező" nem más, mint az adminisztrációhoz szükséges adatok összessége, valamint azok az eljárások és függvények, amelyek a szóban forgó berendezés kezelésével kapcsolatosak. Egy ilyen logikailag összetartozó strukturát Hoare [10] nyomán monitornak nevezünk. A monitor formális megadása például a következő lehet:

monitornév: monitor

begin ... a monitor lokális adatainak leírása ;

procedure eljárásnév (... formális paraméterek);

begin ... eljárástörzs ... end ;

... a monitor további lokális eljárásainak deklarációja; ...

... a monitor lokális adatai kezdő értékének beállítása ;

end

A monitor eljárások aktivizálása formálisan a következőképp történhet:

monitornév . eljárásnév (... aktuális paraméterek ...) ;

A monitor definíción kívül csak az eljárás neve ismert, és a monitor lokális adatai kölcsönösen kizárt elérését az a tény garantálja, hogy egy időben csak



egy monitor eljárás lehet végrehajtás alatt. A monitor eljárások csak a monitor számára lokális adatokhoz férhetnek hozzá (kivéve más monitorokon keresztül).

A monitorok lokális adatai kívülről, az őket kezelő eljárások megkerülésével nem érhetőek el.

A konstrukció tehát biztosítja, hogy a kölcsönös kizárási problémát a monitor mechanizmus saját maga megoldja.

A monitor eljárásokban szükség lehet az őket aktivizáló folyamatoknak (különböző okok miatti) felfüggesztésére. A későbbiekben pedig gondoskodni kell a felfüggesztett folyamatok újraindításáról. Erre a célra bevezetjük a wait és signal műveleteket.

rb: monitor

begin buffer: array [n] of T ;

lp, c: INT ; ne, nf : condition ;

procedure pufferbe (x:T) ;

begin if c = n then nf. wait fi ;

($\times 0 \leq c < n \times$) ;

buffer [lp] := x ; lp := (lp + 1) mod n ;

c := c + 1 ; ne.signal

end ;

procedure pufferből (x:T) ;

begin if c = 0 then ne. wait fi ;

($\times 0 < c \leq n \times$) ;

x := buffer [(lp - c) mod n] ; ($\times -m \bmod n \equiv n - m \bmod n \times$) ;

c := c - 1 ; nf.signal

end ;

c := 0 ; lp := 0

end

(a buffer indexe 0..n-1, lp értéke 0..n-1, míg c értéke 0..n).

2.9.a. algoritmus

rástörzs olyan szervezése praktikus, midőn a jelenleg commentbe írt feltétel szerepel az if után).

Monitor megoldás esetén egy várakozási listán levő folyamat valamely másik folyamat által végrehajtott signal művelet hatására aktivizálódik. A várakozó folyamatok feltételeinek a conditional critical region-ok esetén látott újbóli automatikus kiértékelésére nem kerül sor.

2.2.10. Path kifejezések

A szinkronizáció leírásának a típus definícióba történő teljes koncentrálására Campbell és Habermann [1] dolgoztak ki módszert. Ez az új mechanizmus a szinkronizációt az eljárások szintjén írja le, lehetővé téve különböző folyamatok által végrehajtott eljárások közötti szinkronizáció path kifejezésekkel történő leírását.

A path kifejezések az eljárások szinkronizálásának szervezését írják le. Azokból az eljárásokból épülnek fel, melyeket a folyamatoknak szinkronizáltan kell végrehajtani. A szinkronizáció módját, milyenségét formálisan adják meg, képzésük az alábbi szabályok szerint történik:

- pontosvesszővel elválasztott eljárások csakis a leírás sorrendjében, egymás után hajthatók végre (sorozat, jele: ";")

$p ; q ; r$ a q eljárást hívó folyamat felfüggesztésre kerül mindaddig, amíg a p eljárás végrehajtása tart.

- vesszővel elválasztott eljárások közül (a kifejezés egyszeri végrehajtásakor) mindig csak az egyik hajtható végre (választás, jele: ",")

$p ; (q, r) ; s$ a következő végrehajtási rendet engedi meg: pq vagy prs

- a path és end jelek közé helyezett eljárásokat végrehajtásuk után ismét végre lehet hajtani (implicit ismétlődés). A path-end-ek nem ágyazhatók egymásba

path $p ; (q, (r ; s))$ end pq vagy prs végrehajtása után előlről kezdődhet a kifejezés kiértékelése

- kapcsos zárójelben levő eljárás egyidejűleg többszörösen is végrehajtás alatt állhat (szimultán végrehajtás). A kapcsos zárójeleket nem lehet egymásba ágyazni.

path {read} , write end a read eljárást több folyamat is végrehajthatja időben átlapoltan, de write végrehajtása sem másik write , sem read eljárás végrehajtásával sem lehet átlapolva (A read eljárás itt monopolizálódhat)

A "Readers, writers" feladat megoldását path kifejezések segítségével [1] tartalmazza.

Adat objektumok konstruálására használjuk a type definíciót, melyet a PASCAL nyelv [30] megfelelő fogalmának kiterjesztéseként állítunk elő. Párhuzam huzható a type definíció és az eljárás deklaráció között:

egy típusként deklarált objektum strukturális részének elérése hasonló egy eljárás (procedure) törzsébe való belépéshez.

Például egy buffernek nevezett objektum típus definiálása a következő:

```
type buffer =  
  begin  f : T ; (× adat def ×)  
          (× üres sor × )  
          operations (× operációs rész × )  
            procedure read ... ;  
            procedure write ... ;  
  end
```

Egy programban var A : buffer ; v : T deklarációkat is elhelyezve

A.read(v) hatása: a buffer tartalma olvasásra kerül v-be ;

A.write (v) hatása: a v változó tartalma feliródik az A bufferbe.

Amikor különböző párhuzamos folyamatok szimultán akarnak a közös bufferen műveleteket (read, write) végezni, ez a definíció nem kielégítő. A path kifejezésekkel adott szinkronizációs szabályokat be kell olvasztani az aszinkron folyamatok által közösen használt adatobjektumok leírására szolgáló type de-

```
type buffer =  
  begin f : T ;  
    path write ; read end ;  
    operations procedure read (m:T) = begin m:=f end ;  
      procedure write (m:T) = begin f:=m end ;  
  
  end ;  
  
type pointer =  
  begin p : INT ; p:=0 ;  
    path next end ;  
    operations procedure * next : INT =  
      begin p:=(p+1) mod n ; p end ;  
  
  end  
  
type űzenet-buffer =  
  begin rb: array [n] of buffer ;  
    wsz, rsz: pointer;  
    operations procedure pufferbe (m:T) ; var j:INT =  
      begin j:=wsz.next ; rb [j] . write (m)  
      end ;  
    procedure pufferbűl (m: T) ; var j : INT =  
      begin  
        j:= rsz.next ; rb [j]. read (m)  
      end ;  
  
  end
```

(Az rb tűmb indexe 0..n-1)

2.10. a. algoritmus

*./ A 6.fejezetben erre function-kűnt fogunk hivatkozni

finicióba. Példánk esetén a (* üres sor *) helyére: path write; read end irandó.

A type definíciónak tehát tartalmaznia kell az adatmezők leírását és az inicializálásukat, egy a szükséges szinkronizációt leíró path kifejezést, továbbá a lokális eljárások definíciót (a leírt típusu adatobjektumon végrehajtandó műveleteket):

```
type név =  
    begin adat def ...  
        inicializálás  
        path .. end  
        operations művelet def  
    end
```

Az üzenet-buffer probléma megoldásához szükséges type definíciókat a 2.10.a. algoritmus, míg az eljárásokat aktivizáló folyamatokat a 2.10.b. algoritmus írja le.

```
                var RB: üzenet-buffer;  
  
Pi : begin var y : T ;  
        :  
        RB.pufferbe (y) ;  
        :  
    end ;  
  
Qi : begin var y:T ;  
        :  
        RB.pufferből (y);  
        :  
    end ;
```

2.10.b. algoritmus

A puffer pontereken végzett műveletek kölcsönös kizárását az alkalmazott path kifejezés megoldja, a pointerek függetlenek, így a küldő és fogadó műveletek is azok. Legfeljebb 1 termelő és legfeljebb 1 fogyasztó tudja szimultán elérni vagy módosítani az üzenet-buffert (kizárási feltétel). A termelő-fogyasztó szinkronizációt a buffer path kifejezése biztosítja.



3. PÁRHUZAMOS FOLYAMATOK GRÁF MODELLJE

A programok tervezése, írása során szükséges egy formalizmus, amellyel párhuzamos folyamatok kölcsönhatásai leírhatók. Szükséges egy precíz és lehetőleg szemléletes elméleti eszköz, amelynek segítségével programok párhuzamos feldolgozását megvalósító rendszerek konstruálása megkönnyíthető, specifikáció szerinti működésük ellenőrizhető.

Az utóbbi években a Petri hálókat kezdik alkalmazni párhuzamos folyamatok modellezésére. A Petri háló eszköz párhuzamos folyamatok szinkronizációjának leírására és bizonyos tulajdonságai bizonyítására. Részletes leírásuk [14] és [24] -ben található.

3.1. Petri hálók

D e f i n i c i ó : Tekintsünk egy B ponthalmazt és egy $E \subseteq B \times B$ élhalmazt. A $G = (B, E)$ párt irányított gráfnak nevezzük.

D e f i n i c i ó : Egy Petri háló egy hármast $PH = \langle P, T, A \rangle$, ahol P és T véges diszjunkt halmazok, A pedig bináris reláció:

P : a helyek nem üres halmaza $\{p_1, p_2, \dots, p_n\}$,

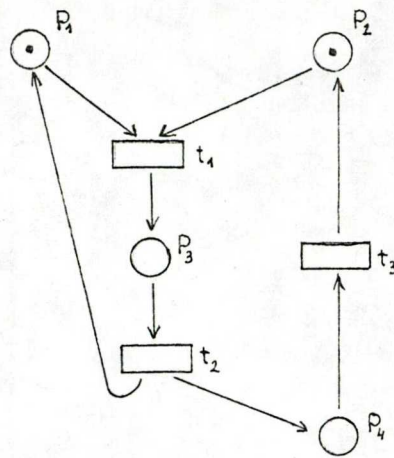
T : az átmenetek nem üres halmaza $\{t_1, t_2, \dots, t_m\}$,

A : az irányított élek halmaza; egy irányított él egy helyet egy átmenettel vagy egy átmenetet egy hellyel köt össze.

Igy PH egy irányított gráf két fajta csúccsal (helyek és átmenetek). A gráf csúcsainak halmaza $P \cup T$, irányított éleinek halmaza:

$A \subseteq (P \times T) \cup (T \times P)$. $p_1 \in P$ input helye a $t \in T$ -nek, ha $(p_1, t) \in (P \times T)$, azaz irányított él vezet p_1 -ből t -be; $p_2 \in P$ output helye $t \in T$ -nek, ha $(t, p_2) \in (T \times P)$, azaz irányított él vezet t -ből p_2 -be. P és T minden elemének szerepelni kell legalább egy reláció elemében.

Egy Petri hálót szemléletesen megadhatunk egy irányított páros gráffal, a helyeket körökkel, az átmeneteket kis téglalapokkal jelölve (3.1. ábra).



3.1. ábra

D e f i n i c i ó : Egy $M : P \rightarrow I$ függvényt a PH háló pontozásának nevezzük. I a nemnegatív egészek halmaza. Ha $p_i \in P$, akkor $M(p_i)$ a p_i helyen levő pontok számát jelenti. Egy p_i helyet pontozottnak nevezünk, ha $M(p_i) \geq 1$. Az M függvényt egy vektorral adhatjuk meg, ahol a vektor elemeit a helyekkel indexelhetjük. A (PH, M) párt pontozott Petri hálónak nevezzük.

D e f i n i c i ó : Egy átmenetet aktivizálhatónak nevezünk, ha minden input helyén van legalább egy pont.
A háló léptetési szabálya a következő: Válasszunk ki az aktivizálható átmenetek közül egyet ; vegyünk el minden input helyéről egy pontot és tegyünk minden output helyére egy pontot ; ismételjük -e két lépést, ha lehet. A háló egy átmenetének a billentése (egy akció) oszthatatlan esemény, ez idő alatt más nem történhet a hálóban.

A háló kiértékelése során az aktivizálható átmenetek közül (ha van egyáltalán) egy tetszőleges kiválasztásra kerül, átbillen. Ezzel előáll egy új pontozás, amelyen folytatódhat a kiértékelés.

D e f i n i c i ó : M_i pontozásból M_j pontozás elérhető, ha a kiértékelés során létezik a billenések olyan S sorozata, amely az M_i pontozást M_j -be viszi át.

D e f i n i c i ó : Egy adott M kezdeti pontozásból elérhető pontozások $[M]$ halmazát a háló M -hez tartozó elérhetőségi halmazának nevezzük.

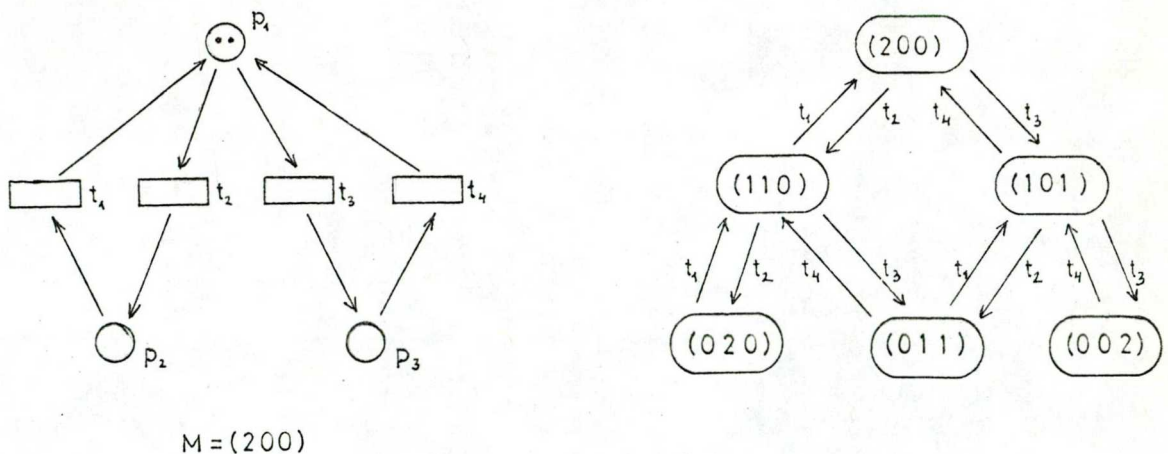
D e f i n i c i ó : Egy $p \in P$ hely biztonságos az M pontozás mellett, ha $(\forall M' \in [M]) (M'(p) \leq 1)$

D e f i n i c i ó : Egy pontozás biztonságos, ha ennél a pontozásnál minden egyes hely biztonságos.

Egy (PH, M) háléhoz hozzárendelhető egy átmenet-gráf (elérhetőségi fa), mely irányított gráf és a következőképp konstruálható.

Indexeljük a szögpontokat az $[M]$, az éleket a T elemeivel. Legyen a kezdő pont M . Képezzük az összes aktivizálható átmenetre az M -ből elérhető pontozásokat. A különböző pontozásoknak különböző szögpontokat feleltessünk meg, s mindegyikbe irányítsunk a megfelelő átmenettel indexelt élt M -ből. Az eljárást ismételjük meg minden új szögpontra.

Ha az M pontozás biztonságos, akkor az eljárás véges lépésben végetér. Példaként tekintsük a 3.2. ábrát.



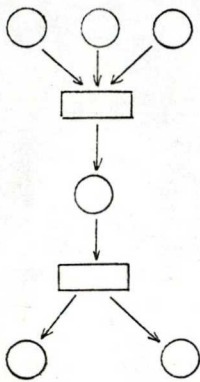
3.2. ábra

Az átmenet gráf alapján az elérhetőség is eldönthető: M_i pontozásból M_j elérhető, ha van olyan irányított út, amely M_i -ből M_j -be vezet.

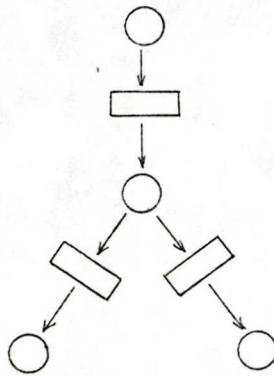
3.2. Petri hálók alkalmazása programokra

Amikor a Petri hálókát modellezésre használjuk, akkor általában a helyeknek feltételeket, az átmeneteknek pedig eseményeket feleltetünk meg. (Az események pillanatszerűek)

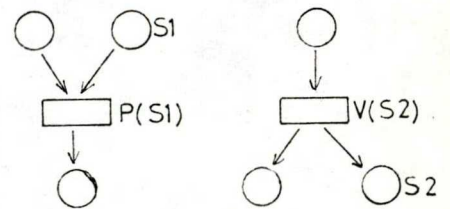
Az olyan folyamatok, amelyek csak induláskor és befejezéskor lépnek kapcsolatba a környezetükkel, méghozzá úgy hogy az indulás feltételektől függ, befejezéskor pedig feltételeket állítanak be, feltétel-esemény szempontjából a 3.3.a. ábra szerint modellezhetők.



3.3.a. ábra



3.3.b. ábra



3.3.c. ábra

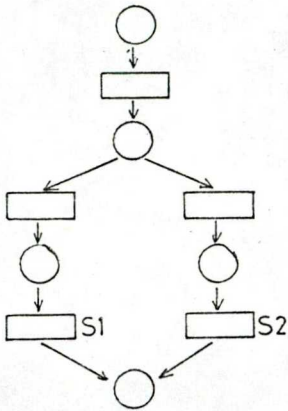
Petri hálókkal csak olyan folyamatokat modellezhetünk, melyekben a feltétel-esemény szerkezet időben nem változik.

Egy program alapján létrehozhatunk egy hálót: feleltessük meg a program változóinak számára történő értékadásnak a 3.3.a. ábrát (egy be- és egy kimeneti feltétellel), a feltételes elágazásnak a 3.3.b. ábrát, míg ha a program tartalmazza a P és V operációkat, akkor azoknak a 3.3.c. ábrát.

További utasítások ábrázolását mutatja a 3.3.d és 3.3.e. ábra.

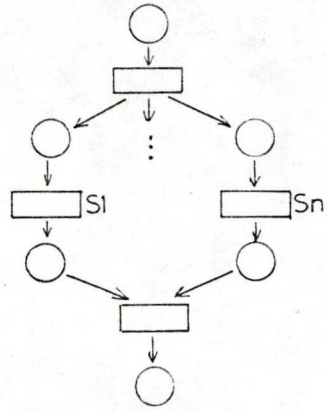
Egy programhoz rendelhető Petri háló a programot alkotó utasításokhoz tartozó Petri hálókból áll elő. Az így kapott háló nem tartalmaz információt a műveletek, predikátumok interpretációjáról, tehát a program olyan tulajdonságai vizsgálhatók, amelyek bármilyen interpretáció mellett igazak.

Az üzenet-buffer probléma 2.5. algoritmus által leírt megoldásához rendelhető Petri háló a 3.4. ábrán látható.



