Christian Blume – Wilfried Jakob

Ipari robotok programozási nyelvei

MŰSZAKI KÖNYVKIADÓ, BUDAPEST, 1987

Előszó

Az ipari robotokról eddig megjelent könyvek a programozás, ill. a programozási nyelvek kérdéseit csupán érintőlegesen tárgyalják. Hiányzik az egyes programozási nyelvek teljesítőképességének, ill. szolgáltatási jellemzőinek összehasonlítása is. Könyvünkben ezért az erre a célra külön kialakított rendszerező elvek segítségével összehasonlítjuk az ipari robotok vezérlésére alkalmazott nyelveket. Öt programozási nyelvet hasonlítunk össze ezzel az eljárással. Ezeket úgy választottuk ki, hogy az egyes nyelvek alapját képező célkitűzésekben lényeges eltérések legyenek felismerhetők. A kiválasztott nyelvek a következők:

- AL az ALGOL magasabb szintű nyelvből kifejlesztett nyelv,
- VAL egy speciálisan az ipari robotok programozására kifejlesztett nyelv,
- HELP magasabb szintű, robotvezérlésekre és real-time (azonos idejű) programozásra alkalmas programnyelvek elemeinek keveréke,
- SIGLA igen egyszerű, a gépi szinthez közel álló nyelv,
- ROBEX az NC-programozáson alapuló nyelv.

Az AL nyelvre vonatkozó ismereteinket elsősorban a Karlsruhei Egyetemen e nyelv implementálása során szereztük. Ez a magyarázata az AL nyelven írt, részletesen kidolgozott programpéldák bemutatásának is. A felsorolt öt programnyelvet minden előítélet és részrehajlás nélkül egyforma részletességgel kívántuk bemutatni. Az egyes nyelvek specifikus szolgáltatásait annak alapján ítéljük meg, hogy milyen mértékben érik el az adott nyelv elé kitűzött célokat. Ezután alkalmazási példákat közlünk.

Könyvünk nem tekinthető a tárgyalt nyelvek elsajátítására alkalmas tankönyvnek, sokkal inkább az ipari robotok programozásának specifikus problémái és eljárásai iránt kívántunk érdeklődést ébreszteni, és egyben tökéletesíteni szeretnénk a programozók érzékét az egzakt és módszeres programozási eljárások iránt. A robotrendszerek fejlesztői és alkalmazói általában jól ismerik az ipari robotok programozását, az algoritmusok szisztematikus megszerkesztésének és megfogalmazásának technikáját azonban már kevésbé. Az információfeldolgozással foglalkozó szakember ezzel szemben ismeri az utóbbi területeket, nem tájékozott azonban a robottechnológia speciális problémáit és követelményeit illetően. Könyvünk ezért az említett szakterületek közötti interdiszciplináris területen való jobb eligazodást segíti elő, hogy a két fél – az ipari robotokat alkalmazó technológus és az információfeldolgozó szakember – azonos fogalmakban gondolkozhasson, és így a partner iránti megértéssel tökéletesebb kommunikációra legyen képes. Jóllehet a könyv súlypontja az információfeldolgozás felé tolódott el, reméljük, hogy mindkét fél hasznosan forgatja majd a könyvet, amellyel újabb gondolatokat sikerül ébresztenünk a további fejlesztésekhez is.

A könyv megírásának alapjául szolgált egyrészt *Christian Blume:* Ipari robotok programozásának nyelvei és rendszerei c. előadása, ezenkívül az említett AL implementáció, majd a VAL rendszer elemzése és számos más robotprogramozási nyelv tanulmányozása során nyert tapasztalat és szakismeret. A hagyományos programozásnál megszokott

fogalmakat, mint pl. a vezérlésátadásokat, az alprogramokat, eljárásokat és ezek szinkronizálását, valamint az adatstruktúrákat az ipari robotokra jellemző specifikus követelmények alapos figyelembevételével tárgyalja *Wilfried Jakob*, e fejezetek szerzője. Az ipari robotokra, végrehajtó szervekre és érzékelőkre, valamint a betanítási eljárás integrációjára, a geometriai adattípusokra és műveletekre vonatkozó leírást, valamint az alapfogalmakat tárgyaló fejezeteket *Christian Blume* dolgozta ki.

Köszönetünket fejezzük ki az Aacheni Műszaki Egyetem szerszámgép-laboratóriumának, amiért készségesen közreműködtek a náluk kifejlesztett ROBEX nyelvre vonatkozó részletkérdések megválaszolásában. A VAL, a HELP és a SIGLA nyelveket jellegzetes módon maguk a robotelőállítók fejlesztették ki. Köszönetünket fejezzük ki az Unimation, a Digital Equipment Automation (DEA) és az Olivetti cégeknek, amiért készségesen rendelkezésünkre bocsátották a nyelvleírásokat. Egy megjelenítő rendszernek szóló utasításrendszerrel kapcsolatban bemutatunk egy részletet a RAIL nyelvből. Ezzel kapcsolatban a megfelelő szakmai dokumentumok átadásáért az Automation cég képviselőit illeti köszönet.

Nem utolsósorban köszönetünket szeretnénk kifejezni a Karlsruhei Egyetem számítógépes folyamatirányítási tanszékén *Dr. Ing. U. Remhold* professzornak, a Karlsruhei Magfizikai Kutatóközpont PFT kutatási témafelelősének, valamint az aschaffenburgi PSI vállalatnak a munka nagyvonalú támogatásáért.

Karlsruhe
Aschaffenburg

Christian Blume
Wilfried Jakob

Tartalomjegyzék

Előszó ... 5

1. Bevezetés ... 11

2. Alapismeretek ... 13
2.1. Alapfogalmak ... 13
2.2. Felfogások az információfeldolgozásban ... 19
2.2.1. Változók ... 19
2.2.1.1. A változók kezelése a programozó részéről ... 19
2.2.1.2. A változók kezelése a program futása alatt ... 20
2.2.1.3. A változók címének meghatározása ... 20
2.2.1.4. A változók címeinek ábrázolása ... 21
2.2.2. A veremtárolás elve ... 22
2.2.2.1. A képletek átírása fordított lengyel jelöléssel ... 23
2.2.2.2. Fordított lengyel jelölésben felírt képletek kiértékelése operandusverem segítségével ... 23
2.2.2.3. Képletek gépi kódolásának előállítása ... 25
2.2.3. A blokkszervezés ... 26
2.2.3.1. A változók érvényességi tartománya és élettartama ... 26
2.2.3.2. Blokkszervezés és a tárkezelés kérdése ... 27
2.2.3.3. A verem elv szerinti tárkezelés ... 30
2.2.4. Alprogramok, eljárások, függvényeljárások és makrók ... 30
2.2.4.1. Az alprogramok ... 30
2.2.4.2. Eljárások, függvényeljárások ... 31
2.2.4.3. A makroeljárások ... 34
2.2.5. Az alprogramok rekurzív hívása ... 35
2.2.6. Folyamatok, taskok és társ-rutinok ... 40
2.2.7. A szinkronizált feldolgozás ... 41
2.3. Programozási nyelvek szintaktikai szabályai ... 44
2.3.1. A nyelvtan szerepe ... 44
2.3.2. A Backus–Naur forma (BNF) ... 47
2.3.3. A szintaxisdiagram ... 51
2.4. A nyelv kialakításának főbb szempontjai az ipari robotok programozásánál ... 51
2.4.1. A „Frame" kísérőkoordináta-rendszer fogalma ... 52
2.4.2. A koordinátatranszformáció és a pályatervezés ... 57
2.4.3. Ipari robotok vezérlési módjai ... 70
2.4.4. Nagyszámítógépeken alkalmazott programozási nyelvek ... 76

2.4.5. Robotfüggő programozási nyelvek ... 77
2.4.6. NC-gépek programozása ... 78
2.4.7. Munkatervek, közhasználatú nyelvek ... 79
2.4.8. A környezeti modell ... 81

3. Adatok, adattípusok ... 83
3.1. Az adatobjektumok ... 83
3.1.1. A típusdeklaráció ... 85
3.1.2. Konstansnevek deklarálása ... 91
3.1.3. A standard adattípusok ... 92
3.1.4. A geometriai adattípusok ... 94
3.1.5. Strukturált adattípusok (Adatszerkezetek) ... 97
3.1.5.1. Az adattömbök ... 97
3.1.5.2. Rekordok (adategyüttesek, kapcsolt adatmezők) ... 98
3.1.5.3. Fájlok ... 100
3.1.6. A pointer-típus és a környezetleíró modell ... 102
3.2. Az adatkezelés ... 108
3.2.1. Műveletek ... 108
3.2.1.1. Aritmetikai műveletek ... 108
3.2.1.2. Geometriai műveletek ... 110
3.2.1.3. Összehasonlítási műveletek ... 114
3.2.1.4. Logikai műveletek ... 114
3.2.2. Standard függvények ... 115
3.2.3. Komplex kifejezések ... 117
3.3. Értékadó utasítások ... 119
3.3.1. Konstans értékadás ... 119
3.3.2. Beviteli utasítások ... 120
3.3.2.1. Adatbevitel párbeszédes technikával (interaktív módszer) ... 120
3.3.2.2. A robot aktuális pozíció- és orientációadatainak bevitele ... 122
3.3.3. Értékadás pointer típusú változónak ... 124
3.4. Szövegek és számadataok kiviteli utasításai ... 125

4. Utasítások ... 127
4.1. A környezeti modellt kezelő utasítások ... 127
4.2. Mozgásvezérlő utasítások ... 133
4.2.1. Implicit mozgásvezérlő utasítások ... 134
4.2.2. Explicit mozgásvezérlő utasítások ... 135
4.2.3. Egyszerű mozgásvezérlő utasítások ... 136
4.2.4. Mozgásvezérlő utasítások paraméterezéssel ... 143
4.2.5. Az érzékelők jeleinek felhasználása a robot mozgásvezérlő utasításainál ... 153
4.2.5.1. A mozgásvezérlő utasítások végrehajtásának ellenőrzése az érzékelők jeleinek segítségével ... 154
4.2.5.2. A mozgásvezérlő utasítások paramétereinek szabályozása az érzékelők jelei alapján ... 159
4.2.6. Mozgásvezérlő utasítások eseményfelügyelet mellett ... 160
4.2.7. Mozgásvezérlő utasítások időellenőrzés (időfelügyelet) mellett ... 165
4.2.8. A robot alaphelyzetbe állítása ... 166

4.3. A végrehajtó szervekre vonatkozó utasítások ... 167
4.3.1. Egyszerű effektorvezérlő utasítások ... 169
4.3.2. Paraméteres effektorvezérlő utasítások ... 171
4.3.3. Az érzékelők jeleinek felhasználása effektorvezérlő utasításoknál .. 173
4.3.4. Szabályozott paraméterű effektorvezérlő utasítások ... 174
4.3.5. Effektorvezérlő utasítások eseményfelügyelettel ... 174
4.4. A robotmozgás, ill. az effektorműködtetés leállítása ... 175
4.5. Érzékelőkkel kapcsolatos utasítások ... 176
4.5.1. Érzékelők jeleitől függő programelágaztatás ... 177
4.5.2. Mérési adatok beolvasása ... 181
4.5.3. Határértékfigyelés ... 182
4.5.4. Vezérlő utasítások vizuális rendszerek esetén ... 183
4.6. A blokkszervezés és utasítássorozatok ... 188
4.7. Futtatásvezérlő utasítások ... 189
4.7.1. Programelágaztatások ... 192
4.7.1.1. Feltételes vezérlésátadás ... 194
4.7.1.2. Programelágaztatás ... 198
4.7.2. Ciklusszervezés ... 201
4.7.2.1. Ciklusszervezés ciklusváltozóval ... 202
4.7.2.2. Feltételtől függő ciklus ... 204
4.7.3. Szinkronizációs utasítások ... 204
4.7.4. Várakozási utasítások ... 207
4.8. Rendszerkapcsolók ... 208
4.9. Kivételes helyzetek kezelése ... 209

5. A betanítási eljárás a programírásban ... 213

6. Alprogramok, eljárások és függvényeljárások ... 219
6.1. Alprogramok ... 220
6.2. Eljárások ... 222
6.3. Függvényeljárások ... 226
6.4. Rekurzív eljárások és függvényeljárások ... 230

7. Eljárások időbeli koordinálása ... 235
7.1. Párhuzamos blokkok ... 235
7.2. Task-kezelés ... 237
7.3. Társrutinok ... 239
7.4. Paralel feldolgozás a VAL, a SIGLA és a ROBEX nyelvekben ... 240

8. A programfejlesztő és a futtatásvezérlő rendszerek ... 241
8.1. Az editor ... 242
8.2. A compiler-program (fordítóprogram) és a processzor ... 243
8.3. Interaktív összetevő ... 250
8.4. A futtatásvezérlő rendszer ... 251
8.4.1. Az interpreter ... 254
8.4.2. A pályavezérlés ... 255
8.5. A szimulátor és a program ellenőrzése ... 257

Általános felhasznált és ajánlott irodalom 261
Függelék 265
Speciális irodalom az egyes fejezetekhez 266
Tárgymutató 268

1. Bevezetés

A legutóbbi időben a termelés és a tömeggyártás egyre több területén alkalmaznak ipari robotokat, így pl. munkadarabok mozgatása, hegesztés, festés stb. során. Ezen feladatok egy részénél a jövőben is megfelelnek majd az egyszerűbb programozási technikák, így pl. a beállítási és betanítási eljárások. A teljes szerelés és a komplex anyagmozgatás terén azonban elengedhetetlen a fejlettebb programozási eljárások és nyelvek használata. Ezért világszerte új programozási nyelvek és programrendszerek kidolgozásán fáradoznak, amelyeknél a fő célkitűzés elsősorban azc érzékelők alkalmazásának és az egyszerű rugalmas programozás feltételeinek megteremtése.

A betanítási eljárással (teach-in) összehasonlítva ugyanis a programozási nyelvek olyan alapvető előnyöket nyújtanak, mint a jó dokumentálhatóság, közérthetőség, korrigálhatóság és az off-line programozás. Az NSZK-ban pl. – egymástól eltérő koncepcióból kiindulva ugyan – mind a Karlsruhei Egyetemen, mind pedig az Aacheni Műszaki Egyetemen fejlesztenek ipari robotok számára magasabb szintű nyelveket.

Mivel könyvünkben e helyen magukat a programozási nyelveket tárgyaljuk, így az informatika területére tartozó problémák tárgyalásának nagyobb teret szánunk, mint az ipari robotok alkalmazásához nélkülözhetetlen robottechnológiai alapok bemutatásának. Elöljáróban azonban a robottechnológia néhány alapfogalmát is megvilágítjuk, ami elsősorban az információfeldolgozó szakemberek számára lehet hasznos. Elsősorban az adatok kezelése kapcsán bemutatjuk a vezérlés oldaláról támasztott speciális követelményeket és a geometriai ábrázolásból kiindulva új adattípusokat vezetünk be. A különféle nyelvek adatstruktúráinak sokrétűségéből és merevségéből adódó nehézségek elkerülése érdekében egy olyan rugalmas adatdefiniálási eljárást választottunk, amely a PASCAL nyelvben előforduló adatstruktúrával áll szoros kapcsolatban.

Rugalmassága, világos szerkezete és széles körben elterjedt volta miatt mintának tekintettük a PASCAL nyelvet, melynek egyre több híve van, és amelyet egyébként több mikroszámítógépre is implementáltak már. Főleg az érzékelők adataival kapcsolatban használható ki az a lehetőség, hogy a rendszerprogramozó új adattípus definiálásával leegyszerűsítheti a különféle érzékelőkkel kapcsolatos adatillesztési feladatokat, mivel ezek logikai struktúrájára nézve még eddig semmilyen egységes szabvány nem alakult ki. A könyvben ez a fejezet így a jövőbeli fejlesztések egyfajta előtanulmányaként fogható fel.

2. Alapismeretek

Ebben a fejezetben a robottechnológia és az információfeldolgozás alapvető fogalmait és célkitűzéseit ismertetjük. Ennek során olyan, a programozók számára már ismert alapfogalmakat is bemutatunk, mint pl. a változókat, alprogramokat stb., hogy az irányítástechnikusok vagy a gyártás területén működő fejlesztőmérnökök is könnyebben bekapcsolódhassanak a programozási eljárásokba. Másfelől viszont a különféle robotprogramozási nyelvek megértéséhez hasznosnak ígérkezik az alapul vett műszaki elgondolások megvilágítása is.

Az informatika egyik részletesen vizsgált problémaköre a nyelvek formális, szintaktikai leírása. Ezen azokat a módszereket és sémákat értjük, amelyek segítségével valamely programnyelv alapszókészletét és ezek formailag helyes használatát be lehet mutatni. Mivel az egyes robotnyelvek ezen módszerek segítségével mutathatók be, a szintaxis leírásának külön fejezetet szentelünk, amely a formális nyelvek elméletének területére vezeti be az Olvasót.

2.1. Alapfogalmak

Mivel az itt tárgyalt fogalmak legtöbbje alapvető jelentőségű, nem elégszünk meg egy formális leírással, hanem a fogalmakat részletesebben magyarázzuk, majd egyszerű példákon keresztül is megvilágítjuk.

Sajnos az egyik legfontosabb fogalomnak, az **ipari robotok** fogalmának még nincs egzakt definíciója. A tudományos-fantasztikus irodalomban használt robot fogalmától való megkülönböztetés kedvéért választott ipari robot elnevezés sem tekinthető általánosan elfogadottnak. Használatosak ezen kívül még a *robot,* a *manipulátor,* ill. az *univerzális mozgatóberendezés,* sőt használják a *robotmanipulátor* fogalmat is.

A német VDI 2860-as szabványtervezetben (az ún. zöldnyomatban) az ipari robot fogalmát a következők szerint definiálják:

Az ipari robotok univerzálisan alkalmazható, több tengely körüli elforgatást lehetővé tevő műveletvégző automaták, amelyeknél az egyes mozgások, a helyzetbeállítási sorrend, az elmozdulási úthosszak, ill. a szögek szabadon programozhatók (tehát mechanikus beavatkozás nem szükséges), ill. adott esetben külső érzékelők által vezérelt, azaz szabályozott mozgást valósítanak meg.

Az ipari robotokra különféle megfogószerkezetek, szerszámok vagy más gyártóeszközök is felszerelhetők, de megoldhatunk velük munkadarab-mozgatási és/vagy gyártási feladatokat is.

Ezzel kapcsolatban fontos kiemelni, hogy olyan programozható készülékről van szó, amely motorok és forgástengelyek segítségével különféle mozgásokat képes megvalósítani a háromdimenziós munkatérben. Azt, hogy a programozás mechanikus beavatkozás nélkül történik, az informatikában magától értetődőnek tekintik, vannak azonban egyszerű adagolókészülékek, amelyek végállásjelzők felszerelésével a végállási helyzet elér-

sére programozhatók. Az ipari robotokhoz ezenkívül érzékelőket is csatlakoztathatunk, melyek gyakorlatilag perifériaként működnek. Az **érzékelők** olyan készülékek, amelyek az érzékszervekhez hasonlóan fizikai jeleket érzékelnek (így pl. fényhullámokat, a csuklós karok nyomását vagy helyzetét), ezeket regisztrálják, elektromos vagy digitális jelekké alakítják és a robot vezérlő számítógépe felé továbbítják. Az érzékelőtől származó jelek ezután kiértékelhetők, és a program menetének megfelelően módosíthatók. A robot karjának végére megfogószerkezet, szerszám vagy valamilyen más gyártóeszköz szerelhető, amelyeket összefoglaló néven **végrehajtó szerveknek** nevezünk. Ezeknek a végrehajtó szerveknek szintén vezérelhetőknek és érzékelőkkel irányíthatóknak kell lenniük. A *2.1. ábra* három tipikusnak tekinthető ipari robotot mutat be.

Az ipari robotok mechanikai felépítésüknél fogva végrehajtó szerveik segítségével különféle **műveletek** sokoldalú elvégzésére alkalmasak. Végrehajthatnak különféle beállításokat, amelyek vezérlése függhet az érzékelőktől, érzékelővezérléssel vagy anélkül működtethetnek megfogóeszközöket vagy szerszámokat, és/vagy módosíthatják a programmenetet.

Az ipari robotok külső kialakítása már a konstrukció fázisában is nagy jelentőségű kérdés – ugyanez áll a *méretekre* és a *geometriára* is –, mivel ezek határozzák meg a *munkateret* és annak lehetőségét, hogy tetszőleges helyzetek beállíthatók legyenek, valamint hogy a végrehajtó szervet tetszőleges irányba lehessen állítani. Ennek biztosításához hat szabadságfokra van szükség.

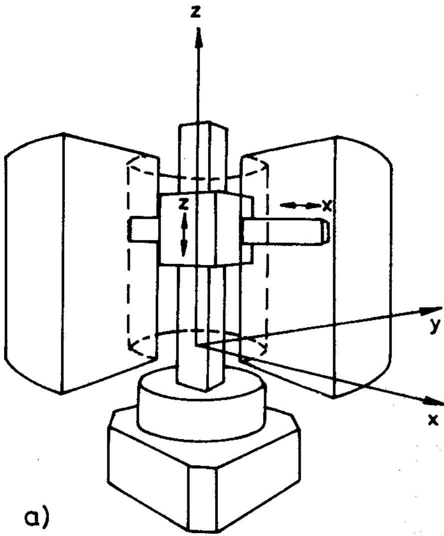
Szabadságfokon egy olyan független mozgástengely meglétét értjük, amelynek mozgása nem állítható elő a többi mozgástengely valamilyen összetett mozgásából. Az ipari robotok **mozgástengelyét** a hajtás, a hajtómű és a csukló kombinációja alkotja.

Az ipari robotok egyes *csuklói* alkalmasak lehetnek egyenesvonalú – más szóval transzlációs – elmozdulás végrehajtására, vagy forgó- – más néven rotációs – mozgás végrehajtására. Az elmozdulást ennek megfelelően az elmozdulási úthossz, vagy szög megadásával írhatjuk elő. Bonyolult műveletek közvetlen – más szóval direkt – programozása a több mozgástengelyű robotok esetében ezért az egyes csuklók szögelfordulásának és az eltolási úthosszak egyenként történő megadása miatt elég fáradságos, hiszen az egyes szögérték-megváltozásoknak a végrehajtó szerv mozgására gyakorolt eredő hatását a programozó már nem tudja minden további nélkül kiszámítani.

Ezért a karok mozgását először a könnyebben kezelhető derékszögű koordináta-rendszerben adjuk meg, ezeket az adatokat azután átszámítjuk a megfelelő csuklókhoz tartozó szögelfordulás- és egyenes vonalú elmozdulásértékekre. Az átszámítást **koordinátatranszformáció** segítségével hajtjuk végre. Hasonlóképpen a robot helyzetéről a helyzetértékelők által szolgáltatott koordináták átszámítása derékszögű koordináta-rendszerbe is koordinátatranszformációval végezhető.

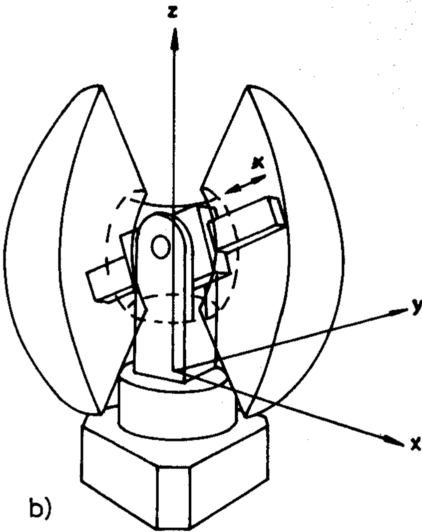
Az ipari robotoknak saját geometriai viszonyaival és a különféle mozgásokkal kapcsolatos problémái a **kinematika** – azaz a mozgástan – keretében tárgyalhatók. Ennek során mindazokat az erőket, amelyek a mozgással kapcsolatban fellépnek, valamint a mozgatott testek tömegét figyelmen kívül hagyjuk. A közelítés általában elegendőnek bizonyul, azonban a robot vezérléséhez már **dinamikus** modellre is szükség van, hogy ennek segítségével lehessen a robotot a pályatervezés során meghatározott térbeli görbe mentén vezérelni. Ez már a robotra ható összes erőt figyelembe veszi, így a tehetetlenségi, a gravitációs, a centrifugális és a *Coriolis*-erőket is.

Ipari robotok programozhatók programszöveg használatával vagy anélkül, mivel a tisztán **mozgásprogramozásra** redukáló eljárásokhoz nincs szükség feltétlenül programozási nyelvre.

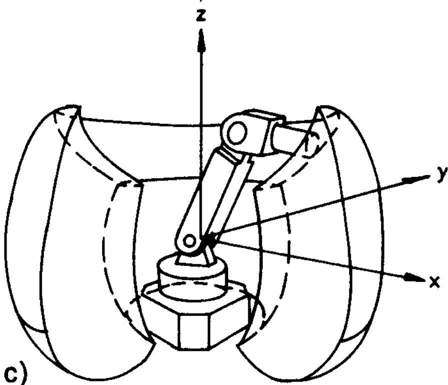


2.1. ábra. Ipari robotok munkatere (működési tartománya)

a) Ipari robot munkatere két transzlációs (egyenes vonalú elmozdulást biztosító csúszka) és egy rotációs (forgási) tengely esetén; 2T + 1R;



b) Ipari robot munkatere egy transzlációs (egyenes vonalú elmozdulást biztosító csúszka) és két rotációs (forgási) tengely esetén (1T + 2R);



c) Ipari robot munkatere három rotációs (forgási) tengely esetén (3R, ún. humanoid)

A mozgásprogramozás eljárásai a következők:
Kézi programozás (végállásjelzők beállítása).
Beállítási eljárás (a robotkart hajtás nélkül a pálya egy pontjában előírt helyzetbe, ún. pozícióba és irányba állítjuk).
Követőprogramozás (a robotkart a hajtás segítségével mozgatjuk a pálya előzetes bejárása céljából).
Master–slave-programozás (a programozó a kicsiny vezető robotkart (master-t) vezeti, amelynek mozgásai átadódnak a nagy, az ún. másoló karra).

A programozás ilyenkor abból áll, hogy a robotkarral bejárjuk a pálya egyes pontjait, az ennek során generált programkód pedig általában csak a pályapontok koordinátáiból áll. Kizárólag **betanítási (teach-in) eljárás** segítségével állíthatók be a tiszta mozgásprogramozáson túlmenő adatok, így pl.:

- a mozgás sebessége,
- a mozgás időtartama,
- a programszünetek,
- egyszerű programhurkok,
- megszakítás esetén egyszerű programelágazások (az érzékelők jeleitől függően),
- különleges funkciók beállítása (pl. a vezérlés módja, lépésköz, osztás).

Ehhez a programozónak le kell nyomnia a megfelelő funkcióbillentyűt. A numerikus paraméterek (pl. időtartam) megadásához meg kell adni a választott konstans számjegyeinek sorozatát. A betanítási eljárás tehát már tartalmaz egy bizonyos szövegrészt is. Mivel minden egyes funkciót, ill. minden egyes parancsot egy-egy megfelelő billentyű lenyomásával választunk ki, így nincs is lényeges elvi különbség a betanítási eljárás és a szövegszerű programozás között, amelynél az egyes műveleteket és adatokat szimbolikusan, azaz jelsorozatok formájában adjuk meg. Az egyes szimbólumok összessége az adott programnyelv **szókészletét** jelenti, azok a nyelvtani szabályok pedig, hogy miképpen lehet az egyes nyelvi elemeket egymással kombinálni, képezik a nyelv ún. **szintaxisát.**

A programozási nyelvek ilyen értelemben vett szintaxisa bizonyos mértékig korlátok közé van szorítva. A szintaxist formális módon definiálják (l. a 2.3. szakaszban). Ennek során figyelmen kívül marad az, hogy mi a szimbólumok vagy ezek kombinációinak jelentése, más néven a nyelv **szemantikája.**

Fontos szempont az is, ahogyan a nyelvet gyakorlatilag alakítjuk, használjuk. Ezt nevezzük a nyelv **pragmatikájának.** Ennek jelentőségét egy példán keresztül világítjuk meg. Egy nagyon egyszerű nyelv nyelvtana álljon az alábbi szókészletből
V = (az autó, a rózsa, nagy, piros, gyors), szintaxisa pedig álljon a következő nagyon egyszerű szabályokból:

1. A szókészlet két eleme egy mondatot alkot.
2. A mondat első eleme vagy „az autó", vagy „a rózsa" szimbólum lehet.
3. A mondat második eleme a „nagy", a „piros" vagy a „gyors" szimbólumok egyike.