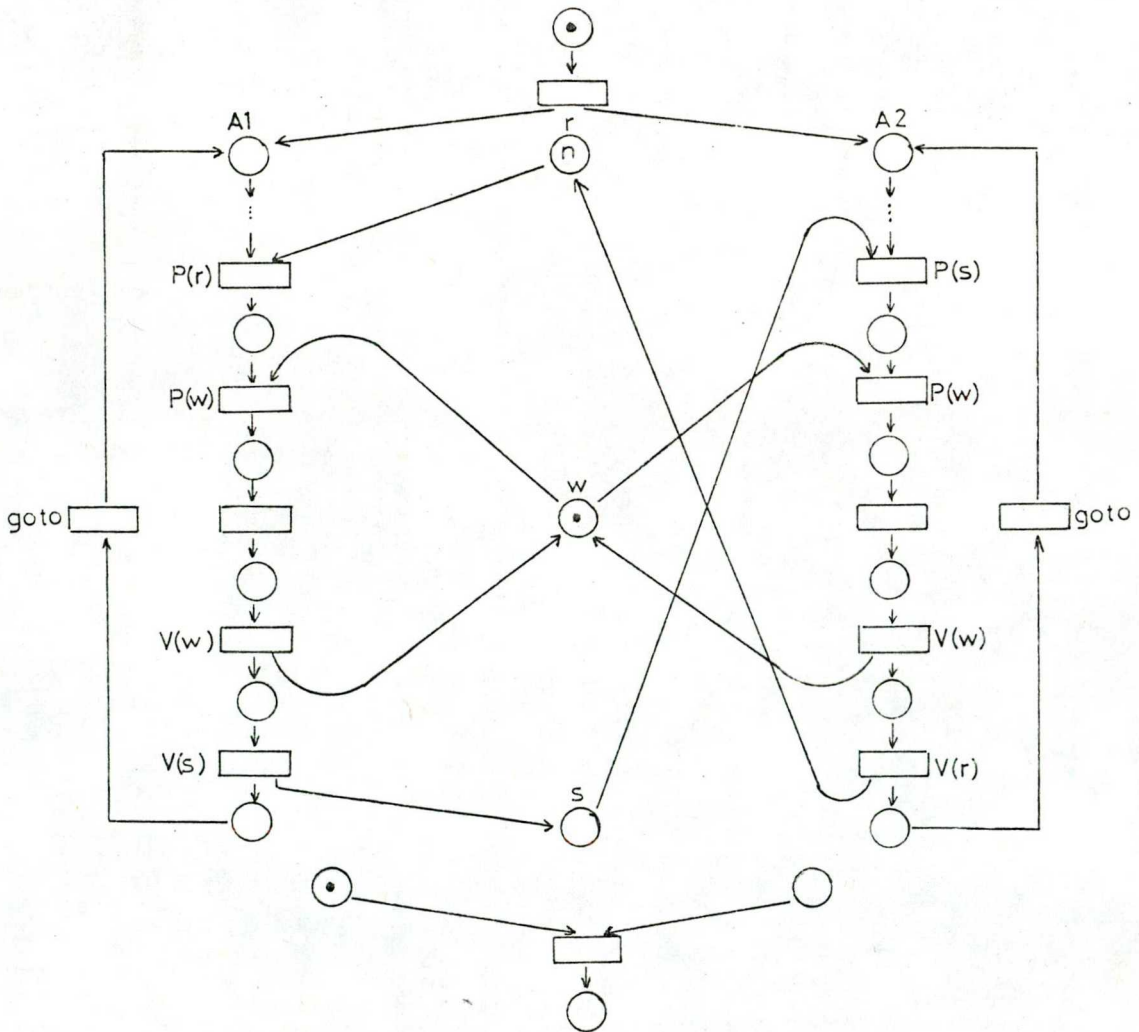
if B then S1 else S2 fi

3.3.d. ábra



cobegin S1; S2; ...; Sn coend

3.3.e. ábra



3.4. ábra

A termelő és fogyasztó oldalt az r, s illetve w helyek kapcsolják össze. Induláskor az r hely n pontot, w hely 1 pontot tartalmaz.

A Petri hálók alkalmazásának előnye, hogy tervezéskor egy szemléletes eszköz áll rendelkezésünkre és bizonyos kérdések a kész programokról algoritmikusan eldönthetőek. Hátrányuk, hogy a program és az azt végrehajtó rendszer számos tulajdonsága a hálók segítségével nem írható le; nem tudunk leírni vele tetszőleges függvényt, ütemezési eljárásokat, időviszonyokat. További hátrány, hogy nagy méretű programok esetén nagy hálókat kapunk (amelyek esetén pl. az átmenet-gráf konstruálása gyakorlatilag szinte kivihetetlen).



4. MAGAS SZINTŰ SZINKRONIZÁCIÓS ESZKÖZÖK MEGVALÓSÍTÁSA SZEMAFOROKKAL

A következőkben a magas szintű szinkronizációs eszközök szemaforral történő megvalósítását és ezen elemi eszközzel való összehasonlítását tárgyaljuk. Ismertetjük az irodalomból ismert algoritmusok általánosításával nyert eredményeinket.

4.1. A conditional critical region és a szemafor viszonya

Dijkstra, E.W. [3]-ban az üzenet-buffer probléma, Courtois, P.J. és szerzőtársai [2]-ben az un. "Readers , writers" feladat szemaforokkal történő megoldását írják le. Hansen, P.B [8] könyvében mindkét problémát critical region-ok és szemaforok, illetve conditional critical region segítségével oldja meg elegáns módon.

Utóbbi két módszert (nevezetesen a critical region-ok és szemaforok ill. a conditional critical region használatán alapulót) összehasonlítva megállapítható, hogy az egymással kapcsolatban álló folyamatok egy bizonyos típusa esetén ideális szinkronizációs eszköz egészen más típusú folyamatok esetén már kevésbé hatékony.

A conditional critical region olyan szituációban alkalmazható eredményesen, ahol egy folyamatot mindaddig várakoztatni kell, amíg a közös adatstruktúra komponensei egy bizonyos feltételt kielégítenek. Hátránya, hogy a szinkronizációs feltételekként használt tetszőleges logikai kifejezések ismételt kiértékelése nagy feldolgozási idővesztést eredményez.

Az egyszerű és a feltételes critical regionok fogalmi egyszerűségét az teszi lehetővé, hogy a várakozó folyamatok régiókba lépésének sorrendjét nem vizsgálják. Ez lazán összekapcsolt, azaz olyan folyamatok esetén, melyek egyazon erőforrást szimultán csak ritkán igényelnek - indokolt.

A szemaforokon végrehajtott P-művelet bizonyos jelzés előfordulásáig történő várakoztatás legdirektebb módja. Gyakran igényelt erőforrások kiosztásakor az operációs rendszernek az igénylő folyamatok sorrendjét is vezérelni kell, ilyenkor célszerű szemaforokat alkalmazni.

Tehát a conditional critical regionokat valóban hatékonyan csak ritkán használt erőforrásokkal kapcsolatban lehet alkalmazni.

Vizsgáljuk most már azt a kérdést, hogyan lehet egy conditional critical region alkalmazásával leírt megoldást szinkronizációs eszközként csak szemaforokat használó programmá fordítani automatikusan. A fordítást két lépésben tárgyaljuk:

- conditional critical region megvalósítása critical region és szemafor segítségével ;
- critical region implementálása szemaforokkal.

A region v when B_i do S_i alaku conditional critical regionok minden B_i ütemezési feltételéhez egy SZ_i szemaforot és egy Q_i jelzõt kell társítani (mindkettõt nulla kezdõértékkel) és a következõ programot kell generálni:

```

region  v do
  if     $B_i$  then       $S_i ; V(SZ_i)$ 
        else  $Q_i := Q_i + 1$  fi ;
P( $SZ_i$ )

```

Emellett minden más folyamat, amely a B_i feltételt igazzá teheti, saját

```

region  v do  $R_i$ 

```

alaku kritikus szakaszában a késleltetett folyamatokat a következõ formában aktivizálja:

```

region  v do
  begin  $R_i ;$ 
    if  $B_i \ \& \ (Q_i > 0)$  then
       $Q_i := Q_i - 1 ;$ 
       $S_i ;$ 
       $V(SZ_i)$  fi ;
  end

```

A region v do R alakú kritikus szakaszok szemaforral történő implementálásához rendeljük a v közös adatstruktúrához egy w szemafort, s ekkor a konstrukció:

$$P(w); R; V(w);$$

Ha olyan párhuzamos folyamataink vannak, melyekben n féle ütemezési feltétellel fordul elő, s a B_i ütemezési feltételtől l_i db folyamat függ (természetesen az l_i db folyamat kritikus szakaszának azonosnak kell lenni, a folyamatok száma $\sum_{i=1}^n l_i$), akkor n db nulla kezdőértékű szemaforra ($SZ[n]$) és n db ugyancsak nulla kezdőértékű jelzőre ($Q[n]$) lesz szükség. Mindegyik feltételes kritikus szakasz lefordítása egy olyan programrészt eredményez, melyben a saját tevékenységének leírásán túl további $n-1$ feltétel vizsgálat és a hozzá tartozó tevékenység végrehajtása is szerepel.

A

region v when B_i do S_i illetve region v when B_k do S_k

conditional critical regionok esetén a 4.1. algoritmus által tartalmazott programot kell generálni.

```
P(w);
if Bi then Si; V(SZ[i])
    else Q[i]:= Q[i]+1 fi;
if Bk & (Q[k]>0) then
    Q[k]:= Q[k]-1;
    Sk; V(SZ[k]) fi;
(× további if utasítások, ×)
(× k=1,...,i-1,i+1,...,n ×)
V(w);
P(SZ[i]);
```

```
P(w);
if Bk then Sk; V(SZ[k])
    else Q[k]:= Q[k]+1 fi;
if Bi & (Q[i]>0) then
    Q[i]:= Q[i]-1;
    Si; V(SZ[i]) fi;
(× további if utasítások ×)
(× i=1,...,k-1,k+1,...,n ×)
V(w);
P(SZ[k]);
```

4.1. algoritmus

A 2.8 algoritmus ez alapján könnyen átirható.

(A következő helyettesítésekre van szükség:

$$i \equiv 1, \quad k \equiv 2, \quad n \equiv 2$$

$$B_i \equiv \text{full} < \text{max}, \quad B_k \equiv \text{full} > 0$$

$$S_i \equiv \text{put}(m, s); \text{full} := \text{full} + 1;$$

$$S_k \equiv \text{get}(m', s); \text{full} := \text{full} - 1;$$

ahol $\text{put}(m, s) \stackrel{\text{def}}{=} \text{buffer}(p) := m; p := (p+1) \bmod \text{max}$

$\text{get}(m', s) \stackrel{\text{def}}{=} m' := \text{buffer}(c); c := (c+1) \bmod \text{max}$

tehát p és c további indexként használt változók, de ezek kifejtésétől a későbbiekben is eltekintünk.)

4.2. A monitor implementálása szemaforokkal

A monitort használva szinkronizációs eszközként folyamatok felfüggesztésére kritikus szakaszok belsejében is sor kerülhet, ilyen esetben a kölcsönös kizárást ideiglenesen fel kell oldani. A probléma megoldását szemaforok alkalmazásával a következő konstrukció szolgáltatja.

Legyen M 1 kezdőértékű, U pedig 0 kezdőértékű szemafor, mindkettő lokális az adott monitorra nézve. Az M biztosítja a monitor lokális eljárásai törzseinek kölcsönösen kizárt végrehajtását. Egy lokális eljárásba lépéskor $P(M)$, míg az eljárás elhagyásakor $V(M)$ hajtandó végre. Amikor egy A folyamat kiad egy olyan feltételre vonatkozó signal-t, melyre egy B folyamat vár, A -nak várakoznia kell mindaddig, amíg a továbbengedett B folyamattól engedélyt nem kap a továbbhaladásra. A signal-t kiadó folyamatok (esetünkben A) magukat a $P(U)$ művelet kiadásával függesztik fel. Egy eljárásból történő kilépés előtt meg kell vizsgálni, vajon valamely más folyamat várakozik-e U -n, s ha igen, akkor a $V(U)$ utasítással lehetővé kell tenni, hogy az folytathassa munkáját. Ezért az U -ra várakozó folyamatok számát a 0 kezdőértékű UC egész változóban őrizzük.

Rendeljünk még minden feltétel változóhoz egy FSZ szemafort és egy FC számlálót, mindkettőt 0 kezdőértékkel.

Alkalmazzuk ezek után a következő algoritmusokat:

Minden monitor eljárás bemenetére $P(M)$, kimenetére

if $UC > 0$ then $V(U)$ else $V(M)$,

a "feltétel.wait" operációk helyére

$FC := FC + 1$;

if $UC > 0$ then $V(U)$ else $V(M)$ fi ;

$P(FSZ)$;

$FC := FC - 1$;

míg a "feltétel.signal" operációk helyére

$UC := UC + 1$;

if $FC > 0$ then $V(FSZ)$; $P(U)$ fi ;

$UC := UC - 1$;

Az

if B_i then felt_i.wait fi;

S_i ;

felt_k.signal

alakú monitor eljárás esetében a következő programot kell generálni:

$P(M)$;

if B_i then $FC[i] := FC[i] + 1$;

if $UC > 0$ then $V(U)$ else $V(M)$ fi ;

$P(FSZ[i])$;

$FC[i] := FC[i] - 1$ fi ;

S_i ;

$UC := UC + 1$;

if $FC[k] > 0$ then $V(FSZ[k])$; $P(U)$ fi ;

$UC := UC - 1$;

if $UC > 0$ then $V(U)$ else $V(M)$ fi ;

4.2. algoritmus

A 2.9. algoritmus első eljárásának ez alapján történő átírásra a következő meg-

feleltetésekkel igényli: $i \equiv 1$, $k \equiv 2$, $B_i \equiv c \equiv n$, $(FC[1] \text{ és } FSZ[1]) \equiv nf$,

$(FC[2] \text{ és } FSZ[2]) \equiv ne$.

4.3. Path kifejezés implementálása szemaforokkal

Vizsgáljuk most path kifejezések P és V műveletekre történő automatikus fordítását.

Egy path kifejezést általában két generált szinkronizációs művelet O_L és O_R közé zárunk. Az O_L művelet P vagy PP művelet, O_R pedig V vagy VV művelet. A PP és VV művelet a fordítás algoritmusának leegyszerűsítésére szolgál a szimultán végrehajtás implementációjában.

Mindkét művelet egy C1 számlálót és egy S3 szemaforot megosztva használ.

Az S3 szemaforral oldjuk meg annak kizárását, hogy a számlálót egynél több folyamat változtassa egy időben. A PP művelet növeli a VV csökkenti a számlálót.

```
procedure PP(C1:INT; S3, Si:SEM);  
begin    P(S3);  
        C1 := C1+1;  
        if C1=1 then P(Si) fi;  
        V(S3);  
  
end;
```

```
procedure VV(C1:INT; S3, Si:SEM);  
begin    P(S3);  
        C1 := C1-1;  
        if C1 = 0 then V(Si) fi;  
        V(S3);  
  
end;
```

4.3. algoritmus

A fordítás algoritmus az alábbi:

1. lépés: Választunk egy S1 szemafor, 1-re inicializáljuk és a
path <path kifejezés> end

path kifejezést a következőképp helyettesítjük:

P(S1) <path kifejezés> V(S1)

Az algoritmus 2. lépését a <path kifejezés> -re végrehajtva készen leszünk.

2. lépés: Kiértékeljük a <path kifejezés> -t és az általa képviselt szinkronizációs sémának megfelelően a következők egyikét hajtjuk végre:

2.1. Szorozat: a $\langle \text{path kifejezés} \rangle$ alakja: $\langle \text{path-kif-1} \rangle$; $\langle \text{path-kif-2} \rangle$

Ekkor választunk egy S_2 szemafort, nullára inicializáljuk és a $\langle \text{path kifejezés} \rangle$ helyére

$\langle \text{path-kif-1} \rangle \quad V(S_2) \quad P(S_2) \quad \langle \text{path-kif-2} \rangle$

kerül. A $\langle \text{path-kif-1} \rangle$ ill. $\langle \text{path-kif-2} \rangle$ -re a 2.lépést ismételten alkalmazva ér véget az algoritmus.

2.2. Választás: a $\langle \text{path kifejezés} \rangle$ alakja: $\langle \text{path-kif-1} \rangle$, $\langle \text{path-kif-2} \rangle$

Ekkor $O_L \langle \text{path-kif-1} \rangle$, $\langle \text{path-kif-2} \rangle O_R$ helyére

$O_L \langle \text{path-kif-1} \rangle \quad O_R \quad O_L \langle \text{path-kif-2} \rangle \quad O_R$ kerül, majd a $\langle \text{path-kif-i} \rangle$ -re ismételten alkalmazzuk a 2.lépést.

2.3. Szimultán végrehajtás: a $\langle \text{path kifejezés} \rangle$ alakja: $\{\langle \text{path-kif} \rangle\}$

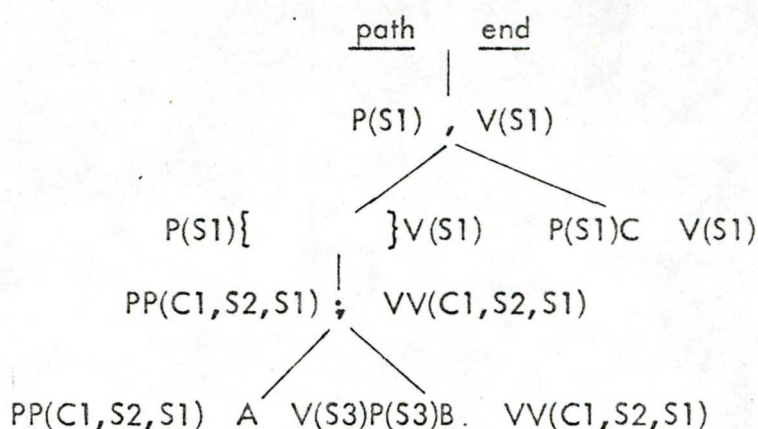
Válasszunk egy C_1 számlálót, inicializáljuk nullára, tovább egy S_3 szemafort, melyet inicializáljunk 1 értékre. A

$P(S_i) \quad \{\langle \text{path-kif} \rangle\} \quad V(S_i)$ helyére
 $PP(C_1, S_3, S_i) \quad \langle \text{path-kif} \rangle \quad VV(C_1, S_3, S_i)$ kerül, majd a $\langle \text{path-kif} \rangle$ -re ismét a 2. lépést alkalmazzuk.

2.4. Eljárás név: a path kifejezés a szinkronizálendő eljárások egyikének neve, ez tovább már nem redukálható, a 2. lépés véget ér.

Példaként megmutatjuk a path ($\{A; B\}$), C end path kifejezés transzformációját P és V műveletekre, a transzformációt fával reprezentálva. Az algoritmus minden lépése a fában egy szögpontnak felel meg, és a lépés során generált szinkronizációs műveletek a megfelelő szögpontok mellé kerültek felírásra.





Az algoritmus eredménye program formában az alábbi:

```

⋮
S1, S2, S3 : SEM;
C1 : INT;
⋮
S1 := 1; S2 := 1; S3 := 0; C1 := 0;
⋮
procedure A = begin PP(C1, S2, S1); <A törzse>; V(S3) end;
procedure B = begin P(S3) ; <B törzse>; VV(C1, S2, S1) end;
procedure C = begin P(S1) ; <C törzse>; V(S1) end
  
```

A path kifejezések eljárásneveket tartalmaznak, de a P és V műveleteket magukba az eljárásokba kell beírni. Az A, B és C eljárások most már a path kifejezés precíz implementációját is adják.

A 2.10 algoritmusban szereplő

path write; read end

path kifejezés implementációja a két eljárás módosítását is jelenti:

```

⋮
S1, S2 : SEM;
⋮
S1 := 1; S2 := 0;
⋮
procedure WRITE (m:T) = begin P(S1); <write törzse>; V(S2) end;
procedure READ (m:T) = begin P(S2); <read törzse>; V(S1) end
  
```

(Megjegyezzük, hogy a $P(S_i)$ műveletet tulajdonképpen már az eljárás hívásakor, a paraméter átadás előtt végre kellene hajtani. Számunkra azonban - mint azt a 6. fejezetben látni fogjuk - az implementáció itt vázolt módja is megfelelő.)

5. FORMÁLIS LEÍRÁS

Ez a fejezet tartalmazza a legfontosabb saját eredményeinket: megmutatjuk, hogy a VDL alkalmassá tehető párhuzamos folyamatok formális leírására is. Ehhez egyrészt definiálunk egy párhuzamos utasításokat is tartalmazó egyszerű programozási nyelvet, melyet a továbbiakban EPPL-nek (Elementary Parallel Programming Language) nevezünk, másrészt bővítjük a VDL eszköztárát. A fejezet végén megadjuk az EPPL formális leírását a bővített VDL segítségével.

Mindezek előtt azonban ismertetjük [15], [17] és [18] alapján azt a módszert, amelyet a továbbiakban használni fogunk.