Így a nyelvből hat mondat állítható össze:

1. Az autó nagy.
2. Az autó piros.
3. Az autó gyors.
4. A rózsa nagy.
5. A rózsa piros.
6. A rózsa gyors.

A kiválasztott szókészlet egyben a magyar nyelvnek is része, így magától értetődő, a nyelv szemantikája tehát ismert. Éppen emiatt azt az egyébként szintaktikusan helyes mondatot, hogy „A rózsa gyors", értelmetlennek érezzük, hiszen nincs olyan rózsa, amelynek az lenne a tulajdonsága, hogy gyors. Emiatt módosíthatnánk a nyelvtan előbb felírt harmadik szabályát, éspedig így:
A mondat második eleme

a) a **nagy, piros** vagy a **gyors** szimbólumok egyike, ha az első elem **az autó** szimbólum,
b) ha pedig a mondat első eleme **a rózsa,** akkor a második elem a **nagy, piros** szimbólumok valamelyike.

Ezen a példán láthatjuk, hogy valamilyen *szemantikai* – azaz *értelmezés szerinti* – korlátozás *szintaktikai* – azaz *nyelvhelyességi* – szabályként is megfogalmazható, a nyelv szintaxisa és szemantikája közötti határ tehát elmosódik. Ezenkívül a módosított 3. szabály arra is rámutat, hogy valamely mondat elemeinek kiválasztása nem független egymástól, hanem a második elem kiválasztása a megelőző szöveggel függ össze, esetünkben az első elemmel.

Az iménti példában használtuk a jelek és szimbólumok fogalmát. Jóllehet ezek a fogalmak ismertek, definiáljuk őket pontosabban. A DIN 44300 szabvány szerint a jel fogalmán a következőket értjük: „A jel a különböző elemek egyezményes véges halmazából vett egyetlen elem. A halmazt **jelkészletnek** nevezzük."
Példák jelkészletekre:

- Az iskolai osztályzatok lehetséges értékeinek halmaza.
- Az adatfeldolgozásban használt ASCII-kód.

Az olyan jelkészletet, amelyben valamennyi jelre nézve meghatározott sorrendiség is érvényes, ABC-nek nevezzük. Meg kell azonban különböztetnünk magát a jelet annak jelentésétől. A ↷ jel jelentheti azt, hogy „jobbra fordítsd" vagy pl. azt, hogy „körkörös fel-le mozgás a levegőben".

A jelhez mindig hozzá tartozik annak jelentése. A jel jelentésével együtt **szimbólumot** alkot. A szimbólum jelentése független az alkalmazott nyelvtől, így pl. ↷ az angolban jelentheti a *„turn right"*-ot is. A szimbólum több jelből is állhat, ekkor alapszónak nevezzük. Az egyes programnyelvek különféle alapszavakat tartalmaznak, pl. BEGIN, GOTO vagy pl. MOVE. Egy jelnek több jelentése is lehet. Így pl. az AL nyelvben a "*" jelnek a következő jelentései lehetnek:

- skalárok szorzása ARÁNYOSSÁGI TÉNYEZŐ * HOSSZÚSÁG,
- vektor skalárral való szorzása 5 * V,
- vektoriális szorzás V1 * V2,
- vektorok elforgatása R * V,
- vektorok transzformációja T * V,
- kétféle rotáció láncolása (összekapcsolása) R1 * R2,
- egy kísérőkoordináta-rendszer ún. „frame" transzformációja T * F,
- két transzformáció láncolása (a transzformáció transzformációja) T1 * T2.

A programozónak ezekre tekintettel kell lennie, mert a programrendszer csak a szövegösszefüggésből ismeri fel az éppen aktuális jelentést, tehát pl. az operandusok típusa alapján. Az operandusok a program által kezelt adatobjektumokhoz tartoznak. Ha az adatok egymástól nem függetlenek, hanem közöttük relációk, kapcsolatok állnak fenn, akkor adatszerkezetről, más szóval *adatstruktúráról* beszélünk. Az adatstruktúrára

egyszerű példa lehet egy lista, amely azonos nemű tárgyak, elemek, objektumok sorba rendezett halmaza. A kapcsolatot itt a következő elemre mutató utalás jelenti (az utolsó listaelemet kivéve), amely megvalósítható pl. a tároló címének megadásával.

A matematikában és az informatikában használt igen fontos és alapvető fogalom továbbá az **algoritmus** fogalma. Ezen általában egy problémakör általános megoldására szolgáló módszert vagy eljárást értünk. Az algoritmusok igen különböző módon írhatók le, akár egészen elvontan, vagy teljes részletességgel. Az információfeldolgozás területén az algoritmusokat programnyelvek segítségével írhatjuk le. Ezek nyelvi felépítése a műveleti lépések (véges) halmazával ábrázolható. Amikor a számítógép végrehajtja a programot, ez az algoritmus egy realizációjának felel meg. Tekintettel kell lenni azonban a következő korlátozásokra:

1. A program csak véges sok – és a tárkapacitás által meghatározott számú – lépésből állhat, amelyek teljes egészében meghatározzák az algoritmust.
2. Minden egyes lépés hatása egyértelműen meghatározott, ugyancsak egyértelműen rögzített az is, hogy valamely lépés végrehajtása után melyik a soron következő lépés.
3. Egy műveleti lépés végrehajtása véges hosszúságú időt vesz igénybe. Ez azonban sajnos nem jelenti azt, hogy a teljes algoritmusnak, ill. az egész programnak a végéhez is mindig eljutunk. Ez az eset pl. akkor fordulhat elő, amikor a program – többnyire programozási hiba folytán – végtelen ciklusba kerül. Ezenkívül léteznek pl. olyan utasítások, amelyek végrehajtása bizonyos körülmények között akár végtelen hosszú ideig is eltarthat, pl. ha az utasítás az, hogy várni kell egy jelre, amely azonban sohasem érkezik meg.

További fontos fogalom a **feldolgozási folyamat** (task) fogalma. Ezen egy olyan programrészletet értünk, amely viszonylag önálló funkciókat hajt végre, és amely ezért a programokat futtató fölérendelt vezérlő- (Supervisor) program számára mint legkisebb önállóan kezelhető egység jelenik meg.

Példák a task fogalmára: felhasználói programok, periferiális készülékek adat be/kiviteli rutinjai. Szigorúan véve itt szekvenciális (azaz sorrendi) feldolgozási folyamatról kellene beszélnünk, mivel hallgatólagosan feltételeztük, hogy az egyes futási lépések **időben egymást követő** sorrendben következnek egymás után. Lehetőség van ezzel szemben arra is, hogy több különböző feladat időben párhuzamosan fusson, akár úgy, hogy több processzor áll rendelkezésünkre, akár úgy, hogy csak **látszólagos időbeli párhuzamosság** áll fenn. Ekkor csak egyetlen processzor áll rendelkezésünkre, amelynek számítási idejét azonban az operációs rendszer részét képező felügyelőprogram tetszőleges részekre bonthatja, és ezeket az időegységeket az egyes feldolgozási folyamatok (taskok) között oszthatja szét. Az egyes taskok futásának vezérlésén kívül az **operációs rendszer** feladata továbbá az input/output műveletek vezérlése, az erőforrások szétosztása (perifériák, fájlok, tárolókapacitás) az egyes taskok között, a fájl-kezelés, valamint egyes különleges üzemállapotok lekezelése.

Ez utóbbi csoportba tartozik a megszakítások kezelése. Egy *programmegszakítás* (interrupt) azt jelenti, hogy egy hardver úton érkező megszakításkérelem az éppen futó programot megszakítani képes, annak állapotát letárolja, valamint hogy az erre az esetre előírt feldolgozási folyamat futását elindítja. A megszakításokat kiválthatják egyes készülékek, érzékelők vagy magának a robotnak a biztonsági berendezései és áramkörei. Az operációs rendszer olyan funkciókat bocsát a futtatásvezérlő rendszer rendelkezésére, amely még tartalmazhatja a fordítóprogramot, a robotvezérlőt, valamint a betanítási rendszernek egy interaktív komponensét, amely a robot vezérlő számítógépére van

implementálva. Implementáláson azt a tevékenységet értjük, amikor egy probléma meghatározását és elemzését követően megtervezzük a megoldási algoritmust, megírjuk az algoritmust megvalósító programot és a futásra kész programot a számítógépen letesztel-jük. A futtatásvezérlő rendszer mellett, amely a felhasználói programok végrehajtását vezérli, szükség van még egy fejlesztő rendszerre is, amelyet többnyire egy saját háttérgépre implementálnak. A fejlesztő és a futtatásvezérlő rendszer sajátosságait a 8. fejezetben tárgyaljuk részletesebben.

2.2. Felfogások az információfeldolgozásban

A programozási nyelvek fejlesztésével kapcsolatosan különféle felfogások alakultak ki, amelyek néhány magasabb szintű nyelv alapját képezik. Ezek a koncepciók elsősorban a program adatainak a szervezésére és felügyeletére vonatkoznak, valamint egy komplex programfutási struktúra vázát érintik. Mindez a strukturált programozhatóságot könnyíti meg, és így a programozó számára a programok jól áttekinthetővé válnak. Az itt bemutatott különféle elképzelések az ipari robotok programozási rendszereinek további fejlődését is befolyásolják, bár igaz, hogy csak az AL és bizonyos megszorításokkal a HELP azok a robotnyelvek, amelyek meg is valósítják ezeket az elveket. A ROBEX nyelv pl., amelyet az NC-gépek programozási technikáiból kiindulva fejlesztettek ki, nem kezel változókat, mivel e téren eddig a szerszámgép minden mozgásjellemzőjét a számítógép már a fordítás során pontosan kiszámította, és számkonstansok formájában állította elő. Tervezik azonban, hogy a jövőben ezt a rendszert mgváltoztatják, és így az információfeldolgozás néhány alapvető sajátossága átvehetővé válik.

2.2.1. Változók

Ebben a pontban a *változók* kezelésének kérdéseivel foglalkozunk, mivel sok gépész és szabályozástechnikai szakember a programírási és programfuttatási munkái során alig ismeri a változók kezelését. Ezért a változókat egyrészt a programozó szemszögéből nézve tárgyaljuk, másrészt áttekintjük, hogy hogyan kezeli a gép a változók számára fenntartott tárolóterületet a programfutás ideje alatt.

2.2.1.1. A változók kezelése a programozó részéről

Az ipari robotok programozásakor változókra van szükség, pl. az érzékelők adatainak feldolgozásánál. Változók segítségével az érzékelőről beolvasott érték tárolható és a program különböző helyein kiértékelhető. Ezenkívül aritmetikai műveleteket is végrehajthatunk velük, pl. egy új helyzetbeállítás kiszámításánál. A változókat a programon belül **nevükkel** ábrázoljuk, a név megválasztása tetszőleges: pl. ALFA vagy BIG. A változóhoz azonban általában hozzá tartozik típusának megadása is, amelyet a programozó az ún. **deklaráció** felírásakor határoz meg. Ez tulajdonképpen egy olyan előzetes **megegyezés,** amely után az egyszer már deklarált változót programunkban bárhol felhasználhatjuk, ahol csak szükségünk van rá, hogy valamilyen **segédeszköznek** tetszőleges értékeket adhassunk, amely azt megőrzi, és amelynek aktuális értékét később felhasználhatjuk.

A *2.1. programrészlet*ben először a BOOLEAN logikai típusú **kapcsoló** változónevet, majd a VECTOR típusú **vektor** változónevet deklaráltuk. Az INTEGER (egész) típusú **aérték** és **adat** nevű változók deklarálása után a programban egy konstans értéket, az

5-öt adjuk az **aérték** nevű változónak. Az **adat** nevű változó értékét kívülről (pl. egy terminálról) olvassuk be. Ezzel az **aérték** változó értékét újraszámoljuk, régi értékét az 5-öt az **adat** nevű változónak a futás során éppen ismertté vált értékéhez adjuk hozzá, és ez lesz az **aérték** új aktuális értéke.

```
VAR kapcsoló: BOOLEAN;
VAR vektor: VECTOR;
VAR aérték,
    adat: INTEGER;
    :
    aérték:= 5;
READ (adat);
aérték:= aérték + adat;
```

VECTOR vektor típusú változó
INTEGER egész típusú változó
A „:=" jelölés értékadást jelent (olv.: „legyen egyenlő")

A fordító megjegyzése:
BOOLEAN logikai típusú változó

2.1. programrészlet. Példa a deklaráció műveletére egy hipotetikus programnyelvben

2.2.1.2. A változók kezelése a program futása alatt

Valamely program futása során egy változót egy **tárolóterület** (azaz egy hely) és egy **tárolótartalom** (tehát egy érték) segítségével hozunk létre.

Az aérték := 5;

értékadási művelet azt jelenti, hogy az **aérték** változó címén (mondjuk az 1020-as címen) levő tárolórekeszbe az 5-ös értéket visszük be. (Az **aérték** ←→ és címének egymáshoz rendelését később tárgyaljuk.) A READ (adat);

utasítás végrehajtásakor a programozó a futási idő alatt bead egy értéket egy adatbeviteli készüléken keresztül, amely azután az **adat** nevű változó tárolóterületébe íródik be. Az *INTEGER* típusmegadás segítségével azt közöljük a rendszerrel, hogy egy tizedespont nélküli számot kell átalakítani 16 vagy 32 bites bináris ábrázolású számmá.

Példa: Hajtsuk végre az előző programrészletet, és a READ-utasítás végrehajtásakor a felhasználó adja meg pl. a 12-es számértéket. A memória tartalma az

aérték := adat + aérték;

utasítás végrehajtása előtt:

1020 című „aérték" változó	tartalma 5
1080 című „adat" változó	tartalma 12

Az utasítás végrehajtása után a memória tartalma így alakul:

1020 című „aérték" változó	tartalma 17
1080 című „adat" változó	tartalma 12.

2.2.1.3. A változók címének meghatározása

Egy felhasználói program *fordításakor* a fordítóprogram (idegen néven compiler) meghatározza a változók *relatív címeit.* Ehhez felállít egy *szimbólumtáblázatot,* amelyben felsorolja a változók elnevezéseit, és a hozzájuk tartozó relatív címeket. A relatív címeket egy számláló segítségével határozza meg, ennek értéke az induláskor nullára áll, és minden egyes új változónak a szimbólumtáblázatban való megjelenésekor akkora értékkel növekszik, amekkora helyre az adott változónak szüksége van.

Példa:

```
VAR magas, pl :        INTEGER;
összeg, kamatok :      REAL;
```

A fordítóprogram a következőképpen számítja ki a relatív címeket:

változónév	relatív cím
magas	0
pl	2
összeg	4
kamatok	8

Itt feltételeztük, hogy az INTEGER változó 2 bájtos, a REAL változó pedig 4 bájt hosszúságú és a címzés a bájtokban kifejezett címek segítségével történik.

A fordítóprogram természetesen az *abszolút címeket* is meghatározhatja, ha erre megfelelő módon utasítjuk, ez azonban különféle bonyodalmakat okoz olyankor, amikor több program is jelen van, vagy ha dinamikus programáthelyezésre is szükség van. Emiatt a változók abszolút címeit többnyire csak a *futási idő* alatt szokták meghatározni oly módon, hogy a változók tartományának kezdőcímét hozzáadják a relatív cím értékéhez.

2.2.1.4. A változók címeinek ábrázolása

Mint már szó volt róla, a fordítóprogram a szimbólumtáblázat, valamint a deklarációk segítségével határozza meg a változók címeit (általában azok relatív címeit). **Blokk-orientált** nyelvek esetében ezenfelül még egy adatra szükség van: a blokkok statikus szegmentálási mélységének megadására (l. a 2.2.3. pontot). Így a teljes címmegadás a

⟨szegmentálási mélység⟩ , ⟨relatív cím⟩

adatpáros segítségével lehetséges.

A blokk szegmentálási mélység adata informál azoknak a blokkoknak a számáról, amelyekbe a változót beágyazzuk. Erre a dinamikus tárkezelés szempontjából van szükség (l. a 2.2.3. pontot).

Változók használata segítségével a felhasználói programnak a fordítóprogram által előállított kódolt formájú változatát tetemes mértékben lerövidíthetjük, mivel pl. egy mozgó pont jellemzőinek megadásához nem lesz szükségünk hat darab valós számérték megadására, csupán két egész számra, tudniillik a cím megadására.

Példa:

MOVE(55.3, 27, 10.5, 0, 70.8, 15)

Ennek a hat valós típusú számnak a jelentése most nem érdekel bennünket, azonban a 2.4.1. és a 4.2.2. pontban még találkozunk ennek az utasításnak a magyarázatával.

Az utasításhoz tartozó kódolt program legyen pl. a következő szerkezetű:

<table>
<tr><td>5000</td><td>Pozicionálás</td></tr>
<tr><td>1200</td><td>MOVE-utasítás</td></tr>
<tr><td>0</td><td>Konstansok megjelölése</td></tr>
<tr><td>1</td><td>A pontok száma</td></tr>
<tr><td>55.3</td><td rowspan="6">A mozgó pont pályaadatai</td></tr>
<tr><td>27.0</td></tr>
<tr><td>10.5</td></tr>
<tr><td>0.0</td></tr>
<tr><td>70.8</td></tr>
<tr><td>15.0</td></tr>
</table>

Legyen most P1 egy *FRAME* típusú (azaz a kísérőkoordináta-rendszer jellemzőit leíró) változó[2.1], melynek értéke valamely célpont meghatározásának leírásához pontosan hat valós számértékből kell, hogy álljon. Ekkor kiadható a következő utasítás

MOVE P1 és az ehhez tartozó programrészlet

5000	Pozicionálás
1200	MOVE-utasítás
1	Változók megjelölése
1	A pontok száma
0	Szegmentálási mélység (jelen esetben nincs)
56	A P1 változó relatív címe

Az 56-os relatív címen az interpretáló program megtalálja a FRAME típusú P1 változó értékét. Az első esetben változók használata nélkül a kódolt program 16 bájt hosszúságú volt, változók használatával pedig a kódolt program hossza 6 bájtra rövidült le.

2.2.2. A veremtárolás elve

Különféle feladatoknál, így paraméterátadásoknál, változók tárterületeinek lefoglalásánál vagy pl. aritmetikai vagy logikai kifejezések kiértékelésekor igen hasznos a **veremtárolás** (angolul stack). Egy verem olyan tárterület, amelybe a Last-in-First-out, rövidítve LIFO (vagyis az utolsónak berakottat először felszabadító) módszer segítségével különböző értékeket vihetünk be, ill. olvashatunk ki onnan. Ennek menete a következő: Először a verem elejére elhelyezünk egy veremmutatót. Ha ezután mondjuk el kell helyezni egy dátumot a veremben, akkor először csökkentjük a mutató értékét, majd „legutolsó" elemként elhelyezzük a dátum értékét a tárban. Ugyanígy mindig csak a „legutolsó" elem olvasható ki: éspedig úgy, hogy kiolvassuk a dátumot és a veremmutató értékét eggyel megnöveljük, amely így a veremben levő előző elemre mutat. A verembe való bevitelt **„push"** műveletnek (betolásnak), az onnan való kivitelt **„pop"** műveletnek (kihúzásnak) nevezzük. A veremtartomány általában egy magasabb címtől az alacsonyabb címek felé „növekszik", ezt úgy fejezzük ki, hogy a vermet csökkenő módon írjuk le. Elvileg természetesen a fordított eljárás is lehetséges, azaz leírható a verem növekvő módon is. Az aritmetikai és logikai kifejezések feldolgozásának példáján a következőkben részletesebben is megvilágítjuk ezt a veremtárolási elv szerinti tárfeldolgozási módszert.

Az információfeldolgozásban a kifejezéseknek a program futása alatt történő feldolgozásánál jól bevált a következő módszer:

- Az aritmetikai és logikai kifejezéseket a fordítóprogram a szintaktikus elemzés során az ún. fordított lengyel jelölésben átírja.
- A fordított lengyel jelölésből egyszerűen előállítja a megfelelő kódolt programrészletet.
- Az interpretáló program ezt a kódolt programrészletet egy operandus verem segítségével hajtja végre.

Ez a módszer hatékony kódolást tesz lehetővé, és a futás alatt a szervezésre fordított idő is minimális.

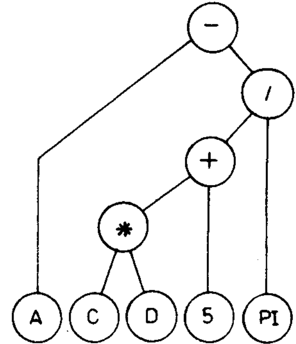
2.1. A FRAME típusú koordinátákról, ill. a velük kapcsolatos kísérőkoordináta-rendszerről később még részletesen lesz szó. Definíciójukat illetően a 2.4.1. pontra hívjuk fel az Olvasó figyelmét.

2.2.2.1. A képletek átírása fordított lengyel jelöléssel

Az aritmetikai és logikai kifejezések ábrázolását legegyszerűbb egy példán keresztül bemutatni. Legyen az utasítás a következő:

A := B–(C * D + 5)/PI

Az értékadási műveletnek csak a jobb oldalával foglalkozunk. A fordítóprogram ehhez előállítja a *2.2. ábra* szerinti fa-struktúrát. A fa-struktúra úgy épül fel, hogy a csomópontokba helyezzük a műveleteket, a két operandust pedig az abból leágazó két szomszédos csomópontból származtatjuk. Bármelyik operandus lehet egy olyan művelet, amelynek ismét csak két operandusa van. Ebből a fastruktúrából előállítható a fordított lengyel jelölés:



2.2. ábra. A fordítóprogram (compiler) képletfeldolgozó fa-struktúrája

Balról kezdve haladunk végig a diagramon és mindegyik előforduló csomópontot azonnal operandusaival írjuk fel, pl. így: B, C, 5 stb. A műveleteket csak akkor írjuk le, miután a kérdéses csomóponton már kétszer áthaladtunk. Így példánkban a következő kifejezéshez jutunk:

BCD * 5 + PI/–

Ez az előbbi kifejezésünk fordított lengyel jelölése. Ez a kifejezés reprezentálja az előre megadott aritmetikai kifejezést, csak most nem szerepelnek zárójelek. Zárójelek használatára a fordított lengyel jelölésben nincs szükség, mivel az zárójelek nélkül is egyértelmű. Egy fordított lengyel jelölésben felírt kifejezés könnyen visszaalakítható algebrai kifejezéssé, ha minden egyes operátort (műveleti jelet) a két megelőző operandusra vonatkoztatunk, azaz:

1. C * D (ez a részletkifejezés a továbbiakban operandusként szerepel)
2. (C * D) + 5 (innen kezdve ezt a részletkifejezést is operandusként fogjuk használni)
3. ((C * D) + 5)/PI (a továbbiakban ez a részletkifejezés is operandusként szerepel)
4. B–(((C * D) + 5)/PI) (a továbbiakban ez a részletkifejezés is operandusként szerepel).

2.2.2.2. Fordított lengyel jelölésben felírt képletek kiértékelése operandusverem segítségével

A fordított lengyel jelölés különösen alkalmas arra, hogy képleteket interpretáló program segítségével közvetlenül értékelhessünk ki (l. még a 8.4.1. pontot is). Ehhez csak egy operandusveremre van szükség, amely az operandusokat és a közbülső eredményeket tárolja, majd ismét előveszi.

Előző példánknál maradva az interpretáló program a fordított lengyel jelölésben felírt BCD * 5 + PI/– kifejezést úgy dolgozza fel, hogy először betölti a B, C és D operandusokat a verembe.

A verem kezdete → B
C
D ← veremmutató
.
.
.

Ezután egy rutin aktivizálódik, amely a veremben levő két legutolsó elemet összeszorozza egymással, és ennek eredményét elhelyezi a veremben a C operandus helyére, mivel a két utolsó – esetünkben a C és D – operandus a kivételkor törlődik, így most az a hely a legutolsó.

A verem ekkor a következő:

verem kezdete→B
a C * D szorzás eredménye←veremmutató
.
. törölt helyek
.

Ezután az 5-ös konstanst töltjük a verembe

verem kezdete→B
a C * D szorzás eredménye
5←veremmutató
.
.
.

majd aktivizálódik az a rutin, amely a verem két legutolsó elemét összeadja és az eredményt ismét csak elhelyezi a törlések után megmaradó legutolsó helyen, esetünkben a B alatti helyen:

verem kezdete→B
a (C * D) + 5 műveletek eredménye←veremmutató
.
. törölt helyek
.

A PI értékének betöltése, majd az osztást végző rutin behívása után a következő helyzet alakul ki:

verem kezdete→B
a ((C * D) + 5)/PI műveletek eredménye←veremmutató
.
. törölt helyek
.

Végül a fordított lengyel jelöléssel felírt kifejezésnek megfelelően aktivizálódik a kivonást végző rutin. Ez levonja a legutolsó elemet az utolsó előttiből (B-ből), majd az eredményt a B helyére teszi, a verem két utolsó eleme törlődik.

verem kezdete a B–((C * D) + 5)/PI művelet eredménye ←.
veremmutató
.
. törölt helyek.

A B–((C * D) + 5)/PI kifejezés eredménye most is a legutolsó helyre került, és ez az érték most már átadható az A változónak, tehát a megfelelő számértéket az A címére kell beírni.

2.2.2.3. Képletek gépi kódolásának előállítása

Miután a fordítóprogram az imént bemutatott módon átalakította az adott aritmetikai vagy logikai kifejezést fordított lengyel jelölésre, egyszerűen előállítható a megfelelő gépi kód is. Ehhez nem kell más szervezési feladatot megoldani, mint „push" műveletet változókra és konstansokra, valamint „pop" műveletet ugyancsak a változókra és a konstansokra. Az olyan aritmetikai műveletekre, mint az összeadás, kivonás, szorzás nem kell további tennivalókat előírni, hiszen ezek gyakorlatilag automatikusan futnak a verem vezérlete alatt. Így az

A := B–(C * D + 5)/PI,

utasítás, amelynek lengyel jelölése

BCD * 5 + PI/–

a következő struktúrájú gépi kóddá alakítható át:

100	„push" művelet változó bevitelére
0	blokkolási mélység
20	a B változó relatív címe
100	„push" művelet
0	blokkolási mélység
24	a C változó relatív címe
100	„push" művelet
0	blokkolási mélység
42	a D változó relatív címe
1003	szorzási művelet az operandusveremre
150	„push" művelet konstans bevitelére
1	típus megadás: REAL
5.0	konstans, értéke 5
1001	összeadási művelet az operandusverem két elemére
100	„push" művelet
0	blokkolási mélység
10	a PI változó relatív címe
1004	osztási művelet az operandusverembe
1002	kivonási művelet az operandusverembe
2001	a verem egy elemének értékét add át a következő változónak
0	blokkolási mélység
16	az A változó relatív címe

Az első pillantásra ez az eljárás körülményesnek tűnik, figyelembe kell azonban venni, hogy ezzel a módszerrel tetszőleges mélységben egymásba ágyazott blokkolt aritmetikai és logikai kifejezések lefordíthatók gépi kódra. A gépi program pedig viszonylag egyszerűen számítja ki a közbülső eredményeket. A „push" és „pop" műveleteket széleskörűen használják még egyebek mellett az eljárásparaméterek, a különféle elágazási feltételek veremjeinek a kezelésénél. Ez az eljárás a magasabb szintű programnyelvek, így az ALGOL, a PASCAL, sőt az AL robotprogramozási nyelv esetében is jól bevált.

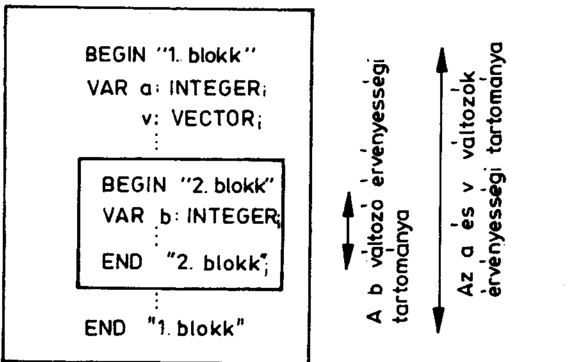
2.2.3. A blokkszervezés

Több, tágabb értelemben az ALGOL nyelvcsaládhoz sorolható programozási nyelv rendelkezik a **blokkszervezés** lehetőségével. A programozó viszonylag önálló zárt egységekre tagolhatja programját, ezek kezdetét, ill. végét egy-egy BEGIN-nel és END-del jelöli. Mindegyik blokk tartalmazhat adatdeklarációkat, utasításokat, valamint további blokkokat, a blokkok egymásba ágyazhatók.

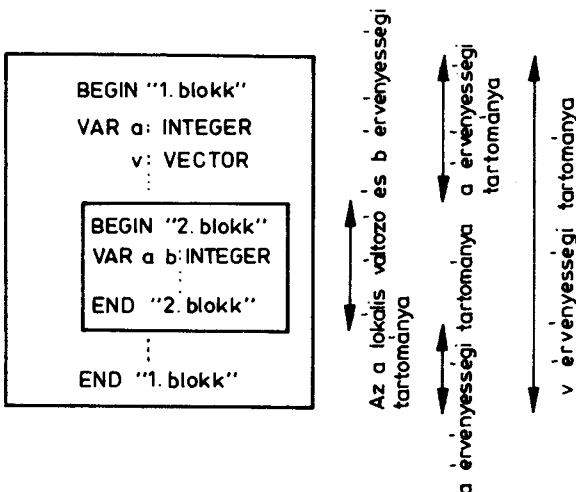
2.2.3.1. A változók érvényességi tartománya és élettartama

Az ún. eljárásoktól, ill. függvényeljárásoktól eltekintve a program blokkokra osztása nem befolyásolja a program végrehajtását. Azok a változók azonban, amelyek valamelyik adott blokkon belül lettek deklarálva, csak ezeknek a blokkoknak a belsejében használhatóak, valamint az ebben a blokkban megnyitott belső blokkokban, de nem dolgozhatunk velük a blokkokon kívül. Ilyen értelemben beszélünk a változók **érvényességi tartományáról.**

A *2.3. ábrán* a 2. blokkban a b változó ún. lokális változó, az a és v változók pedig ún. globális változók.



2.3. ábra. A változók érvényességi tartománya



2.4. ábra. A változók érvényességi tartományai a belső és külső blokkokban

Mármost, ha egy belső blokknak valamely változója ugyanazt a nevet viseli, mint az őt tartalmazó külső blokkban egy változó, akkor a belső blokkbeli változó mintegy kiszorítja a külső blokkban definiált változót, ekkor az már nem hozzáférhető a belső blokkban.

Az 1. blokkban definiált a globális változónak van **élettartama** is, ez az 1. blokk elejétől annak végéig terjed, érvényességi tartománya azonban a 2. blokkra nem terjed ki (l. a *2.4. ábrát*). A v változó élettartama ezzel szemben a teljes 1. blokkra kiterjed, míg a második blokkban levő a és b lokális változók élettartama azonban csak erre a 2. blokkra terjed ki.

A BEGIN és END alapszavakon kívül még eljárásdeklarációk, függvénydeklarációk, ciklusutasítások vagy akár task-deklarációk segítségével is definiálhatunk blokkokat.

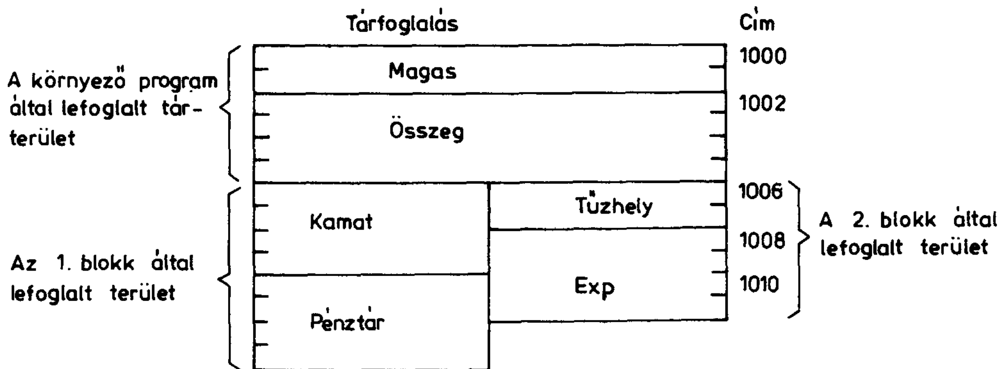
2.2.3.2. Blokkszervezés és a tárkezelés kérdése

A blokkszervezés – amellett, hogy áttekinthetőbbé és strukturáltabbá teszi a programot – a programbeli adatok számára fenntartott *tárterület* jobb kihasználását is elősegíti. A változók élettartama ugyanis arra a blokkra korlátozódik, amelyben deklaráltuk, így tárterületre is csak olyankor van szükségük, amikor éppen az adott blokk feldolgozása folyik. Emiatt a program futása során egy-egy blokk utasításainak végrehajtását megelőzően a program lefoglalja a blokk változóihoz szükséges tárterületet, majd e blokk feldolgozásának befejeztével ismét szabaddá teszi azt (l. a *2.2. programrészletet*).

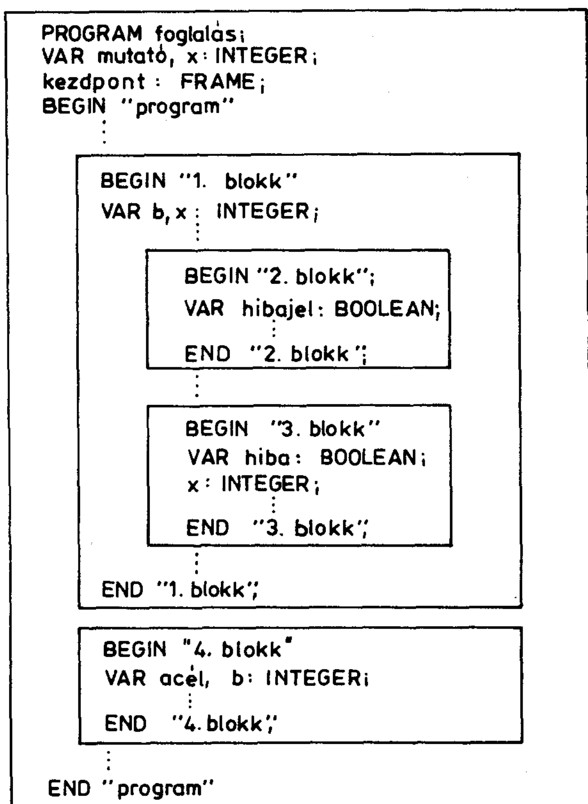
2.2. programrészlet. Példa dinamikus tárkezelésre PASCAL-ban

```
PROGRAM szetosztas;
  VAR
    magas: integer;
    összeg: real;
PROCEDURE blokk1;
  VAR
    kamat: real;
    kassza: real;
BEGIN (* a blokk1 törzs kezdete *)
        :
       (* utasítások *)
        :
  END (* a blokk1 törzs vége *);
PROCEDURE blokk2;
  VAR
    tüzhely: integer;
    exp: real;
    BEGIN (* a blokk2 törzs kezdete *)
       :
      (* utasítások *)
       :
END (* a blokk2 törzse *);
BEGIN (* a szétosztás törzsének kezdete *)
  :
  (* utasítások *)
END (* a szétosztás törzsének vége *)
```

A *2.2. programrészlet* futása során a *dinamikus tárkezelés* folyamatát a *2.5. ábra* szemlélteti. Látható, hogy az 1006-tól 1010-ig terjedő című memóriarekeszek egyaránt felhasználhatók a **kamat** és **kassza** nevű változók tárolására (az 1. blokk végrehajtása alatt), valamint a **tuzhely** és **exp** változók számára a 2. blokk végrehajtásának időtartama alatt.



2.5. ábra. Dinamikus tárkezelés változók esetében



2.6. ábra. A változók helyfoglalása a tárban, élettartamuk és érvényességi tartományuk a program futása alatt

Tárfoglalás

Élettartam és érvényességi tartomány

Program

1. blokk

2. blokk

3. blokk

4. blokk

mutató

x

kezdpont

b

x

Hibajel

Hiba

x

Acél

b

Program:

Mutató

x

Kezdpont

1. blokk

b

x

2. blokk

Hibajel

3. blokk

Hiba

x

4. blokk

Acél

b

2.2.3.3. A verem elv szerinti tárkezelés

Mivel egymásba ágyazott blokkok esetében a mindig az utolsónak lefoglalt blokkot szabadítjuk fel először, az adatok számára lefoglalt helyekkel való gazdálkodás is megvalósítható a *veremfeldolgozás* mintájára. A blokk feldolgozásának megkezdésekor az adatok számára fenntartott veremben lefoglalunk egy tárterületet a blokk változóira. Minden további belső blokk számára új adatterületet csatolunk a verem végéhez. Mihelyt befejeződik a blokk feldolgozása, a blokkhoz tartozó adattár területe ismét felszabadul, éspedig úgy, hogy ez a terület mindig a verem legfelső eleméhez tartozik.

A *2.6. ábra* szemlélteti, hogy a példaként felvett program futása során hogyan alakul a változók helyfoglalása és élettartama, valamint érvényességi tartománya.

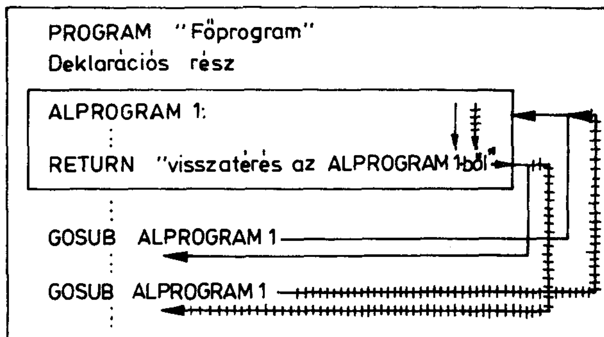
Ha az adott programnyelv nem teszi lehetővé rekurzív eljárások használatát (l. a 2.2.5. pontot), akkor már a fordítóprogram is meg tudja szervezni a tárfoglalás teljes folyamatát, és meg tudja határozni a változók címeit is.

2.2.4. Alprogramok, eljárások függvényeljárások és makrók

A programírási munka egyszerűsítése és a program hosszának csökkentése érdekében a programban több helyen lényegében változatlanul megismétlődő programrészletek kényelmes és áttekinthető szerkesztési módszereként bevezették az alprogramok, az eljárások és a függvényeljárások használatát. A makrók ezzel szemben szövegbehelyettesítési célt szolgálnak, ezek feldolgozására még a program futása előtt sor kerül, így ezek nem csökkentik a program hosszát. A makrók és mindenekelőtt az eljárások használatának az a jelentősége, hogy az egyszer megírt algoritmus más programokban ismételten felhasználható, ugyanakkor különösen az eljárások használata a program strukturáltságának – azaz a program viszonylag önálló, jól áttekinthető szerkezeti egységekre osztásának – legfontosabb segédeszköze. A program strukturáltsága elvezethet egy hierarchikus felépítéshez, melyben az egyes eljárások egy fölérendelt eljárás vezérlete alatt oldják meg a rájuk tartozó feladatot, majd az eredményt átadják a futtató eljárásnak.

2.2.4.1. Az alprogramok

Az alprogram olyan programrészlet, amelyet a program többi részletétől elkülönítve írunk, és amely a programban tetszőleges helyről hívható. Az alprogram feldolgozása után a vezérlés az alprogramot behívó utasítást követő utasításnak adódik át (l. *a 2.7. ábrát*).



2.7. ábra. Alprogram behívása

A visszatérés szervezését a *futtató rendszer* veszi át oly módon, hogy az alprogram meghívásakor tárolja a következő utasítás címét, ahonnan az alprogram befejezése után a főprogram folytatódik. Az alprogramok legegyszerűbb fajtájánál nem történik paraméterátadás, és lokális változókat sem lehet deklarálni. Ezenkívül a feldolgozás módja szekvenciális, azaz a főprogram az alprogram futásának ideje alatt várakozik. Egy másik különleges eset az alprogram ún. *rekurzív* aktivizálási lehetősége. Ekkor az alprogram – futása közben – meghívhatja saját magát is. Az alprogramok szervezésének és használatának különféle eseteivel a következő fejezetekben még részletesebben is foglalkozunk.

2.2.4.2. Eljárások, függvényeljárások

Az alprogramoknak a széles körű felhasználásuk érdekében a különböző alkalmazási feladatokhoz rugalmasan kell illeszkedniök. Ezért célszerű, ha az alprogramok *paraméterezhetőek*. E paramétereknek a különféle aktivizálások során különféle értékeket lehet adni. Az „alprogram" – vagy idegen szóval szubrutin – és az „eljárás" szavakat általában egymás szinonimáiként használjuk. Ennek ellenére azoknál a programnyelveknél, ahol csak lokális változók és paraméterátadási lehetőség nélküli alprogramok lehetségesek, nem beszélhetünk az eljárásszervezési elv meglétéről. Ez az oka annak, hogy az eljárások és függvényeljárások már bizonyos mértékig „fejlettebb" szintű alprogramoknak tekinthetők.

Az itt következő példa bemutatja a paraméterek használatát eljárások esetében. Az eljárásban a robotkar pillanatnyi helyzete és egy tetszőleges pályapont közötti távolságot kell meghatározni (ezzel kapcsolatos az ún. keret (frame) fogalma, melyet részletesebben a 2.4.1. pontban tárgyalunk), majd az így meghatározott értéket össze kell hasonlítani egy olyan értékkel, amelyet az eljárás egy-egy meghívásakor különféleképpen adhatunk meg. Ha ez a távolság túllép egy megadott értéket, akkor az eljárás beállít egy globális hibajel értéket, egyébként pedig végrehajtja a robotkarnak a megadott pályapontra vezérlését. A feladat megoldásához definiáljuk a **frame-távolság**[2.2] nevű eljárást a következő két paraméterrel:

1. egy frame (kísérőkoordináta-rendszer), amelyhez képest a mozgás értelmezett,
2. egy megadott távolságérték.

A *2.3. programrészlet* bemutatja, hogyan kell egy eljárást az AL nyelvben programozni. Ezt az eljárást pl. a következő módon hívhatjuk:

frame-távolság (horog, 2),

de helyes a következő hívási mód is:

frame-távolság (FRAME (xrot, VECTOR (5, 10, 7) * CM), bazis + 3);

Az első esetben a **horog** nevű frame és a kar helyzete közötti távolságot határozzuk meg. Ha ez kisebb 2 cm-nél, akkor a robot karját a **horog** pozíciójába mozdítjuk el, egyébként pedig beállítjuk a **hibajel** változót.

2.2. A *frame* angol szó, keretet, gépállványt jelent. A robottechnikában a robot egyes ízületeihez kötött koordináta-rendszerekről, ezekben mért koordinátákról van szó a frame elnevezéssel kapcsolatban. A könyv magyar kiadásának előkészítése során felmerült, hogy a szót valamilyen megfelelő magyar szóval helyettesítsük, azonban tekintettel arra, hogy a megfelelő programozási nyelvben is a szó eredeti angol formájában jelenik meg, továbbá, ha „frame" helyett a magyar „keret" elnevezést használnánk az egyes szóösszetételeknél (pl. célkeret, karkeret, kerettávolság stb.) az eredeti értelem jelentős torzulást szenvedne. Ezért döntöttünk az angol kifejezés meghagyása mellett, amit az is igazolni látszik, hogy a német szerzők – akik egyébként következetesen német kifejezésekkel helyettesítették a legtöbb angol terminus technicust – a „frame" esetében maradtak az angol szónál. (A szerk. megjegyzése).

```
PROCEDURE framedistanz (VALUE FRAME célframe;
                        VALUE SCALAR distanz);
BEGIN "framedistanz"
FRAME karframe;
karframe←ARM (* a robotkar aktuális helyzetét meghatározó frame értékeit átadjuk a karframe
              változónak *)
IF | POS (célframe) - POS (karframe) < distanz THEN
MOVE ARM TO célframe
ELSE
  hibajel← TRUE;
END "framedistanz";
```

2.3. programrészlet. Eljárás definiálása az AL nyelvben

```
PROCEDURE koordtengelyertek (p: INTEGER);
BEGIN
  tengely := YTENGELY;          (* megváltoztatja a „tengely"     *)
                                (* globális változó értékét       *)
  vektor.XTENGELY := 10;        (* megváltoztatja a „vektor"      *)
                                (* globális változó értékét       *)
x := p;                         (* a p paramétert átadja az x glo- *)
                                (* bális változónak               *)
END;
```

2.4. programrészlet. Eljárás deklarálása egy hipotetikus nyelvben a különféle paraméterátadási módok szemléltetésére

```
tengely := XTENGELY;
vektor := VECTOR (20, 7, 13);
koordtengelyertek (vektor.tengely);
```

2.5. programrészlet. A 2.4. programrészlet eljárásának aktivizálása

A második esetben a robotkart egy explicit módon definiált frame által meghatározott helyzetbe vezéreljük, majd meghatározzuk az (5, 10, 7) vektorokból mért távolságot. Azt a távolságadatot, amellyel az összehasonlítást el kell végezni, először ki kell számítani. Ezt úgy adtuk meg, hogy a **„bázis"** nevű változó és a 3 számértékű konstans értékének összege legyen.

Az eljárás deklarálása abból áll, hogy megadjuk az eljárás nevét, jelen esetben frametávolság nevet, a paraméterlistát, itt a **cél-frame** és a **távolság** paraméterekkel, majd egy deklarációs részt az eljárás lokális változóival – jelen esetben a **kar-frame** változóval – ezután pedig leírjuk az eljárás törzsét, amely tulajdonképpen az eljárásban megvalósított algoritmus. A paraméterlistában és az eljárás törzsében előforduló paramétereket *formális* paramétereknek nevezzük, mivel ezek a későbbi eljárásaktivizálások során konkrétan megadott ún. *aktuális* paraméterek helyett szerepelnek. Arra vonatkozóan, hogy hogyan kell a formális paramétereket az aktuális paraméterekkel helyettesíteni, különféle *paraméterátadási módszerek* léteznek. Általánosságban érvényes azonban, hogy a paraméterek sorrendje – azaz a paramétereknek a listában elfoglalt helye – határozza meg az aktuális és formális paraméterek egymással történő megfeleltetését, ezenkívül az aktuális és a formális paraméterek típusainak egyezniük kell. Ebből a célból az eljárás fejében meg kell határozni a formális paraméterek típusát.

A következő példán megvilágítjuk az aktuális–formális paraméterátadások fontosabb típusait. Tekintsük a *2.4. programrészlet* szerinti eljárást.

Az eljárás megváltoztatja a két globális változónak a **tengely**-nek és a VECTOR típusú **vektor** változónak az értékét. Az INTEGER típusú **p** formális paraméter értékét átadjuk az **x** változónak. Mármost a következő utasítássorozat a paraméterátadási módtól függően az **x** változó különböző értékeit szolgáltatja (l. a *2.5. programrészletet*).

a) Call by value (Érték szerinti paraméterátadás)
Az aktuális paraméter értékét a program az eljárás hívásakor határozza meg, és az ekkor kiszámított értéket helyettesíti be a formális paraméter helyébe mindenütt, ahol a kérdéses formális paraméter az eljárás törzsében csak előfordul. A példában tehát a program először kiértékeli a

vektor · tengely = VECTOR (20, 7, 13) XTENGELY

kifejezést, azaz kiválasztja a vektor X koordinátáját, és ennek értékét számítja ki.

Az **x** globális változó tehát az eljárás hívása után a 20-as értéket veszi fel.

b) Call by reference (hivatkozás szerinti paraméterátadás)
Az eljárás hívásakor átadódik egy az aktuális paraméterre utaló hivatkozás, azaz a formális paraméter most azt a változót képviseli, amelyet a híváskor aktuális paraméterként jelöltünk meg. Ez azt eredményezi, hogy ha az eljárás törzsében a változók értékei megváltoznak, akkor ez az aktuális paraméter értékét is megváltoztatja. A példabeli eljárás hívásakor tehát megvalósítunk egy a vektor X-koordinátájára vonatkozó hivatkozást. Az eljárás törzsében a

vektor · XTENGELY := 10;

utasítás hatására a **vektor** nevű változó X-koordinátájának átadódik a 10-es érték, így ezt az értéket kapja **x** az eljárás aktivizálása után.

c) Call by name (név szerinti paraméterátadás)
Ezt a paraméterátadási módot ritkábban valósítják meg, mert implementálása körülményesebb, és a program futásakor is időigényesebb. Ez annak a következménye, hogy az aktuális paraméter szövegszerűen helyettesíti a formális paramétert, értékét a program csak akkor határozza meg, amikor az eljárás feldolgozása közben egy olyan helyre ér, ahol az adott paraméter előfordul. Ha a paraméter több helyen is előfordul, akkor minden egyes alkalommal újra kiértékeli. A paraméter minden egyes kiértékelésekor új értéket kaphat, és ez sajnos ahhoz vezet, hogy a paraméterátadás áttekinthetetlenné válik. Példánkban a **vektor, tengely** aktuális paraméter értéke csak akkor lesz kiértékelve, amikor a **p** paraméter értékét átadjuk az **x** globális változónak. Mivel a **tengely** nevű változó az eljárás törzsében az YTENGELY értékét kapta, így a **vektor** Y-koordinátáját választjuk ki, ezért **x** végül a 7-es értéket kapja.

Különösen a *hivatkozás szerinti* paraméterátadás és a *név szerinti* paraméterátadás alkalmazása esetén különféle *mellékhatások* adódhatnak abból a körülményből, hogy az eljárásban értéket adhatunk a paramétereknek, ill. általában globális változóknak. Ez egyrészt azt a célt szolgálja, hogy az eljárás különféle számítási eredményeket adhasson át, másrészről azonban vigyázni kell, nehogy tévedésből olyan változók értékeit is megváltoztassuk, amelyeknek a főprogramban konstansoknak kellene maradniuk.

Az előbbiekben tárgyalt eljárások mellett léteznek még az ún. függvényeljárások, amelyek segítségével az ilyen mellékhatások egyszerűen elkerülhetőek, és amelyeknél a függvényeljárás eredménye másként adódik át a főprogram algoritmusának. Ekkor a függvényeljárás nevét egy olyan változónak tekintjük, amelynek típusát külön deklarálni kell, és az ebben a függvényeljárásban kiszámított számérték ennek a változónak adódik át. Ez a név közvetlenül használható a főprogram utasításaiban és kifejezéseiben, mivel a függvényeljárás hívása nem egy külön zárt utasítás, hanem egyszerűen a függvényeljárás név és a paraméterlista megadásával történik. Példaként álljon itt egy függvényeljárás, amely egy vektor komponenseinek összegét állítja elő:

```
SCALAR PROCEDURE vektorszumma (VECTOR v);
RETURN (v.XTENGELY + v.YTENGELY + v.ZTENGELY);
```

A következő egyszerű utasításban egy **alapérték** nevű skalár változónak adunk értéket, amely egyben a függvényeljárás hívása is:

alapérték := 5 + vektorszumma (célvektor);

Különleges szerepet játszanak az ún. standard függvények. Ezeket a programozónak definiálnia sem kell, mivel ezek a rendszer szolgáltatásai és többnyire a legfontosabb matematikai alapfüggvények kiszámítását biztosítják, így többek között a gyökvonást, a szinusz-, a tangens-, a logaritmus kiszámítását.

Például távolság := SIN (rotáció) * vektorhossz;

Itt kiszámítjuk a **rotáció** szögérték szinuszát, ezt megszorozzuk a **vektorhossz** nevű változó értékével, és a művelet eredménye a **távolság** nevű változónak adódik át.

2.2.4.3. A makroeljárások

Az alprogramokkal ellentétben a makroeljárások, vagy röviden makrók nem a program futása során hajtódnak végre, hanem már a program fordításakor feldolgozzuk azokat, és így a futtatható programban ezek már nem ismerhetők fel. A makrók kizárólag a *szöveghelyettesítés* céljait szolgálják, amely állhat egy egyszerű, előre megadott jelsorozatból álló szöveg bemásolásából, de segítségükkel akár a legbonyolultabb szöveg-, vagy programgenerálás is végrehajtható. Lehetőség van paraméterek és belső definíciók használatára, sőt további makrók behívása is megengedett (l. pl. COLE [2.1]).

Egy *makrodefiníción* egy makro névnek egy makro törzshöz való hozzárendelését értjük. A *makro név* – ugyanúgy, mint bármely változónév – egy szabadon választható karaktersorozatból áll. A *makrotörzs* karakterek és/vagy makrohívások sorozatából állhat. A *makro hívása* úgy történik, hogy megadjuk a makro nevét (esetleg paramétereivel együtt), mire az ún. *makrogenerátor* behelyettesíti a makro törzséhez tartozó karaktersorozatot.

Példa: DEFINE inch = # * 2.54 cm # ;

Ez a makrodefiníció az **inch** makronévhez egyértelműen hozzárendeli a * 2,54 cm szövegből álló makrotörzset. A # karakterek határolójelekként szolgálnak, ezeket a „környező" programszövegben nem célszerű használni.

Egy ilyen makrohívás szerepel pl. a

```
távolság := bázis + 35 inch;
```

utasításban. Ez a makrogenerátorral való feldolgozás után a

```
távolság := bázis + 35 * 2,54 cm;
```

szöveget eredményezi.

Hasonlóképpen történik a makro paraméteres definiálása is:

```
DEFINE mozgásvez (célframe) =
            # MOVE kar TO célframe
            WITH DURATION = 5 sec; #
```

És az utasítássorozat közbeni hívása:

```
x := 200;
mozgásvez (lerakás)
OBEN kéz;
mozgásvez (tartály)
```

utasítássorozat a bennük levő makrohívásokkal a következő szöveget eredményezi:

```
x := 200;
MOVE kar TO lerakás
```

```
    WITH DURATION = 5 sec;
  OBEN kéz;
  MOVE kar TO tartály
    WITH DURATION = 5 sec;
```

A makrodefiníciók a makrogenerátor típusától függően akár a program elején, akár pedig a program tetszőleges helyén is beolvashatók a külső könyvtárból. Magát a makrodefiníciót azonban minden esetben már a makro hívása előtt deklarálnunk kell, mivel a makrogenerátor csak így tudja kezelni.

A makrók használatának a következő előnyei vannak:

- a programírási munka lerövidül;
- a gyakran ismétlődő fontosabb utasítássorozatokat tapasztaltabb programozók makroként definiálhatják, így azokat később mindenki használhatja;
- a makrohívások konzekvens alkalmazása esetén a program jobban olvasható és egyszerűbben szabványosítható.

Hátránya, hogy az így generált program tárkapacitás-igénye nagyobb, és a fordítási idő is megnövekszik. Ezért makrók alkalmazásánál mérlegelni kell az előnyöket és a hátrányokat.

2.2.5. Az alprogramok rekurzív hívása

A magasabb szintű programnyelvek egy részénél lehetőség van arra, hogy egy eljárást magában az ezt behívó eljárásban, vagy bármely más alprogramból hívni lehessen. Az első esetben *közvetlen rekurzív* eljáráshívásról beszélünk, a második eset pedig a *közvetett rekurzív* hívás esetéhez vezethet – pl. akkor, ha két eljárás kölcsönösen egymást aktivizálja (l. a *2.8. ábrát*).

Egészen a legutóbbi időkig az általánosan elterjedt vélemény szerint az ipari robotok programozástechnikájában szükségtelennek és teljesen elvont játszadozásnak minősítették a rekurzív eljáráshívásokat, ezért a következőkben teljes részletességgel bemutatunk egy példát rekurzív eljáráshívásra. Látni fogjuk, hogy ugyanaz a probléma más módszerekkel csak igen körülményesen lenne megoldható.

Egy kisebb demonstrációs program (a *2.6. program*) egy robotkar működtetésének vezérlését valósítja meg tetszőleges kezdő- és végpontok mellett, ugyanakkor a vezérlést egy – a programban megadott – állandó gyorsítás mellett kell végrehajtani mindaddig, amíg a kar el nem éri a legnagyobb megengedett sebességet, vagy amíg életbe nem lép a fékezési utasítás. A mozgást állandó lassítással fékezik, amelynek értéke megegyezik a gyorsítás (negatív) értékével. Arra nincs lehetőség, hogy a robotkart egy

⟨mozgasd a robotkart A-ból B-be konstans C gyorsítással/lassítással⟩

típusú utasítás szerint közvetlenül vezérelhessük. A nyelvben azonban létezik egy VIAMOVE utasítás, amelynek hatására a robot karja elmozdul a pálya egy közbülső pontja felé.

A közbülső pontot egy előírt sebességgel kell elérni, majd ezt a sebességet a következő közbülső pont felé irányuló mozgássá kell átalakítani. A feladat az, hogy előállítsuk a VIAMOVE utasítások egy sorozatát, éspedig úgy, hogy a közbülső pontokon adott sebességgel való áthaladáshoz egy állandó gyorsulást, ill. lassulást alkalmazunk.

A gyorsulás, a sebesség, az út és az idő között a következő összefüggések állnak fenn:

$$\Delta s = \frac{a}{2}\Delta t^2 + v_1 \Delta t \text{ és}$$

$$\Delta v = a\Delta t = v_2 - v_1,$$

ahol

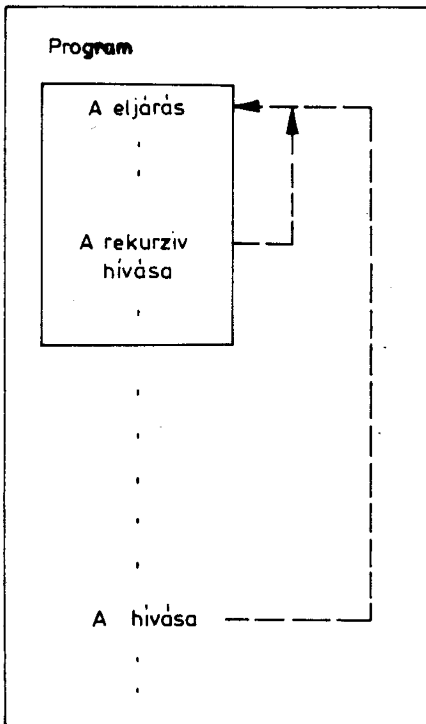
Δs = két közbülső pont közötti szakasz hossza;
Δv = sebességváltozás két közbülső pont között;
Δt = adott **dt** időintervallum;
a = a választott **a**, ill. **b** gyorsulás;
v_1 = sebesség az első közbülső pont elérésekor;
v_2 = sebesség a második közbülső pont elérésekor.