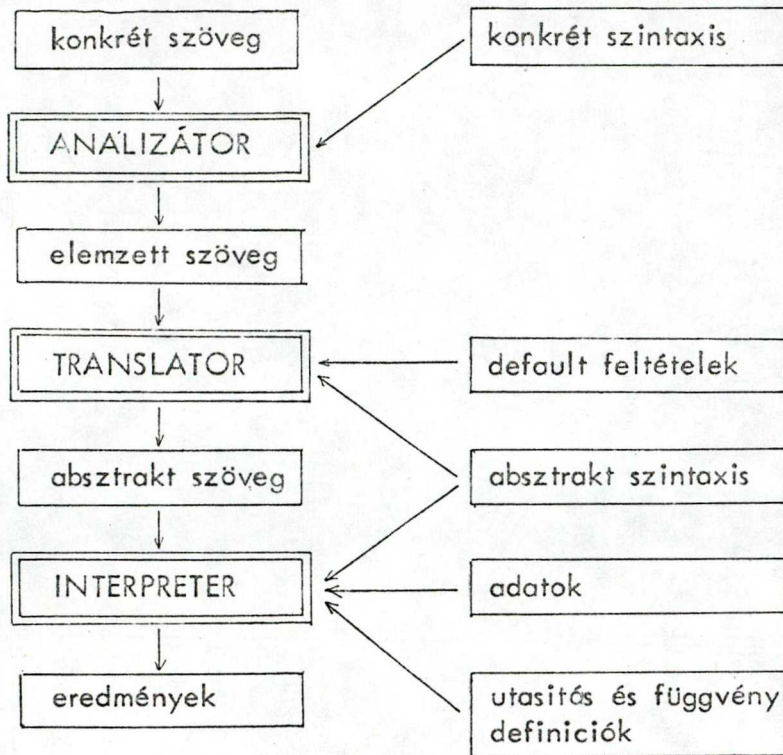
A módszer alapja egy absztrakt interpreter, amely a konkrét (jelsorozat-ként adott) programból a lexikális és szintaktikus analízis végrehajtása után nyert ún. absztrakt program interpretációjával definiálja a jelentést.

Három gépet definiálunk az adathalmazainkkal együtt, melyek a nyelv definíció alapját képező 3 tevékenységet végrehajtják:

ANALIZÁTOR	(lexikális analízist hajtja végre)
TRANSLATOR	(szintaktikus analízist hajtja végre)
INTERPRETER	

A nyelv definíciós menetét az 5.1. ábrán tüntettük fel.

A konkrét program (konkrét szöveg) a program értelmezése szempontjából felesleges lexikális egységeket is tartalmaz (elválasztó jelek, zárójelek stb.). A konkrét szintaxis a nyelvet alkotó szövegeknek a halmazát definiálja.



5.1. ábra

Az ANALIZÁTOR a programozó által írt program kiértékelését végzi, vizsgálva, hogy az kielégíti-e a konkrét szintaxist, és egy transzformált szöveget hoz létre. Egy program szövegre konkrét szöveggént hivatkozunk:

$$\text{is-konkrét-szöveg} = \text{is-karakter-list}$$

ahol is-karakter-list a nyelvbeli karakterek és utasítás határolók komplett halmaza. Az ANALIZÁTOR által generált szöveg az elemzett szöveg, melyet formája alapján elemzési fának is neveznek.

A programozási nyelv absztrakt szintaxisa a nyelvet alkotó programoknak az értelmezés szempontjából lényeges szerkezetét definiálja. Az absztrakt szintaxist használva absztrakt programokat (absztrakt szöveg) definiálunk, melyek a programok csak az interpretációjuk szempontjából lényeges szerkezetét tartalmazzák.

E tevékenységet végzi el a TRANSLATOR, mely egy olyan absztrakt

szöveget állít elő, melyben az összes implicit nyelvi jellemző explicite le van írva, viszont már nem tartalmazza az értelmezés szempontjából felesleges információkat.

Az absztrakt szöveg már alkalmas arra, hogy inputja legyen egy INTERPRETER-nek, mely a nyelv szemantikáját definiálja.

A továbbiakban először az EPPL konkrét szintaxisának leírását adjuk meg. Utána az EPPL programok absztrakt formára hozását tekintjük át, részletesen csak az absztrakt szintaxisukat ismertetve. Ezt követi az absztrakt interpreter követelményeinek kifejlesztése, mely biztosítani tudja párhuzamos programozási nyelvek formális leírását. A fejezet utolsó része tartalmazza az EPPL interpreter részletes leírását.

5.1. Az EPPL konkrét szintaxisa

Definiáljuk az EPPL nyelvet. Konkrét szintaxisának formális leírását egy szintaktikus metanyelven, a BNF-en adjuk meg (lásd 5.1. táblázat).

Egy program deklarációs részből és utasításrészből áll. Minden a programban használt változót deklarálni kell. A deklarációk a program elején állnak, a <párhuzamos-folyamat> -okban lokális változók használata nem megengedett. Egész (INT), logikai (BOOL) és szemafor (SEM) típusu változókat, ilyen típusu elemekből felépített tömböket (pl. ARRAY [n] OF INT) valamint INT és BOOL elemtípusu puffereket (pl. BUFFER [n] OF INT) használunk. A puffer LIFO szerkezetű. Tömb és puffer mérete csak egész konstans, struktúra változó indexeként használt változó csak egész típusu lehet.

Példa értékadó utasításokra:

$M := 1 ; Q[M] := 4 ; Q[2] := Q[M] ; Q[3] := Q[2],$

ahol Q egy egész típusu tömb, M pedig egy egész típusu egyszerű változó.
A $Q := 5$ tömbértékkadás jelentése: Q minden eleme értékül 5-öt vesz fel.



[K1]	⟨program⟩	::=	PROGRAM ⟨deklarációs rész⟩; ⟨utasítás rész⟩ END.
[K2]	⟨deklarációs rész⟩	::=	⟨vált-dekl⟩ {; ⟨vált-dekl⟩}
[K3]	⟨vált-dekl⟩	::=	⟨azonosító-lista⟩ : ⟨típus⟩
[K4]	⟨azonosító-lista⟩	::=	⟨azonosító⟩ {, ⟨azonosító⟩}
[K5]	⟨típus⟩	::=	⟨egyszerű típus⟩ ⟨strukturált típus⟩
[K6]	⟨egyszerű típus⟩	::=	⟨skalár típus⟩ ⟨szemafor típus⟩
[K7]	⟨skalár típus⟩	::=	⟨egész típus⟩ ⟨logikai típus⟩
[K8]	⟨strukturált típus⟩	::=	⟨tömb típus⟩ ⟨puffer típus⟩
[K9]	⟨tömb típus⟩	::=	ARRAY [⟨méret⟩] OF ⟨t-komponens típus⟩
[K10]	⟨méret⟩	::=	⟨egész konstans⟩
[K11]	⟨t-komponens típus⟩	::=	⟨egyszerű típus⟩
[K12]	⟨puffer típus⟩	::=	BUFFER [⟨méret⟩] OF ⟨p-komponens típus⟩
[K13]	⟨p-komponens típus⟩	::=	⟨skalár típus⟩
[K14]	⟨utasítás rész⟩	::=	⟨utasítás⟩ {; ⟨utasítás⟩}
[K15]	⟨utasítás⟩	::=	⟨cimke⟩ : ⟨cimkétlen-utasítás⟩ ⟨cimkétlen-utasítás⟩
[K16]	⟨cimkétlen utasítás⟩	::=	⟨értékadó-ut⟩ ⟨goto-ut⟩ ⟨input-ut⟩ ⟨output-ut⟩ ⟨párhuzamos-ut⟩
[K17]	⟨értékadó-ut⟩	::=	⟨változó⟩ := ⟨kifejezés⟩
[K18]	⟨goto-ut⟩	::=	GOTO ⟨cimke⟩
[K19]	⟨input-ut⟩	::=	INPUT ⟨azonosító⟩
[K20]	⟨output-ut⟩	::=	OUTPUT ⟨kifejezés⟩
[K21]	⟨párhuzamos-ut⟩	::=	COBEGIN ⟨párhuzamos-folyamat⟩ {; ⟨párhuzamos folyamat⟩} COEND
[K22]	⟨párhuzamos-folyamat⟩	::=	BEGIN ⟨párh-ut⟩ {; ⟨párh-ut⟩} END
[K23]	⟨párh-ut⟩	::=	⟨cimke⟩ : ⟨p-cimkétlen-ut⟩ ⟨p-cimkétlen-ut⟩
[K24]	⟨p-cimkétlen-ut⟩	::=	⟨értékadó-ut⟩ ⟨goto-ut⟩ ⟨feltételes-ut⟩ ⟨input-ut⟩ ⟨output-ut⟩ ⟨P-ut⟩ ⟨V-ut⟩
[K25]	⟨feltételes-ut⟩	::=	IF ⟨kifejezés⟩ THEN ⟨p-cimkétlen-ut-lista⟩ FI IF ⟨kifejezés⟩ THEN ⟨p-cimkétlen-ut-lista⟩ ELSE ⟨p-cimkétlen-ut-lista⟩ FI
[K26]	⟨p-cimkétlen-ut-lista⟩	::=	⟨p-cimkétlen-ut⟩ {; ⟨p-cimkétlen-ut⟩}

[K27]	⟨P-ut⟩	::=	P (⟨változó⟩)
[K28]	⟨V-ut⟩	::=	V (⟨változó⟩)
[K29]	⟨kifejezés⟩	::=	⟨konstans⟩ ⟨változó⟩ ⟨bináris⟩ ⟨unáris⟩
[K30]	⟨bináris⟩	::=	(⟨kifejezés⟩ ⟨bináris operátor⟩ ⟨kifejezés⟩)
[K31]	⟨bináris operátor⟩	::=	⟨aritmetikai bin opr⟩ ⟨logikai bin opr⟩ ⟨reláció bin opr⟩
[K32]	⟨unáris⟩	::=	⟨unáris operátor⟩ ⟨kifejezés⟩
[K33]	⟨változó⟩	::=	⟨egyszerű változó⟩ ⟨indexes változó⟩
[K34]	⟨egyszerű változó⟩	::=	⟨azonosító⟩
[K35]	⟨indexes változó⟩	::=	⟨struktura változó⟩ [⟨elemi kifejezés⟩]
[K36]	⟨elemi kifejezés⟩	::=	⟨egyszerű változó⟩ ⟨egész konstans⟩
[K37]	⟨struktura változó⟩	::=	⟨azonosító⟩
[K38]	⟨konstans⟩	::=	⟨egész konstans⟩ ⟨logikai konstans⟩
[K39]	⟨egész típus⟩	::=	INT
[K40]	⟨szemafor típus⟩	::=	SEM
[K41]	⟨logikai típus⟩	::=	BOOL
[K42]	⟨aritmetikai bin.opr⟩	::=	+ -
[K43]	⟨logikai bin.opr⟩	::=	& v
[K44]	⟨reláció bin opr⟩	::=	> >= = < <= <>
[K45]	⟨unáris operátor⟩	::=	~ + -
[K46]	⟨egész konstans⟩	::=	⟨szám⟩
[K47]	⟨logikai konstans⟩	::=	TRUE FALSE
[K48]	⟨cimke⟩	::=	⟨szám⟩
[K49]	⟨szám⟩	::=	⟨számjegy⟩ {⟨számjegy⟩}
[K50]	⟨számjegy⟩	::=	0 1 2 ... 9
[K51]	⟨azonosító⟩	::=	⟨betű⟩ {⟨betű⟩}
[K52]	⟨betű⟩	::=	A B ... Z

5.1. táblázat (folytatás)

Az utasítások általában címkézettek lehetnek, kivételt csak a feltételes utasítás THEN illetve ELSE ágát alkotó utasítás lista elemei képeznek, melyeknek nem lehet címkéje. Ezen utasítás listákból így vezérlésátadó utasítással (GOTO) csak kiugrani lehet, bennük feltételes utasítás pedig csak olyan fordulhat elő, mely THEN illetve ELSE ágában további feltételes utasítást már nem tartalmaz. A feltételes utasításban szereplő kifejezés értéke logikai típusu kell hogy legyen.

Mint azt a 4. fejezetben megmutattuk, a magas szintű szinkronizációs eszközök megvalósíthatóak szemaforokkal. Így szinkronizációs eszközként választhatjuk a szemaforokat és a rajtuk értelmezett P és V műveleteket.

A <P-ut> és <V-ut> utasítások a korábbiakban bevezetett P illetve V műveleteket valósítják meg.

SEM típusu változó vagy tömbelem értékét csak a <P-ut> és a <V-ut> utasításokkal, továbbá olyan értékadó utasítással lehet változtatni, mely nem része egyetlen <párhuzamos-folyamat> -nak sem. A <P-ut> és <V-ut> utasításokban csak SEM típusu változóra vagy tömbelemre történhet hivatkozás. Példa <P-ut> -ra : P(SZ [M]), ahol SZ egy 2 elemű szemafor tömb, M pedig egy egész típusu egyszerű változó.

Puffer művelet csak <párhuzamos-folyamat> -beli értékadásban fordulhat elő. Input utasítással csak egyszerű változónak adható érték. A címke egy egész szám lehet, míg az azonosítók csak betűkből épülhetnek fel. Megköveteljük, hogy különböző <párhuzamos-folyamat> -okban különböző címkék szerepeljenek.

A PROGRAM, END, ARRAY, BUFFER, OF, GOTO, INPUT, OUTPUT, COBEGIN, COEND, BEGIN, IF, THEN, ELSE, FI, P és V az EPPL kulcsszavai. A lexikális egységeket legalább egy betűközzel kell elválasztani egymástól. A "(x" és "x)" jelek közé helyezett szöveg kommentárnak számít, csak a példa programjainkban használjuk, a nyelvnek nem elemei.

Két példát adunk EPPL programokra:

1. EPPL példa: két párhuzamos folyamat tetszőleges sorrendben kerülhet végrehajtásra.

A program végrehajtása után X értéke 3. Mindegyik folyamat csak egyszer hajtódik végre.

PROGRAM

SZ : SEM;

X : INT;

SZ := 1;

X := 0;

COBEGIN

BEGIN P(SZ);

X := X+1;

V(SZ)

END;

BEGIN P(SZ);

X := X+2;

V(SZ)

END;

COEND;

END.

2. EPPL példa: két párhuzamos ciklikus folyamat dolgozik.

Ha FELT a 0 értéket veszi fel, akkor a folyamatok csak ciklusuk végéig dolgoznak, ismételt végrehajtásukra már nem kerül sor.

Valamely folyamat (pl.1.) utasításai között lehet INPUT utasítás, mely beolvas egy értéket, s ha az 0, akkor leállnak a párhuzamos folyamatok.

PROGRAM

SZ : SEM;

FELT, SZAML : INT;

SZ := 1;

FELT := 1; SZAML := 0;

COBEGIN

BEGIN

1 : P(SZ);

(× 1.kritikus szakasz ×)

IF SZAML=100 THEN INPUT FELT; SZAML := 0 FI;

V(SZ);

(× az 1. ciklus további utasításai ×)

IF FELT <> 0 THEN GOTO 1 FI;

END;

BEGIN

2 : P(SZ);

(× 2. kritikus szakasz ×)

V(SZ);

(× a 2. ciklus további utasításai ×)

IF FELT <> 0 THEN GOTO 2 FI;

END;

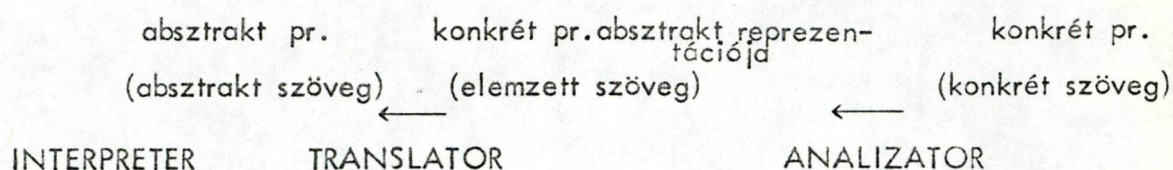
COEND;

END.

(Megjegyezzük, hogy a dolgozat egyéb helyein - az áttekinthetőség érdekében kisbetűket is használunk, s a kulcsszavakat aláhuzzuk.

5.2. Konkrét programok fordítása absztrakt programmá

Mint a korábbiakban már vázoltuk, a nyelvdefiniálás elvileg az alábbi séma szerint történhet:



A konkrét programok absztrakt programmá fordítása a parse és a translate függvények segítségével két lépésben hajtható végre. Ha txt egy konkrét program, akkor a megfelelő absztrakt programot a következő függvény definiálja:

$\text{translate} \circ \text{parse} (\text{text})$

A parse függvény eredménye és egyben a translate függvény argumentuma egy t strukturált objektum, melyet txt absztrakt reprezentációjának nevezünk és ez nem más, mint txt elemzési fája.

5.2.1. Konkrét programok absztrakt reprezentációja

Definiálhatnánk egy elemzőt, mely minden konkrét program esetén előállítaná annak elemzési fáját. Ennél egyszerűbb módszert követünk.

Először az absztrakt reprezentációkat jellemző is-c-program predikátumot vezetjük be, tisztázzuk mit értünk konkrét program absztrakt reprezentációján. Ezután definiáljuk a generate függvényt, mely az absztrakt reprezentációkat konkrét programokká képezi le. Végül a parse függvény implicit definícióját adjuk meg, mint a generate inverzét.

Egy konkrét program t absztrakt reprezentációja egy objektum, mely kielégíti az is-c-program predikátumot:

$\text{is-c-program} = (\langle s_1 : \text{is-PROGRAM} \rangle , \langle s_2 : \text{is-c-dcl-rész} \rangle , \langle s_3 : \text{is-} \rangle , \\ \langle s_4 : \text{is-c-ut-rész} \rangle , \langle s_5 : \text{is-END} \rangle , \langle s_6 : \text{is-} \rangle)$

Strukturája a konkrét program szintaktikus strukturáját tükrözi, elemi komponensei a konkrét programot alkotó karakter értékek. Az is-c-program predikátum predikátum-halmazzal van definiálva, melynek elemeit a konkrét szintaxis szerkesztési szabályainak újrainásával definiálhatjuk. A vázolt eljárások leírása terjedelme miatt elhagyjuk.



5.2.2. Az EPPL absztrakt szintaxisa

Egy program absztrakt szerkezetét kézenfekvő egy VDL objektummal ábrázolni. Az elemi objektumokat a programnak azok a szimbólumai képezhetik, amelyek a program értelmezése szempontjából szerepet játszanak. Az összetett objektumok a terminális szimbólumokból felépülő szintaktikai egységeknek feleltethetők meg. A szelektorok ezeknek a szintaktikus egységeknek a kiválasztására szolgálnak. Az EPPL absztrakt szintaxisát az 5.2. táblázat tartalmazza, míg utasítások absztrakt szintaxisára példákat az 5.5. ábrán találhatunk.

5.2.3. A translátor

Az absztrakt program előállítását a megfelelő konkrét program absztrakt reprezentációjából egy algoritmussal, a translátorral definiáljuk. A translátor függvények halmazával definiált, melyek között rekurzív függvények is előfordulnak, a legfelső szinten a `translate (t)` függvény áll.

A `translate (t)` függvény az is-c-program predikátumot kielégítő t objektumot egy is-párh-program predikátumot kielégítő objektumba képezi le:

$$\begin{aligned} \text{translate (t)} &= \\ \text{is-c-program(t)} &\rightarrow \mu_c \{ \langle \text{s-dekl-rész} : \text{trans-dekl-list (s}_2\text{(t))} \rangle, \\ &\quad \langle \text{s-ut-rész} : \text{trans-utlist (s}_4\text{(t))} \rangle \} \\ \text{TRUE} &\longrightarrow \text{error} \\ \text{ahol is-c-dcl-rész (s}_2\text{(t)) és is-c-ut-rész (s}_4\text{(t))} \\ &\quad \vdots \end{aligned}$$

E leképezés olyan objektumot hoz létre, melynek mnemonikus nevű szelektorai vannak. A translátor a többszörös deklarációkat kiszűrheti.

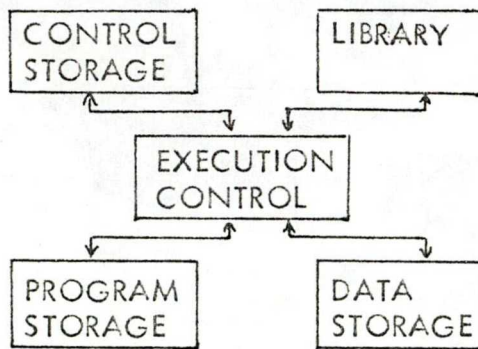
Definíciójának részletes leírását terjedelme miatt elhagyjuk.

[A23]	is-V-ut	=	(< s-v : is-V > , < s-par : is-változó >)
[A24]	is-kifejezés	=	is-konstans $\vee$ is-változó $\vee$ is-bináris $\vee$ is-unáris
[A25]	is-bináris	=	(< s-bal-oldal: is-kifejezés > , < s-bin-opr: is-bináris-operátor > , < s-jobb-oldal : is-kifejezés >)
[A26]	is-bináris-operátor	=	is-aritm-bin-opr $\vee$ is-log-bin-opr $\vee$ is-reláció-bin-opr
[A27]	is-unáris	=	(< s-un-opr : is-unáris-operátor > , < s-operandus: is-kifejezés >)
[A28]	is-változó	=	is-egyszerű-változó $\vee$ is-indexes-változó
[A29]	is-egyszerű-változó	=	is-azonosító
[A30]	is-indexes-változó	=	(< s-tv : is-tömb-vált > , < s-index: is-elemi-kifejezés >)
[A31]	is-elemi-kifejezés	=	is-egyszerű-változó $\vee$ is-egész-konst
[A32]	is-tömb-vált	=	is-azonosító
[A33]	is-konstans	=	is-egész-konst $\vee$ is-logikai-konst
[A34]	is-egész-típus	=	is-INT
[A35]	is-szemafor-típus	=	is-SEM
[A36]	is-logikai-típus	=	is-BOOL
[A37]	is-aritm-bin-opr	=	is-ADD $\vee$ is-SUBTR
[A38]	is-log-bin-opr	=	is-AND $\vee$ is-OR
[A39]	is-reláció-bin-opr	=	is-GT $\vee$ is-GE $\vee$ is-EQ $\vee$ is-LE $\vee$ is-LT $\vee$ is-NE
[A40]	is-unáris-operátor	=	is-NOT $\vee$ is-PLUS $\vee$ is-MINUS
[A41]	is-egész-konst	=	is-szám
[A42]	is-logikai-konst	=	is-TRUE $\vee$ is-FALSE
[A43]	is-címke	=	is-szám
[A44]	is-szám	=	is-számjegy-lista
[A45]	is-számjegy	=	{ 0, 1, 2, ..., 9 }
[A46]	is-azonosító	=	is-betű-lista
[A47]	is-betű	=	{ A, B, ..., Z }
[A48]	is-párh	=	{ PARH }
[A49]	is-P	=	{ P }
[A50]	is-V	=	{ V }
[A51]	is- < >	=	{ NIL }

5.3. Az absztrakt interpreter

Mint a fejezet elején vázoltuk, a nyelv definiálás alapja egy absztrakt interpreter. A konvencionális interpreterek ismertetése után definiálunk egy absztrakt interpretert. Megmutatjuk, hogy a VDL-eszköztárának bővítésével, nevezetesen a megvárakoztató makróutasítás bevezetésével - alkalmassá tehető párhuzamos folyamatok formális leírására is.

A programozási nyelvek szemantikájának definiálására használt absztrakt interpreterek nagyon hasonlóak a konvencionális interpreterekhez, utóbbiak struktúrája [19] alapján az 5.2. ábrán tekinthető át.



5.2. ábra

EXECUTION CONTROL: a konvencionális interpreter vezérlője, az interpreter akcióit irányítja. Egy gépi nyelvű program reprezentálja.

CONTROL STORAGE: az értelmezéshez használt utasításokat tartalmazza. Az utasítás sorozatok rendszerint az értelmezés alatt makró utasítások kifejtéseként vagy utasítások módosításával kerülnek kialakításra.

PROGRAM STORAGE: az értelmezendő programot tartalmazza absztrakt formában.

DATA STORAGE: a program adatait és a végrehajtást befolyásoló közbűlő adatokat tartalmazza.

LIBRARY: a makródefiníciókat tartalmazza.

Definiálunk most egy absztrakt gépet, melynek komponensei fák.

Az absztrakt gép állapotai VDL objektumok, így a gép állapotainak halmaza $G = \{ \xi \mid \text{is-állapot}(\xi) \}$. Az absztrakt gép egy $(\xi_0, \text{is-állapot})$ kettes, ahol ξ_0 a gép kezdőállapota, az is-állapot pedig a gép állapotainak, mint VDL objektumoknak a halmazát meghatározó predikátum. Természetesen $\text{is-állapot}(\xi_0) = \text{IGAZ}$.

Egy program végrehajtása az absztrakt gépen a

$$\xi_i \rightarrow \xi_{i+1} \quad \xi_i \in G, \quad i = 0, 1, \dots, n$$

leképezéssorozatot jelenti. Egy $\xi_i \rightarrow \xi_{i+1}$ állapotváltozást az absztrakt gépen egy utasítás végrehajtása eredményez.

Az absztrakt gép is-állapot predikátuma komponenseit az szabja meg, hogy mire kívánjuk használni az absztrakt gépet.

Általában:

$$\begin{aligned} \text{is-állapot} = (& \langle s\text{-adat}_1 : \text{is-adat} \rangle, \\ & \vdots \\ & \langle s\text{-adat}_n : \text{is-adat} \rangle, \\ & \langle s\text{-vezérlés} : \text{is-vezérlés} \rangle, \\ & \langle s\text{-könyvtár} : \text{is-utasításkészlet} \rangle) \end{aligned}$$

Az $s\text{-adat}_i(\xi)$ komponensek azt az objektumot tartalmazzák, amellyel az absztrakt gép műveleteket végez (, pl az értelmezendő program és annak adatai).

Az $s\text{-vezérlés}(\xi)$ komponens (az absztrakt gép által) végrehajtandó absztrakt program. Az absztrakt program makróutasításokból alkotott fa, melynek csomópontjaiban utasítások foglalnak helyet. Ezt vezérlési fának nevezzük.

Az $s\text{-könyvtár}(\xi)$ komponens a makróutasítások kifejtéseit, definícióit tartalmazza. Így a VDL programot lényegében ez a két utóbbi komponens együtt határozza meg.

A vezérlési fa felépítését a végrehajtás során maga az absztrakt gép végzi.

A

cobegin ut₁ ; ut₂ ; ... ; ut_n coend

összetett utasítás végrehajtása akkor fejeződik be, ha az összes ut₁, ..., ut_n utasítás végrehajtása befejeződött.

Ennek az összetett utasításnak az értelmezést végző absztrakt gép által végrehajtandó absztrakt programozási nyelven

null ;
 ut₁ ,
 ut₂ ,

 ut_n

(5.3. ábra)

feleljen meg (5.3. ábra)

(Megjegyezzük, hogy a sorosan végrehajtandó begin ut₁ ; ... ; ut_n end utasításnak a

null ;
ut_n ;

ut₁ ;

konstrukció felelne meg.)

Ez az összetett utasítás az egyprocesszoros gépen az ut₁, ..., ut_n utasítások előre meg nem határozott sorrendben való végrehajtását jelenti, míg a többprocesszoros gépen az utasítások párhuzamosan is végrehajthatóak.

A többprocesszoros gépeken olyan műveleteket is definiálni kell, amelyek egymással párhuzamosan nem hajthatók végre. Ezek az un. primitív műveletek (pl. Load/Store művelet pár)

Azokat az utasításokat, amelyek egymással párhuzamosan nem hajthatók végre, nevezzük primitív utasításoknak.

Követeljük meg, hogy egy makróutasítás végrehajtásának kezdetekor legyen egyértelműen meghatározott annak eredménye (tehát pl. vezérlési fát adó makróutasítás esetén az, hogy az utasítást milyen vezérlési fával kell helyettesíteni.) Egy másik utasítás végrehajtása menetközben nem zavarhatja azt meg.

A VDL-ben két féle makróutasítás létezik:

- értéket adó makró
- vezérlési fát adó makró.

Legyen minden makróutasítás is primitív utasítás és a makróutasítás végrehajtásának minden művelete primitív művelet. Ekkor egy absztrakt program végrehajtásának eredménye az egyprocesszoros gépen és a többprocesszoros gépen ugyanaz. Ez esetben az egyprocesszoros absztrakt gépen a többprocesszoros gépen futó feladatok megoldása is definiálható, ha a párhuzamos programrészek szinkronizációjának megvalósításához a VDL utasításkészletét kibővítjük a megvárakoztató makróutasítással.

D e f i n i c i ó j a :

is-várj-def = (<s-par : is-formal-par-lista> ,
 <s-def : is-esemény>)

is-esemény = (<s-egyik : is-egyik> , <s-másik : is-másik>)

is-egyik = (<s-feltétel : is-feltétel> , <s-ut : is-makró-ut>)

is-másik = (<s-feltétel : is-TRUE> , <s-ut : is-várj-def>)

Felírásának formája:

$$\underline{ut} (x_1, x_2, \dots, x_n) =$$

$$\begin{array}{lcl} \text{VÁRJ} \longrightarrow & [p(x_1, \dots, x_n; \xi) \longrightarrow & t(x_1, \dots, x_n; \xi) \\ & \text{TRUE} \longrightarrow & \underline{ut}(x_1, \dots, x_n)] \end{array}$$

ahol $p(x_1, \dots, x_n; \xi)$ predikátum és $t(x_1, \dots, x_n; \xi)$ olyan utasításfa, amely megvárakoztató utasítást már nem tartalmaz.

Jelentése:

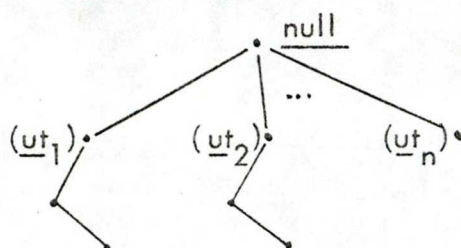
- Ha a processzor végre akarja hajtani az $\underline{ut}$ megvárakoztató utasítást,

akkor először meghatározza a p predikátum értékét.

- Ha a p predikátum értéke igaz, akkor a processzor végrehajtja a t vezérlési fát az aktuális paraméterek értéke mellett.

- Ha a p predikátum értéke hamis, akkor a processzor rátér egy másik terminális pontban lévő makróutasítás végrehajtására és a vezérlési fa vizsgált levelére a megvárakoztató utasítást helyezi vissza.

Többprocesszoros gép esetén az 5.3. ábrán látott utasítások feldolgozásán több processzor is dolgozhat párhuzamosan (5.4. ábra).



5.4. ábra

Amennyiben az ut_i -k közös adatokon dolgoznak, mivel azokhoz egyidőben csak egy folyamat férhet hozzá, a többi folyamatnak várakozni kell. A megvárakoztató utasítás végrehajtása ezt biztosítja; ha a folyamat vizsgált utasítása nem hajtható végre, akkor a folyamat leáll, a processzor elválik tőle, rátér egy másik folyamat terminálisában helyet foglaló makróutasítás végrehajtására. (A processzorok és folyamatok egymáshoz rendelését az operációs rendszer végzi el.)

5.4. Az EPPL interpreter

Adjuk meg az EPPL nyelv szemantikáját absztrakt értelmezővel. Az absztrakt gép állapotát meghatározó komponensek:

- az értelmezendő program
- az értelmezendő program adatai és utasítás számlálói
- az értelmező program

Igy

is- állapot = ($\langle$ s-program : is-absztrakt-program $\rangle$,
 $\langle$ s-adatok : is-adatok $\rangle$,
 $\langle$ s-mutató : is-egész-konst $\rangle$,
 $\langle$ s-párh-mutató : is-egész-konst-lista $\rangle$,
 $\langle$ s-vezérlés : is-vezérlés $\rangle$,
 $\langle$ s-könyvtár : is-utasításkészlet $\rangle$).

Az értelmezést végző absztrakt gépre megadjuk a nyelv értelmezőjének programját. Az absztrakt értelmező ξ_0 kezdeti állapota legyen a következő:

$\xi_0 = \mu_0(\langle$ s-program : is-párh-program $\rangle$,
 $\langle$ s-mutató : 1 $\rangle$,
 $\langle$ s-párh-mutató : is- $\langle \rangle$ $\rangle$,
 $\langle$ s-adatok : $\mu_0(\langle$ s-tul: is- $\langle \rangle$ $\rangle$, $\langle$ s-jel: is - $\langle \rangle$ $\rangle$, $\langle$ s-input: is-inp $\rangle$ $\rangle$)
 $\langle$ s-vezérlés : pr-végrehajtás $\rangle$,
 $\langle$ s-könyvtár : HALM $\rangle$)

ahol tehát:

- az értelmezendő program egy VDL objektummal megadott absztrakt EPPL nyelvű program, azaz is-absztrakt-program (is-párh-program) = TRUE
- a mutató értéke 1
- a párhuzamos mutató és az adatkomponens üres (kivéve az s-input komponenst, melyen az input adatok helyezkednek el)
- az értelmező program (a végrehajtandó absztrakt program) az s-vezérlés komponensen helyezkedik el. A vezérlési fa most egy szögpontból áll, melyhez a pr-végrehajtás makróutasítás van rendelve
- a szükséges makródefiníciókat (HALM) a könyvtár tartalmazza .

Két utasításslálót használunk. Az s-mutató (ξ) az értelmezendő utasítás kiválasztására szolgál:



elem (s-mutató (ξ) . s-ut-rész.s-program (ξ)

Ha a kiválasztott utasítás egy párhuzamos utasítás, akkor az ezt alkotó párhuzamos folyamatok utasításai közül értelmezésre az s-párh-mutató (ξ) utasításszám-láló segítségével történik egy utasítás kiválasztása. Ha

elem(i).s-párh-mutató (ξ) = k

azaz az i-edik folyamat k-adik utasítása a kiválasztandó, akkor az

elem (k).(elem(i) . s-folyamatok(t))

az i-edik párhuzamos folyamat értelmezésre következő utasítását választja ki.

Az absztrakt gép egy másik komponense az értelmezendő program adatainak halmaza. A programban szereplő azonosítók tulajdonsága és értéke az s-tul ill. s-jel komponenséről az azonosító nevekként mint szelektorokkal választhatók ki. Az s-jelz komponensen az értelmező program egy munkaváltozójának aktuális értéke áll. Az s-input komponensen egy input file, míg az s-output komponensen egy output file állhat. A felfüggesztett folyamatok nyilvántartására az s-sem komponens szolgál.

is-adatok = ($\langle$ s-tul : is-tul-táblázat $\rangle$, $\langle$ s-jel: is-jel-táblázat $\rangle$,
 $\langle$ s-sem : is-sem-táblázat $\rangle$, $\langle$ s-jelz : is-érték $\rangle$,
 $\langle$ s-input : is-inp $\rangle$, $\langle$ s-output : is-out $\rangle$)

is-tul-táblázat = ({ $\langle$ az : is-tip $\rangle$ || is-egyszerű-változó(az) v is-címke(az)})

is-jel-táblázat = ({ $\langle$ az : (is-érték v is-érték-lista) $\rangle$ ||
is-egyszerű-változó (az) v is-címke(az)})

is-sem-táblázat = ({ $\langle$ az : is-érték-lista $\rangle$ || is-egyszerű-változó(az)})

ahol:

is-tip = is-vált-tip v is-címke-tip

is-vált-tip = is-egyszerű-típus v is-tömb v is-m-buffer

is-m-buffer = ($\langle$ s-p-méret : is-egész-konst $\rangle$, $\langle$ s-j: is-egész-konst $\rangle$,
 $\langle$ s-el-tip : is-skálár-típus $\rangle$)

is-érték = is-egész-konst

is-cimke-tip	=	is-e-cimke $\vee$ is-p-cimke
is-inp	=	is-érték-lista $\vee$ is- $\langle \rangle$
is-out	=	is-érték-lista $\vee$ is- $\langle \rangle$
is-p-cimke	=	($\langle$ s-ct : PCIMKE $\rangle$, $\langle$ s-fsz : is-érték $\rangle$)
is-e [^] -cimke	=	{ ECIMKE }

Az absztrakt gép egy ξ , állapotában az s-program komponensre az 5.5. ábra, míg az s-adatok komponensre az 5.6. ábra ad egy példát.

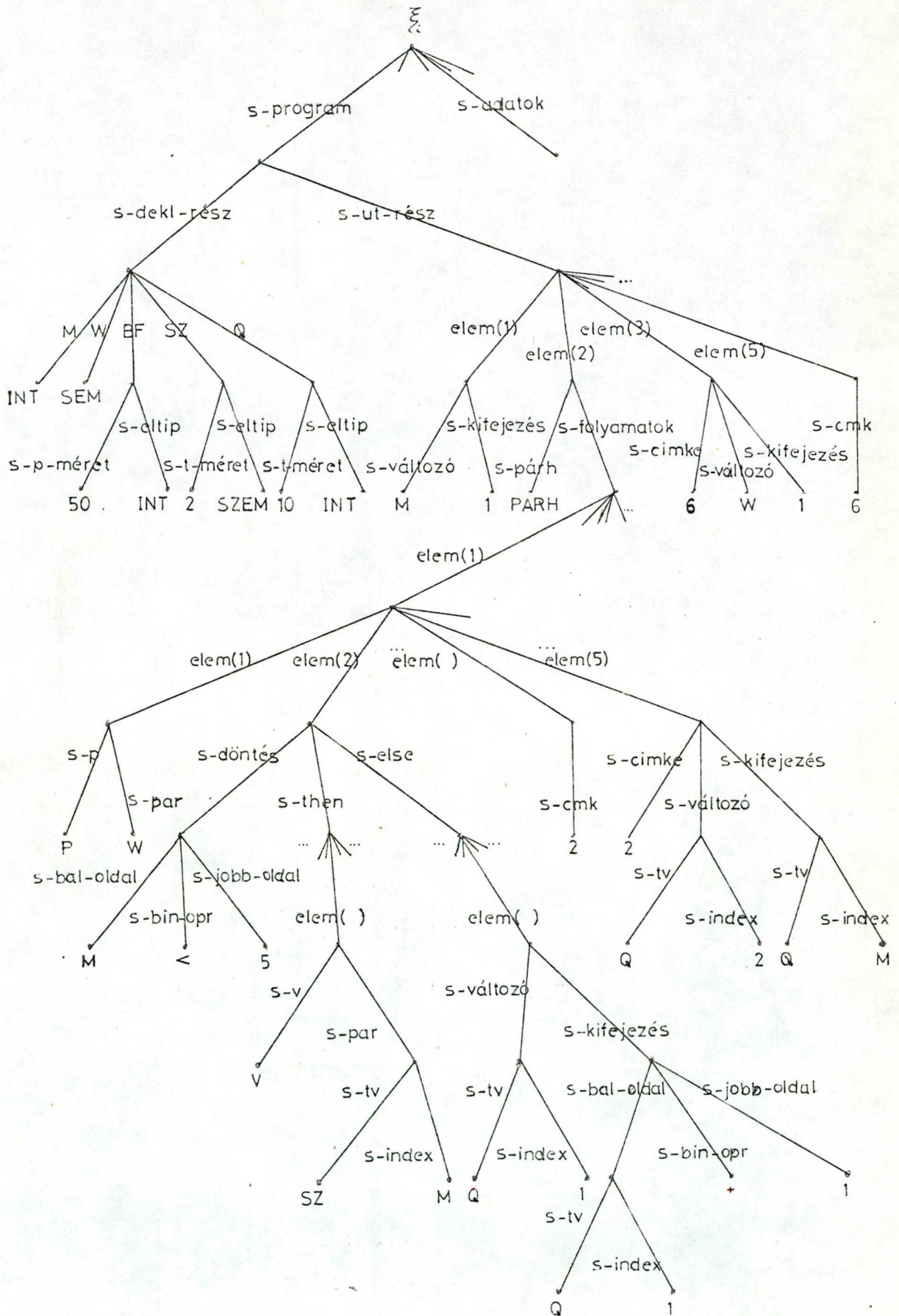
A program utasításainak tényleges értelmezését két előkészítő lépés vezeti be : a deklarációk feldolgozása és a címkegyűjtés. (A formális leírást a MELLÉKLET tartalmazza.)