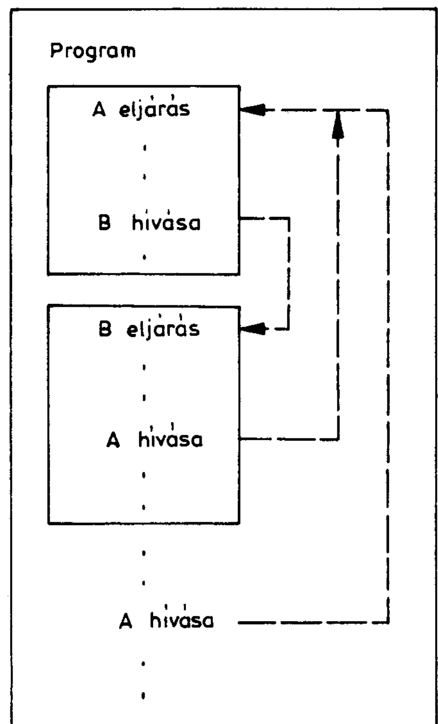
Legyenek most „**kezdp**" a mozgás tetszőlegesen választott kezdőpontja, „**célp**" pedig a végpontja. Az egyszerűség kedvéért a kar haladjon az X-tengely mentén, így elegendő a **kezdp.XTENGELY** és a **célp.XTENGELY** X-koordinátaértékeivel foglalkozni. A matematikai standardfüggvények közül demonstrációs programunkhoz szükség van az SQR (négyzetre emelés) és az **ABS** (abszolútérték-képzés) rutinjaira.

A program futása a *2.9. ábra* szerinti *rekurzív hívásokat* és VIAMOVE utasításokat eredményezi az ott feltüntetett paraméterekkel. A főprogramban a vezérlést a

MOVE kar TO 150;

utasítás zárja le, miáltal a robotkar ott megáll.





2.8. ábra. Rekurzív eljáráshívási lehetőségek

```
 1 PROGRAM                                   harmonikusmozgasvezerles;
 2 CONST
 3  vmax = 80;                               (* max robotsebesség             *)
 4  dt = 0.25;                               (* 250 msec-os időintervallum    *)
                                             (* a vezerles vegrehajtasahoz    *)
 5  VAR
 6  kezdp          ,                         (* kiinduló helyzet              *)
 7  célp           : FRAME;                  (* véghelyzet                    *)
 8  feltavols      ,                         (* a szakasz fele                *)
 9  b : REAL       ;                         (* gyorsulas/lassulas            *)
10 PROCEDURE mozgasvez (szakasz, v: REAL);
   VAR
11  a: REAL;                                 (* konstans vagy változó         *)
                                             (* gyorsulas                     *)
12 BEGIN                                     (* mozgasvez                     *)
13  a := b;
14  szakasz := abs(szakasz) + 0.5 * a * sqr(dt) + v*dt;
15  IF feltavols < 0 THEN
16     szakasz := - szakasz;
17  v := v+a * dt;
18  IF (ABS(szakasz) < ABS(feltavols)) AND (v <= vmax) THEN
19  BEGIN
20     VIAMOVE kar TO kezdp.XTENGELY + szakasz WHERE VELOCITY = v;
21     mozgasvez (szakasz, v);
22     VIAMOVE kar TO célp.XTENGELY - szakasz WHERE VELOCITY = v;
23  END
24 END (* a mozgasvez vége *)
25 BEGIN "harmonikusmozgas"
26  kezdp.XTENGELY := 50;                    (* x-koord: 50 cm                *)
27  celp.XTENGELY := 150;                    (* x-koord: 150 cm               *)
28  feltavols := (celp.XTENGELY-kezdp.XTENGELY)/ 2;
29  b := 40;                                 (* konstans 40 cm/(sec * sec)
                                             (* gyorsitas/lassitas            *)
30  mozgas (0,0);
31  MOVE kar TO celp.XTENGELY;
32 END "harmonikusmozgas"
```

2.6. programrészlet. Adott gyorsulású mozgás végrehajtására hipotetikus nyelven írt program rekurzív eljáráshívással

A gyorsítás és a lassítás szimmetrikusan történik, a gyorsulás, ill. lassulás értéke tetszőlegesen megadható (természetesen a technikai lehetőségek szabta határokon belül) *(2.10. ábra)*. A figyelmes Olvasó bizonyára észrevette, hogy a 13. sor felesleges, a 14. sorban a 0.5 *** a *** SQR (dt) kifejezés pedig konstans, akárcsak a 17. sorban az **a * dt** kifejezés értéke.

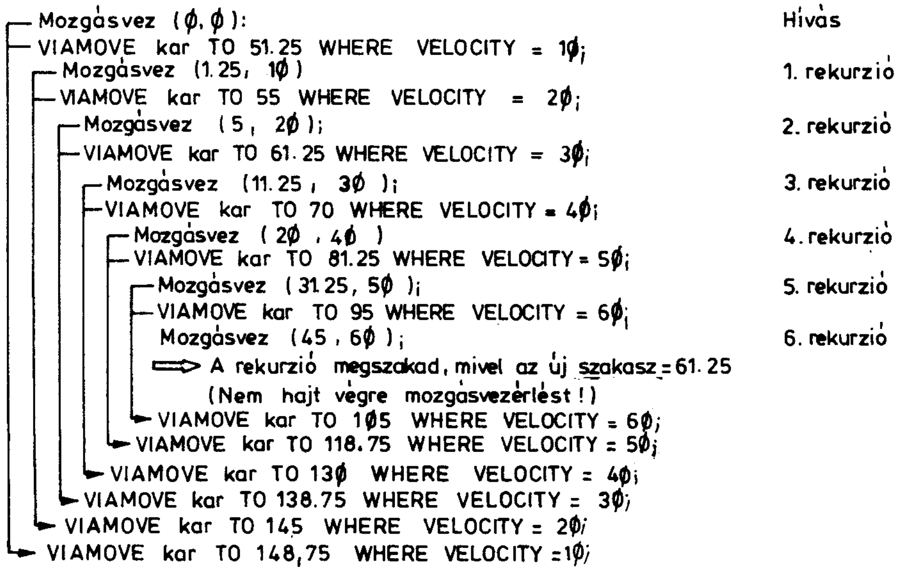
Ezeket a konstans kifejezéseket konstans gyorsulás esetén elegendő egyszer kiszámítani, nem szükséges minden rekurzív hívásnál újból meghatározni. Ha azonban a gyorsulás értékét egy vezérlési művelet alatt is változtatni kell tudnunk, pl. úgy, hogy a művelet elején és végén kisebb, a középső részen ellenben nagyobb legyen, akkor a 13. sorban minden egyes rekurzív hívás során új gyorsulási értéket kell kiszámítanunk. Példánk így:

13 **a := 10 + b * szakasz/feltavols;**

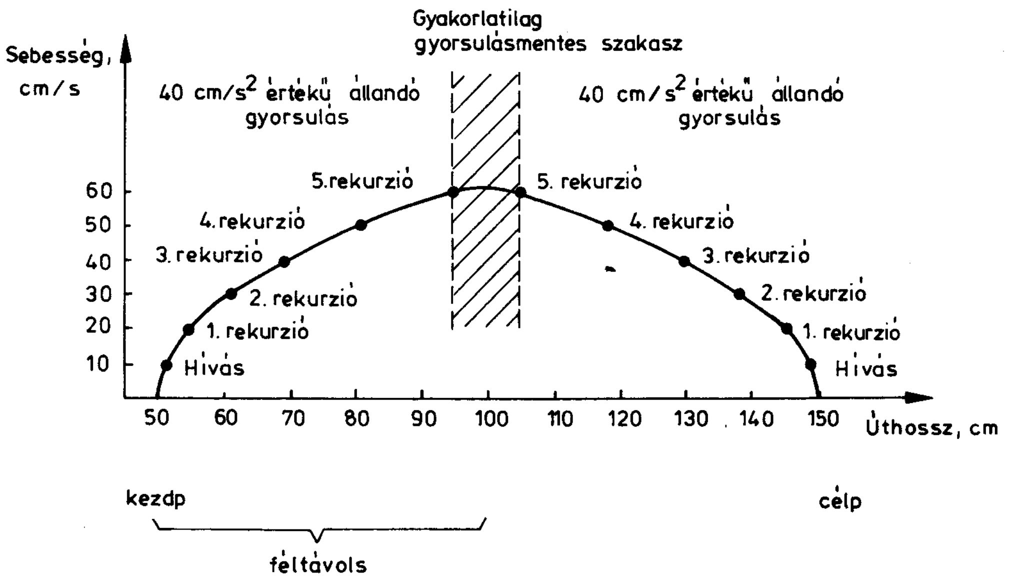
Így tehát egy rövid kis program segítségével szabadon beprogramozhatunk tetszőleges harmonikus mozgástípusokat. Ilyen mozgástípusokra lehet szükségünk pl. festőüzemben vagy pl. köszörülési technológiák vezérlésénél *(2.11. ábra)*.

A rekurzív algoritmusban az iterációsorozat akkor szakad meg, amikor teljesül a 18. sorban álló következő kritérium:

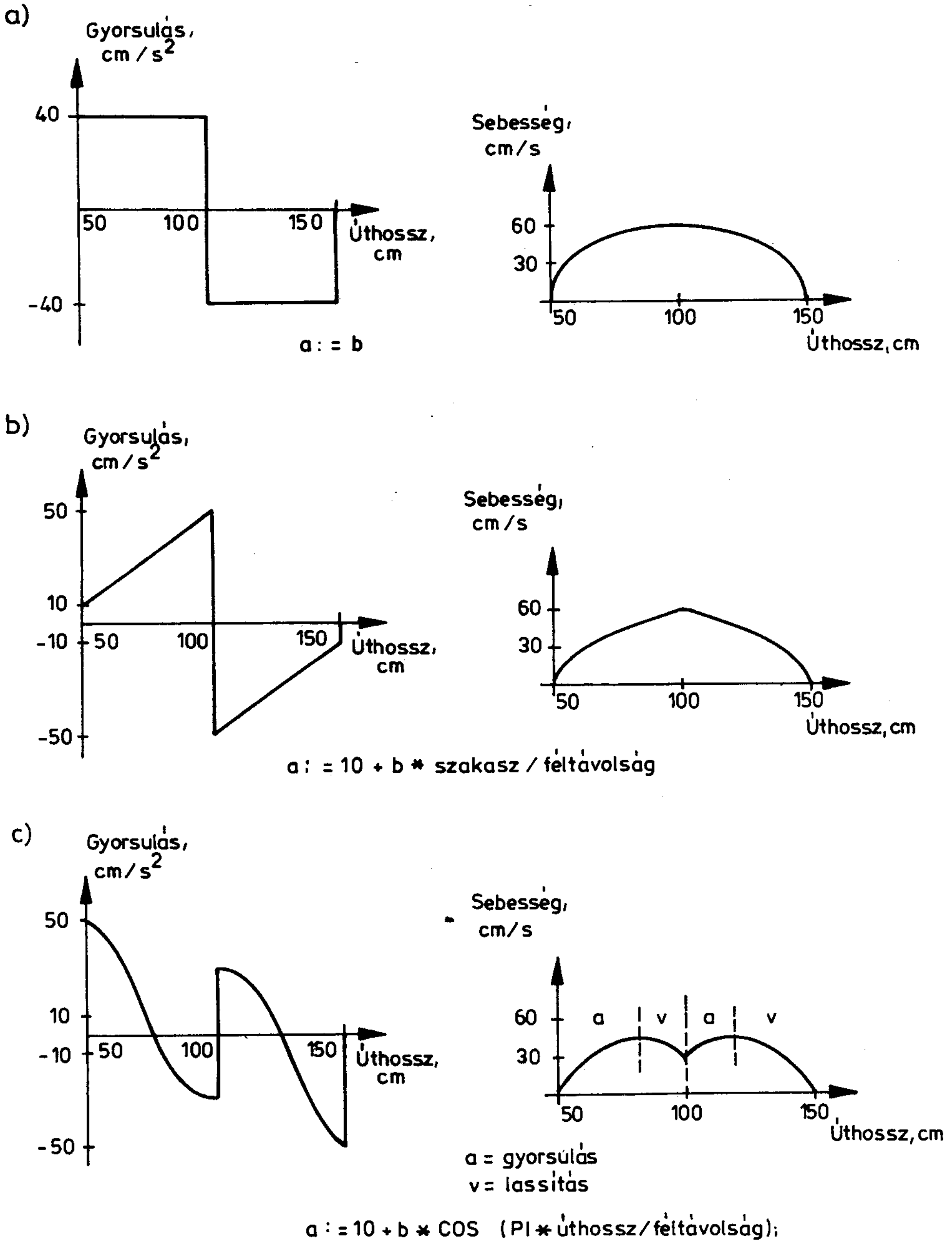
ABS **(szakasz)** < ABS **(feltavols)** AND **v** $\leqq$ **vmax**



2.9. ábra. Sorozatos rekurzív eljáráshívások feldolgozása



2.10. ábra. A mozgásvez programkészlet rekurzív eljáráshívásaiban aktivált VIAMOVE-utasítások sebesség–út diagramja



2.11. ábra. A robotkar mozgását leíró diagramok *a)* állandó gyorsulás, ill. lassulás; *b)* lineárisan növekvő, ill. csökkenő gyorsulás, ill. lassulás; *c)* koszinusz függvény jellegű gyorsulás, ill. lassulás

Az első összehasonlításnál azt kérdezzük le, hogy a kezdőponttól a kiszámított közbülső pontig terjedő távolság az előírt szakasz első felében helyezkedik-e el. Ez a feltétel a hatodik iterációs lépésben már nem teljesül, így nem történik további rekurzív programhívás. A második összehasonlításban már csak azt vizsgáljuk, hogy túlléptük-e a maximálisan megengedett sebességet. Ha ez fennáll, akkor a rekurzív programhívás ugyanúgy befejeződik, mint az előző esetben, és így a robotkart a fékezés megkezdéséig közel maximális sebességgel vezéreljük.

Az eljáráshívás és a rekurzió fogalmának általánosabb megértéséhez hasznos segítséget nyújt a *felélesztés* fogalmának megismerése. Egy eljárás felélesztésén az eljárás hívását követően annak egyszeri lefutását értjük. Példánkban a **„mozgásvez"** nevű eljárás első felélesztése a főprogramban levő eljáráshívást követően jön létre, ahol a 30. sorban a következő utasítás áll:

30. sor mozgásvez (0,0);

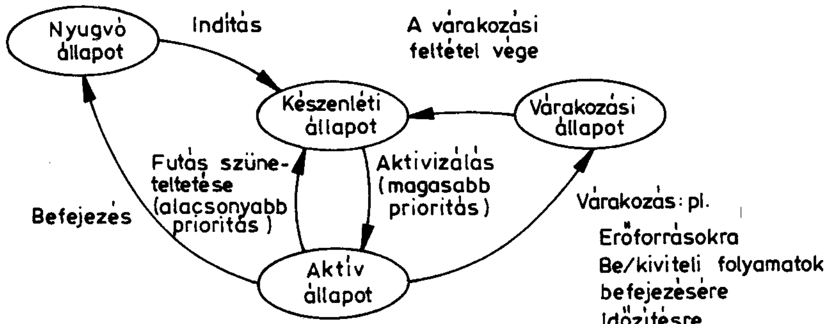
Mielőtt azonban ez a felélesztés befejeződne, megkezdődik maga a második felélesztés, éspedig az eljárásban található első rekurzív eljáráshívás hatására, amely a következő:

21. sor mozgásvez(szakasz,v);

És azt jelenti, hogy a program a **szakasz** és a **v** paraméterek értékeit lokális változókként tárolja le (ugyanúgy, mint a deklararált a lokális változó értékét) és az eljárás második felélesztésében az eljárás egy új adategyüttessel indul (mintegy új adategyüttest fektet fel). Eszerint tehát minden egyes további eljáráshívás mindaddig megszakítja az eljárás végrehajtását, amíg a felírt kritérium teljesül. Ezután befejeződik az eljárás *utolsó* felélesztése, majd folytatódik az *utolsó előtti* felélesztés, más szóval a célp.TENGELY–távols = 106.25 és v = 60 értékekkel végrehajtódik a 22. sorban levő VIAMOVE-utasítás. Ezután pedig végrehajtódnak a további felélesztések a hozzájuk tartozó adatokkal és a megfelelő működtetésvezérlési utasítások is, majd befejeződik az adott felélesztés. A főprogramban a vezérlést a célpontra vonatkozó MOVE-utasítással zárjuk le (31. sor). A rekurzív eljáráshívás lehetőségének abban mutatkozik meg a legnagyobb előnye, hogy tetszőlegesen sok pályapont számítható ki anélkül, hogy előzetesen valamilyen adattömböt kellene felvennünk (az adattömbökre vonatkozóan l. a 3.1.5.1. pontban az Array specifikációt). A mozgás lefékezésekor szükségünk van a gyorsulási adatokra is, mivel azt kívánjuk, hogy a mozgás szimmetrikus lefolyású legyen, és ezeknek az adatoknak a segítségével meghatározható a fékezés pontos kezdőpontja. A számítás végrehajtása előtt azonban nem ismerjük és nem is tudjuk meghatározni, hogy hány közbülső pályapontra lesz szükségünk, ezért egy túlméretezett adattömböt kellene felvennünk, és ezzel sok tárterületet veszítenénk. Dinamikus adatstruktúra felvétele szintén nagy tárterületet igényel és tetemes járulékos számítási időt is maga után von. Erre a problémára tehát a rekurzív eljáráshívás módszere nemcsak elegáns, hanem a probléma természetét is híven tükröző megoldás.

2.2.6. Folyamatok, taskok és társ-rutinok

A feldolgozási *folyamatnak* és a *tasknak* a 2.1. szakaszban bevezetett fogalmait most már pontosabban is körülírhatjuk. Egy feldolgozási folyamaton egy olyan programszakaszt értünk, amely minden egyes aktivizálásakor az eljárásokhoz hasonlóan végigfut egy felélesztésen. A folyamat valamely felélesztését nemcsak maga a program indíthatja el, hanem olyan külső esemény is, mint pl. egy megszakításkérés. A feldolgozási folyamat lefutása más folyamatokkal vagy programokkal párhuzamosan is végbemehet. Így az is előfordulhat, hogy ugyanannak a feldolgozási folyamatnak időben egymás mellett több felélesztése is létezzen, vagy hogy a felélesztések egész sora keletkezzen, és ezeket szekven-



2.12. ábra. Példa folyamatállapotok és állapotátmenetek rendszerére

ciálisan, azaz egymást követően dolgozzák fel. Ugyanúgy a feldolgozási folyamat várakozhat valamilyen meghatározott eseményre, pl. egy másik folyamat befejeződésére. Ezzel kapcsolatban igen hasznosnak bizonyult a folyamatállapotok bevezetése. Egy feldolgozási folyamat az indítást megelőzően lehet pl. *nyugalmi* állapotban. Az indítás után előfordulhat, hogy egyéb folyamatok végrehajtása miatt „készenléti" állapotban marad, mielőtt „aktívvá" válna. Valamely más folyamat – pl. periferiális készülékek beviteli/kiviteli folyamatának – befejeződésére való várakozás a *várakozási állapot.*

A *2.12. ábra* az iménti négy folyamatállapottal jellemzett folyamatdiagramra mutat be egy példát. (E négy folyamatállapoton kívül azonban továbbiak is lehetségesek.) Az egyik folyamatállapotból a másikba való átlépés az operációs rendszer futtatásvezérlésének hatására következik be. Egyes programnyelvekben a feldolgozási folyamatok felélesztései az eljárásokéhoz hasonlóan különböző pontokon kezdődhetnek és különböző pontokon fejeződhetnek be. Ennek a gondolatnak az általánosítása elvezet a *társrutinokhoz,* amelyek egy felélesztés létrejöttét követően egy meghatározott utasítással megszakíthatók, majd ettől a ponttól újból folytathatók (l. még ROHLFING [2.2]). Ezzel a társrutinok közötti egyenrangú, felváltva végbemenő osztott működés válik lehetővé, ellentétben a folyamatok hierarchikus futtatási struktúrájával.

A *2.13. ábra* egy példát mutat be. Az FP főprogram először a K1 rutin ciklikus futását indítja. Ez az első ciklusban még elindítja a K2 rutint is, amely azután szintén ciklusban folytatódik. Az egész folyamatot vagy rendszerparanccsal, vagy a számítógép kikapcsolásával lehet leállítani.

2.2.7. A szinkronizált feldolgozás

Az előző szakaszban tárgyalt egymás mellett párhuzamosan futó szekvenciális feldolgozási folyamatokat egyes esetekben még *szinkronizálni* is kell. Az ilyen szinkronizált feldolgozásra elsősorban akkor van szükség, ha egy folyamat csak valamely másik folyamatrészlet végrehajtását követően folytatható *(2.14. ábra).*

A program szinkronizálása megvalósítható lenne egy közösen használt s *szinkronizáló változó* segítségével, amelyhez mindkét feldolgozási folyamat hozzáférhet *(2.15. ábra).* Ez a megoldás mégsem előnyös, mert az 1. folyamatnak így állandóan le kell kérdeznie az s szinkronizáló változó állapotát, és ezzel fölösleges gépidőt használ (l. pl. WETTSTEIN [2.3]).

Ez a megoldás egyébként még hibás is lehet, ha pl. az 1. folyamat prioritása magasabb, mint a 2. folyamaté, mivel ekkor az 1. folyamatban az ismétlődő lekérdezések következ-

tében a 2. folyamat nem képes beállítani az s szinkronizáló változó értékét. Ezért a szinkronizáló változó fölötti vezérlés lehetőségét célszerűbb az *operációs rendszer* feladatkörébe utalni, különösen olyankor, amikor több folyamat is várakozhat. Ebből a célból ún. *szemaforokat* (jelzőket) használnak. Ezekhez a programozó csak úgy férhet hozzá, ha az operációs rendszer funkcióit működteti. Bináris szemafor esetén pl. a P-művelet – amelyet többnyire WAIT-utasításként (várakozási utasításként) állítanak be – azt idézi elő, hogy az operációs rendszer, ill. az interpretáló program lekérdez egy belső szinkronizáló változót. Ha ez zérussal egyenlő, akkor az éppen futó feldolgozási folyamatot WAIT-utasítással megállítja, egyébként pedig folytatja.

A V-művelet – amelyet többnyire SIGNAL-utasításként állítanak be – azt idézi elő, hogy a WAIT-utasítással megállított feldolgozási folyamat folytatódik *(2.16. ábra)*. Ha az operációs rendszer WAIT-utasítással várakozásra kényszerít egy folyamatot, az nem vesz igénybe számítási időt, és később ezt a folyamatot egy SIGNAL-utasítást követően ismét aktivizálhatja, azaz processzor-időt bocsát rendelkezésre.

Általános formájában a szemaforhoz hozzárendelnek egy *várakozási sort* (listát) is, és így egyszerre több feldolgozási folyamat várakozhat (l. WETTSTEIN [2.3]). Ekkor a belső szinkronizáló változót számlálóként alakítják ki, amely előre, ill. visszafelé számlál. A SIGNAL és WAIT-utasítások hatása a következő:

WAIT-utasítás:

s := s − 1;

IF s≧0 THEN a folyamat feldolgozása folytatódik. ELSE a folyamat várakozási állapotba kerül, és ezt az operációs rendszer bejegyzi az s szemafor várakozási listájába is.

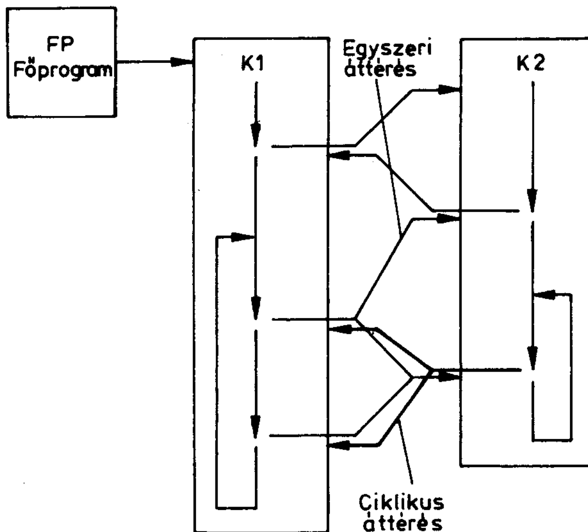
SIGNAL-utasítás:

s := s + 1;

IF s≦0 THEN a várakozási sorból egy folyamat aktivizálódik;

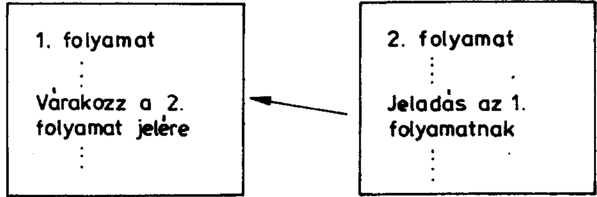
A folyamat feldolgozása folytatódik;

A szemafor általában 1 karakterrel inicializálható. A *2.17. ábra* három feldolgozási folyamatot mutat be. Ezek mindegyike egymást követően ciklikusan egy alkatrészraktárhoz fordul. Egyidejűleg azonban csak egy feldolgozási folyamatnak szabad hozzáférnie a szóban forgó alkatrészraktárhoz, mivel egyébként a robotkarok vagy a rakodóberen-

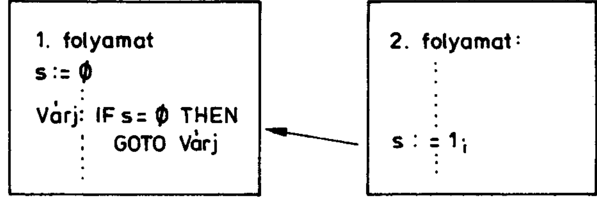


2.13. ábra. A K1 és K2 együttfutó társrutinok ciklikusan váltakozó működésének szemléltetése a visszatérési címekkel

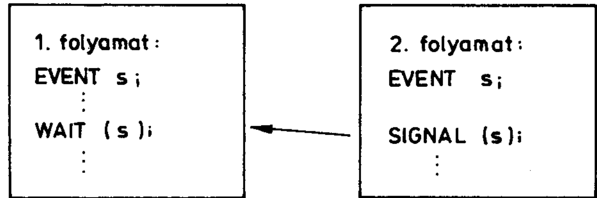
2.14. ábra. Két folyamat szinkronizálása

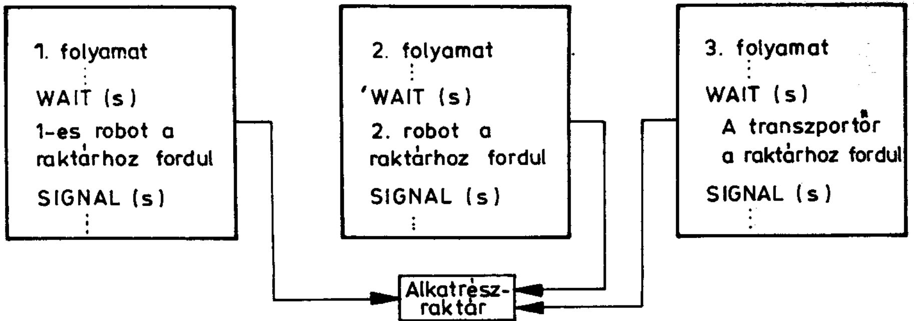


2.15. ábra. Példa az s szinkronizáló változó segítségével vezérelt szinkronizálásra



2.16. ábra. Példa az s szemafor segítségével vezérelt szinkronizálásra





2.17. ábra. Különböző berendezések szinkronizált hozzáférése egy alkatrészraktárhoz

dezés egymással összeütközhetnének. Azt a folyamatrészletet, amelyben a raktárhoz való fordulás bekövetkezik, kizáró vagy más néven *kritikus szakasznak* nevezik. Ha a 2. folyamat elsőnek érkezik a kritikus szakaszhoz, akkor a WAIT-utasítás beállítja a szemafor értékét s = 0-ra. Ha eközben a 3. folyamat a kritikus szakaszhoz ér, akkor ennek WAIT-utasítása már szüneteltetni fogja a 3. folyamatot, és ezt be is jegyzi az s szemafor várakozási sorába.

A szemafor állapota ekkor −1 lesz. Mihelyt a 2. folyamat elhagyja a kritikus szakaszt, a SIGNAL-utasítás hatására s nullára áll, és a várakozási sorból a 3. folyamat aktivizálódik.

Ha eközben még az 1. folyamat is a kritikus szakaszba érkezik, akkor annak futása is felfüggesztődik. A szemafor értéke −1-re állítódik, és csak a 3. folyamat SIGNAL-utasításának hatására lép be az 1. folyamat a kritikus szakaszba.

2.3. Programozási nyelvek szintaktikai szabályai

Minthogy könyvünkben az ipari robotok programozásával kapcsolatban példaként többféle nyelvet is bemutatunk, egyértelműen, tömören és világosan be kell mutatnunk ezek szintaktikai szabályait is. Ehhez elsősorban az ALGOL és a PASCAL nyelv leírásaival összefüggésben olyan formális segédeszközöket kellett létrehozni, amelyeket egyfajta etalonként tudunk használni. Hangsúlyozni kell azonban, hogy az egyes programozási nyelvek szintaktikai szabályainak hiánytalan formalizálása bizonyos nehézségeket vet fel. E probléma megoldására bonyolult nyelvleírási módszereket fejlesztettek ki, ezek tárgyalása azonban túllépné könyvünk kereteit (l. KAPPATSCH [18]). További problémát okoz a szintaktikai és a szemantikai szabályok szétválasztása. Valamely nyelv egyik vagy másik szabályát ugyanis csak nagyon terjedelmes és nehezen áttekinthető szintaktikai leírással lehetne bemutatni. Erre példaként említhető az a szabály, hogy értékadásnál (l. a 3.3. szakaszt) a bal oldali változó típusának meg kell egyeznie a jobb oldali kifejezés kiértékelésénél kapott eredmény típusával.

Ilyen és hasonló okok miatt egyes szabályokat – mint pl. azt, hogy az indexek számának meg kell egyeznie a megfelelő tömbdeklarációban szereplő indexpárok számával – még akkor is a szemantikai szabályok között sorolunk fel, ha azok tisztán szintaktikai követelményeknek tekinthetők.

2.3.1. A nyelvtan szerepe

Mielőtt a szintaktikai leírás formájának részleteire rátérnénk, egy szemléltető példa kapcsán megvilágítjuk a *„nyelvtan"*, azaz a nyelv grammatikájának fogalmát.

A „nyelvtan" kétféle célt szolgál:

Egyrészt lehetővé teszi a szintaktikailag helyes mondatokba foglalható jelsorozatok létrehozását. E lehetőség szövegek leírásánál játszik nagyobb szerepet. A nyelvtan másrészről annak is alapját képezi, hogy elemezni lehessen egy szöveg szintaktikai helyességét. Ez utóbbi lehetőséget a könyvkiadók lektorai minden valószínűség szerint megerősítenék. A programozási nyelvek esetében ezt a feladatot speciális programok veszik át (többnyire valamilyen fordítóprogram).

Az itt következő szemléltető példában a vasúti szerelvények összeállításához a következő nyelvtani jellegű szabályokat használhatjuk. Egy ilyen mintegy nyelvtani szabályok szerint összeállított vasúti szerelvény egy-egy mondatnak felel meg e nyelvtani szabályok által definiált „szerelvény-összeállító nyelvben". Példánkban a vonat „elemei" a következők lehetnek:

E1 : mozdony (m)
E2 : utasszállító személyvagonok (u)
E3 : tehervagonok (t)
E4 : étkezőkocsi (é)
E5 : postavagon (p)

Ezekből összeállítható pl. a következő jelsorozattal felírható szerelvény, ami e nyelv egy „mondatának" tekinthető:

m u u u é u u u p

Ez a jelsorozat egy mozdonyból, három személyvagonból, étkezőkocsiból, majd ismét három személyvagonból és egy postavagonból álló szerelvényt határoz meg. Álljon most a nyelv „nyelvtana" a következő szabályokból:

R1 : Az (S) szerelvény lehet személyszállító vonat (Sz) vagy tehervonat (T)

R2 : A (T) tehervonat-szerelvényt egy vagy két mozdony húzhatja, amely után egy (V) vagonsort kell csatolni

R3 : A (V) vagonsor tetszőleges számú tehervagonból állhat

R4 : Egy (Sz) személyszállító szerelvény lehet helyiérdekű forgalmat lebonyolító személyvonat (H), vagy távolsági forgalmat lebonyolító gyorsvonat (Gy)

R5 : A (H) helyiérdekű személyvonatok a mozdony után kapcsolt (U) utasszállító személyvagonokból, valamint postavagonból állnak

R6 : A (T) távolsági gyorsvonatok a mozdony után kapcsolt (U) utasszállító személyvagonsorból, étkezőkocsiból, egy másik (U) utasszállító vagonsorból és postavagonból állnak.

R7 : Egy (U) utasszállító személyvagonsor tetszőleges számú személyvagonból állhat.

A nyelvtani szabályok leírásához használjuk a következő segédszimbólumokat:

H1 : szerelvény (S)
H2 : személyszállító vonat (Sz)
H3 : tehervonat (T)
H4 : vagonsor (V)
H5 : helyiérdekű személyvonat (H)
H6 : távolsági gyorsvonat (Gy)
H7 : utasszállító vagonsor (U)

Ezek a jelek nem tartoznak a nyelvhez, csupán a vonatkozó nyelvtani szabályok rögzítésére szolgálnak.

Valamely nyelvben használt jelek és jelölések összességét a nyelv *terminális szimbólumainak* (T) vagy más néven alapszimbólumoknak nevezzük, ellentétben a segédszimbólumok halmazával, amelyet *nemterminális szimbólumoknak* (N) nevezünk.

Példánkban tehát

T = {, m, u, t, e, p}

N = {S, Sz, T, V, H, Gy, U}

E két halmaz – T és N – segítségével, valamint a szintaktikai szabályok

R = {R1, R2, R3, R4, R5, R6, R7}

halmazának segítségével most már szerelvényeket (S) állíthatunk össze. Induljunk ki egy *kezdő szimbólumból,* jelen esetben S-ből, majd alkalmazzuk sorban az R szabályok valamelyikét mindaddig, amíg erre lehetőség van. Ezzel az eljárással vasúti szerelvények állíthatók össze. Egy szabály alkalmazását kifejtésnek nevezzük. Az egész eljárást ezek után általánosan úgy írhatjuk le, hogy a kezdő szimbólumból kiindulva egy egész kifejtéssorozatot vezetünk le, és ennek során a nemterminális és terminális szimbólumokból álló jelsorozat végül egy tisztán terminális szimbólumokból álló jelsorozattá alakul.

Példa:

Kezdőszimbólum S

R1 : S→Sz

R4 : Sz→Gy

R6 : Gy→mUeUp

R7 : mUeUp m u u u e u u u p

A szabályok alkalmazását úgy jelöltük, hogy a nyíl bal oldalán a szabály alkalmazása előtti jelsorozatot, jobb oldalán pedig az eredményt tüntettük fel. Az m u u u e u u u p eredmény a már korábban bemutatott vasúti szerelvényt adja.

Ezek után formális módon leírhatjuk szerelvény-összeállító nyelvünk (G) grammatikáját, nyelvtanát:

G = {T, N, R, S}

Ez a T terminális szimbólumok, az N nemterminális szimbólumok, az R szabályok és az S kezdeti szimbólumok halmazából áll. Az R1...R7 szabályokat a megszokott köznapi nyelv segítségével fogalmaztuk meg. További absztrakcióként most már csak a jelsorozatokkal és ezeknek a szabályok adta átalakítási lehetőségeivel kell foglalkoznunk. Példánkból a következő kifejtési szabályokat, ún. produktumokat (P) nyerjük:

P1 : S→Sz P2 : S→T	R1
P3 : T→mV P4 : T→mmV	R2
P5 : V→tV P6 : V→t	R3
P7 : Sz→H P8 : Sz→Gy	R4
P9 : H→mUp	R5
P10: Gy→mUeUp	R6
P11: UeU→uUeUu P12: UeU→ueu P13: Up→uUp P14: Up→up	R7 módosított változatai

Az előbb megfogalmazott R1...R7 szabályok egyikét-másikát produktumokra bontottuk. Az R7-es szabályban megfogalmazott állítást élesebbé tettük azzal, hogy az étkezőkocsi előtt és mögött mindig azonos számú személyszállító vasúti kocsi rákapcsolását írtuk elő. Ez a korábbi szóbeli megfogalmazásból nem derülhetett ki.

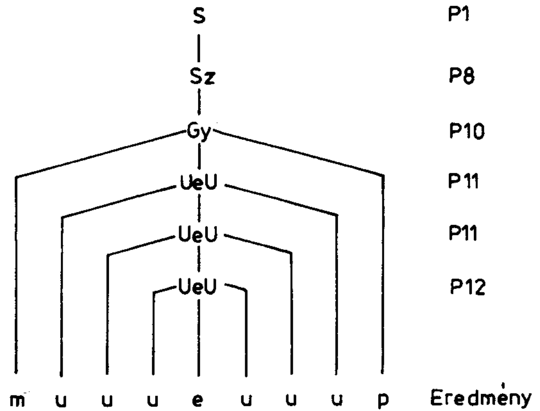
Az iménti példa így a következőképpen is leírható:

P1 : S→Sz
P8 : Sz→Gy
P10: Gy→mUeUp
P11: mUeUp→muUeUup
P11: muUeUup→muuUeUuup
P12: muuUeUuup→muuueuuup

Különösen érdekesek a P11, ill. a P13 kifejtési szabályok. Ezek olyan jelsorozat előállításához vezetnek, amelyek az eredeti jelsorozatból és még két további „u" jelből állnak. Ettől nem változnak meg a szabály alkalmazhatóságának lehetőségei, tehát (elméletileg) egymás után akár végtelen sokszor is alkalmazhatók, és így *tetszőleges* hosszúságú jelsorozatok, ill. vasúti szerelvények is összeállíthatók. Az ilyen szabályokat *rekurzív* tulajdonságú szabályoknak nevezzük, mivel akár „egymásba ágyazva" is alkalmazhatók. A P11 kifejtési szabályból ezenkívül az a tulajdonság is adódik, hogy a gyorsvonati szerelvények mindig azonos számú (u) utasszállító személyvagont fognak tartalmazni az (e) étkezőkocsi előtt, mint mögötte, és így a szerelvény bizonyos szimmetriával fog rendelkezni.

Az úgyszintén rekurzív P5 kifejtési szabály figyelembevételével a (G) grammatika által meghatározott vasúti szerelvény-összeállító nyelv halmazelméleti írásmódban a következőképpen adható meg:

2.18. ábra. Az muuueuuup sorrendben összeállított vasúti szerelvény származtatásakor használt szintaxisdiagram (szintaxisfa)



$$VNy = \{x|x = mt^k|x = mmt^k|x = mu^kp|x = mu^keu^kp\},$$

ahol

k = 1, 2, 3, ...

k = 3-ra x pontosan a példánkban a nyelv egyik elemeként kapott muuueuuup sorrenddel fog megegyezni.

A kifejtés egész folyamatát igen áttekinthetően szemléltethetjük az ún. szintaxis diagram segítségével, ami grafikusan mutatja be azt, ahogyan a kezdeti szimbólumtól elindulva és sorban egymás után alkalmazva az egyes szabályokat, eljutunk a terminális szimbólumok végső jelsorozatáig *(2.18. ábra)*.

2.3.2. A Backus–Naur forma (BNF)

Ezek után egész pontosan meghatározhatjuk a programozási nyelvek grammatikájához tartozó kifejtési szabályok felírásának módját. Az ALGOL 60-as nyelvleírás előkészítésekor kifejlesztettek egy a szintaktikai szabályok leírására szolgáló mintát, amely később a *Backus–Naur forma (BNF)* néven vált ismertté, és a későbbi programnyelvek megalkotásakor is mintául szolgált. Ez a forma elsősorban az előző szakaszban tárgyalt kifejtési eljárás kényelmes és áttekinthető jelölésére alkalmas.

Először is helyettesítsük a kifejtésnél használt nyilat két egymást követő kettősponttal és egy egyenlőségjellel! Így tehát példánk

A→abW

új jelölése

A ::= abW

lesz. Valamely „nyelvtani" rendszer segédszimbólumait, amelyek természetesen nem képezik a nyelv részét, tetszőlegesen jelölhetjük, de kisbetűvel írjuk, és csúcsos zárójelek közé tesszük.

Példa:

A mozgásvezérlési utasítás→MOVE ARMszámjegy TO plista utasítást a

⟨mozgásvezérlési utasítás⟩ ::= MOVE ARM ⟨számjegy⟩ TO ⟨plista⟩

alakba írjuk át.

A MOVE, ARM, TO alapszavakat többnyire csupa nagybetűvel írjuk, még akkor is, ha az éppen definiált nyelv megkülönbözteti a nagy- és a kisbetűket. Újabban a csúcsos zárójeleket el is szokták hagyni.

Lényeges egyszerűsítésekhez vezettek a következő konvenciók, amelyek már a bővített BNF sémát képezik:

1. Ha egy szimbólum többféleképpen alakítható át, akkor az alternatív lehetőségeket egyetlen kifejtési műveletben is összefoglalhatjuk. Ekkor az egyes alternatívákat egymástól függőleges vonallal választjuk el.

   Példa:

   utasítás→mozgásvezérlési utasítás
   utasítás→értékadás

   Ezt a BNF-ben a következőképpen is írhatjuk:

   ⟨utasítás⟩ ::= ⟨mozgásvezérlési utasítás⟩|⟨értékadás⟩

2. Ha egy szimbólum vagy szimbólumok egy sorozata opcionálisan választható, akkor a kifejtési művelet képletében az opcionális szimbólumokat szögletes zárójelbe tesszük.

   Példa:

   Az előző szakaszban példaként tárgyalt vasúti szerelvény-összeállító nyelvben a (T) tehervonatszerelvénynél opcionálisan megadható egy második (m) mozdony is. Az R2-es szabálynak megfelelő

   P3 : T→mV
   P4 : T→mmV

   kifejtési lehetőségeket tehát a BNF-ben így írhatjuk

   ⟨tehervonatszerelvény⟩ ::= MOZDONY [MOZDONY] ⟨tehervagonsor⟩

3. Ha szimbólumok vagy szimbólumok egy sorozata többször is megismételhető, akkor ezeket a jeleket kapcsos zárójelbe tesszük, és felső indexben megadjuk a megengedhető ismétlések számát. Így főleg rekurzív szabályok fejthetők ki igen kényelmesen. Az indexet egyszerűen n jelöli.

   Példa:

   Az előző példában a

   P9: H→mUp
   P13: Up→uUp
   P14: Up→up

   szabályok BNF-ben a következőképpen írhatók:

   ⟨helyiérdekű személyvonat⟩ ::= MOZDONY {SZEMÉLYVAGONOK}$_1^n$ POSTAVAGON

   Az alsó indexben szereplő 1 szám azt jelenti, hogy a „SZEMÉLYVAGONOK"-nak legalább egyszer szerepelnie kell. Ha az ismételhető szimbólumsorozat akár teljesen el is hagyható, akkor az alsó indexbe zérust írunk.

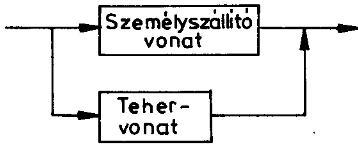
4. Ha egy kifejtési szabály különféle alternatívákat nyújt és ezeknek van közös része, akkor ez zárójelbe téve kiemelhető, az alternatív lehetőségek többi részét pedig ilyenkor kapcsos zárójelbe tesszük.

Befejezésül megemlítjük még, hogy a BNF-nek vannak további változatai. Ezek a jelölésmód egyes részleteiben különbözhetnek ugyan egymástól, a programozási nyelvek szintaktikai szabályait azonban összességükben, teljességükben és áttekinthetően írhatjuk le segítségükkel. A bemutatott jelölésmód alkalmazásánál nehézségek merülhetnek fel olyan esetben, amikor a leírni kívánt nyelvben használt jelkészlet olyan jeleket is

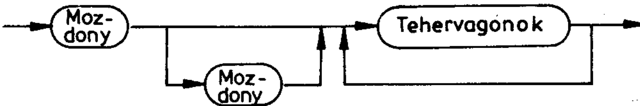
Megnevezés	BNF	Szintaxisdiagram
Alapszimbólum	A	A
Nemterminális szimbólum	⟨a⟩	a
Jelsorozat	⟨a⟩ A B ⟨b⟩ ...	a A B b
Kifejtési szabály	⟨z⟩: : =⟨x⟩ A	z x A
Alternatívák	⟨a₁⟩|⟨a₂⟩|...|⟨aₙ⟩	a_1 a_2 a_n
Opcionális lehetőségek	⟨z⟩: : = [⟨opt⟩] ⟨a⟩	z opt a
"Üres" kifejtési lehetőség, mint alternatíva	⟨z⟩: : = ⟨x⟩|	z x
Ismétlés	⟨x⟩ $\{\langle y\rangle\}_1^n$ B	x y B
Az ismétlés, mint opcinális lehetőség	⟨x⟩ $\{\langle y\rangle\}_0^n$ B	x y B

2.19. ábra. A BNF ábrázolása szintaxisdiagramokkal

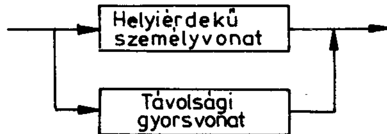
Szerelvény



Tehervonat



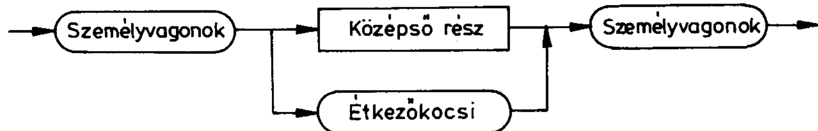
Személyszállító vonat



Helyiérdekű személyvonat

Távolsági gyorsvonat

Középső rész



2.20. ábra. A vasúti szerelvényt összeállító nyelv szintaxisdiagramjai

2.21. ábra. A „távolsági gyorsvonat" jelölésére szolgáló szimbólumok szintaxisdiagramja

tartalmaz, amelyeket a BNF is használ, pl. az alternatív lehetőségek, vagy az ismétlések jelölésére. Ebben az esetben pl. úgy segíthetünk magunkon, hogy a kérdéses jeleket aláhúzzuk.

Példa:

⟨mozgásvezérlési utasítás⟩ ::= MOVE ⟨plista⟩

⟨plista⟩ ::= {P ⟨számjegy⟩| }$_0^n$ P ⟨számjegy⟩

Itt most a függőleges vonal nem azt jelenti, hogy alternatív lehetőség következik, hanem a „plista" elemeit választja el egymástól. Egy ilyen mozgásvezérlési utasításra tehát a következő utasítás lehet egy példa:

MOVE P3|P5|P6

2.3.3. A szintaxisdiagram

Bár a félreértések elkerülésére a BNF segítségével felírt formális definíciók nélkülözhetetlenek, ezek a definíciók ugyanakkor grafikus segédeszközök használatával lényegesen szemléletesebbé is tehetők. Erre a célra az ún. grafikus *szintaxisdiagramok* használatosak. A BNF különféle kifejezéseihez egyértelműen hozzárendelhetők bizonyos grafikus szimbólumok *(2.19. ábra)*. Ezeknek a grafikus szimbólumoknak a segítségével áttekinthetően bemutatható a 2.3.1. pontban tárgyalt szerelvény-összeállító nyelv összes szabálya *(2.20. ábra)*. Mivel olyan szimmetrikus elrendezést akartunk elérni, amelyben az étkezőkocsi előtt és mögött levő utaskocsik száma azonos, be kellett vezetni a „középső rész" feliratú újabb nemterminális szimbólumot. Ha ezt a feltételt nem kötjük ki, akkor a távolsági gyorsvonat egyszerűbben is ábrázolható *(2.21. ábra)*.

2.4. A nyelv kialakításának főbb szempontjai az ipari robotok programozásánál

Az ipari robotok a számítógépek felhasználásának egy új területét jelentik. Ez megköveteli egy sor egészen sajátságos programozási eljárás és módszer alkalmazását. Az új programozási technikák jellemvonásait éppen a vezérelni kívánt ipari robotok alakították ki és alakítják ma is. Ez a tény az ipari robotok programozási nyelveinek megtervezésekor is teljesen nyilvánvaló, hiszen magukat az egyes nyelveket kell a követelmények súlypontjától függően mindig új felfogásban megtervezni. A felfogásbeli különbségek ezenkívül három fontos cél elérését is előmozdítják:

1. Az alkalmazók kezelési és programozási kényelmének megjavítását;
2. A meglevő szabványokhoz való illeszthetőséget;
3. A rendszerfejlesztéssel kapcsolatos ráfordítások csökkentését.

Az első pont bővebb magyarázatra szorul. Az, hogy minden programozási rendszer a kezelési és programozási kényelmet is javítja, eléggé magától értetődő. Az ipari robotok programozástechnikájában azonban azt a különleges kérdést kell megoldani, hogy hogyan lehet nyelvi kifejezési eszközökkel térbeli mozgásokat és teljes mozgássorozatokat leírni. A gépi implementálás szemszögéből nézve ennek a legegyszerűbb módja az, hogy a robotkoordináták (kéz és ízületi koordináták) egész sorozatát adjuk meg, melyek értékeit azután a program kiadja a robotkar vezérlőegységének. A programozó szempontjából azonban egy ilyen módszer nagyon fáradságos és időrabló lenne, hiszen a robot mozgását a forgómozgást végző csuklók pontos szögadataival, ill. az ún. csúszkák (pl. teleszkópok) pontos eltolási értékeivel kellene előre meghatároznunk.

Bizonyos mértékű javulást jelentene, ha a programozás derékszögű koordináta-rendszerben történne. Ekkor azonban a programrendszernek gondoskodnia kell a derékszögű koordináta-rendszerből a robot saját, ún. ízületi koordináta-rendszereibe történő koordinátatranszformációról és az inverz transzformációkról. Ezen kívül azonban meg kellene még adni az összes pályapontot, azaz a pályák kezdőpontjait, valamint azokat a közbülső pontokat, ahol a mozgás iránya, ill. a sebessége megváltozik, valamint a pálya végpontját is. Ez pedig bonyolultabb mozdulatoknál már igen komoly követelményeket támaszt a programozó képességeivel, főleg geometriai látásmódjával szemben. Ez az az ok, ami miatt a programozás során magasabb szintű absztrakciót kell megvalósítani, hogy ezután a programozónak már ne az egyes pályapontokat és magukat a mozdulatokat kelljen leírnia, hanem azt a műveletet, amelyet végre kíván hajtatni a robottal. Gondolunk itt olyanokra, mint pl. „fogj meg egy tárgyat" vagy általában „szereld az A tárgyat a B tárgyra". Az ilyen utasítások, amelyek az ember térbeli látásmódjához és gondolkodásmódjához tökéletesen illeszkednek, megkövetelik egy olyan rendszer kialakítását, amely képes a hiányzó, vagy csak rejtett, implicit módon jelenlevő információt a meglevő adatbázisból előteremteni, és ennek segítségével generálni is tudja a megfelelő mozdulatokat.

2.4.1. A „Frame" kísérőkoordináta-rendszer fogalma

Csaknem valamennyi programozási nyelvben különféle formában szerephez jutott a „Frame" fogalma[2.3.], amely támpontot nyújt a robotkar (vagy valamely ízülete) pozíciójának és orientációjának geometriai leírásához. A robotkar pozícióján (helyzetén) a robot által megfogott szerszám végpontját, vagy pedig a megfogószerkezet ún. megfogási pontját értjük (angolul Tool Center Point, TCP). Az orientáció megadja, hogy melyik irányból (felülről, alulról, oldalról stb.) és a szerszám vagy megfogószerkezet milyen mértékű elfordításával közelítjük meg az adott pozíciót (helyzetet). A robotkoordinátákban – más néven ízületi koordinátákban – történő programozással ellentétben (azaz a csukló szögadataival, ill. az eltolási adatokkal ellentétben) a „Frame" kísérőkoordináta-rendszer a pályapontbeli pozíciót a derékszögű koordinátákkal írja le, az orientációt pedig a koordinátatengelyek, ill. az ízületek vagy a megfogószerkezet tengelyei körüli elfordulási szögek segítségével adja meg. Ezekhez az adatokhoz a viszonyítási rendszert egy valós térbeli báziskoordináta-rendszer adja, amelynek az origója legyen pl. a munkapad valamelyik sarka, Z tengelye mutasson felfelé, X tengelye előre, Y tengelye pedig oldalra. A báziskoordináta-rendszert már a programrendszer installálásakor definiálni

2.3. A frame – amint azt a 2.2. lábjegyzetnél már módja volt látnia az Olvasónak – keretet jelent. Később „tömb" értelemben is használatos, amely egy kísérőkoordináta-rendszert, azaz frame koordináta-rendszert határoz meg (A ford. megjegyzése)

lehet. Ezután a „Frame" kísérőkoordináta-rendszer úgy is elképzelhető, mint a báziskoordináta-rendszer egyfajta másolata, melynek origóját egy eltolási vektor határozza meg. Ennek a „Frame" kísérőkoordináta-rendszernek a térbeli orientációja a főtengelyek körüli forgatási adatokkal változtatható meg. Az AL nyelvben pl. a programozó úgy is meghatározhat egy pályapontot (a megfogószerkezet vagy szerszám orientációjával együtt), hogy először deklarál egy Frame típusú programozási objektumot, és ennek az objektumnak – gyakorlatilag egy adattömbnek – explicit értékadással átadja egy háromdimenziós vektor és egy vagy több rotáció értékét.

Példa:

```
FRAME box;          (* a „box", Frame típusú változó *)
    :               (* deklarációja *)
box←FRAME (ROT(XHAT, 180 * GRAD),
        VECTOR (60, 80, 20) * CM);
```

A „Frame" kísérő rendszer tehát egy olyan koordináta-rendszert jelent, amelynek origója báziskoordináta-rendszerben az X = 60 cm, Y = 80 cm és Z = 20 cm-es koordinátákkal jellemezhető. Ezt a „Frame" kísérőkoordináta-rendszert ezenkívül 180°-kal elforgattuk az X tengely körül, aminek következtében a Z tengely függőlegesen lefelé mutat, az Y tengely iránya pedig megfordul (*2.22. ábra*). Az X tengelyt ebben a rendszerben XHAT-nak neveztük el.

Amkor egy ipari robotot a frame által meghatározott pozícióba és orientációba akarunk állítani, ez az AL-programrendszerben a következőt jelenti:

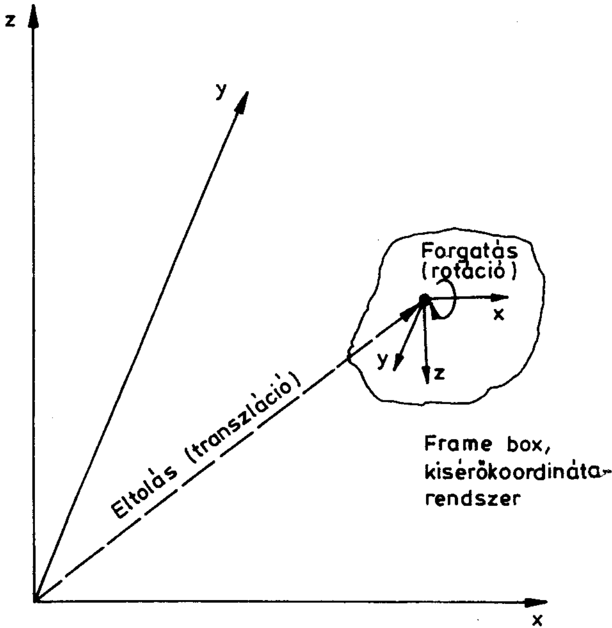
- az eltolási vektor végpontja az ún. TCP-be, azaz a megfogószerkezet pofáinak középpontjába mutat;
- a megfogószerkezet a frame által meghatározott koordináta-rendszer Z tengelyének irányába mutat;
- az Y tengely keresztülhalad a megfogási ponton (*2.23. ábra*).

Definiálunk egy frame-rendszert úgy is, hogy megadjuk valamely másik frame-rendszerhez viszonyított relatív helyzetét. Ekkor az eltolási vektort a frame-rendszer origójához irányítjuk, az elforgatási adatokat pedig a főtengelyekre vonatkoztatva adjuk meg. Ennek a műveletnek a geometriai jelentését a *2.24. ábra* szemlélteti.