A deklarációkat tartalmazó s-dekl-rész . s-program (ξ) komponens elemei tetszőleges sorrendben kerülnek feldolgozásra ([I2] - [I9]), mindegyik egy bejegyzést eredményez az s-tul.s-adatok(ξ) és az s-jel.s-adatok(ξ) komponensre. A változókhoz értékük definiálásáig az utóbbi komponensen a NEMDEF érték kerül feljegyzésre.

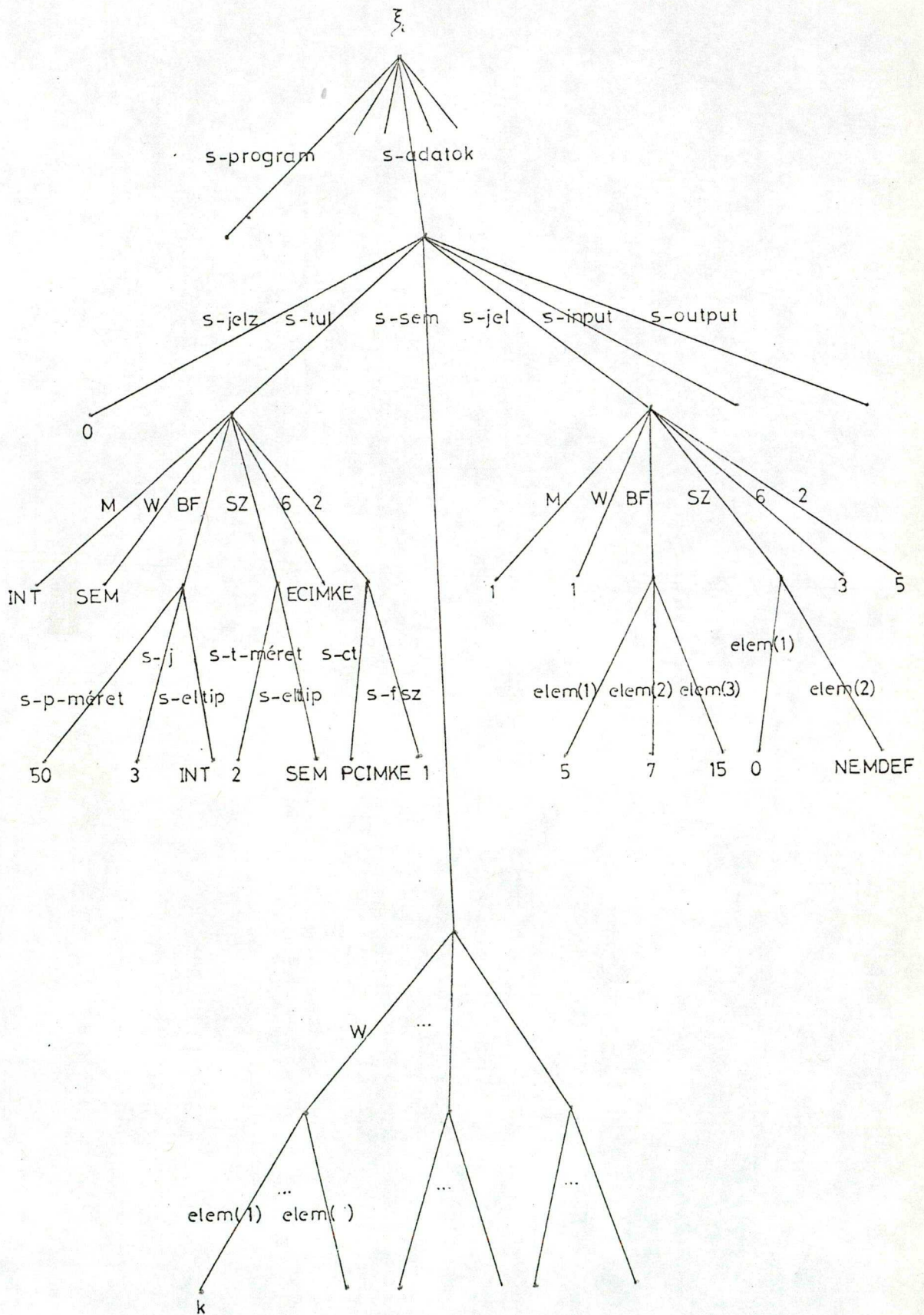
A pufferek deklarált változó a deklaráció feldolgozásakor módosított adatszerkezettel kerül át az s-tul.s-adatok(ξ) komponensre, nevezetesen kiegészül az s-j komponenssel, melynek értéke a pufferben aktuálisan levő elemek x számát adja meg; $1 \leq x \leq$ s-p-méret. Üres puffer esetén értéke 0. Az s-jel.s-adatok(ξ) komponensen a puffer változó egy x elemű listát azonosít.

Többszörös deklaráció esetén hibajelzés ([I5]) történik, bár ezt a vizsgálatot a Translátor is elvégezheti.

A címkegyűjtés során ([I10] - [I21]) minden címkéről egy bejegyzés készül az s-tul.s-adatok(ξ) és az s-jel.s-adatok(ξ) komponensre. Ha több utasításnak is ugyanaz a címkéje ([I13]), vagy ha több folyamatban is előfordul ugyanaz a címke utasítás előtt ill. egy folyamaton belül többször is ([I18]), akkor ez hibajelzést eredményez. A párhuzamos mutató építés ([I20]) biztosítja, hogy a párhuzamos folyamatok számának megfelelő elemszámu lista elemei az egyes folyamatok utasításainak számát tartalmazzák a címkegyűjtés befejezésekor.



5.5 ábra



5.6 ábra

Az utasítások értelmezése előtt az s -jelz. s -adatok (ξ) komponensre nullát írunk, s ezt csak a párhuzamos folyamatok feldolgozásának idejére módosítjuk 1-re.

Az értékadó utasítás értelmezése eltérő tevékenységeket is tartalmaz, attól függően, hogy párhuzamos folyamatokon kívül ([I26] - [I49]) vagy azokban ([I62] - [I68]) fordul elő. A leírásban a tömörségre törekvés céljából a lehetséges részeket összevonjuk.

Igy a kifejezések értelmezését ugyanaz a kif-értelmezés makróutasítás ([I27]) végzi értad-értelmezés és p-értad-értelmezés esetén is, működésében csupán s -jelz. s -adatok (ξ) értéke alapján van eltérés. Nevezetesen értad-értelmezés végrehajtásakor a kifejezésekben (s -jelz. s -adatok (ξ) = 0) INT, SEM és BOOL típusu változók és ugyanilyen típusu tömbelemek fordulhatnak elő; míg p-értad-értelmezés esetén a kifejezésekben (s -jelz. s -adatok (ξ) = 1) ezeken túl még INT és BOOL elemtípusu pufferre is történhet hivatkozás, viszont SEM típusu változóra vagy tömbelemre nem.

A $v := e$ értékadás végrehajtása a v . s -jelz. s -adatok(ξ) szelektorra az e érték felírását eredményezi. Előbb azonban szükséges a v változó és az e érték típusa kompatibilitásának vizsgálata. (Pl. INT típusu változónak TRUE érték adása értelmetlen). Ezt végzi el a konvert ill. p-konvert makróutasítás.

Működésük előfeltétele, hogy egy kifejezés kiértékelésekor egy olyan két komponensű ob objektum épüljön fel, melyet jellemző predikátum formális definíciója:

$$is-ob = (\langle s-típus: is-egyszerű-típus \rangle , \langle s-érték: is-konstans \rangle)$$

Ezen objektum felépítését az ft-ob makróutasítás végzi ([I43])

Az ft-ob által felépített ob objektum s -típus komponensén INT, BOOL vagy SEM állhat.

Ha egy tömb indexe egy egyszerű változó ([I35]), akkor ennek értékét a megkeres makróutasítás ([I36]) szolgáltatja az s -jelz. s -adatok (ξ) komponensből.

Ha az operátor és az operandusz(ok) típusa nem kompatibilis, akkor hibajelzés történik ([I38] , [I39]) .

Az értékadás két oldala kompatibilitásának vizsgálatát végző konvert makróutasítás ([I45]) megengedi:

INT, BOOL vagy SEM típusu változóhoz	azonos típusu érték,
SEM típusu változóhoz	INT típusu érték,
INT, BOOL vagy SEM típusu tömbelemhez	azonos típusu érték,
SEM típusu tömbelemhez	INT típusu érték,

míg a p-konvert makróutasítás ([I64])

INT típusu változóhoz	INT típusu érték,
BOOL típusu változóhoz	BOOL típusu érték,
INT típusu tömb vagy puffer elemhez	INT típusu érték,
BOOL típusu tömb vagy puffer elemhez	BOOL típusu érték

rendelését. Egyéb esetekben hibajelzésre kerül sor.

A tömbértékadást [I47] ill. [I66] írja le.

A formális leírás során LIFO szerkezetű puffert használunk, a pufferműveletek végrehajtásának menete vázlatosan az alábbi (legyen BF egy puffer-változó).

Pufferbe töltés:

- s-j . BF . s-tul . s-adatok (ξ) értékének növelése 1-gyel ([I67]),
- (ha a növelés előtti érték = s-p-méret. BF . s-tul.s-adatok(ξ), akkor hibajelzés ([I66]),
- az aktuális érték felírása az elem (s-j) . BF . s-jel . s-adatok (ξ) szelektorra ([I68]);

Ürités a pufferből:

- az elem (s-j) . BF.s-jel.s-adatok(ξ) szelektoron levő érték leakasztása ([I31]),
- az s-j . BF.s-tul. s-adatok(ξ) értékének csökkentése 1-gyel ([I32] - [I34]),
- (üres puffer esetén hibajelzés ([I30]))

- a puffer utolsó elemének törlésekor a BF.s-jel.s-adatok (ξ) szelektorra NEMDEF elhelyezése.

(Megjegyezzük, hogy ha FIFO szerkezetű pufferrel dolgoznánk, ez utóbbi puffer művelet megvalósítása a következő lenne:

- elem (1).BF . s-jel.s-adatok (ξ) szelektoron levő érték leakasztása,
- BF. s-jel . s-adatok $\leftarrow \mu(\text{BF.s-jel. s-adatok}(\xi); \{ \langle \text{elem}(i) : \text{elem}(i+1)(x) \rangle \mid 1 \leq i \leq \text{length}(x) - 1 \})$ ahol $x = \text{BF.s-jel.s-adatok}(\xi)$,
- az s-j. BF. s-tul.s-adatok (ξ) értékének csökkentése 1-gyel)

A bináris logikai és reláció műveletek tömör leírását adtuk. ([I41], [I42])

Részletezve pl. az is-AND(opr) $\rightarrow v_1$ & v_2 helyett a következő írható:

is-AND(opr) $\rightarrow$ logikai-and (v_1, v_2)

logikai-and (a,b) =

a = FALSE $\rightarrow$ PASS $\leftarrow$ FALSE

TRUE $\rightarrow$ PASS $\leftarrow$ b

Az input utasítás az s-input.s-adatok(ξ) komponensről leválaszt egy elemet ([I51] - [I52]), míg az output utasítás az s-output.s-adatok(ξ) komponensen levő listát bővíti egy elemmel ([I53] - [I54]).

A párhuzamos folyamatok végrehajtásakor ([I55] - [I60]) egy folyamat (k) első utasításának értelmezése után az értelmezésre következő utasítás kiválasztását a párh-mut-növelés(k) makróutasítás segíti elő. Egy goto utasítás értelmezését követően az s-párh-mutató megfelelő (k-adik) komponensére a goto utasítás által kijelölt címkéjű utasítás sorszama - 1 kerül, így a párh-mut-növelés(k) végrehajtása ez esetben is biztosítja a ténylegesen értelmezésre következő utasítás kiválasztását.

Az utasítás lista utolsó utasításának értelmezését követően a végrehajt makróutasítás ismételt felhívásakor $i > \text{length}(t1)$, tehát a folyamat értelmezése megtörtént.

A goto utasítás értelmezését végző p-goto-értelmezés (k,t) makróutasítás ([I69]) biztosítja a helyes vezérlésátadást.

A párhuzamos folyamat utasításai között szerepelhet feltételes utasítás, melynek ELSE ágán üres utasítás is állhat. A két ágon helyet foglaló utasítás



listán szerepelhet minden olyan utasítás, mely a párhuzamos folyamatban megengedett - két kivétellel:

- az utasításoknak nem lehet címkéjük,
- csak olyan feltételes utasítások fordulhatnak elő, melyek THEN illetve ELSE ágán már nem szerepel feltételes utasítás. Értelmezésüket ([I70] - [I74]) írja le. A feltételes utasítások egymásba ágyazottsági szintjét 1 tükrözi.

A P és V utasításokban csak SEM típusu változó vagy tömbem fordulhat elő.

A P utasítás értelmezése során ([I75] - [I80]) végrehajtásra kerül az 5.3-ban definiált megvárakoztató makróutasítás. [I76] eredményeképp, ha a t.s-jel. s-adatok (ξ) > 0 feltétel teljesül, akkor a leírásnak megfelelően folytatódik az értelmezés.

Ha a feltétel nem teljesül (a szemafor értéke nulla), akkor

- az adatkomponens s-sem szelektorára a szemafor nevével (pl. w) ellátott lista következő (ha a szemafor most kerül az s-sem szelektorra, akkor első) elemeként felkerül a folyamat sorszáma (k),
- a vezérlési fa vizsgált levelére a megvárakoztató utasítás (P-ért-1(k,t)) kerül visszairásra,
- a processzor elválik-e (k-adik) folyamattól, rátér egy másik terminális pontban levő makróutasítás végrehajtására.

[I78] értelmezése a fentivel analóg.

A V utasítás értelmezésének menetét ([I81] - [I84]) tartalmazza. Ha a szemafor (pl.w) értéke nulla és az s-sem.s-adatok(ξ) komponensen a (w) szemafor szerepel, akkor a szemaforon alvó egyik folyamatot fel kell ébreszteni, ezt végzi el az ébresztés (w) makróutasítás ([I82], [I84]). Eredményeképp az elem (i).w.s-sem.s-adatok (ξ) által kiválasztott folyamat lekerül a várakozási listáról. i értéke az ütemezési stratégiától függ, s a folyamat ébresztés az operációs rendszer feladata.

6. AZ MPPL ÉS FORDITHATÓSÁGA EPPL-RE

A 4. fejezetben megmutattuk, hogy a magas szintű szinkronizációs eszközök implementálhatóak szemaforokkal, ezt használjuk fel a továbbiakban.

Definiálunk egy magas szintű párhuzamos programozási nyelvet, az MPPL-t, melynek az EPPL-től való eltérése főként abból adódik, hogy szinkronizációs eszközként a P és V műveletek mellett megengedi critical region (CR), conditional critical region (CCR), monitor (M) és path kifejezés, type definíció (PT) használatát is.

A 2.7 - 2.10 algoritmusok tehát MPPL nyelvű programrészek.

A 4. fejezetben leírt módszerek megvalósításával konstruálható olyan fordítóprogram, mely az MPPL nyelven írt programokat EPPL nyelvűre fordítja.

Az MPPL nyelvű programok strukturájának áttekintése után a különböző magas szintű szinkronizációs eszközök fordítási mechanizmusát vázoljuk. A fordítás algoritmusának részletes leírása meghaladja a dolgozat kereteit. Csak annyit mutatunk meg, hogy az EPPL elég általános ahhoz, hogy az MPPL nyelvű programokat arra lefordítsuk.

Az MPPL szintaxisának leírásánál (6.1. táblázat) - kihasználva a két nyelv közös elemeit - lehetőség szerint utalunk az EPPL konkrét szintaxisa leírására (5.1. táblázat) "lásd [Ki]" formában.

Az MPPL szemantikájának megadása fordítóprogramjának (MPPL→EPPL) leírásával történhet.

Néhány a fordításkor figyelembe veendő jellemzőre a 6.1 táblázatban megjegyzés formájában utalunk, míg a fontosabb nyelvi elemek fordításának mechanizmusát 6.1. - 6.3. pontokban tárgyaljuk.

A leírást a nyelv strukturájának, s a párhuzamosság nyelvi kifejezésének érzékeltetéséhez szükséges szintig végezzük el. Az MPPL is bővíthető.

$\langle \text{program} \rangle ::= \text{PROGRAM } \langle \text{deklarációs rész} \rangle \text{ BEGIN } \langle \text{utasítás rész} \rangle \text{ END.}$
 $\langle \text{deklarációs rész} \rangle ::= \langle \text{type-def rész} \rangle \langle \text{vált-dekl rész} \rangle \langle \text{monitor-def rész} \rangle$
 $\langle \text{type-def rész} \rangle ::= \langle \text{üres} \rangle | \langle \text{type-def} \rangle \{ ; \langle \text{type-def} \rangle \};$
 $\langle \text{üres} \rangle ::=$
 $\langle \text{type-def} \rangle ::= \text{TYPE } \langle \text{type-név} \rangle = \langle \text{type-törzs} \rangle$
 $\langle \text{type-név} \rangle ::= \langle \text{azonosító} \rangle$
 $\langle \text{type-törzs} \rangle ::= \text{BEGIN } \langle \text{lok-vált-dekl} \rangle \langle \text{lok-adat-inic} \rangle \langle \text{path kif} \rangle^1$
 $\text{OPERATIONS } \langle \text{művelet def} \rangle \text{ END}$
 $\langle \text{lok-vált-dekl} \rangle ::= \langle \text{ált-vált-dekl} \rangle \{ ; \langle \text{ált-vált-dekl} \rangle \};$
 $\langle \text{lok-adat-inic} \rangle ::= \langle \text{értékadó-ut} \rangle \{ ; \langle \text{értékadó-ut} \rangle \};$

Megj.: 1/ egy path kifejezés

$\langle \text{művelet-def} \rangle ::= \{ \langle \text{t-elj-dekl vagy fgv-dekl} \rangle \};$
 $\langle \text{t-elj-dekl vagy fgv-dekl} \rangle ::= \langle \text{t-elj-dekl} \rangle | \langle \text{fgv-dekl} \rangle$
 $\langle \text{t-elj-dekl} \rangle ::= \text{PROCEDURE } \langle \text{type-elj-név} \rangle (\langle \text{formális paraméter szekció} \rangle$
 $\{ ; \langle \text{formális paraméter szekció} \rangle \} ; \langle \text{vált-dekl rész} \rangle$
 $= \langle \text{elj-törzs} \rangle$
 $\langle \text{type-elj-név} \rangle ::= \langle \text{azonosító} \rangle$
 $\langle \text{formális paraméter szekció} \rangle ::= \langle \text{ált-vált-dekl} \rangle$
 $\langle \text{fgv-dekl} \rangle ::= \text{FUNCTION } \langle \text{azonosító} \rangle : \langle \text{eredménytípus} \rangle = \langle \text{elj-törzs} \rangle$
 $\langle \text{eredmény típus} \rangle ::= \langle \text{típus} \rangle$
 $\langle \text{vált-dekl rész} \rangle ::= \langle \text{üres} \rangle | \text{VAR } \langle \text{vált-dekl} \rangle \{ ; \langle \text{vált-dekl} \rangle \};$
 $\langle \text{vált-dekl} \rangle ::= \langle \text{ált-vált-dekl} \rangle | \langle \text{osztott-vált-dekl} \rangle$
 $\langle \text{ált-vált-dekl} \rangle ::= \langle \text{azonosító} \rangle \{ , \langle \text{azonosító} \rangle \} : \langle \text{típus} \rangle^2$
 $\langle \text{osztott-vált-dekl} \rangle ::= \langle \text{azonosító} \rangle \{ , \langle \text{azonosító} \rangle \} : \text{SHARED}^3 \langle \text{típus} \rangle$

Megj: 2/ $\langle \text{type-def} \rangle$ -ben definiált típus hátrább levő $\langle \text{type-def} \rangle$ -ekben

$\langle \text{típus} \rangle$ -ként szerepelhet (a TYPE-ok sorrendjét ez megszabja, előre hivatkozás tilos)

3/ Csak $\langle \text{CR-ut} \rangle$ és $\langle \text{CCR-ut} \rangle$ -ban hivatkozhatunk rá.

$\langle \text{típus} \rangle ::= \langle \text{egyszerű típus} \rangle | \langle \text{strukturált típus} \rangle$
 $\langle \text{egyszerű típus} \rangle ::= \langle \text{skalár típus} \rangle | \langle \text{szemafor típus} \rangle | \langle \text{feltétel típus} \rangle$

<skalár típus> ::= <egész típus> | <logikai típus>
 <feltétel típus> ::= CONDITION⁴

Megj: 4/ csak monitor definícióban fordulhat elő

<strukturált típus> ::= <rekord típus> | <tömb típus> | <puffer típus>
 <rekord típus> ::= RECORD <rekord szekció> { <rekord szekció> } END
 <rekord szekció> ::= <ált-vált-dekl>
 <monitor-def rész> ::= <üres> | MON <monitor def> { <monitor def> } ;
 <monitor def> ::= <monitornév> : MONITOR <monitor-törzs>
 <monitornév> ::= <azonosító>
 <monitor-törzs> ::= BEGIN <lok-vált-dekl> <mon-elj-dekl>
 { <mon-elj-dekl> } ; <lok-adat-inic> END
 <mon-elj-dekl> ::= PROCEDURE <mon-elj-név> (<formális paraméter szekció>
 { <formális paraméter szekció> }) ; <elj-törzs>
 <mon-elj-név> ::= <azonosító>
 <eljárás-dekl rész> ::= <elj-dekl> { <elj-dekl> } ; | <üres>
 <elj-dekl> ::= PROCEDURE <eljárásnév> (<formális paraméter szekció> {
 <formális paraméter szekció> }) ; <elj-törzs>⁵

Megj: 5/ szinkronizációt leíró utasítások közül <P-ut>, <V-ut>, <CR-ut>
 és <CCR-ut> szerepelhet benne.

lásd [K9] - [K21]

<párhuzamos-folyamat> ::= BEGIN <vált-dekl rész> <párh-ut>
 { ; <párh-ut> } END

lásd [K23]

<p-cím kétlen-ut> ::= <értékadó ut> | <goto-ut> | <feltételes-ut> |
 <input-ut> | <output-ut> | <P-ut> | <V-ut> |
 <eljárás-utasítás> | <CR-ut> | <CCR-ut> | <M-ut> |
 <PT-ut>

lásd [K25 - K28]

<eljárás-utasítás>⁶ ::= <eljárás név> (<aktuális paraméter> { , <aktuális paraméter> })
 <eljárásnév> ::= <azonosító>
 <aktuális paraméter> ::= <változó>

Megj: 6/ <CR-ut> és <CCR-ut> alkalmazásakor fordulhat elő

<CR-ut> ::= REGION <változó>⁷ DO <elj-törzs>
<CCR-ut> ::= REGION<változó>⁷ WHEN <kifejezés>⁸ DO <elj-törzs>

Megj: 7/ SHARED típusu változó

8/ a kifejezés értéke logikai típusu kell hogy legyen

<M-ut> ::= <monitornév> . <mon eljnév>(<aktuális paraméter>
{, <aktuális paraméter>})

<PT-ut> ::= <type-név> . <type eljnév>(<aktuális paraméter>
{, <aktuális paraméter>})

<elj-törzs> ::= BEGIN <utasítások>⁹ END

Megj: 9/ Az<utasítások>tartalma attól függően, hogy az <elj-törzs>
milyen utasításban szerepel - eltérő. (Ezek kifejtése [K29]
módosítását, s további utasítás fajták bevezetését igényli)

lásd [K29] - [K47]

<címke> ::= <betű> <szám>

lásd [K49] - [K52]

6.1. táblázat (folytatás)

A<párhuzamos-folyamat>-okban szereplő lokális változókat a fordításkor általános változók váltják fel. Mivel az EPPL nyelvben eljárások definiálására nincs lehetőség, egy eljárás hívás hatására az aktuális-formális paraméter cseréje végrehajtásával aktualizált eljárástörzs generálásra kerül az EPPL nyelvű program aktuális helyére, eközben a deklarációs rész újabb általános változókkal bővül.

6.1. A conditional critical region fordíthatósága

A (6.1.a. ábrán feltüntetett szerkezetű MPPL program) fordítását két lépésben végezzük:

- conditional critical region megvalósítása critical region és szemafor segítségével (6.1.b. ábra)
- critical region implementálása szemaforokkal (6.1.c. ábra, EPPL pr).

Vizsgáljuk először azt az esetet, amikor a programban nem szerepelnek eljárások:

- Az egyazon v_e shared változóhoz tartozó conditional critical region-okat és critical region-okat egy csoportba soroljuk, s minden ilyen változóhoz a fordítás idejére egy M_e táblázatot rendelünk. Csoportonként a B_i ütemezési feltételek mindegyikéhez egy bejegyzést rendelünk M_e -ben a következő tartalommal:

B_i , $SZ[i]$, $Q[i]$, folyamatonként a B_i -hez tartozó utasítássort leíró mutatók

- Az M_e felhasználásával a 4.1.-ben részletezett algoritmus alapján a conditional critical regionokat megvalósítjuk critical region és szemafor segítségével, illetve a csoportbeli critical regionokat kibővítjük a látott módon. A deklarációk közé az SZ és Q tömbök deklarációja, a szekvenciális utasítások közé kezdeti értékadások kerülnek.

- A critical region-t a 4.1.-ben leírt módon megvalósítjuk szemaforokkal, felépítjük a komplett EPPL nyelvű programot, mely a deklarációk átírását is jelenti. A v_e shared változónak egy w_e szemafort feleltetünk meg.

A fordítás közben ellenőrizni kell, hogy shared változóra csak a hozzá tartozó region-okban történhet hivatkozás. Kikötés, hogy a conditional critical regionokban azonos feltételekhez azonos végrehajtandó utasítássor tartozhat.

A 6.1.a., b., c. ábrákon $i \neq k$.

A 2.8. algoritmus MPPL és EPPL-beli szerkezetét a 6.2.a, b. ábra ($i \neq j$) szemlélteti (n az ütemezési feltételek száma; T az EPPL által megengedett tetszőleges típus).



program

var v: shared record

BF: array[max] of T

;

end;

begin

cobegin

;

Pi : begin

;

region v when B_i do S_i;

;

end;

;

Pk : begin

;

region v when B_k do S_k;

;

end;

coend;

end

program

var v: shared record

BF: array [max] of T

end;

SZ: array [n] of SEM;

Q: array [n] of INT;

SZ := 0 ; Q:= 0;

cobegin

Pi : begin

region v do

begin

if B_i then

S_i ; V(SZ[i])

else Q[i]:= Q[i]+1 fi;

if B_k & (Q[k]>0) then

Q[k]:= Q[k] - 1;

S_k ; V(SZ[k]) fi ;

end;

P(SZ[i]);

end;

Pk :

;

coend;

end

program

BF : array[max] of T;
SZ : array [n] of SEM;
Q : array [n] of INT;
W : SEM;

SZ := 0; Q := 0; W := 1;

cobegin

i : begin
P(W);
if B_i then
S_i; V(SZ[i])
else Q [i] := Q[i] + 1 fi;
if B_k & (Q[k] > 0) then
Q[k] := Q[k] - 1 ;
S_k;
V(SZ[k]) fi ;
V(W) ;
P(SZ[i]) ;
end;
k :
:

coend;

end.

program

```

var   v : shared record
        BF: array [50] of INT;
        FULL : INT;
        :
        end;
        M,N : INT ;
        :
begin
    FULL := 0;
    :
    cobegin
        :
        Pi: begin
            :
            (x az M érték beírása a
             BF pufferba x)
            region v when FULL<50 do
                begin
                    put (M,BF) ;
                    FULL := FULL +1
                end;
            Qi: begin
                :
                (x a BF pufferból egy érték
                 kiemelése N-be x)
                region v when FULL>0 do
                    begin
                        get (N,BF);
                        FULL := FULL-1
                    end;
                coend;
            end
        coend;
    end

```

6.2.a. ábra

program

```

        BF : array [50] of INT;
        FULL, M, N : INT;
        SZ: array [2] of SEM;
        Q : array [2] of INT;
        W: SEM;

        FULL := 0;
        SZ := 0 ; Q:=,0; W := 1;
        cobegin
            i: begin
                P(W);
                if FULL < 50 then
                    put (M,BF);
                    FULL := FULL + 1;
                    V(SZ [1])
                else Q[1]:= Q[1]+1 fi;
                if (FULL > 0) & (Q[2]>0)
                then
                    Q[2]:= Q [2] -1;
                    get (N,BF) ;
                    FULL := FULL - 1;
                    V(SZ [2] ) fi;
                V(W);
                P(SZ [1])
            end;
            i: :
        coend;
    end.

```

6.2.b. ábra

Lehetőség van eljárások alkalmazására is, a fordítás eredménye ez esetben is ugyanaz kell hogy legyen, mint amikor nem éltünk az eljárások lehetőségével.

Példaként csak utalunk a 6.1. ábra-beli MPPL program ilyen módon történő átírására. A rövidebb írás kedvéért jelölje D az alábbi típust:

```
shared record BF : array [max] of T
                        ⋮
                        end
```

```
Ekkor
    ⋮
var    $\mu$  : D ;
    ⋮

procedure   Eljnév (  $v$  : D, ... ) ;
begin
    region  $v$  when    $B_i$  do  $S_i$  ;
end;
    ⋮
 $P_i$ :
    ⋮
    Eljnév (  $\mu$ , ... )
    ⋮
```

6.3. ábra

Az eljárások fordítására a monitorral kapcsolatban a következő pontban utalunk.

6.2. A monitor fordíthatósága

Monitoronként 2 szemaforot és egy egész típusu változót, valamint minden feltétel változóhoz további egy-egy szemaforot és egész típusu változót vezetünk be.

A monitorbeli deklarációkat lefordítjuk EPPL-beli megfelelőjükre, s ott az előbb említett és további szükséges közös változók deklarációját is elhelyezzük. A monitorbeli kezdőérték adásokat a további kezdőérték adásokkal együtt

mon rb : monitor

begin BF : array [50] of INT;

lp, c : INT;

ne, nf : condition;

procedure pufferbe (x:INT);

begin

BF [lp] := x ; } (x)

end;

procedure pufferból (x : INT) ;

begin

x := BF [] ; } (x x)

end;

C := 0; lp := 0

end

:

P_i :

var y : INT ;

rb.pufferbe (y);

:

Q_i :

var y : INT ;

rb.pufferból (y);

:

program

BF: array [50] of INT;

lp, c : INT;

M, U : SEM ; UC : INT;

FSZ: array [2] of SEM;

FC : array [2] of INT;

y1, y2 : INT;

:

c := 0; lp := 0;

M := 1; U := 0; UC := 0;

FSZ := 0; FC := 0;

:

i :

BF [lp] := y1 ; } (x')

i :

y2 := BF [] ; } (xx')

end.

6.4. ábra

a szekvenciális utasításrészben helyezzük el.

Ezt követően az egyes folyamatokba a monitornév.eljárásnév (akt paraméter) hívósorok helyére a hívott eljárás törzsének az aktuális-formális paramétercsere végrehajtásával aktualizált változatából a 4.2.-ben leírt algoritmussal generált utasítás sorozatot szerkesztjük be.

A fordítás közben ellenőrizni kell, hogy a monitorban deklarált változókhoz csak a monitor eljárás hívások során lehet hozzáférés.

A 2.9. algoritmus MPPL és EPPL-beli szerkezetét a 6.4. ábra szemlélteti.

6.3. Path kifejezésekkel szinkronizált folyamatok fordíthatósága

A type definíciók adat definíciós része alapján változó, tömb ill. puffer deklarációk, míg a path kifejezésekkel leírt szinkronizáció megvalósításához szemafor változók ill. tömbök generálása szükséges. A folyamatokból aktivizált eljárásokban szereplő lokális változókhoz általános változókat, míg a kölcsönös kizárás megvalósításához szemaforokat vezetünk be.

A 2.10. algoritmus fordításának eredménye a 6.5. ábrán látható EPPL nyelvű programrész lehet.

A példában az üzenet buffer szerepét a BF tömb játssza. Vele azonos elemszámu SZ1 és SZ2 szemafor tömbökre van szükség, mivel mindegyik puffer elemből olvasni csak írást követően lehet. A termelő-fogyasztó szinkronizációt az SZ1 és SZ2 szemafor tömbök oldják meg.

(Legyen $n \equiv 50$, $T \equiv \text{INT}$)

Legfeljebb 1 termelő és legfeljebb csak 1 fogyasztó tudja szimultán elérni vagy módosítani BF-et. (W ill. R szemafor biztosítja ezt).

Az eljárásokban szereplő lokális változók (j) szerepét WSZ, illetve RSZ veszi át.

program

BF : array [50] of INT;

WSZ, RSZ : INT ; W, R : SEM ;

SZ1, SZ2: array [50] of SEM;

Y1, Y2 : INT ;

:

WSZ := 0; RSZ := 0 ; W := 1; R := 1 ; SZ1 := 1; SZ2 := 0 ;

:

cobegin

:

i: begin

(× Y1 értékének előállítása ×)

P(W);

WSZ := WSZ + 1;

if WSZ = 51 then WSZ := 1 fi ;

P(SZ1 [WSZ]) ;

BF [WSZ] := Y1 ;

V (SZ2 [WSZ]) ;

V(W);

:

end;

j : begin

P(R); RSZ := RSZ + 1;

if RSZ = 51 then RSZ := 1 fi;

P(SZ2[RSZ]); Y2 := BF[RSZ]; V (SZ1 [RSZ]);

V(R); (× Y2 feldolgozása ×)

:

end;

coend;

end.

IRODALOMJEGYZÉK

- [1] Campbell, R.H., Habermann, A.N.: The specification of process synchronization by path expressions.
Lecture Notes in Comput. Sci. Vol 16, Springer Verlag, (1974), 89-101.
- [2] Courtois, P.J., Heymans, F., Parnes, D.L.: Concurrent control with "Readers and writers"
Comm. ACM 14,10 (1971) 667-668.
- [3] Dijkstra, E.W.: Cooperating sequential processes" in Programming Languages, F. Genuys, (Ed.) Academic Press, New York, (1968), 43-112.
- [4] Dijkstra, E.W.: Hierarchical ordering of sequential processes.
Acta Informatica, 1,2 (1971), 115-138.
- [5] Habermann, A.N.: Synchronization of communicating processes.
Comm. ACM 15,3 (1972) 171-176.
- [6] Hansen, P.B.: Concurrent programming concepts
Computing Surveys, 5,4 (Dec, 1973), 223-245.
- [7] Hansen, P.B.: Distributed processes: A concurrent programming concept
Comm.ACM 21,11 (1978), 934-941.
- [8] Hansen, P.B.: Operating system principles.
Prentice-Hall, New Jersey (1973)
- [9] Hansen, P.B. Structured Multiprogramming
Comm. ACM 15,7 (1972), 574-578.
- [10] Hoare, C.A.R. Monitors: an operating system structuring concept
Comm. ACM 17,10 (1974), 549-557.



- [11] Hoare, C.A.R.: "Towards a theory of parallel programming" in Operating Systems Techniques, C.A.R. Hoare and R.H. Perrot, (Eds), Academic Press, London (1972), 61-71.
- [12] Howard, J.H.: Proving Monitors
Comm. ACM 19,5 (1976), 273-279.
- [13] Kessels, J.L.W.: A conceptual framework for a nonprocedural programming language
Comm. ACM 20,12 (1977), 906-913.
- [14] Knuth, E.: Remarks on concurrent programming problems.
MTA SZTAKI Közl. 18 (1977), 131-137.
- [15] Lauer, P.: Formal definition of ALGOL 60
TR 25-088, IBM Laboratory Vienna (Dec 1968)
- [16] Lee, A.N.: Computer semantics
Van Nostrand Reinhold Co. (1972)
- [17] Lucas, P., Lauer, P., Stigleitner, H.: Method and notation for the formal definition of programming languages.
TR 25.087, IBM Laboratory Vienna (July 1970)
- [18] Marcotty, M., Ledgard, H.F., Bochmann, G.V.: A sampler of formal definitions.
Computing Surveys. 8,2 (1976)
- [19] Neuhold, E.H.: The formal description of programming languages
IBM Systems Journal 10 (1971)
- [20] Neuhold, E.J.: The formal semantics of operating systems
North-Holland Publishing Company (1974)
- [21] Neuhold, E.H.: Towards the formal description of operating systems.
Proc. of the SIGPLAN/SIGOPS Workshop, Savannah, Ga, SIGPLAN Notices (Sept, 1973)

- [22] Presser, L., White, J.R.: Linkers and loaders
Computing Surveys, 4, 3 (1972)
- [23] Show, A.C.: The logical design of Operating Systems
Prentice-Hall, 1974.
- [24] Valette, R., Diaz, M.: Top-down formal specification and verification
of parallel control systems
Digital Processes, 4 (1978) 181-199.
- [25] Varga, L.: Rendszerprogramok elmélete és gyakorlata
Akadémiai kiadó, Bp. (1978)
- [26] Varga, L.: The abstractions of machine dependent program
forms
KFKI-76-11
- [27] Wadge, W.W., Ashcroft, E.A.: LUCID, a Nonprocedural Language
with Iteration
Comm. ACM 20, 7 (1977), 519-526.
- [28] Wegner, P.: The Vienna Definition Language
Computing Surveys 4, 1 (1972)
- [29] Winograd, T.: Beyond Programming Languages
Comm. ACM 22, 7 (1979), 391-401.
- [30] Wirth, N.: The programming language Pascal
Acta Informatica 1, 1 (1971) 35-63.