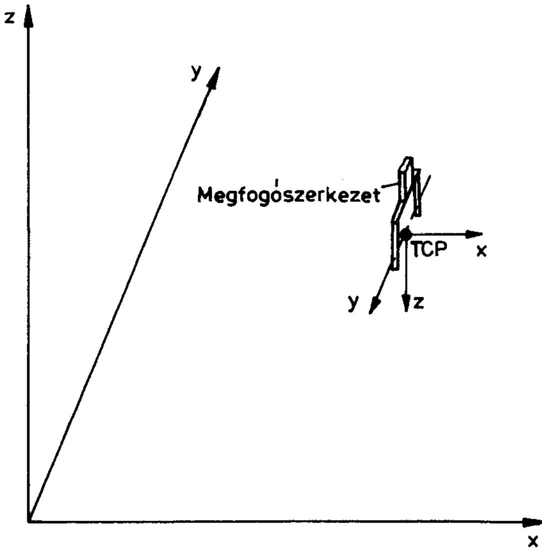
A frame-rendszereket elsősorban olyankor célszerű ilyen relatív módon definiálni, amikor az egyik frame-rendszer valamely adatának megváltozásakor – a transzformáció szabályainak megfelelően – automatikusan megváltoznak az ehhez a frame-rendszerhez relatív módon definiált frame-rendszerek adatai is. Ez olyankor fordulhat elő, amikor pl. magát a robotot kell elmozdítani, vagy elforgatni, vagy amikor a munkadarab több megfogási ponttal rendelkezik, és ezek adatait a munkadarab helyzetváltoztatásával együtt célszerű transzformálni. Robotkoordinátákban végrehajtott programozás esetén ilyenkor minden egyes pályapontot újra kellene programozni. A frame-rendszer derékszögű koordináta-rendszere ugyanakkor közös viszonyítási rendszert jelent mind a robot, mind pedig az érzékelők koordinátái számára.

A frame-rendszereket a programban egy 3×3-as $\mathbf{R}$ rotációs mátrix és egy 1×3-as $\mathbf{v}$ vektor segítségével írhatjuk le. A rotációs mátrixot egy külön adattípus, az ún. „rotációs" adattípus segítségével is deklarálhatjuk. E mátrixot úgy értelmezzük, hogy ennek a mátrixnak egy $\mathbf{v}$ vektorral vett szorzata egy olyan $\mathbf{v}'$ vektort eredményez, amely $\mathbf{v}$-ből a rotációs mátrix által meghatározott forgástengely körül adott szöggel való elforgatással adódik. Azaz

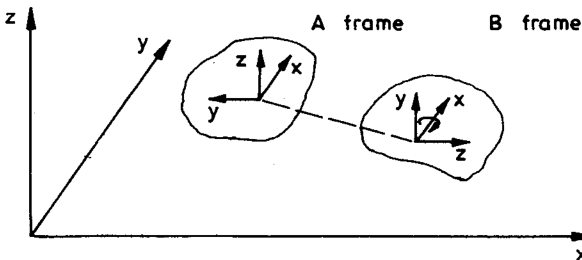
$$\mathbf{v}' = \mathbf{R} \cdot \mathbf{v}$$



2.22. ábra. A **box** nevű frame (kísérő) koordináta-rendszer geometriai értelmezése



2.23. ábra. A megfogószerkezet pozíciója és orientációja, valamint a frame (kísérő) koordináta-rendszer közötti összefüggés



2.24. ábra. Különféle (pl. ízületi) frame koordináta-rendszerek egymáshoz viszonyított helyzete

Az X tengely körüli α szöggel való elforgatást az

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}$$

mátrix adja meg, a *2.25. ábra* szerinti geometriai értelmezésnek megfelelően.
Az Y tengely körüli elforgatást az

$$\mathbf{R}_y = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix}$$

mátrixszal reprezentálhatjuk. A Z tengely körüli forgatást pedig

$$\mathbf{R}_z = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

adja.

Példa:
Forgassuk el a $\mathbf{v}^T = (15, 100, 20)$ vektort az X tengely körül 30°-kal. Az $\mathbf{R}_x$ rotációs mátrix, akkor

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0{,}87 & -0{,}5 \\ 0 & 0{,}5 & 0{,}87 \end{bmatrix}$$

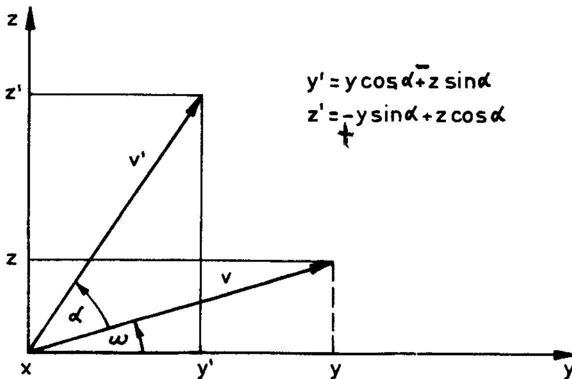
Az elforgatás után a v′ vektor a következő lesz

$$\mathbf{v}' = \mathbf{R}_x \cdot \mathbf{v} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0{,}87 & -0{,}5 \\ 0 & 0{,}5 & 0{,}87 \end{bmatrix} * \begin{bmatrix} 15 \\ 100 \\ 20 \end{bmatrix} = \begin{bmatrix} 15 \\ 77 \\ 67{,}4 \end{bmatrix}$$

Belső számításokhoz célszerű a rotációs mátrixot és az eltolási mátrixot összevonni egy 4×4-es ún. *Denavit–Hartenberg-mátrixba* (DH-mátrixba) (l. DENAVIT [2.4])

$$\textbf{DH}\text{-mátrix} = \begin{bmatrix} [\mathbf{R}] & [\mathbf{v}] \\ 000 & 1 \end{bmatrix}$$

A DH-mátrix egy *homogén transzformációt,* ill. egy frame-rendszert ír le. Ennek a transzformációnak a matematikai hátterét azonban részletekbe menően nem áll módunkban tárgyalni (ezzel kapcsolatban l. BLUME [2.5]).



2.25. ábra. Az x koordinátatengely körüli rotáció értelmezése

Példa:
Tegyük fel, hogy a programozó egy „cél" nevű frame-et határoz meg a következők szerint
cel←FRAME (ROT (ZHAT, 180 * GRAD), VECTOR(55, −37, 23,) * CM);
Ekkor a FRAME belső ábrázolására használt DH-mátrix a következő

$$\mathbf{DH}_{cel} = \begin{bmatrix} -1 & 0 & 0 & 55 \\ 0 & -1 & 0 & -37 \\ 0 & 0 & 1 & 23 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ebből a felírásmódból azonnal kitűnik, hogy ez a transzformáció a frame X, ill. Y tengelyeit megfordítja.

Egy frame-nek a DH-mátrix – más néven *transzformációs mátrix* – segítségével való ábrázolása lényegesen megkönnyíti a frame rotációjának és eltolásának kiszámítását. Ehhez csak fel kell állítani a 4 × 4-es DH-mátrixot a rotációs mátrix, ill. az eltolási vektor alapján, és ezt be kell szorozni a frame-mátrixszal. Vegyük észre, hogy a rotációs mátrixszal való szorzás során a frame-mátrix helyzetvektora változatlan marad.

Példa:
Forgassuk el a **cél**-frame-et a Z tengely körül −180°-kal majd toljuk el az X= −55 cm, Y=37 cm, Z=0 cm pontba. Legyen a transzformált-frame neve **ujcel:**
ujcel= eltolás * rotáció * cel

$$\text{ujcel} = \begin{bmatrix} 1 & 0 & 0 & -55 \\ 0 & 1 & 0 & 37 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 0 & 55 \\ 0 & -1 & 0 & -37 \\ 0 & 0 & 1 & 23 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{ujcel} = \begin{bmatrix} 1 & 0 & 0 & -55 \\ 0 & 1 & 0 & 37 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 55 \\ 0 & 1 & 0 & -37 \\ 0 & 0 & 1 & 23 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{ujcel} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 23 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

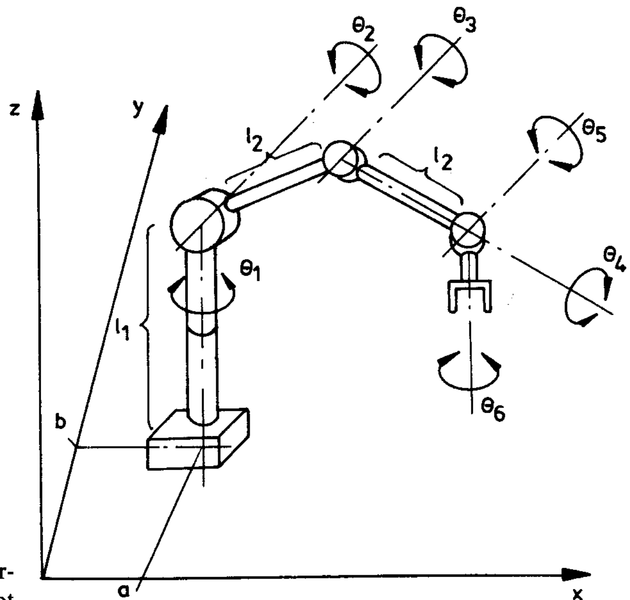
Az eredményül kapott „ujcel" nevű frame orientációja tehát ugyanaz, mint a báziskoordináta-rendszeré, azaz Z tengelye ismét függőlegesen felfelé mutat, origója pedig az X=0, Y=0, Z=23-as pontban van.

2.4.2. A koordinátatranszformáció és a pályatervezés

A frame-rendszer használata lehetővé teszi, hogy a programozó a pozíció és az orientáció megadásánál tetszőleges térbeli (környezeti) koordináta-rendszert, tehát derékszögű koordinátákat vagy polárkoordinátákat, vagy bármi mást használhasson. A robotkart azonban mindig robotkoordinátákban (ún. ízületi koordinátákban) vezérlik, ezért csak kivételes esetben vehetők át közvetlenül a környezeti koordináták. Általában tehát a programrendszernek vagy a futtató rendszernek kell olyan modulokkal rendelkeznie, amelyek végrehajtják a koordinátatranszformációt. A környezeti koordináta-rendszernek robotkoordinátákba (ízületi koordinátákba) történő áttranszformálására azért van szükség, hogy a robotkart valamilyen explicit értékmegadással beprogramozott frame-helyzetbe is közvetlenül be lehessen állítani. Ha a programozáskor egy betanítási eljárást követő vezérléssel elért pályapont frame-rendszerét akarjuk előállítani, akkor fordított irányú (inverz) transzformációra van szükségünk, azaz robotkoordinátákból környezeti koordinátákba való transzformációra.

A következőkben egy példa kapcsán részletesebben is megvilágítjuk az imént említett két irányba történő transzformáció lényegét (l. PAUL [2.6]-ot). Röviden érintjük még a pályatervezés kérdését is, amelynek segítségével a programozó által megadott két vagy több frame-helyzet között irányított pályavezérlést lehet létrehozni.

A transzformációs egyenletek levezetését egy hipotetikus hattengelyű robot példáján demonstráljuk. Ennek felépítése nagyon hasonlít a működésben levő szerelőrobotok szerkezetéhez, így pl. a PUMA 600-as robotéhoz. A *2.26. ábrán* látható a robot geomet-



2.26. ábra. Egyetlen megfogószerkezettel ellátott hattengelyű robot geometriai viszonyai

riája, valamint az egyes szögek elnevezései. Azt a helyzetet, amikor valamennyi csukló elfordulási szögértéke nullával egyenlő, alaphelyzetnek nevezzük. Definiáljuk az alaphelyzetet a következőképpen (l. még a *2.27. ábrát*):

- Θ_1 értéke akkor nulla, amikor a robotkar a báziskoordináta-rendszer x tengelyének irányába mutat;
- Θ_2 értéke akkor nulla, amikor a robot felső karja az xy síkkal párhuzamos;
- Θ_3 értéke akkor nulla, amikor az alsó kar függőlegesen felfelé mutat;
- Θ_4 értéke akkor nulla, amikor a befogópofák megfogási pontjain (TCP-n) átmenő egyenes az y tengellyel párhuzamos;
- Θ_5 értéke akkor nulla, amikor a megfogószerkezet a legutolsó robotízület irányába mutat; végül
- Θ_6 értéke akkor nulla, amikor a Θ_4-nél felírt feltétel teljesül.

Mármost a pozíciónak és az orientációnak a vezérlését fel lehet úgy bontani, hogy a Θ_1, Θ_2 és Θ_3 szögeket az előírt pozíció megközelítésének érdekében változtatjuk, míg Θ_4, Θ_5 és Θ_6 szögeket az előírt orientáció beállítására tartjuk fenn. Ekkor a Θ_1, Θ_2, Θ_3 segítségével beállított pont nem lehet azonos a tulajdonképpeni megfogási ponttal (azaz a TCP-vel, a Tool Center Point-tal), hanem csak azzal a P ponttal, amelyet úgy kapunk, hogy a megfogási pontból az előírt megfogási iránnyal ellenkező (negatív) irányba felmérjük a megfogószerkezet hosszát *(2.28. ábra)*.

Tekintsük a következő off-line programrészletet, amely azt az F frame-helyzetet írja le, amelybe a robotkart be kívánjuk állítani:

F := FRAME (ROT (YHAT,180) * ROT(XHAT,30),VECTOR(24,18,10) *
* CM);

A megfogási pont tehát x = 24 cm, y = 18 cm és z = 10 cm-nél van. A frame koordinátarendszert 180°-kal elforgattuk az y tengely körül és 30°-kal az x tengely körül. Az F frame-rendszert a következő mátrix segítségével ábrázolhatjuk:

$$\mathbf{F} = \begin{bmatrix} 1 & 0 & 0 & 24 \\ 0 & 1 & 0 & 18 \\ 0 & 0 & 1 & 10 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0{,}5 * \sqrt{3} & -0{,}5 & 0 \\ 0 & 0{,}5 & +0{,}5 * \sqrt{3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{F} = \begin{bmatrix} -1 & 0 & 0 & 24 \\ 0 & 0{,}866 & -0{,}5 & 18 \\ 0 & -0{,}5 & -0{,}866 & 10 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

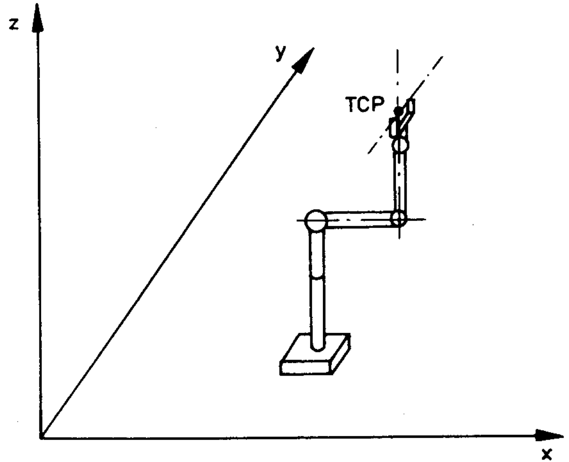
A mátrix harmadik oszlopa a frame koordináta-rendszer z tengelyirányú egységvektorát jelenti a báziskoordináta-rendszer koordinátáival kifejezve:

$$(0,0,1)^T_{\text{frame}} = (0,\ -0{,}5,\ -0{,}866)^T_{\text{bázis}})$$

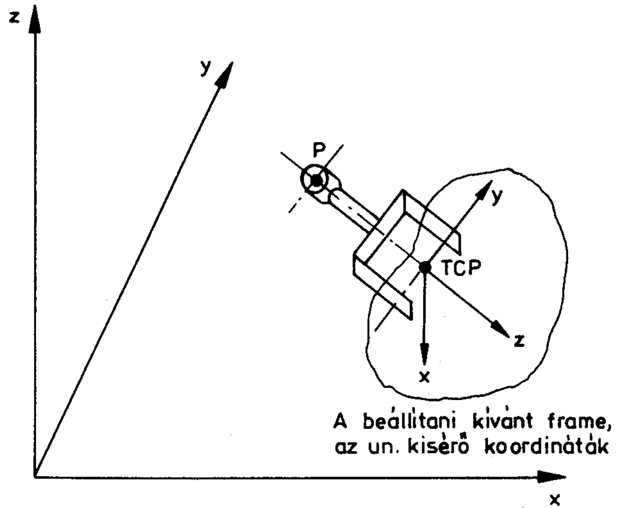
Legyen a megfogószerszám hossza a megfogási ponttól a Θ_5-ös szög állítására való ízületi forgáspontjáig L = 6 cm. Ekkor a vezérléssel beállítandó P pont a következőképpen számítható ki:

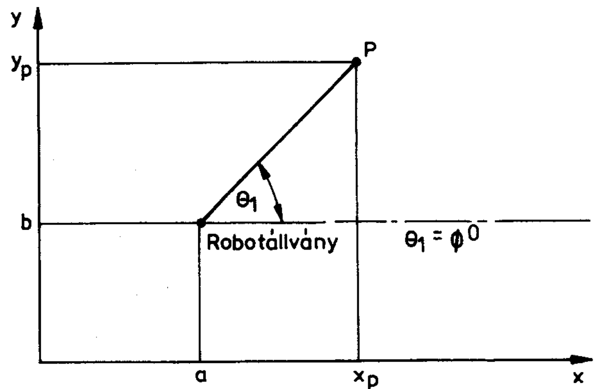
$$P = P_{\text{frame}} - (0,\ -0{,}5,\ -0{,}866)^T * L$$
$$P = (24,18,10)^T - (0,-0{,}5,-0{,}866)^T * 6$$
$$P = (24,21,15{,}2)^T$$

2.27. ábra. A robotkarok alaphelyzetének definiálása



2.28. ábra. A megfogószerkezet jellemző adatai: frame (keret) a megfogási pont, a vezérlendő P pont





2.29. ábra. A Θ_1 szög kiszámítása

A Θ_1 szög segítségével először beállítjuk a P pont xy síkban vett vetületét. A *2.29. ábra* leegyszerűsítve mutatja a Θ_1 szög származtatását. Legyen a robotállvány középpontja az $x=a$, $y=b$ pontban (a báziskoordináta-rendszerhez viszonyítva), a P pont koordinátái pedig $x_p=24$, $y_p=21$. Ekkor a Θ_1 szögre fennáll

$$\text{tg}\ \Theta_1 = \frac{y_p-b}{x_p-a}$$

$$\Theta_1 = \text{arctg}\ ((y_p-b)/(x_p-a))$$

Legyen pl.

$a=14$

$b=11,$

ekkor

Θ_1 értéke

$$\Theta_1 = \text{arctg}((21-11)/(24-14))$$

$$\Theta_1 = 45°$$

adódik.

A Θ_2 és Θ_3 szög kiszámításához fel kell használnunk a következő hosszúságadatokat:

l_1 = 40 cm = az xy síktól a második csuklóig mért távolság;

l_2 = 20 cm = a robot alsó karjának és felső karjának hossza.

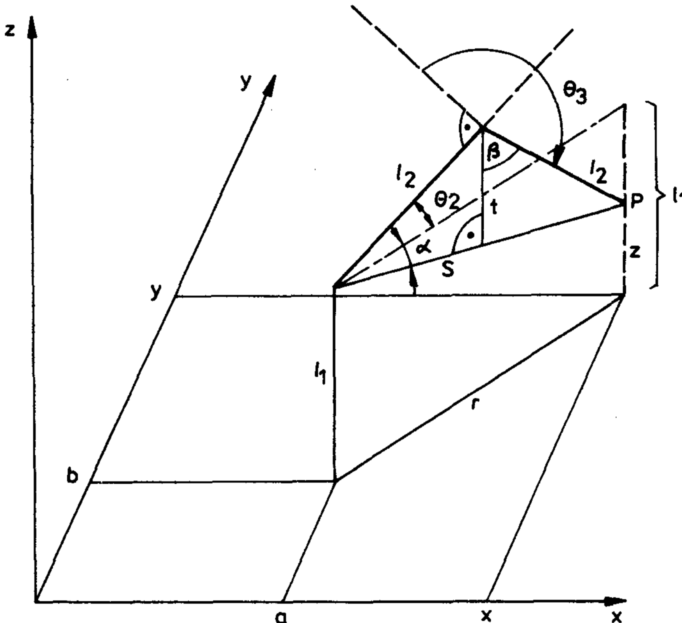
A *2.30. ábrán* látható vázlat a Θ_2 és Θ_3 szögek levezetéséhez szükséges geometriai összefüggéseket szemlélteti.

Fennáll:

$$r = \sqrt{(x_p-a)^2+(y_p-b)^2}$$

$$s = \sqrt{(x_p-a)^2+(y_p-b)^2+(z_p-l_1)^2}$$

$$t = 0{,}5\sqrt{4*l_2^2-s^2}$$



2.30. ábra. A Θ_2 és Θ_3 szögek meghatározása

Minthogy az s, l_2 és l_3 szakaszok egy egyenlő oldalú háromszöget

$$\mathrm{tg}\,(\alpha-\Theta_2) = 2 * t/s$$
$$\alpha-\Theta_2 = \mathrm{arctg}\,((\sqrt{4 * l_2^2 - s^2})/s)$$
$$\Theta_2 = \alpha - \mathrm{arctg}\,(\sqrt{(4 * l_2^2-(x_p-a)^2-(y_p-b)^2-(z_p-l_1)^2)/((x_p-a)^2+(y_p-b)^2+(z_p-l_1)^2)})$$

Az α szögre fennáll

$$\mathrm{tg}\,\alpha = (l_1-z_p)/r$$
$$\alpha = \mathrm{arctg}\,((l_1-z_p)/\sqrt{(x_p-a)^2+(y_p-b)^2})$$

Tehát

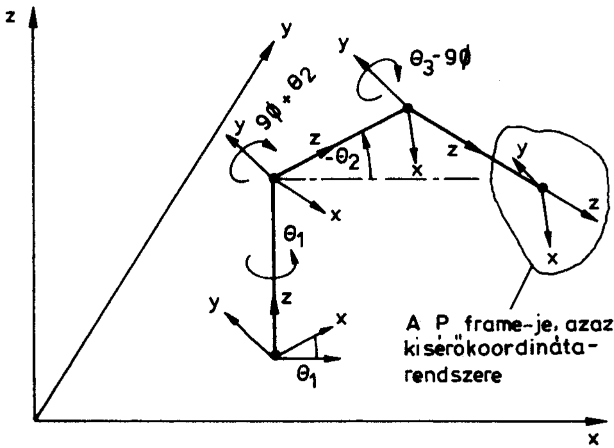
$$\Theta_2 = \mathrm{arctg}\,((l_1-z_p)/\sqrt{(x_p-a)^2+(y_p-b)^2}) - \mathrm{arctg}\,(\sqrt{(4 * l_2^2-(x_p-a)^2-(y_p-b)^2-(z_p-l_1)^2)/((x_p-a)^2+(y_p-b)^2+(z_p-l_1)^2)})$$

Behelyettesítve a felvett számértékeket, kapjuk, hogy

$$\Theta_2 = \mathrm{arctg}\,((40-15{,}2)/\sqrt{(24-14)^2+(21-11)^2}) -$$
$$- \mathrm{arctg}\,(\sqrt{4 * 20^2-(24-14)^2-(21-11)^2-(15{,}2-40)^2)/((24-14)^2+(21-11)^2+(15{,}2-40)^2)})$$
$$\Theta_2 = \mathrm{arctg}\,1{,}7536 - \mathrm{arctg}\,0{,}98\,137$$
$$\Theta_2 = 60{,}3^\circ - 44{,}5^\circ$$
$$\Theta_2 = 15{,}8^\circ$$

A Θ_3 szögre fennáll, hogy

$$\Theta_3 = 270 - 2 * \beta$$
$$= 270 - 2 * (90-\alpha+\Theta_2)$$
$$\Theta_3 = 90 + 2 * \alpha - 2 * \Theta_2$$



2.31. ábra. A P frame-rendszer orientációjának kiszámítása

Az α-ra és a Θ_2-re kiszámított értékeket behelyettesítve kapjuk

$$\Theta_3 = 90 + 2 * 60{,}3 - 2 * 15{,}8$$
$$\Theta_3 = 179°$$

Ezek után az orientáció meghatározására szolgáló Θ_4, Θ_5 és Θ_6 kiszámításához más utat választunk. (Ez amiatt lehetséges, mert a robot speciális geometriai kialakítása folytán e három forgástengely egy pontban metszi egymást.) Abból a megfontolásból indulunk ki, hogy a már ismert Θ_1, Θ_2, Θ_3 szögekből a legutolsó ízület orientációja egyszerűen meghatározható transzformációk segítségével. Minthogy a megfogószerkezet orientációját abban a pozícióban, ahová vezérelni kívánjuk, az F frame-rendszer adja meg, így kiszámítható az a transzformációs mátrix is, amely a legutolsó ízület orientációját az F frame-rendszerbe viszi át. Ebből a szempontból elegendő, ha csak az orientációt vesszük figyelembe, az eltolási értékeket pedig zérusnak vesszük. A *2.31. ábra* az elforgatásnak azt a rendszerét szemlélteti, amelyek segítségével a báziskoordináta-rendszerből először a P frame-rendszerbe jutunk, ahol a rendszer tengelye a legutolsó ízület irányába mutat.

Az első forgatás a báziskoordináta-rendszer z tengelye körüli Θ_1 szöggel való elforgatás, miáltal az x tengely a második robotkar irányába fog mutatni. Ezután az új y tengely körül egy $90° + \Theta_2$ szöggel történő elforgatást végzünk, aminek eredményeképpen az új z tengely a második ízület irányába fog mutatni. Végül az y tengely körül $\Theta_3 - 90°$-kal forgatjuk a rendszert, aminek következtében az új z tengely a harmadik ízület irányába mutat, amelyen a megfogószerkezet van rögzítve. A P frame koordináta-rendszere a következő P orientációs mátrixszal jellemezhető

$$P = ROT(z, \Theta_1) * ROT(y, 90 + \Theta_2) * ROT(y, \Theta_3 - 90)$$

A következő részben a következő rövidített jelöléseket használjuk:

$\sin \Theta_i = si$

$\cos \Theta_i = ci$

$\sin(\Theta_i + \Theta_k) = si * ck + ci * sk = sik$

$\cos(\Theta_i + \Theta_k) = ci * ci - si * sk = cik$

Ekkor P-t a következőképpen írhatjuk:

$$\mathbf{P} = \begin{bmatrix} c1 & -s1 & 0 & 0 \\ s1 & c1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -s2 & 0 & c2 & 0 \\ 0 & 1 & 0 & 0 \\ -c2 & 0 & -s2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s3 & 0 & -c3 & 0 \\ 0 & 1 & 0 & 0 \\ c3 & 0 & s3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{P} = \begin{bmatrix} -c1 * s2 & -s1 & c1 * c2 & 0 \\ -s1 * s2 & c1 & s1 * c2 & 0 \\ -c2 & 0 & -s2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s3 & 0 & -c3 & 0 \\ 0 & 1 & 0 & 0 \\ c3 & 0 & s3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{P} = \begin{bmatrix} -c1 * s2 * s3 + c1 * c2 * c3 & -s1 & c1 * s2 * c3 + c1 * c2 * s3 & 0 \\ -s1 * s2 * s3 + s1 * c2 * c3 & c1 & s1 * s2 * c3 + s1 * c2 * s3 & 0 \\ -c2 * s3 - s2 * c3 & 0 & c2 * c3 - s2 * s3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{P} = \begin{bmatrix} c1 * c23 & -s1 & c1 * s23 & 0 \\ s1 * c23 & c1 & s1 * s23 & 0 \\ -s23 & 0 & c23 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Annak érdekében, hogy az F frame koordináta-rendszerrel egybevágó rendszert kapjunk, a P frame-rendszerét el kell forgatnunk, éspedig a z tengely körül a Θ_4 szöggel, az új y tengely körül Θ_5 szöggel, az új z tengely körül pedig Θ_6 szögértékkel. (Itt most csak a rotációkat vettük figyelembe.)

$F = P * ROT(z, \Theta_4) * ROT(y, \Theta_5) * ROT(z, \Theta_6)$

Ezt balról szorozva az inverz P^{-1} mátrixszal, a

$P^{-1} F = ROT(z, \Theta_4) * ROT(y, \Theta_5) * ROT(z, \Theta_6)$

$$\mathbf{P}^{-1}\mathbf{F} = \begin{bmatrix} c4 & -s4 & 0 & 0 \\ s4 & c4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c5 & 0 & s5 & 0 \\ 0 & 1 & 0 & 0 \\ -s5 & 0 & c5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c6 & -s6 & 0 & 0 \\ s6 & c6 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{P}^{-1}\mathbf{F} = \begin{bmatrix} c4 & -s4 & 0 & 0 \\ s4 & c4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c5c6 & -c5s6 & s5 & 0 \\ s6 & c6 & 0 & 0 \\ -s5c6 & s5s6 & c5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{P}^{-1}\mathbf{F} = \begin{bmatrix} c4c5c6 - s4s6 & -c4c5s6 - s4c6 & c4s5 & 0 \\ s4c5c6 + c4s6 & -s4c5s6 + c4c6 & s4s5 & 0 \\ -s5c6 & s5s6 & c5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

egyenletet kapjuk.