MELLÉKLET

$$\begin{aligned} [11] \quad & \underline{\text{pr-végrehajtás}} = \\ & \quad \underline{\text{cimkegyűjtés}} ; \\ & \quad \underline{\text{deklaráció-feldolgozás}} (\text{s-dekl-rész.s-program}(\xi)) \end{aligned}$$

$$\begin{aligned} [12] \quad & \underline{\text{deklaráció-feldolgozás}} (t) = \\ & \quad \text{is } -\langle \rangle (t) \longrightarrow \underline{\text{null}} \\ & \quad \text{is } - \text{deklarációs-rész}(t) \rightarrow \underline{\text{dekl-leválasztás}} (t) \end{aligned}$$

$$[13] \quad \underline{\text{null}} = \text{PASS} \leftarrow \Omega$$

$$\begin{aligned} [14] \quad & \underline{\text{dekl-leválasztás}}(t) = \\ & \quad \underline{\text{null}} ; \\ & \quad \{ \underline{\text{ért-deklaráció}} (\text{az}, \text{az}(t)) \parallel \text{az}(t) \neq \Omega \} \\ & \quad \text{ahol: is-deklarációs-rész}(t) \ \& \ \text{is-típus}(\text{az}(t)) \ \& \\ & \quad \text{is-egyszerű-változó}(\text{az}) \end{aligned}$$

$$\begin{aligned} [15] \quad & \underline{\text{ért-deklaráció}}(n, \text{tulsg}) = \\ & \quad n.\text{s-tul.s-adatok}(\xi) \neq \text{NIL} \rightarrow \text{error} \\ & \quad \text{TRUE} \longrightarrow \underline{\text{ért-dekl}} (n, \text{tulsg}) \end{aligned}$$

$$\begin{aligned} [16] \quad & \underline{\text{ért-dekl}}(n, \text{tulsg}) = \\ & \quad \text{is-egyszerű-típus}(\text{tulsg}) \longrightarrow \underline{\text{épit-1-tul}} (n, \text{tulsg}) \\ & \quad \text{is-tömb}(\text{tulsg}) \longrightarrow \underline{\text{épit-2-tul}} (n, \text{tulsg}) \\ & \quad \text{is-buffer}(\text{tulsg}) \longrightarrow \underline{\text{épit-3-tul}} (n, \text{tulsg}) \\ & \quad \text{ahol: is-egyszerű-változó}(n) \ \& \ \text{is-típus}(\text{tulsg}) \end{aligned}$$

$$\begin{aligned} [17] \quad & \underline{\text{épit-1-tul}} (n, \text{tulsg}) = \\ & \quad \text{s-tul.s-adatok} \leftarrow \mu(\text{s-tul.s-adatok}(\xi); \langle n: \text{tulsg} \rangle) \\ & \quad \text{s-jel.s-adatok} \leftarrow \mu(\text{s-jel.s-adatok}(\xi); \langle n: \text{NEMDEF} \rangle) \end{aligned}$$

[I8] épít-2-tul (n, tulsg) =
 $s\text{-tul.s-adatok} \leftarrow \mu(s\text{-tul.s-adatok}(\xi); \langle n: \text{tulsg} \rangle)$
 $s\text{-jel.s-adatok} \leftarrow \mu(s\text{-jel.s-adatok}(\xi); \langle n:$
 $\mu_0(\{ \langle \text{elem}(i): \text{NEMDEF} \rangle \mid 1 \leq i \leq s\text{-i-méret.tulsg} \}) \rangle)$

[I9] épít-3-tul (n, tulsg) =
 $s\text{-tul.s-adatok} \leftarrow \mu(s\text{-tul.s-adatok}(\xi); \langle n:$
 $\mu_0(\langle s\text{-p-méret} : s\text{-p-méret.tulsg} ,$
 $\langle s\text{-i} : 0 \rangle, \langle s\text{-el tip} : s\text{-el tip.tulsg} \rangle) \rangle)$
 $s\text{-jel.s-adatok} \leftarrow \mu(s\text{-jel.s-adatok}(\xi); \langle n: \text{NEMDEF} \rangle)$

[I10] cimkegyűjtés =
 $s\text{-mutató}(\xi) \leq \text{length.s-ut-rész.s-program}(\xi) \longrightarrow$
cimkegyűjtés;
mutatónövelés;
cimkeleválasztás(elem(s-mutató(ξ)). s-ut-rész.s-program(ξ))
 $s\text{-mutató}(\xi) > \text{length.s-ut-rész.s-program}(\xi) \longrightarrow$
értelmezés;
mutatóállítás;
jelzőállítás

[I11] cimkeleválasztás(t) =
 $\text{is-párhuzamos-ut}(t) \longrightarrow \text{gyűjt-pf-lista}(s\text{-folyamatok}(t), 1, 1);$
leválaszt(t)
TRUE $\longrightarrow \text{leválaszt}(t)$
ahol is-utasítás(t)

[I12] leválaszt(t) =
 $\text{is-cimkés}(t) \longrightarrow \text{c-értékadás}(s\text{-cimke}(t), s\text{-mutató}(\xi))$
 $\text{is-cimkétlen}(t) \longrightarrow \text{null}$

[I13] c-értékkadás (név, érték) =
név.s-tul.s-adatok(ξ) $\neq$ NIL $\longrightarrow$ error
TRUE $\longrightarrow$ c-értad(név, érték)

[I14] c-értad (név, érték) =
s-tul.s-adatok $\leftarrow \mu$ (s-tul.s-adatok(ξ); $\langle$ név: ECIMKE $\rangle$)
s-jel.s-adatok $\leftarrow \mu$ (s-jel.s-adatok(ξ); $\langle$ név: érték $\rangle$)
ahol: is-cimke(név) &
is-szám(érték)

[I15] gyűjt-pf-lista(t, i, l) =
is- $\langle$ X(t) $\longrightarrow$ null
TRUE $\longrightarrow$ gyűjt-pf-lista (tail(t), i+1, l);
gyűjt-p-ut-lista (head(t), i, l)
ahol: is-párh-folyamat-lista (t)

[I16] gyűjt-p-ut-lista(t, i, j) =
is- $\langle$ X(t) $\longrightarrow$ párh-mut-építés(i, j-1)
TRUE $\longrightarrow$ gyűjt-p-ut-lista(tail(t), i, j+1);
p-cimke-leválasztás (head(t), i, j)
ahol: is-párh-ut-lista(t)

[I17] p-cimke-leválasztás(t, i, j) =
is-p-cimkés(t) $\longrightarrow$ pc-értékkadás(s-cimke(t), i, j)
is-p-cimkétlen(t) $\longrightarrow$ null
ahol: is-párh-ut (t)

[I18] pc-értékkadás (név, i, érték) =
név.s-tul.s-adatok(ξ) $\neq$ NIL $\longrightarrow$ error
TRUE $\longrightarrow$ pc-ért-ad (név, i, érték)



$$\begin{aligned}
 [I19] \quad & \underline{\text{pc-ért-ad}}(\text{név}, i, \text{érték}) = \\
 & \text{s-tul.s-adatok} \leftarrow \mu(\text{s-tul.s-adatok}(\xi); \langle \text{név}: \\
 & \quad \mu_0(\langle \text{s-ct:PCIMKE} \rangle, \langle \text{s-fsz: } i \rangle) \rangle) \\
 & \text{s-jel.s-adatok} \leftarrow \mu(\text{s-jel.s-adatok}(\xi); \langle \text{név} : \text{érték} \rangle)
 \end{aligned}$$

$$\begin{aligned}
 [I20] \quad & \underline{\text{párh-mut-építés}}(k, l) = \\
 & \text{s-párh-mutató} \leftarrow \mu(\text{s-párh-mutató}(\xi); \langle \text{elem}(k) : l \rangle)
 \end{aligned}$$

$$\begin{aligned}
 [I21] \quad & \underline{\text{mutatónövelés}} = \\
 & \text{s-mutató} \leftarrow \text{s-mutató}(\xi) + 1
 \end{aligned}$$

$$\begin{aligned}
 [I22] \quad & \underline{\text{jelzőállítás}} = \\
 & \text{s-adatok} \leftarrow \mu(\text{s-adatok}(\xi); \langle \text{s-jelz} : 0 \rangle)
 \end{aligned}$$

$$\begin{aligned}
 [I23] \quad & \underline{\text{mutatóállítás}} = \\
 & \text{s-mutató} \leftarrow 1
 \end{aligned}$$

$$\begin{aligned}
 [I24] \quad & \underline{\text{értelmezés}} = \\
 & \text{s-mutató}(\xi) \leq \text{length.s-ut-rész.s-program}(\xi) \longrightarrow \\
 & \quad \underline{\text{értelmezés;}} \\
 & \quad \underline{\text{mutatónövelés;}} \\
 & \quad \underline{\text{jelzőállítás;}} \\
 & \quad \underline{\text{ut-értelmezés}}(\text{elem}(\text{s-mutató}(\xi)).\text{s-ut-rész.s-program}(\xi)) \\
 & \text{s-mutató}(\xi) > \text{length.s-ut-rész.s-program}(\xi) \longrightarrow \underline{\text{null}}
 \end{aligned}$$

$$\begin{aligned}
 [I25] \quad & \underline{\text{ut-értelmezés}}(t) = \\
 & \text{is-értékadó-ut}(t) \longrightarrow \underline{\text{értad-értelmezés}}(t) \\
 & \text{is-goto-ut}(t) \longrightarrow \underline{\text{goto-értelmezés}}(t) \\
 & \text{is-input-ut}(t) \longrightarrow \underline{\text{input-értelmezés}}(t) \\
 & \text{is-output-ut}(t) \longrightarrow \underline{\text{output-értelmezés}}(t) \\
 & \text{is-párhuzamos-ut}(t) \longrightarrow \underline{\text{párh-végrehajt-elők}}(t) \\
 & \text{ahol: is-utasítás}(t)
 \end{aligned}$$

$$\begin{aligned}
 [I26] \quad \underline{\text{értad-értelmezés}(t)} &= \underline{\text{konvert-értékkadás}}(s\text{-változó}(t), ob); \\
 ob: &\underline{\text{kif-értelmezés}}(s\text{-kifejezés}(t)) \\
 \text{ahol: } is\text{-változó}(s\text{-változó}(t)) \ \& \ is\text{-kifejezés}(s\text{-kifejezés}(t)) \ \& \ is\text{-ob}(ob)
 \end{aligned}$$

$$\begin{aligned}
 [I27] \quad \underline{\text{kif-értelmezés}(t)} &= \\
 is\text{-konstans}(t) &\longrightarrow \underline{\text{konst-értelmezés}}(t) \\
 is\text{-változó}(t) &\longrightarrow \underline{\text{vált-értelmezés}}(t) \\
 is\text{-unáris}(t) &\longrightarrow \underline{\text{un-értelmezés}}(a, s\text{-un-opr}(t)); \\
 &\quad a: \underline{\text{kif-értelmezés}}(s\text{-operandus}(t)) \\
 is\text{-bináris}(t) &\longrightarrow \underline{\text{bin-értelmezés}}(a, b, s\text{-bin-opr}(t)); \\
 &\quad a: \underline{\text{kif-értelmezés}}(s\text{-bal-oldal}(t)), \\
 &\quad b: \underline{\text{kif-értelmezés}}(s\text{-jobb-oldal}(t)) \\
 \text{ahol: } &is\text{-kifejezés}(t)
 \end{aligned}$$

$$\begin{aligned}
 [I28] \quad \underline{\text{konst-értelmezés}(t)} &= \\
 is\text{-logikai-konst}(t) &\longrightarrow PASS \leftarrow \underline{\text{ft-ob}}(BOOL, t) \\
 is\text{-egész-konst}(t) &\longrightarrow PASS \leftarrow \underline{\text{ft-ob}}(INT, t)
 \end{aligned}$$

$$\begin{aligned}
 [I29] \quad \underline{\text{vált-értelmezés}(t)} &= \\
 is\text{-egyszerű-változó}(t) &\longrightarrow \underline{\text{tartalom-1}}(t) \\
 is\text{-indexes-változó}(t) &\longrightarrow \underline{\text{tartalom-2}}(t)
 \end{aligned}$$

$$\begin{aligned}
 [I30] \quad \underline{\text{tartalom-1}(v)} &= \\
 v_{tip} &= (INT \vee SEM \vee BOOL)^* \& (v_{ért} \neq NIL) \longrightarrow \\
 &\quad PASS \leftarrow \underline{\text{ft-ob}}(v_{tip}, v_{ért}) \\
 s\text{-jelz.s-adatok}(\xi) = 1 \ \& \ s\text{-eltyp.} v_{tip} &= (INT \vee BOOL) \ \& \ f > 0 \longrightarrow \\
 &\quad \underline{s-j-csökkentés}(v, f); \\
 &\quad \underline{\text{pufferből}}(v) \\
 TRUE &\longrightarrow \text{error}
 \end{aligned}$$

 Megj.: */ A ($is\text{-INT}(v_{tip}) \vee is\text{-SEM}(v_{tip}) \vee is\text{-BOOL}(v_{tip})$)
 reláció tömör leírása. E jelölést a mellékletben több
 helyen használjuk.

ahol : $v_{tip} \stackrel{\text{def}}{=} v.s\text{-tul.s-adatok}(\xi) \ \& \ v_{ért} \stackrel{\text{def}}{=} v.s\text{-jel.s-adatok}(\xi)$
 $\& \ f \stackrel{\text{def}}{=} s\text{-j.v}_{tip}$

és ez a jelölés érvényes [I30] - [I37]-ig

[I31] pufferból (v) =
 PASS $\leftarrow$ ft-ob (s-el_{tip}.v_{tip}, elem(f).v_{ért})

[I32] s-j-csökkentés (v, f) = s-j-cs(v);
töröl(v, f)

[I33] töröl(v, f) =
 $f > 1 \rightarrow v.s\text{-jel.s-adatok} \leftarrow \mu(v_{ért}; \langle \text{elem}(f) : \Omega \rangle)$
 $f = 1 \rightarrow s\text{-jel.s-adatok} \leftarrow \mu(v_{ért}; \langle v : \text{NEMDEF} \rangle)$

[I34] s-j-cs(v) =
 $v.s\text{-tul.s-adatok} \leftarrow \mu(v_{tip}; \langle s\text{-j} : s\text{-j.v}_{tip}-1 \rangle)$

[I35] tartalom-2(t) =
 $\text{is-egész-konst}(s\text{-index}(t)) \rightarrow \text{tartalom-3}(s\text{-tv}(t), s\text{-index}(t))$
 $\text{is-azonosító}(s\text{-index}(t)) \rightarrow \text{tartalom-3}(s\text{-tv}(t), a);$
 $a: \text{megkeres}(s\text{-index}(t))$
 TRUE $\rightarrow$ error

[I36] megkeres(b) =
 $b.s\text{-tul.s-adatok}(\xi) = \text{INT} \ \& \ b.s\text{-jel.s-adatok}(\xi) \neq \text{NEMDEF} \rightarrow$
 PASS $\leftarrow b.s\text{-jel.s-adatok}(\xi)$
 TRUE $\rightarrow$ error

[I37] tartalom-3(v, c) =
 $\text{is-tömb}(v_{tip}) \ \& \ c \leq s\text{-t-méret}(v_{tip}) \ \& \ \text{elem}(c).v_{ért} \neq \text{NEMDEF} \rightarrow$
 PASS $\leftarrow$ ft-ob(s-el_{tip}.v_{tip}, elem(c).v_{ért})
 TRUE $\rightarrow$ error

$$\begin{aligned}
 [I38] \quad & \underline{\text{un-értelmezés}}(ob, opr) = \\
 & \text{is-BOOL}(tipus_1) \ \& \ \text{is-NOT}(opr) \rightarrow \text{PASS} \leftarrow \underline{\text{ft-ob}}(\text{BOOL}, \text{érték}_1) \\
 & \text{is-INT}(tipus_1) \ \& \ \text{is-PLUS}(opr) \rightarrow \text{PASS} \leftarrow ob \\
 & \text{is-INT}(tipus_1) \ \& \ \text{is-MINUS}(opr) \rightarrow \text{PASS} \leftarrow \underline{\text{ft-ob}}(\text{INT}, -\text{érték}_1) \\
 & \text{TRUE} \rightarrow \text{error} \\
 & \text{ahol: } tipus_1 \stackrel{\text{def}}{=} s\text{-tipus}(ob) \ \& \ \text{érték}_1 \stackrel{\text{def}}{=} s\text{-érték}(ob)
 \end{aligned}$$

$$\begin{aligned}
 [I39] \quad & \underline{\text{bin-értelmezés}}(ob-1, ob-2, opr) = \\
 & \text{is-INT}(tipus_1) \ \& \ \text{is-INT}(tipus_2) \ \& \ \text{is-aritm-bin-opr}(opr) \rightarrow \\
 & \quad \text{PASS} \leftarrow \underline{\text{ft-ob}}(\text{INT}, \underline{\text{aritm-bin-érték}}(v_1, v_2, opr)) \\
 & \text{is-BOOL}(tipus_1) \ \& \ \text{is-BOOL}(tipus_2) \ \& \ \text{is-log-bin-opr}(opr) \rightarrow \\
 & \quad \text{PASS} \leftarrow \underline{\text{ft-ob}}(\text{BOOL}, \underline{\text{log-bin-érték}}(v_1, v_2, opr)) \\
 & \text{is-INT}(tipus_1) \ \& \ \text{is-INT}(tipus_2) \ \& \ \text{is-reláció-bin-opr}(opr) \rightarrow \\
 & \quad \text{PASS} \leftarrow \underline{\text{ft-ob}}(\text{BOOL}, \underline{\text{reláció-bin-érték}}(v_1, v_2, opr)) \\
 & \text{TRUE} \rightarrow \text{error} \\
 & \text{ahol: } tipus_1 \stackrel{\text{def}}{=} s\text{-tipus}(ob-1) \ \& \ v_1 \stackrel{\text{def}}{=} s\text{-érték}(ob-1) \ \& \\
 & \quad tipus_2 \stackrel{\text{def}}{=} s\text{-tipus}(ob-2) \ \& \ v_2 \stackrel{\text{def}}{=} s\text{-érték}(ob-2)
 \end{aligned}$$

$$\begin{aligned}
 [I40] \quad & \underline{\text{aritm-bin-érték}}(v_1, v_2, opr) = \\
 & \text{is-ADD}(opr) \rightarrow \text{PASS} \leftarrow v_1 + v_2 \\
 & \text{is-SUBTR}(opr) \rightarrow \text{PASS} \leftarrow v_1 - v_2
 \end{aligned}$$

$$\begin{aligned}
 [I41] \quad & \underline{\text{log-bin-érték}}(v_1, v_2, opr) = \\
 & \text{is-AND}(opr) \rightarrow v_1 \ \& \ v_2 \\
 & \text{is-OR}(opr) \rightarrow v_1 \ \vee \ v_2
 \end{aligned}$$

$$\begin{aligned}
 [I42] \quad & \underline{\text{reláció-bin-érték}}(v_1, v_2, opr) = \\
 & \text{is-GT}(opr) \rightarrow v_1 > v_2 \\
 & \text{is-GE}(opr) \rightarrow v_1 \geq v_2 \\
 & \text{is-EQ}(opr) \rightarrow v_1 = v_2 \\
 & \text{is-LE}(opr) \rightarrow v_1 \leq v_2 \\
 & \text{is-LT}(opr) \rightarrow v_1 < v_2 \\
 & \text{is-NE}(opr) \rightarrow v_1 \neq v_2
 \end{aligned}$$

$$[I43] \quad \underline{ft-ob(tipus, \acute{e}rt\acute{e}k)} = \mu_0(\langle s-tipus:tipus \rangle, \langle s-\acute{e}rt\acute{e}k : \acute{e}rt\acute{e}k \rangle)$$

$$[I44] \quad \underline{konvert-\acute{e}rt\acute{e}kad\acute{a}s} (valt, ob) = \underline{\acute{e}rt\acute{e}kad\acute{a}s}(valt, w); \\ w: \underline{konvert}(valt, ob)$$

$$[I45] \quad \underline{konvert}(valt, ob) = \\ v_t = s-tipus(ob) \longrightarrow PASS \leftarrow s-\acute{e}rt\acute{e}k(ob) \\ is-SEM(v_t) \ \& \ is-INT(s-tipus(ob)) \longrightarrow PASS \leftarrow s-\acute{e}rt\acute{e}k(ob) \\ s-el\acute{t}ip.v_t = s-tipus(ob) \ \& \ is-t\ddot{o}mb(v_t) \longrightarrow PASS \leftarrow s-\acute{e}rt\acute{e}k(ob) \\ is-SEM(s-el\acute{t}ip.v_t) \ \& \ is-INT(s-tipus(ob)) \ \& \ is-t\ddot{o}mb(v_t) \longrightarrow \\ PASS \leftarrow s-\acute{e}rt\acute{e}k(ob) \\ TRUE \longrightarrow error \\ ahol: v_t \stackrel{def}{=} valt.s-tul.s-adatok(\xi)$$

$$[I46] \quad \underline{\acute{e}rt\acute{e}kad\acute{a}s}(v\acute{a}lt, \acute{e}rt\acute{e}k) = \\ is-egyszer\ddot{u}-v\acute{a}ltoz\acute{o}(v\acute{a}lt) \longrightarrow \underline{\acute{e}rt\acute{e}kad\acute{a}s-1}(v\acute{a}lt, \acute{e}rt\acute{e}k) \\ is-indexes-v\acute{a}ltoz\acute{o}(v\acute{a}lt) \longrightarrow \underline{\acute{e}rt\acute{e}kad\acute{a}s-2}(v\acute{a}lt, \acute{e}rt\acute{e}k)$$

$$[I47] \quad \underline{\acute{e}rt\acute{e}kad\acute{a}s-1}(v, e) = \\ v.s-tul.s-adatok(\xi) = (INT \ v \ SEM \ v \ BOOL) \longrightarrow \\ s-jel.s-adatok \leftarrow \mu(s-jel.s-adatok(\xi); \langle v:e \rangle) \\ is-t\ddot{o}mb(v_{tip}) \longrightarrow s-jel.s-adatok \leftarrow \mu(s-jel.s-adatok(\xi); \\ \langle v: \mu_0(\{ \langle elem(i):e \rangle \mid 1 \leq i \leq s-t-m\acute{e}ret.v_{tip} \}) \rangle) \\ TRUE \longrightarrow error \\ ahol: v_{tip} \stackrel{def}{=} v.s-tul.s-adatok(\xi)$$

$$[I48] \quad \underline{\acute{e}rt\acute{e}kad\acute{a}s-2}(v, e) = \\ is-eg\acute{e}sz-konst(s-index(v)) \longrightarrow \underline{\acute{e}rt\acute{e}kad\acute{a}s-3}(s-tv(v), s-index(v), e) \\ is-azonosító(s-index(v)) \longrightarrow \underline{\acute{e}rt\acute{e}kad\acute{a}s-3}(s-tv(v), a, e); \\ a: \underline{megkeres}(s-index(v))$$