Ha most invertáljuk a már ismert P mátrixot és F helyébe behelyettesítjük az általános rotációs mátrix jelölést, kapjuk hogy

$$P^{-1}F = \begin{bmatrix} c1c23 & s1c23 & -s23 & 0 \\ -s1 & c1 & 0 & 0 \\ c1s23 & s1s23 & c23 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} tx & ox & ax & 0 \\ ty & oy & ay & 0 \\ tz & oz & az & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P^{-1}F = \begin{bmatrix} c1c23tx+s1c23ty-s23tz & c1c23ox+s1c23oy-s23oz & c1c23ax+s1c23ay-s23az & 0 \\ -s1tx+c1ty & -s1ox+c1oy & -s1ax+c1ay & \\ c1s23tx+s1s23ty+c23tz & c1s23ox+s1s23oy+c23oz & c1s23ax+s1s23ay+c23az & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ekkor a $P^{-1}F$ szorzatot kifejező két mátrix elemeit egyenként egymással egyenlővé tehetjük, amiből a

$$c4s5 = c1c23ax + s1c23ay - s23az$$
$$s4s5 = -s1ax + c1ay$$

egyenletrendszert kapjuk.
Ebből a Θ_4 szögre

$$\Theta_4 = \mathrm{arctg}\left(\frac{-s1ax+c1ay}{c1c23ax+s1c23ay-s23az}\right)$$

adódik.

A Θ_6 szög a

$$-s5c6 = c1s23tx + s1s23ty + c23tz$$
$$s5s6 = c1s23ox + s1s23oy + c23oz$$

egyenletekből számítható, amiből

$$\Theta_6 = \mathrm{arctg}\left(\frac{c1s23ox+s1s23oy+c23oz}{-c1s23tx-s1s23ty-c23tz}\right)$$

Végezetül a Θ_5 szögre az

$$s4s5 = -s1ax+c1ay$$
$$c5 = c1s23ax+s1s23ay+c23az$$

egyenletekből

$$\Theta_5 = \mathrm{arctg}\left(\frac{-s1ax+c1ay}{s4(c1s23ax+s1s23ay+c23az)}\right) \quad \text{adódik.}$$

A gyakorlati számítás megvalósítása érdekében ezt az algoritmust nyilvánvalóan lehet még tökéletesíteni és esetleg optimálni is. Így pl. a Θ_5 kiszámításakor célszerű megvizsgálni, hogy Θ_4 értéke egyenlő-e 0°-kal, ill. 180°-kal. Ebben az esetben más képletet kell választanunk Θ_5 kiszámításához, hogy elkerüljük a nullával való osztást és hogy Θ_4 előjelére vonatkozó hiányzó információnkat is pótoljuk:

Ehhez a következő két egyenletből indulunk ki:

$$c4s5 = c1c23ax + s1c23ay - s23az$$
$$c5 = c1s23ax + s1s23ay + c23az$$

Ebből a Θ szög

$$\Theta_5 = \operatorname{arctg}\left(\frac{c1c23ax + s1c23ay - s23az}{c4(c1s23ax + s1s23ay + c23az)}\right)$$

Behelyettesítve példánk számértékeit:

$$\Theta_4 = \operatorname{arctg} + \left(\frac{-\sin 45 * 0 + \cos 45 * (-0{,}5)}{\cos 45 * \cos(194{,}8) * 0 \sin 45 * \cos(194{,}8) * (-0{,}5) - \sin(194{,}8) * (-0{,}866)}\right)$$

$\Theta_4 = -71{,}2°$

$$\Theta_5 = \operatorname{arctg}\left(\frac{-\sin 45 * 0 + \cos 45 * (-0{,}5)}{\sin\ -71{,}2(\cos 45 * \sin 194{,}8 * 0 + \sin 45 * \sin 194{,}8 * (-0{,}5) + \cos 194{,}8 * (-0{,}866))}\right)$$

$\Theta_5 = 21{,}9°$

$$\Theta_6 = \operatorname{arctg}\left(\frac{0 + \sin 45 * 194{,}8 * 0{,}866 + \cos 194{,}8 * (-0{,}5)}{-\cos 45 * \sin 194{,}8 * (-1) - 0 - 0}\right)$$

$\Theta_6 = -61°$

Ezekután a megfelelő F frame koordináta-rendszer segítségével történő pozicionáláshoz a következő programrészlet használható:

```
F := FRAME(ROT(y,180) * ROT(x,30),VECTOR(24,18,10));
:
MOVE kar TO F;
```

Itt az F frame koordináta-rendszernek megfelelő pozíció és orientáció beállításához a vezérléssel be kell állítani a következő szögértékeket:

$\Theta_1 = 45°$
$\Theta_2 = 15{,}8°$
$\Theta_3 = 179°$
$\Theta_4 = 71{,}2°$
$\Theta_5 = 21{,}9°$
$\Theta_6 = 61°$

Ezek abszolút szögértékek, tehát a vezérlés az aktuális „A" robothelyzethez képest csak az inkrementumokat, azaz a szögértékek megváltozásait számítja ki

$\Delta\Theta_i = \Theta_i - \Theta_{Ai}$

formában

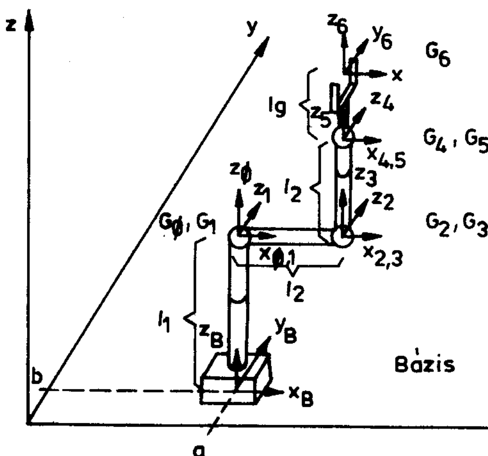
Az új Θ_i szögértékeknek megfelelő vezérlés végrehajtására különféle eljárások lehetségesek (l. még a 2.4.3. pontot). Magasabb szintű nyelvek esetében a programozó nemcsak a kezdeti és a végpont segítségével adhatja meg a robot általános pályáját, hanem közbülső pontokat is megadhat, sőt esetleg azt is előírhatja, hogy ezekben a közbülső pontokban milyen sebességgel mozogjon a robot. Ehhez az szükséges, hogy egy pályatervezési modulban a teljes mozgási folyamatot kielemezzük, a pályát megfelelő módon szakaszokra bontsuk, és a közbülső pontokról egy táblázatot állítsunk össze, amelyeket azután a robot egy megadott (ismert) időtartam alatt fog bejárni (l. idevonatkozóan PAUL [2.7]-et).

Végezetül tekintsük át a robotkoordinátáknak térbeli koordináta-rendszerbe (környezeti koordinátákba) való áttranszformálását! Erre pl. akkor van szükség, amikor a programozó a frame koordináta-rendszert nem annak explicit koordinátaértékeinek segítségével definiálja, hanem betanítási eljárással. Ekkor a robot beáll a kívánt pozícióba és orientációba, majd beolvassuk a különféle szögértékeket és meghatározzuk a frame koordináta-rendszer transzformációs mátrixát. A keresett koordinátatranszformációt a következőképpen nyerjük: a báziskoordináta-rendszer origóját először a robot bázispontjába helyezzük át *(2.32. ábra)*. Ezután minden egyes ízülethez hozzárendelünk egy ízületi koordináta-rendszert megadó frame-et, tehát pl. a G_1-től G_6-ig terjedő frame-eket. Figyelembe kell venni, hogy a két egymás után következő ízületi koordináta-rendszer között a következő fontos szabály teremt kapcsolatot:

1. A z_{i-1} tengely az i-edik csukló forgástengelyével esik egybe.
2. Az x_i tengely merőlegesen áll a z_{i-1} tengelyre és attól elfelé mutat.

Az $i-1$-edik ízületi koordináta-rendszerből G_{i-1}-ből a következő műveletsorozattal juthatunk az i-edik ízületi koordináta-rendszerbe, G_i-be:

1. Az z_{i-1} tengely körüli olyan Θ_i szöggel való elforgatás, hogy x_{i-1} párhuzamos legyen x_i-vel.
2. A z_{i-1} tengely menti d_i távolsággal végrehajtott eltolás.
3. Az x_i (tehát a már elforgatott x_{i-1}) tengely mentén a_i távolsággal eszközölt eltolás.
4. Az x_i tengely körül olyan α_i szöggel való elforgatás, hogy a z tengelyek ismét egybeessenek.



2.32. ábra. Az egyes csuklók G_i ízületi koordináta-rendszereinek, az ún. frame-eknek alaphelyzet szerinti értelmezése egy hattengelyű robot esetén

2.1. táblázat. A robotok frame-transzformációjának paraméterértékei

θ	α	d	a
θ_1	-90	0	0
θ_2	0	0	l_2
θ_3	90	0	0
θ_4	-90	l_2	0
θ_5	90	0	0
θ_6	0	l_2	0

A *2.32. ábrán* bemutatott robot alaphelyzetében – tehát amikor valamennyi θ_i elforgatási szög értéke nulla – az α_i, d_i és a_i értékeit a *2.1. táblázatban* tüntettük fel.

A G_{i-1} ízületi koordináta-rendszernek a G_i rendszerbe való transzformálását az előbbi négy műveletnek megfelelően a következő négy mátrixszal való szorzással végezhetjük el:

$$T_{i-1,i} = ROT(z_{i-1}, \theta_i) * TRANSL(O,O,d_i) * TRANSL(A_i,O,O) * ROT(x_i\alpha_i)$$

$$T_{i-1,i} = \begin{bmatrix} \cos\Theta_i & -\sin\Theta_i & 0 & 0 \\ \sin\Theta_i & \cos\Theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_i & -\sin\alpha_i & 0 \\ 0 & \sin\alpha_i & \cos\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{i-1,i} = \begin{bmatrix} \cos\Theta_i & -\sin\Theta_i\cos\alpha_i & \sin\Theta_i\sin\alpha_i & a_i\cos\Theta_i \\ \sin\Theta_i & \cos\Theta_i\cos\alpha_i & -\cos\Theta_i\sin\alpha_i & a_i\sin\Theta_i \\ 0 & \sin & \cos & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Kiindulva abból a G_0 koordináta-rendszerből, amely a báziskoordináta-rendszerhez viszonyítva csak egy – a robot helyzetének megfelelő – eltolást tartalmaz, azaz

$$G_0 = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & l_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}\text{-ból}$$

a $T_{i-1,i}$ transzformációs mátrixokkal sorozatosan jobbról végrehajtott mátrixszorzásokkal nyerjük a soron következő G_i ízületi koordináta-rendszert. (A transzformációt mindig a frame koordináta-rendszerhez viszonyítva végezzük el.) Legutolsónak G_6 megadja a megfogószerkezet pozícióját és orientációját, ami egyben azt az F frame-rendszert jelenti, ami ekkor a robotkart jellemzi.

Tehát

$$F = G_0 * T_{0,1} * T_{1,2} * T_{2,3} * T_{3,4} * T_{4,5} * T_{5,6}$$

Ha itt az α_i, d_i és a_i értékeit behelyettesítjük az egyes $T_{i-1,i}$ transzformációs mátrixokba, akkor kapjuk hogy

$$T_{0,1} = \begin{bmatrix} c1 & 0 & -s1 & 0 \\ s1 & 0 & c1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{1,2} = \begin{bmatrix} c2 & -s2 & 0 & l_2c2 \\ s2 & c2 & 0 & l_2s2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{2,3} = \begin{bmatrix} c3 & 0 & s3 & 0 \\ s3 & 0 & -c3 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{3,4} = \begin{bmatrix} c4 & 0 & -s4 & 0 \\ s4 & 0 & c4 & 0 \\ 0 & -1 & 0 & l_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{4,5} = \begin{bmatrix} c5 & 0 & s5 & 0 \\ s5 & 0 & -c5 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{5,6} = \begin{bmatrix} c6 & -s6 & 0 & 0 \\ s6 & c6 & 0 & 0 \\ 0 & 0 & 1 & l_g \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A szorzásokat elvégezve:

$$T_{0,1} * T_{1,2} = \begin{bmatrix} c1c2 & c1s2 & -s1 & c1c2l_2 \\ s1c2 & -s1s2 & c1 & s1c2l_2 \\ -s2 & -c2 & 0 & -s2l_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{0,1} * T_{1,2} * T_{2,3} = \begin{bmatrix} c1c2c3 - c1s2s3 & -s1 & c1c2s3 + c1s2c3 & c1c2l_2 \\ s1c2c3 - s1s2s3 & c1 & s1c2s3 + s1s2c3 & s1c2l_2 \\ -s2c3 - c2s3 & 0 & -s2s3 + c2c3 & -s2l_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{0,1} * T_{1,2} * T_{2,3} = \begin{bmatrix} c1c23 & -s1 & c1s23 & c1c2l_2 \\ s1c23 & c1 & s1s23 & s1c2l_2 \\ -s23 & 0 & c23 & -s2l_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{0,1} * T_{1,2} * T_{2,3} * T_{3,4} =$$

$$\begin{bmatrix} c1c23c4 - s1s4 & -c1s23 & -c1c23s4 - s1c4 & (c1s23 + c1c2)l_2 \\ s1c23c4 + c1s4 & -s1s23 & -s1c23s4 + c1c4 & (s1s23 + s1c2)l_2 \\ -s23c4 & -c23 & s23s4 & (c23 - s2)l_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{0,1} * T_{1,2} * T_{2,3} * T_{3,4} * T_{4,5} =$$

$$\begin{bmatrix} c1c23c4c5-s1s4c5-c1s23s5 & -c1c23s4-s1c4 & c1c23c4s5-s1s4s5+c1s23c5 & (c1s23+c1c2)l_2 \\ s1c23c4c5-c1s4c5-s1s23s5 & -s1c23s4+c1c4 & s1c23c4s5+c1s4s5+s1s23c5 & (s1s23+s1c2)l_2 \\ -s23c4c5-c23s5 & s23s4 & -s23c4s5+c23c5 & (c23-s2)l_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{0,1} * T_{1,2} * T_{2,3} * T_{3,4} * T_{4,5} * T_{5,6} =$$

$$\begin{bmatrix} c1c23c4c5c6-s1s4c5c6-c1s23s5c6-c1c23s4s6-s1c4s6 & -c1c23c4c5s6+s1s4c5s6+c1s23s5s6-c1c23s4c6-s1c4c6 & c1c23c4s5-s1s4s5+c1s23c5 & (c1c23c4s5-s1s4s5+c1s23c5)l_g+(c1s23+c1c2)l_2 \\ s1c23c4c5c6-c1s4c5c6-s1s23s5c6-s1c23s4s6+c1c4s6 & -s1c23c4c5s6+c1s4c5s6+s1s23s5s6-s1c23s4c6+c1c4c6 & s1c23c4s5+c1s4s5+s1s23c5 & (s1c23c4s5+c1s4s5+s1s23c5)l_g+(s1s23+s1c2)l_2 \\ -s23c4c6-c23s5c6+s23s4s6 & s23c4c5s6+c23s5s6+s23s4c6 & -s23c4s5+c23c5 & (-s23c4s5+c23c5)l_g(c23-s2)l_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Mivel ismerjük, hogy

$$G_0 = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A frame koordináta-rendszerre azt kapjuk, hogy

$$F = G_0 * (T_{0,1} * T_{1,2} * T_{2,3} * T_{3,4} * T_{4,5} * T_{5,6})$$

$$F = \begin{bmatrix} tx & ox & ax & px \\ ty & oy & ay & py \\ tz & oz & az & pz \\ o & 0 & 0 & 1 \end{bmatrix}$$

ahol az egyes mátrixelemek jelentése a következő:

$$
\begin{aligned}
tx &= (c1c23c4 - s1s4)c5c6 - c1s23s5s6 + (-c1c23s4 - s1c4)s6 \\
ty &= (s1c23c4 - c1s4)c5c6 - s1s23s5c6 + (-s1c23s4 + c1c4)s6 \\
tz &= (-s23c4c5 - c23s5)c6 + s23s4s6 \\
ox &= (-c1c23c4 + s1s4)c5s6 + c1s23s5s6 + (-c1c23s4 - s1c4)c6 \\
oy &= (-s1c23c4 + c1s4)c5s6 + s1s23s5s6 + (-s1c23s4 + c1c4)c6 \\
oz &= (s23c4c5 + c23s5)s6 + s23s4c6 \\
ax &= (c1c23c4 - s1s4)s5 + c1s23c5 \\
ay &= (s1c23c4 + c1s4)s5 + s1s23c5 \\
az &= -s23c4s5 + c23c5 \\
px &= ((c1c23c4 - s1s4)s5 + c1s23c5)l_g + c1(s23 + c2)l_2 + a \\
py &= ((s1c23c4 + c1s4)s5 + s1s23c5)l_g + s1(s23 + c2)l_2 + b \\
pz &= (-s23c4s5 + c23c5)l_g + (c23 - s2)l_2 + l_1
\end{aligned}
$$

A nagyobb kényelmet és több szolgáltatást nyújtó programrendszereknél mind a betanítási fázisban, mind pedig a program végrehajtása során szükség van a kétféle koordinátatranszformációra, valamint pályaszámításra is, ami a megnövekvő számításigény miatt már többprocesszoros rendszerek (esetleg külön aritmetikaprocesszoros rendszerek) vagy pedig többszámítógépes rendszerek alkalmazását igényli. Ellenkező esetben ugyanis a numerikus számítások a vezérlő számítógépet már annyira leterhelnék, hogy ez akadályozná a tulajdonképpeni vezérlést, ill. a programfutás ellenőrzést.

Ha az előbbi képletekben a frame koordináta-rendszer mátrixának elemeiben behelyettesítjük a példában megadott $\Theta_1 \ldots \Theta_6$ szögértékeket, valamint az l_1, l_2 és l_g hosszúságértékeket, akkor az F frame-mátrixot ismét derékszögű koordináta-rendszerbeli adatokkal kapjuk meg. A numerikus kiértékelés végrehajtásakor a bonyolult képletszámítások miatt nagy körültekintéssel kell eljárni.

2.4.3. Ipari robotok vezérlési módjai

A programozás menete az, hogy a programozó először egy frame-rendszer alakjában írja elő a megközelítendő pozíciót és orientációt. Ezután a compiler-program, vagy az interpreter ezt a frame-rendszert egy DH-mátrix formájában ábrázolja. Ebből a mátrixból számítja ki a számítógép a koordinátatranszformáció elvégzése után a csuklók szög-, ill. elmozdulási adatait. A vezérlés feladata ezek után az, hogy az egyes robottengelyeket az előírt Θ_i szögekre állítsa be. Ez csak a legegyszerűbb pontvezérléses (ún. PTP) (point-to-point) eljárás esetében valósítható meg anélkül, hogy újabb közbülső számértékeket kellene meghatározni az egyes tengelyek elforgatási szögeire, ill. a közbülső pontokban érvényes frame-rendszerekre. Általában azonban interpolációt kell végrehajtani, ami úgy történik, hogy a vezérlés gondoskodik a program szerinti kiinduló közbülső és végpontok között további interpolációs pontokban a frame-rendszereknek, valamint az egyes robottengelyek elforgatási szögértékeinek a meghatározásáról (l. még BINDER [2.8]). Ha az interpoláció segítségével a bejárandó út növekményének az értékeit (az ún. útinkrementumokat) határozzuk meg, akkor az ún. útraszteres interpolációról beszélünk. Az egyes útinkrementumok bejárásához szükséges időtartamok különbözőek lehetnek, ez függ a pálya geometriájától és a beprogramozott sebességtől. Ezzel szemben ún. időraszteres interpolációról beszélünk, amikor a mindenkori új tengelyadatok azonos időtartamonként kerülnek interpolálásra.

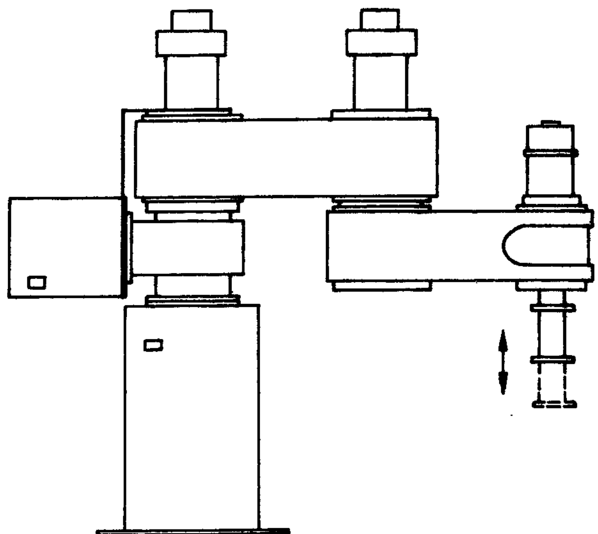
A vezérlőrendszernek meg kell adni a kiinduló és a végpontbeli frame-rendszert – robotkoordinátákban vagy derékszögű, ill. valós térbeli koordinátákban – sőt esetleg további közbülső pontokra vonatkozó frame-rendszereket is, valamint a sebességértékeket. Lényegében kétféle vezérlési eljárás ismeretes:

1. Pontvezérlés (**P**oint to **P**oint, azaz ún. PTP-vezérlés);
2. Pályavezérlés (**C**ontinuous **P**ath, CP, másként **C**ontrolled **P**ath).

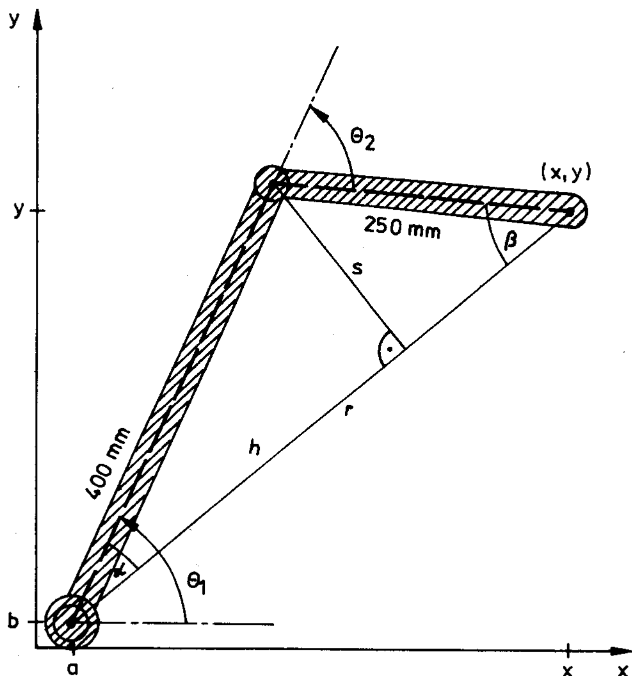
A PTP-vezérlésnél a robot összes mozgatandó tengelyét azonnal a maximális gyorsítással és a megengedett legnagyobb sebességgel működtetjük, míg el nem érjük az egyes tengelyekre előírt szöghelyzetet. A vezérlés az egyes tengelyek mozgatását egyidejűleg kezdi el, azonban mivel a különféle tengelyeknek különböző szögértékekkel kell elfordulniuk, így az egyes karok és csuklók különböző időpontokban állnak meg, vagyis hiányzik a robottengelyek közötti működtetési összhang, és így a megfogószerkezetnek, az ún. kéznek vagy TCP-nek a pályája a programozó szempontjából nézve nem eléggé meghatározott.

A *pályavezérlésnél* az egyes tengelyek működtetése összehangolt. A legegyszerűbb esetben a vezérlés a tengelyek működtetését olyan mértékben késlelteti, hogy azok mozgásukat egyszerre kezdjék és egyszerre fejezzék be. Ezt az eljárást a tengelyek együttes interpolációjának nevezzük. Emellett a derékszögű koordináta-rendszerbeli lineáris interpolációnak van még nagyobb jelentősége. Ekkor a vezérlés úgy határozza meg a robot egyes tengelyeinek mozgását, hogy a megfogási pont (ill. a TCP) pályája a kiinduló és végpont között egy egyenes mentén haladjon. Emellett létezik még a köríves és a parabolikus interpoláció, amikor a pontok között köríves, ill. hiperbolaszakaszok mentén fut a pálya, de ismert a komplex függvények szerinti pályavezérlés, amely munkadarabok felületének bejárására használható.

A következő példában bemutatjuk a három alapvető vezérlési módot, a PTP-bezérlést, a tengelyek együttes interpolációját és a lineáris interpolációt. Legyen adva a *2.33. ábra* szerinti kis négytengelyű robot, amely kis alkatrészek kezelésére és szerelésére alkalmas.



2.33. ábra. Egy japán gyártmányú két vízszintes pozicionáló tengellyel rendelkező kisméretű robot



2.34. ábra. A robot pozicionáló tengelyei

Csak a két pozicionálótengellyel a Θ_1 és Θ_2 szögek vezérlésével foglalkozunk. A *2.34. ábrából* a koordinátatranszformációra a következő adatokat nyerhetjük:

a) A robotkoordináták derékszögű koordinátákká transzformálása:

$$x = 400 * \cos\Theta + 250 * \cos(\Theta_1 + \Theta_2) + a$$
$$y = 400 * \sin\Theta_1 + 250 * \sin(\Theta_1 + \Theta_2) + b$$

b) A derékszögű koordinátáknak robotkoordinátákká való transzformálása:

$$r = \sqrt{(x-a)^2 + (y-b)^2}$$
$$h^2 + s^2 = 400^2$$
$$(r-h)^2 + s^2 = 250^2$$
$$h = \frac{48\,750}{r} + \frac{r}{2}$$
$$s = \sqrt{100\,000 - h^2}$$
$$\alpha = \mathrm{arctg}\left(\frac{s}{h}\right)$$
$$\beta = \mathrm{arctg}\left(\frac{s}{r-h}\right)$$

$$\Theta_1 = \mathrm{arctg}\left(\frac{y-b}{x-a}\right) + \alpha$$
$$\Theta_2 = -\alpha - \beta$$

A feladat az, hogy a robotot az

$x = 435, y = 510$, ill. $\Theta_1 = 60°, \Theta_2 = 25°$-os

pozícióból az

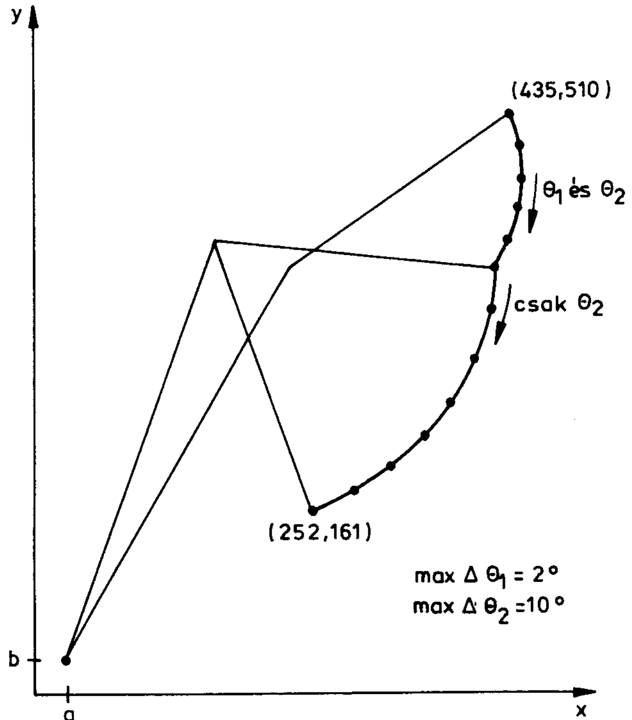
$x = 256, y = 161$, ill. $\Theta_1 = 70°, \Theta_2 = -140°$-os

pozícióba vezéreljük.

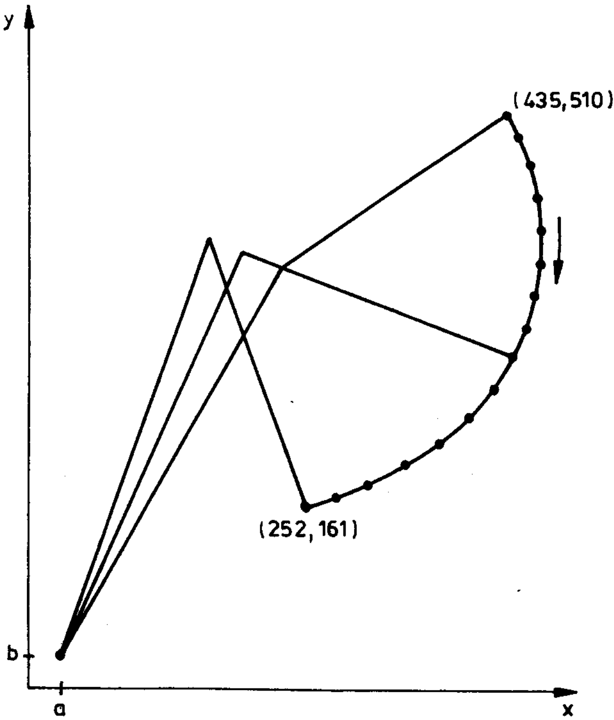
Pontvezérlésnél a Θ_1 tengelyt maximális sebességgel, azaz időegységenként 2°-os sebességgel mozgatjuk, a Θ_2 tengelyt pedig időegységenként 10°-os sebességgel. A *2.35. ábra* szemlélteti az így adódó pályát. A pálya kígyózó alakja úgy jön létre, hogy a Θ_1-es tengely a teljes működtetési idő mintegy harmadának elteltével már be is fejezi a mozdulatot. A *2.39. ábra* mutatja a tengelyek mozgatásának folyamatát mindhárom interpolációs eljárás esetében.