[I49] értékkadás-3(v, c, e) =
 $\text{is-tömb}(v_{\text{tip}}) \ \& \ c \leq \text{s-t-méret}(v_{\text{tip}}) \longrightarrow$
 $\text{s-jel.s-adatok} \leftarrow \mu(\text{s-jel.s-adatok}(\xi); \langle \text{elem}(c). \text{v}_{\text{ért}} : e \rangle)$
 $\text{TRUE} \longrightarrow \text{error}$
 $\text{ahol: } v_{\text{tip}} \stackrel{\text{def}}{=} v.\text{s-tul.s-adatok}(\xi) \ \&$
 $v_{\text{ért}} \stackrel{\text{def}}{=} v.\text{s-jel.s-adatok}(\xi)$

[I50] goto-értelmezés(t) =
 $\text{s-cmk}(t) . \text{s-tul.s-adatok}(\xi) = \text{ECIMKE} \longrightarrow$
 $\text{s-mutató} \leftarrow \text{s-cmk}(t) . \text{s-jel.s-adatok}(\xi) - 1$
 $\text{TRUE} \longrightarrow \text{error}$

[I51] input-értelmezés(t) =
értékkadás(s-olvas(t), b);
 $b: \underline{\text{read}}$

[I52] read =
 $\text{is} \leftarrow \langle \rangle (\text{s-input.s-adatok}(\xi)) \longrightarrow \text{error}$
 $\text{TRUE} \longrightarrow \mu(\text{s-adatok}(\xi); \langle \text{s-input:tail}(\text{s-input}(\xi) \quad \rangle \rangle);$
 $\text{PASS} \leftarrow \text{head}(\text{s-input.s-adatok}(\xi))$

[I53] output-értelmezés(t) =
write(b);
 $b: \underline{\text{kif-értelmezés}}(\text{s-kiir}(t))$

[I54] write(v) =
 $\mu(\text{s-adatok}(\xi); \langle \text{s-output: s-output}(\xi) \quad v \rangle)$

[I55] párh-végrehajt-elők(t) = párh-végrehajt(t);
jelzőnövelés

[I56] jelzőnövelés =
 $\text{s-adatok} \leftarrow \mu(\text{s-adatok}(\xi); \langle \text{s-jelz} : 1 \rangle)$

$$\begin{aligned}
 [157] \quad & \underline{\text{párh-végrehajt}(t)} = \\
 & \quad \underline{\text{null}}; \\
 & \quad \underline{\text{tevékenység-1}(1, t)}, \\
 & \quad \vdots \\
 & \quad \underline{\text{tevékenység-k}(k, t)}, \\
 & \quad \vdots \\
 & \quad \underline{\text{tevékenység-m}(m, t)} \\
 & \text{ahol: is-párhuzamos-ut}(t) \ \& \ L \stackrel{\text{def}}{=} \text{s-párh-mutató}(\xi) \ \& \\
 & m = (1i) \ (\text{elem}(i) \ (L) \neq \Omega \ \& \ \text{elem}(i+1) \ (L) = \Omega)
 \end{aligned}$$

$$\begin{aligned}
 [158] \quad & \underline{\text{tevékenység-k}(k, t)} = \\
 & \quad \underline{\text{végrehajt}}(k, \text{elem}(k).s\text{-folyamatok}(t), 1); \\
 & \quad \underline{\text{p-mutató-ind}(k)} \\
 & \text{ahol: is-párh-ut-lista}(\text{elem}(k).s\text{-folyamatok}(t))
 \end{aligned}$$

$$\begin{aligned}
 [159] \quad & \underline{\text{p-mutató-ind}(k)} = \\
 & \quad \text{s-párh-mutató} \leftarrow \mu(\text{s-párh-mutató}(\xi); \langle \text{elem}(k) : 1 \rangle)
 \end{aligned}$$

$$\begin{aligned}
 [160] \quad & \underline{\text{végrehajt}}(k, t1, i) = \\
 & \quad \text{length}(t1) < i \rightarrow \underline{\text{null}} \\
 & \quad \text{TRUE} \longrightarrow \underline{\text{végrehajt}}(k, t1, \text{elem}(k).s\text{-párh-mutató}(\xi)); \\
 & \quad \quad \underline{\text{párh-mut-növelés}(k)}; \\
 & \quad \quad \underline{\text{p-ut-értelmezés}(k, \text{elem}(i).t1)} \\
 & \text{ahol: is-szám}(\text{elem}(k).s\text{-párh-mutató}(\xi)) \ \& \\
 & \quad \text{is-párh-ut}(\text{elem}(i).t1)
 \end{aligned}$$

$$\begin{aligned}
 [161] \quad & \underline{\text{p-ut-értelmezés}(k, t)} = \\
 & \quad \text{is-értékadó-ut}(t) \longrightarrow \underline{\text{p-értad-értelmezés}(t)} \\
 & \quad \text{is-goto-ut}(t) \longrightarrow \underline{\text{p-goto-értelmezés}(k, t)} \\
 & \quad \text{is-input-ut}(t) \longrightarrow \underline{\text{input-értelmezés}(t)} \\
 & \quad \text{is-output-ut}(t) \longrightarrow \underline{\text{output-értelmezés}(t)} \\
 & \quad \text{is-feltételes-ut}(t) \longrightarrow \underline{\text{feltételes-értelmezés}(k, 0, t)} \\
 & \quad \text{is-P-ut}(t) \longrightarrow \underline{\text{P-értelmezés}(k, t)} \\
 & \quad \text{is-V-ut}(t) \longrightarrow \underline{\text{V-értelmezés}(t)} \\
 & \text{ahol: is-párh-ut}(t)
 \end{aligned}$$

$$[I62] \quad \underline{p\text{-értad-értelmezés}(t)} = \underline{\text{konvert-p-értékkadás}(s\text{-változó}(t), ob);}$$

$$ob: \underline{kif\text{-értelmezés}(s\text{-kifejezés}(t))}$$

ahol: $is\text{-változó}(s\text{-változó}(t)) \ \& \ is\text{-kifejezés}(s\text{-kifejezés}(t)) \ \& \ is\text{-ob}(ob)$

$$[I63] \quad \underline{\text{konvert-p-értékkadás}(valt, ob)} = \underline{p\text{-értékkadás}(valt, w);}$$

$$w: \underline{p\text{-konvert}(valt, ob)}$$

$$[I64] \quad \underline{p\text{-konvert}(valt, ob)} =$$

$$v_t = s\text{-tipus}(ob) \ \& \ is\text{-INT}(v_t) \longrightarrow \text{PASS} \leftarrow s\text{-érték}(ob)$$

$$v_t = s\text{-tipus}(ob) \ \& \ is\text{-BOOL}(v_t) \longrightarrow \text{PASS} \leftarrow s\text{-érték}(ob)$$

$$s\text{-el tip. } v_t = s\text{-tipus}(ob) \ \& \ is\text{-INT}(s\text{-el tip. } v_t) \longrightarrow \text{PASS} \leftarrow s\text{-érték}(ob)$$

$$s\text{-el tip. } v_t = s\text{-tipus}(ob) \ \& \ is\text{-BOOL}(s\text{-el tip. } v_t) \longrightarrow \text{PASS} \leftarrow s\text{-érték}(ob)$$

$$\text{TRUE} \longrightarrow \text{error}$$

ahol $v_t \stackrel{\text{def}}{=} valt.s\text{-tul.s-adatok}(\xi)$

$$[I65] \quad \underline{p\text{-értékkadás}}(valt, érték) =$$

$$is\text{-egyszerű-változó}(valt) \longrightarrow \underline{p\text{-értékkadás-1}}(valt, érték)$$

$$is\text{-indexes-változó}(valt) \longrightarrow \underline{értékkadás-2}(valt, érték)$$

$$[I66] \quad \underline{p\text{-értékkadás-1}}(v, e) =$$

$$v.s\text{-tul.s-adatok}(\xi) = (\text{INT } v \text{ BOOL}) \longrightarrow$$

$$s\text{-jel.s-adatok} \leftarrow \mu(s\text{-jel.s-adatok}(\xi); \langle v:e \rangle)$$

$$is\text{-tömb}(v_{tip}) \longrightarrow s\text{-jel.s-adatok} \leftarrow \mu(s\text{-jel.s-adatok}(\xi);$$

$$\langle v : \mu(\{ \langle \text{elem}(i):e \rangle \mid 1 \leq i \leq s\text{-t-méret. } v_{tip} \}) \rangle)$$

$$s\text{-el tip. } v_{tip} = (\text{INT } v \text{ BOOL}) \ \& \ (s\text{-j. } v_{tip} < s\text{-p-méret. } v_{tip}) \longrightarrow$$

$$\underline{\text{pufferbe}}(v, e);$$

$$\underline{s\text{-j-növelés}}(v)$$

$$\text{TRUE} \longrightarrow \text{error}$$

ahol: $v_{tip} \stackrel{\text{def}}{=} v.s\text{-tul.s-adatok}(\xi)$
és ez a jelölés érvényes [I66] - [I68] -ig



$$[167] \quad \underline{s\text{-}j\text{-}növelés(v)} = \\ v.s\text{-}tul.s\text{-}adatok \leftarrow \mu(v_{tip} ; \langle s\text{-}j : s\text{-}j.v_{tip} + 1 \rangle)$$

$$[168] \quad \underline{pufferbe(v, e)} = \\ v.s\text{-}jel.s\text{-}adatok \leftarrow \mu(v.s\text{-}jel.s\text{-}adatok(\xi); \\ \langle elem(s\text{-}j.v_{tip}) : e \rangle)$$

$$[169] \quad \underline{p\text{-}goto\text{-}értelmezés(k, t)} = \\ s\text{-}ct.s\text{-}cmk(t) . s\text{-}tul.s\text{-}adatok(\xi) = PCIMKE \ \& \\ s\text{-}fsz.s\text{-}cmk(t).s\text{-}tul.s\text{-}adatok(\xi) = k \longrightarrow \\ s\text{-}párh\text{-}mutató \leftarrow \mu(s\text{-}párh\text{-}mutató(\xi); \langle elem(k) : d_t - 1 \rangle) \\ TRUE \longrightarrow error \\ ahol: d_t \stackrel{def}{=} s\text{-}cmk(t).s\text{-}jel.s\text{-}adatok(\xi)$$

$$[170/a] \quad \underline{feltételes\text{-}értelmezés(k, l, t)} = \underline{felt\text{-}ért(k, l, t, ob)}; \\ ob: \underline{kif\text{-}értelmezés(s\text{-}döntés(t))}$$

$$[170/b] \quad \underline{felt\text{-}ért(k, l, t, ob)} = \\ s\text{-}tipus(ob) = BOOL \ \& \ s\text{-}érték(ob) = TRUE \longrightarrow \underline{értelmez\text{-}ut\text{-}lista(k, l, s\text{-}then(t))} \\ s\text{-}tipus(ob) = BOOL \ \& \ s\text{-}érték(ob) = FALSE \longrightarrow \underline{értelmez\text{-}ut\text{-}lista(k, l, s\text{-}else(t))} \\ TRUE \longrightarrow error \\ ahol: is\text{-}p\text{-}cimkétlen\text{-}lista(s\text{-}then(t)) \ \& \\ (is\text{-}p\text{-}cimkétlen\text{-}lista(s\text{-}else(t)) \vee is\text{-}\langle \rangle(s\text{-}else(t))) \ \& \\ is\text{-}kifejezés(s\text{-}döntés(t)) \ \& \ is\text{-}ob(ob)$$

$$[171] \quad \underline{értelmez\text{-}ut\text{-}lista(k, l, t)} = \\ is\text{-}\langle \rangle(t) \longrightarrow \underline{null} \\ TRUE \longrightarrow \underline{értelmez\text{-}ut\text{-}lista(k, l, tail(t))}; \\ \underline{értelmez\text{-}ut(k, l, head(t))}$$

$$[172] \quad \underline{értelmez\text{-}ut(k, l, t)} = \\ l = 0 \longrightarrow \underline{ért\text{-}0\text{-}ut(k, t)} \\ l = 1 \longrightarrow \underline{ért\text{-}1\text{-}ut(k, t)} \\ TRUE \longrightarrow error$$

[173] $\underline{\text{ért-0-ut}}(k, t) =$

$\text{is-értékadó-ut}(t)$	$\longrightarrow$	$\underline{\text{p-értad-értelmezés}}(t)$
$\text{is-goto-ut}(t)$	$\longrightarrow$	$\underline{\text{p-goto-értelmezés}}(k, t)$
$\text{is-input-ut}(t)$	$\longrightarrow$	$\underline{\text{input-értelmezés}}(t)$
$\text{is-output-ut}(t)$	$\longrightarrow$	$\underline{\text{output-értelmezés}}(t)$
$\text{is-feltételes-ut}(t)$	$\longrightarrow$	$\underline{\text{feltételes-értelmezés}}(k, 1, t)$
$\text{is-P-ut}(t)$	$\longrightarrow$	$\underline{\text{P-értelmezés}}(k, t)$
$\text{is-V-ut}(t)$	$\longrightarrow$	$\underline{\text{V-értelmezés}}(t)$

ahol: $\text{is-p-cimkétlen}(t)$

[174] $\underline{\text{ért-1-ut}}(k, t) =$

$\text{is-értékadó-ut}(t)$	$\longrightarrow$	$\underline{\text{p-értad-értelmezés}}(t)$
$\text{is-goto-ut}(t)$	$\longrightarrow$	$\underline{\text{is-goto-értelmezés}}(k, t)$
$\text{is-input-ut}(t)$	$\longrightarrow$	$\underline{\text{input-értelmezés}}(t)$
$\text{is-output-ut}(t)$	$\longrightarrow$	$\underline{\text{output-értelmezés}}(t)$
$\text{is-P-ut}(t)$	$\longrightarrow$	$\underline{\text{P-értelmezés}}(k, t)$
$\text{is-V-ut}(t)$	$\longrightarrow$	$\underline{\text{V-értelmezés}}(t)$

ahol: $\text{is-p-cimkétlen}(t)$

[175] $\underline{\text{P-értelmezés}}(k, t) =$

$\text{is-egyszerű-változó}(s\text{-par}(t))$	$\longrightarrow$	$\underline{\text{P-ért-1}}(k, s\text{-par}(t))$
$\text{is-indexes-változó}(s\text{-par}(t))$	$\longrightarrow$	$\underline{\text{P-ért-2}}(k, s\text{-par}(t))$

[176] $\underline{\text{P-ért-1}}(k, t) =$

$\text{VÁRJ} \longrightarrow [t.s\text{-jel}.s\text{-adatok}(\xi) > 0 \longrightarrow \underline{\text{érték-csökkent-1}}(t)$	
$\text{TRUE} \longrightarrow \underline{\text{P-ért-1}}(k, t)]$	

[177] $\underline{\text{P-ért-2}}(k, t) =$

$\text{is-egész-konst}(s\text{-index}(t))$	$\longrightarrow$	$\underline{\text{P-ért-3}}(k, s\text{-tv}(t), s\text{-index}(t))$
$\text{is-azonosító}(s\text{-index}(t))$	$\longrightarrow$	$\underline{\text{P-ért-3}}(k, s\text{-tv}(t), a) ;$
		$a: \underline{\text{megkeres}}(s\text{-index}(t))$

$$\begin{aligned}
 [178] \quad & \underline{P\text{-}\acute{e}rt\text{-}3(k, t, b)} = \\
 & \text{V}\acute{A}R\text{J} \longrightarrow [\text{elem}(b).t.s\text{-jel}.s\text{-adatok}(\xi) > 0 \longrightarrow \underline{\acute{e}rt\acute{e}k\text{-cs\acute{o}kkent\text{-}2}(t, b)} \\
 & \text{TRUE} \longrightarrow \underline{P\text{-}\acute{e}rt\text{-}3(k, t, b)}]
 \end{aligned}$$

$$\begin{aligned}
 [179] \quad & \underline{\acute{e}rt\acute{e}k\text{-cs\acute{o}kkent\text{-}1}(t)} = \\
 & t.s\text{-tul}.s\text{-adatok}(\xi) \neq \text{SEM} \longrightarrow \text{error} \\
 & \text{TRUE} \longrightarrow s\text{-jel}.s\text{-adatok} \leftarrow \mu(s\text{-jel}.s\text{-adatok}(\xi); \langle t:\acute{e}rt_t - 1 \rangle) \\
 & \text{ahol: is-v\acute{a}ltoz\acute{o}}(t) \text{ \acute{e}s} \\
 & \quad \acute{e}rt_t \stackrel{\text{def}}{=} t.s\text{-jel}.s\text{-adatok}(\xi)
 \end{aligned}$$

$$\begin{aligned}
 [180] \quad & \underline{\acute{e}rt\acute{e}k\text{-cs\acute{o}kkent\text{-}2}(t, b)} = \\
 & s\text{-el\acute{t}ip}.t.s\text{-tul}.s\text{-adatok}(\xi) \neq \text{SEM} \longrightarrow \text{error} \\
 & \text{TRUE} \longrightarrow t.s\text{-jel}.s\text{-adatok} \leftarrow \mu(t.s\text{-jel}.s\text{-adatok}(\xi); \\
 & \quad \langle \text{elem}(b): \acute{e}rt_t - 1 \rangle) \\
 & \text{ahol: } \acute{e}rt_t \stackrel{\text{def}}{=} \text{elem}(b).t.s\text{-jel}.s\text{-adatok}(\xi)
 \end{aligned}$$

$$\begin{aligned}
 [181] \quad & \underline{V\text{-}\acute{e}rtelmez\acute{e}s}(t) = \\
 & \text{is-egyszer\acute{u}-v\acute{a}ltoz\acute{o}}(s\text{-par}(t)) \longrightarrow \underline{V\text{-}\acute{e}rt\text{-}1}(s\text{-par}(t)) \\
 & \text{is-indexes-v\acute{a}ltoz\acute{o}}(s\text{-par}(t)) \longrightarrow \underline{V\text{-}\acute{e}rt\text{-}2}(s\text{-par}(t))
 \end{aligned}$$

$$\begin{aligned}
 [182] \quad & \underline{V\text{-}\acute{e}rt\text{-}1}(t) = \\
 & t.s\text{-tul}.s\text{-adatok}(\xi) \neq \text{SEM} \longrightarrow \text{error} \\
 & \acute{e}rt_t = 0 \ \& \ t.s\text{-sem} \neq \text{NIL} \longrightarrow \underline{\acute{e}breszt\acute{e}s}(t) \\
 & \text{TRUE} \longrightarrow s\text{-jel}.s\text{-adatok} \leftarrow \mu(s\text{-jel}.s\text{-adatok}(\xi); \langle t:\acute{e}rt_t + 1 \rangle) \\
 & \text{ahol: } \acute{e}rt_t \stackrel{\text{def}}{=} t.s\text{-jel}.s\text{-adatok}(\xi)
 \end{aligned}$$

$$\begin{aligned}
 [183] \quad & \underline{V\text{-}\acute{e}rt\text{-}2}(t) = \\
 & \text{is-eg\acute{e}sz-konst}(s\text{-index}(t)) \longrightarrow \underline{V\text{-}\acute{e}rt\text{-}3}(s\text{-tv}(t), s\text{-index}(t)) \\
 & \text{is-azonosit\acute{o}}(s\text{-index}(t)) \longrightarrow \underline{V\text{-}\acute{e}rt\text{-}3}(s\text{-tv}(t), a); \\
 & \quad a: \underline{\text{megkeres}}(s\text{-index}(t))
 \end{aligned}$$



- [184] V-ért-3(t,b)=
 $s\text{-el tip.t.s-tul.s-adatok}(\xi) \neq \text{SEM} \longrightarrow \text{error}$
 $\text{ért}_t = 0 \wedge \text{elem(b).t.s-sem} \neq \text{NIL} \longrightarrow \text{ébresztés}(\text{elem(b).t})$
 $\text{TRUE} \longrightarrow t.s\text{-jel.s-adatok} \leftarrow \mu(t.s\text{-jel.s-adatok}(\xi); \langle \text{elem(b): } \text{ért}_t + 1 \rangle)$
 ahol: $\text{ért}_t \stackrel{\text{def}}{=} \text{elem(b).t.s-jel.s-adatok}(\xi)$
- [185] párh-mut-növelés(k) =
 $s\text{-párh-mutató} \leftarrow \mu(s\text{-párh-mutató}(\xi); \langle \text{elem(k): } pm_k + 1 \rangle)$
 ahol: $pm_k \stackrel{\text{def}}{=} \text{elem(k).s-párh-mutató}(\xi)$