A tengelyek együttes interpolációjánál a Θ_1 tengely működtetését időegységenként 0,87°-ra lassítjuk le, és így a két tengely mozgása egyidejűleg fejeződik be *(2.39. ábra)*. Az így adódó pálya körív alakú *(2.36. ábra)*.

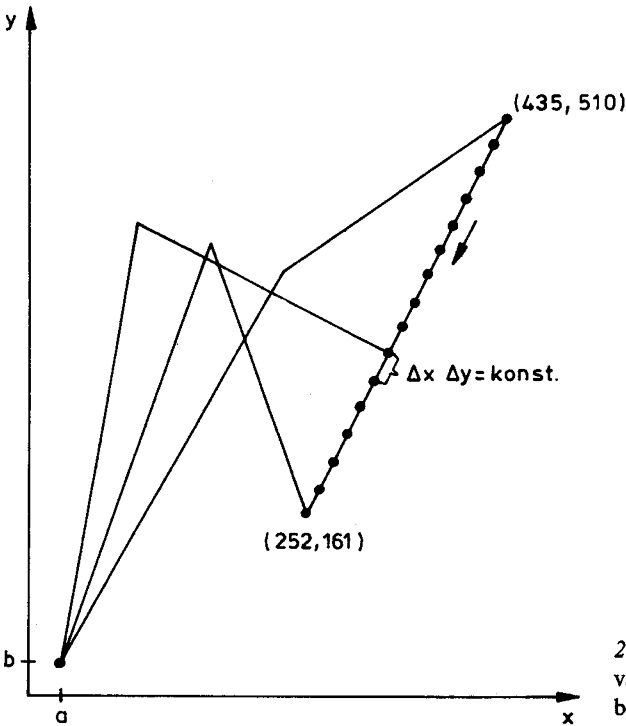
A derékszögű koordináták szerint lineáris interpolációnál a pálya közbülső pontjait nem robotkoordinátákban, hanem valós térbeli koordinátákban (derékszögű koordinátákban) számoljuk. Az így meghatározott közbülső pályapontokat ilyenkor vissza kell transzformálni robotkoordinátákra, és ezek szerint kell a vezérlést végrehajtani. Ez lényegesen nagyobb számítási igényű eljárás, és emiatt az interpolációt általában nem lehet ugyanolyan finom osztásközökkel megvalósítani, mint a tengelyek együttes interpolációjánál. Az eredményül adódó pálya ezért nem lesz egészen pontos és egyenes, ahogy azt a *2.37. ábra* idealizáltan mutatja, hanem egymáshoz csatlakozó kicsiny körívekből fog összetevődni. Aritmetikai processzorok alkalmazásával azonban az eltérés minimális



2.35. ábra. Pontvezérlés (az ún. PTP, azaz point-to-point vezérlés)

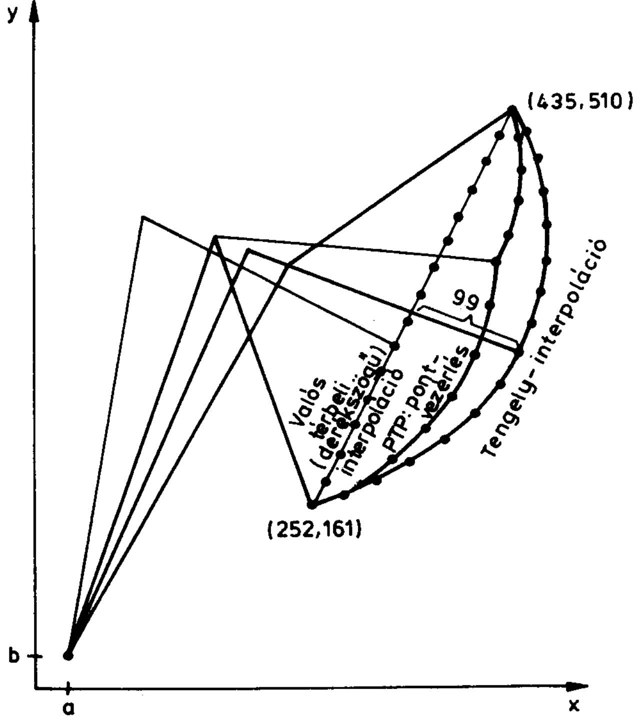


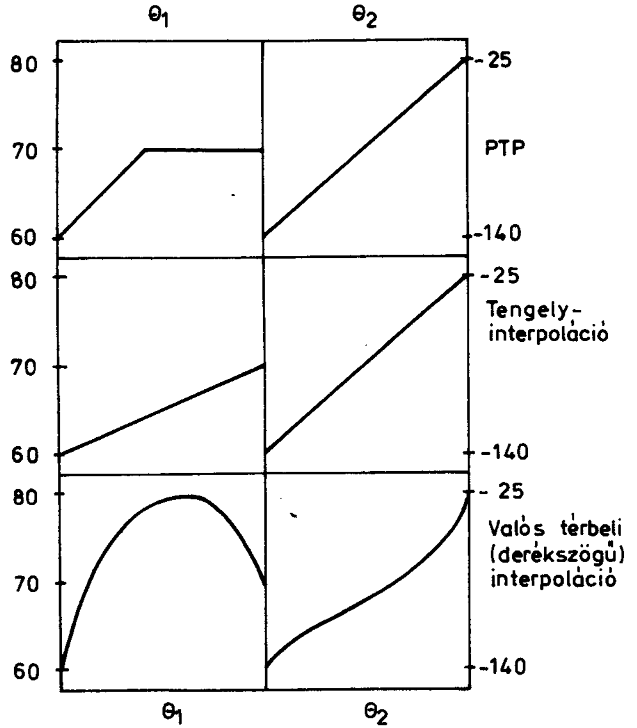
2.36. ábra. Lineáris tengelyinterpoláció



2.37. ábra. Lineáris interpoláció a valós térbeli koordináta-rendszerben pályavezérlésnél

2.38. ábra. A különféle vezérlési módok pályagörbéinek összehasonlítása





2.39. ábra. A robotok vezérlési módjai

mértékű. A Θ_1 és Θ_2 tengelyek szögelfordulását leíró időfüggvények nemlineáris jellegűek, sőt a Θ_1 először túl is lépi a 70°-os végértéket, majdnem 80°-ot ér el, majd ezután áll vissza 70°-ra.

A *2.38. ábrán* összehasonlíthatjuk a háromféle pályát. Kitűnik, hogy a PTP-vezérlésnél a programozó lényegében nem ismeri a pálya menetét, és ezért elég széles folyosót kell hagynia az összeütközések elkerülésére. Ugyanez fennáll a tengelyek együttes interpolációjánál is, amelynél ugyan kiszámítható a pálya, de ez az egyenestől akár 99 mm-rel is eltérhet, jóllehet a teljes úthossz csak kb. 400 mm.

2.4.4. Nagyszámítógépeken alkalmazott programozási nyelvek

Az ipari robotok programozási nyelveinek a kifejlesztéséhez – elsősorban persze a magasabb szintű nyelvek fejlesztéséhez – a nagyszámítógépeken használatos programozási nyelveket vették alapul. Ezzel mintegy automatikusan meghonosították az újabb robotvezérlő programozási nyelvekben is a már ismert programozási módszereket, a változók, deklarációk és programeljárások használatát, a blokkszervezést stb. Ugyanígy a vezérlésátadási utasításokat, valamint az aritmetikai és értékátadási utasításokat is az alapul vett ALGOL, FORTRAN, ill. a PL/1 nyelvből vették át, mivel ezekre a robotvezérlések programozásánál is szükség volt.

Nem kerültek be a robotvezérlő programozási nyelvekbe az olyan nyelvi szerkezetek, amelyekre a robotok vezérlésénél ez idő szerint nincs szükség – mint pl. a szövegkezelés, a karakter típusú változók, vagy a fájlkezelések –, mindezzel a nyelv leegyszerűsödött. Figyelembe kell azonban venni, hogy a programrendszerek implementálásával szemben jelenleg még korlátokat jelentő tárkapacitási és sebességi adatok már talán a közeli jövőben jelentősen kedvezőbbek lesznek, mivel a nagy kapacitású tárak és a többprocesszoros rendszerek árának lényeges csökkenése várható. Ezért elképzelhető, hogy a robotvezérlések programozásánál szerephez jut pl. a karakter típusú adatok feldolgozása is. Például olyankor, amikor a program futása alatt kerülnek kiértékelésre esetleges felhasználói üzenetek.

A nagyszámítógépes rendszerektől örökölt nyelveket azonban – az esetleges kisebb megszorítások mellett – mindenképpen ki kell egészíteni a robotok vezérlésére alkalmas parancsokkal. Többnyire új adattípusokat is be kell vezetni, így pl. a VECTOR, a ROTATION és a FRAME adattípusokat, amelyek a pályabeli pozíciók egyszerűbb leírását hivatottak elősegíteni. Ezenkívül további parancsokat kellett felvenni a robotvezérlő nyelvekbe a megfogószerkezetek és szerszámok nyitásának és zárásának vezérlésére, valamint a különféle feldolgozási folyamatok egymáshoz rendelésére (l. a 2.26. pontot és a 7.2. szakaszt is), a szinkronizálási kérdések megoldására, az érzékelők adatainak kezelésére, a jelek bevitelére és kiadására és a különféle robotfüggő hardver-paraméterek beállítására. A nyelv megtervezésekor, különösen pedig az aritmetikai műveletek kialakításakor a nyelv ortogonális struktúráját kell szem előtt tartani.

Ez közelebbről azt jelenti, hogy pl. egy VECTOR adattípus bevezetésekor az összeadás műveletének kiterjeszthetőnek kell lennie vektorokra is, és a programozónak kell ügyelnie arra, hogy az összeadásjel ezután szigorúan véve már két különböző műveletet jelent (egyrészt számok összeadását, másrészt vektorok összeadását). E kettő nem keverhető össze, mert pl. egy vektornak egy valós (real) típusú számmal való összeadása már típushibát okoz. Az ortogonális nyelvi szerkezet megköveteli azt is, hogy komplexebb adattípushoz tartozó változóknak való értékadáskor az egyszerűbb adattípusú változók felhasználhatók legyenek. Ez sajnos nem teljesül minden nyelvnél.

```
SCALAR vx, vy, vz;
VECTOR vektor;
vx←5.1;
vy←10;
vz←17.5;
vektor← VECTOR (vx, vy, vz);
```

2.7. programrészlet. A standard adattípusok alkalmazása az AL nyelv geometriai adattípusainál

A *2.7. programrészlet* egy ilyen példát mutat be. Egy a VECTOR geometriai adattípushoz tartozó változó értékét (l. a 3.1.4. pontot) a standard REAL típusú változók segítségével állítjuk be.

Egyes nyelvek még a komplexebb típusú változók komponensenkénti lekérdezését sem teszik lehetővé, jóllehet ez egészen természetes követelmény lenne, pl. a következő típusú utasítással:

IF vektor [1] > 5 THEN PRINT («Nagy a távolság»); itt vektor [1] a vektor x-komponensét jelenti.

Összefoglalva azt mondhatjuk, hogy a nagyszámítógépes rendszerekből átvett nyelvi megoldásoknál a már kialakult mechanizmusok és rendszerprogram-részletek megkönnyítik a robotvezérlési programrendszerek implementálását. Ugyanakkor a programozásnál az, hogy informatikai ismeretekre is szükség van, más oldalról hátrányt is jelent, mivel a robotvezérlések programozói elsősorban gyártástechnológiai irányú előképzettségűek.

Mint a *2.2. táblázatból* látható, az ismertebb programnyelvek többségéből már fejlesztettek robotvezérlő nyelveket is. Mind ez ideig nem létezik azonban olyan robotvezérlési programnyelv, amelynek kiindulópontja a PASCAL nyelv lenne, csupán PAUL [2.6] tett erre vonatkozó megfelelő kezdeményezést. Feltűnő emellett az is, hogy alig alkalmazzák a folyamatirányító programnyelveket, jóllehet ezek természetüknél fogva eleve tartalmazzák a különféle task-kezelési, folyamatperiféria-kezelési lehetőségeket, a megszakításkérések és a különleges üzemállapotok lekezelésére alkalmas eljárásokat (l. a 4.9. szakaszt).

2.4.5. Robotfüggő programozási nyelvek

Főleg az ipari robotok előállítói azok, akik olyan módon fejlesztik gyártmányaikhoz a programrendszereket, hogy elsősorban a vezérlés nyújtotta lehetőségekből kiindulva, egy viszonylag egyszerű nyelvet fejlesztenek tovább a vezérlési funkciók kiegészítésére. Az így kifejlesztett nyelvek konkrétan a robotvezérlés követelményeihez illeszkednek. Tehát pl. a működtetési utasításokat olyan kifinomultan alakították ki, hogy a programozó jól

2.2. táblázat. Az ismertebb magasabb szintű programnyelvek és az azokból származtatott robotvezérlő programnyelvek

Programnyelv	Robotvezérlő programnyelv
ALGOL (ALGOrithmic Language)	AL (Assembly Language)
PL/1 (Programming Language)	AUTOPASS (AUTOmated Parts ASsembly System)
FORTRAN (FORmula TRANslation)	névtelen (Standford Research Institute)
BASIC (Beginners All purpoSe Instruction Code)	MAL (Multipurpose Assembly Language)
RTL/2 (Real Time Language)	INDA (INDustrial Automation)

megválaszthassa a különféle vezérlési és interpolációs eljárásokat. Szintaktikailag ezek a nyelvek többnyire elég egyszerűek, hogy az interpretert (fordítóprogramot), aránylag csekély tárkapacitás igénybevételével magán a vezérlő számítógépen lehessen implementálni. A programban a vezérlésátadások, az alprogramok és aritmetikai segédeljárások használatának lehetőségei erősen korlátozottak, ami miatt összetettebb programoknál a strukturált programozás megvalósítása már nehézkes.

2.4.6. NC-gépek programozása

A numerikusan vezérelt szerszámgépek terén már az 50-es évektől alkalmaznak programrendszereket az NC-rendszerek (Numerical Control rövidítése) vezérlésére (l. idevonatkozóan WECK [2.9] művét). Az ekkor kialakult programozási nyelvek nemcsak hogy explicit, funkciómegadási utasításokat tartalmaznak – ilyeneket, mint pl. „előtolás", „lyukfúrás", „maróbekapcsolás" stb. –, hanem tartalmaznak egy leíró részt is, amelyben leírják a munkadarab geometriáját. E célból olyan geometriai alapelemeket használnak, mint a pont, egyenes kör, négyszög, amelyek segítségével le lehet írni a munkadarabok felületét és a munkadarabok helyzetét. Ezeket az adatokat a fordítóprogram értékeli ki a fordítás során, majd később úgy használják fel, mint a szerszámvezérlésekre vonatkozó funkcionális utasítások egyes paramétereit. Ez tehát ún. implicit programozás, mivel a pályavezérlési utasításokat nem explicit adják meg, hanem a geometriai adatokból vezetik le. Gyakran az összes adatot konstansként kell megadni a futtató rendszernek, így tehát nem történik a szó igazi értelmében vett változókezelés, az érzékelők adatainak feldolgozása pedig csak korlátozott mértékben lehetséges.

Különleges jelentőségre tett szert az APT (**A**utomatically **P**rogrammed **T**ools). Ez egy sor további nyelv kifejlesztéséhez szolgált alapul. Az NSZK-ban elsősorban az EXAPT (Extended **APT**) vált szélesebb körben ismertté, amely az APT-nek egy különleges technológiákhoz illesztett bővített változata. A program felosztható a programfejben specifikált adatokra, amelyek magát a szerszámgépet specifikálják, geometriai definíciókra, technológiai definíciókra, amelyeknek segítségével pl. egy menetvágást lehet specifikálni, valamint arra a részre, amely a végrehajtásra vonatkozó utasításokat tartalmazza.

Az Aacheni Műszaki Egyetem szerszámgép-laboratóriumában kifejlesztett ROBEX nyelv (**ROB**oter **EX**apt) tulajdonképpen az EXAPT nyelvnek robotvezérlésekre történő alkalmazása. A *2.8. programrészlet* pl. robotra is alkalmazható, ha a program szerszámgép-specifikációs részében megadjuk a ROBOT1 megnevezést, majd a robotot egy GOTO utasítással a P1 pontba vezéreljük. A későbbiekben még részletesebben bemutatjuk a ROBEX nyelvet, amely az NC-gépek programozásából kialakult nyelvi irányzat egyik jellegzetes tagja. Az NC-programozás lényegében véve egy olyan környezeti modellt alkalmaz, amely túlnyomórészt geometriai adatokat tartalmaz.

```
MACHIN/WM1                 A szerszámgép specifikációja
TRANS/100,200,0            A munkadarab helyzete a munkapadon
P1 = POINT/100,60,25       A P1 pont adatai
                           X = 100, Y = 60, Z = 25
FROM/10,-10,0              Az előírt művelet kiindulási pontja
GOTO/P1                    A szerszám beállítása P1-be
```

2.8. programrészlet. Példaprogram az NC-gépek vezérlésére alkalmas APT nyelvben

2.4.7. Munkatervek, közhasználatú nyelvek

Magasabb szintű nyelveken végrehajtott programozáshoz megfelelő előképzettség szükséges, ezért az ipari robotok beállításakor az addig ott dolgozó szakembereket általában nem tudják programozóként tovább foglalkoztatni. Emiatt olyan kezdeményezések születtek, hogy a robotvezérlési programozási nyelvet kell jobban hozzáigazítani az emberi munkaerő számára készült munkatervi utasítások nyelvezetéhez. Egy ilyen nyelv egyszerűbben megtanulható és a beérkező feladatok rövidebb idő alatt beprogramozhatók. Ehhez azonban lényegesen intelligensebb fejlesztőrendszerre és futtatásvezérlő rendszerre van szükség.

Egy olyan program, amely a manuális munkatervi utasítások nyelvezete szerint íródik, rengeteg információt kénytelen nélkülözni. Ezekre a munkapadnál dolgozó szakmunkásnak nincs is szüksége, hiszen ő rengeteg járulékos információt és tapasztalatot halmoz fel a munkapad és a szerszámok geometriai adottságairól, bizonyos logikai feltételekről, fizikai törvényszerűségekről és különféle segédeszközökről stb. Ha pl. egy olyan utasítást veszünk, hogy

⟨illeszd a fedlapot az alapkeretre⟩,

ez az ember számára tökéletesen elegendő ahhoz, hogy el tudja végezni az illesztést. Ugyanakkor egy programrendszernek előbb meg kellene oldania a következő feladatokat:

- ☐ Meg kell határoznia a fedlap megfogási pontjait.
- ☐ Tisztáznia kell, hogy a fedlap gyűrűkbe bepattintható kapcsokkal illeszkedik.
- ☐ Ezzel kapcsolatban meg kell határoznia az alkatrészek pontos helyzetét és irányát.

A következő explicit utasításokat kell generálnia:

- Állítsd rá a robotkart a fedlap megfogási pontjára.
- Fogd meg a fedlapot.
- Vezesd a robotkart a megfelelő összeillesztési helyzetbe.
- Lassan süllyeszd a robotkart, és a megfogóeszköz erőváltozása alapján, vagy valamilyen vizuális rendszer segítségével állapítsd meg, hogy valamennyi kapocs bepattant-e.
- Engedd el a megfogást.
- Távolítsd el a robotkart.

A programrendszer egy tervezőmoduljának az a feladata, hogy miután a kezelő megjelölte a végrehajtandó célt (a fedlapnak az alapkeretre illesztését), utasítástervet készítsen, amely az ismert kiinduló állapotból az előírt végállapothoz vezet. Ehhez igen intelligens környezeti modellre van szükség (l. a 2.4.8. pontot), amely az embernél meglevő előzetes ismereteket közvetíti.

Az IBM AUTOPASS (**AUTO**mated **P**ars **AS**sembly **S**ystem) elnevezésű rendszernek készítői ezt a célt tűzték maguk elé (részletesen l. LIEBERMAN [2.10]). Sajnos nem ismeretes, hogy hol tart a programrendszer implementálása és kipróbálása. A fordítóprogram tartalmaz egy problémamegoldó részt, amely ellenőrzi, hogy teljesülnek-e a felhasználói utasítások végrehajtásához szükséges előfeltételek, majd elvégzi a szükséges kiegészítéseket és előállítja a munkamozzanatra vonatkozó komplett utasítás megvalósításához szükséges vezérlési utasításokat. Ha ez mégsem sikerül, akkor a felhasználónak ki kell egészítenie az adatokat.

Az erre a célra kialakított nyelv olyan szerelési utasításokból áll, amely az alkatrészek elhelyezésére és pontos beillesztésére, valamint szerszámok és speciális segédeszközök használatára vonatkozó műveleteket tartalmaz. Ebben a nyelvben az előbbieken kívül deklarálni lehet még a manipulátor különféle jellemzőit, valamint az egyes tárgyak és berendezések egymáshoz viszonyított térbeli elhelyezkedését is. A *2.40. ábra* bemutatja egy fékberendezés szerelési utasítását és az ezt megvalósító AUTOPASS nyelven írt programrészletet.

Egy másik, az előbbinél még messzebb menő elképzelés azt tűzte ki célul, hogy a robotberendezés feladatát a természetes beszélt nyelvhez igen közelálló nyelven lehessen megadni. A „közelálló" szó úgy értendő, hogy mivel a beszélt nyelv szintaktikai és szemantikai elemzése igen komplex feladat lenne, ezért a természetes nyelvnek csak egy külön, erre a célra kiválasztott részét használhatnánk. A milánói egyetemen a mesterséges intelligencia kutatásával kapcsolatos kutatási tervben kifejlesztettek egy DONAU

9 1. ASM SUPPORT BRACKET
10 P/U AND POSITION THE NUT IN THE NEST OF THE FIXTURE
11 1090037 NUT, CAR RET TAB QTY 01
12 P/U, ORIENT AND POSITION THE BRACKET INTO THE FIXTURE WITH ITS TAB OVER THE NUT
13 1115191 BRKT ASM RAIL SUPPORT QTY 01
14 P/U SCREW AND LOAD DRIVER
15 1107379 STUD, CR TAB INTLK QTY 01
16 P/U, ORIENT AND POSITION THE INTERLOCK OVER THE BRACKET HOLE, WITH THE NOTCHED LUG UP
17 1117637 INTERLOCK, CR + TAB QTY 01
18 P/U AIR DRIVER
19 DRIVE SCREW TIGHT
20 TORQUE 12.0 IN/LBS
21 ASIDE AIR GUN

1. OPERATE *nutfeeder* WITH *car-ret-tab-nut* AT *fixture nest*
2. PLACE bracket IN *fixture* SUCH THAT *bracket.bottom* CONTACTS *car-ret-tab-nut.top*
   AND *bracket.hole* IS ALIGNED WITH *fixture.nest*
3. PLACE *interlock* ON *bracket* SUCH THAT *interlock.hole* IS ALIGNED WITH *bracket.hole*
   IS ALIGNED WITH *bracket.hole*
   AND *interlock-base* CONTACTS *bracket.top*
4. DRIVE IN *car-ret-intlk-stud* INTO *car-ret-tab-nut* AT *interlock.hole*
   SUC THAT TORQUE IS EQ 12.0 IN-LBS USING *air-driver* ATTACHING *bracket* AND *interlock*
5. NAME *bracket interlock car-ret-intlk-stud car-ret-tab-nut* ASSEMBLY *support-bracket*

AUTOPASS program for support bracket assembly.

2.40. ábra. Fékszerelés munkaterve és a hozzá tartozó programrészlet LIEBERMANN [2.10] nyomán

(Domain Oriented NAtural language Understanding system) elnevezésű rendszert (l. BERNORIO [2.11], MAROY [2.12]). Ez a rendszer a robotoknak (de csak szimulált robotoknak) a beszélt olasz nyelv egy kiemelt részhalmazának segítségével való programozását teszi lehetővé, és eredményként egy MICROPLANNER robotnyelven írt programot állít elő. A rendszer egy UNIVAC 1108 gépen fut a LISP operációs rendszer alatt, így jelenleg még nem megoldható ipari alkalmazása a túlságosan bonyolult és költséges hardver miatt.

2.4.8. A környezeti modell

A robotvezérlő programozási nyelvek fordítására alkalmas compilerek egy ún. környezeti modellt is generálnak, ill. használnak fel. Ez a környezeti modell azt a környezetet definiálja, ahol a robotberendezést működtetik (TAYLOR [2.13]). A modell tulajdonképpen egy adatbázis, amely adatokat tartalmaz:

- a munkadarabok helyzetéről;
- a munkadarabok egymáshoz viszonyított helyezetéről;
- a munkadarabok geometriai adottságairól;
- a megfogási pontokról és a megfogási irányokról;
- a munkadarabok fizikai tulajdonságairól, így tömegükről, felületi minőségükről, merevségükről stb.
- a szerszámok pozicionálásának eddigi módosításairól (a kezelési protokollnak megfelelően);
- a mindenkori robotpozícióról és orientációról;
- a hozzávezetések, szállítóberendezések, határolók és lerakóhelyek rögzített helyzetéről;
- a helyzetváltoztatások és a funkcióváltások időpontjáról;
- egy-egy kezelési művelet befejezéséről, pl. két munkadarab összeszerelésekor.

A környezeti modellben a mindenkor futó robotvezérlő programra érvényes és a csakis rájuk vonatkozó adatokon kívül szerepelnek még az általánosabb előismereteket közvetítő információk is:

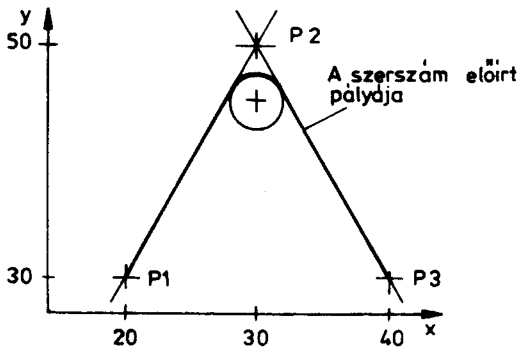
- a robotberendezés és a munkaterület térbeli geometriai adatai;
- a tiltott területek adatai;
- a logikai összefüggések és korlátozások adatai, így pl. ha nagyobb tömegű alkatrészek szerelésénél azok nem fordíthatók bizonyos pozíciókba;
- az egyes időbeli korlátozási feltételek, ill. események adatai.

A környezetleíró modelleket az NC-programok és az implicit robotvezérlő programok fordításakor használják. NC-berendezések programozásakor a programozó olyan geometriai alapelemek segítségével, mint pl. a pont, egyenes vagy kör egy pályát definiál, a compiler pedig (NC üzem esetén ezt *processzornak* nevezik) ebből meghatározza a vezérelt pálya egyes pontjainak adatait.

Példa (l. a *2.41. ábrát*):

Lépések:

- P1 pont definiálása $x = 20$, $y = 30$ koordinátákkal;
- P2 pont definiálása $x = 30$, $y = 50$ koordinátákkal;
- P3 pont definiálása $x = 40$, $y = 30$ koordinátákkal;
- Egyenes definiálása a P1, P2 pontok segítségével;



2.41. ábra. Geometriai adatok NC-programozás esetén

2.42. ábra. A fékberendezés szerelési technológiájához tartozó környezetleíró modell az AUTOPASS-ban

- Egyenes definiálása a P2, P3 pontok segítségével;
- A két egyenest érintő 5 sugarú kör definiálása;
- A marószerszámot működtető utasítás kiadása úgy, hogy az végighaladjon a két egyenesen és a körív mentén.

Az NC-programozás az implicit programozásnak egy – csupán a technológiai és a geometriai adatokra korlátozott – különleges esete. Általánosabb modellt tartalmaz a 2.4.7. pontban bemutatott AUTOPASS programrendszer. A *2.42. ábra* a fékberendezés szerelési folyamatához tartozó környezetleíró modell egy részletét mutatja be.

3. Adatok, adattípusok

A legelterjedtebb modern programozási nyelv a PASCAL, amely egyben kitűnő adatszervezési lehetőségekkel rendelkezik. Ezért most a könyvben tárgyalt robotvezérlő nyelvek adatleírási lehetőségeit a PASCAL-éval hasonlítjuk össze.

Ezt a látszólag egészen elvont témát azért tárgyaljuk ennyire részletesen mert egy programozási nyelv teljesítőképessége (használhatósága) ugyanúgy függ a nyelvben kezelhető **adatstruktúrákban** rejlő lehetőségektől, mint az utasításkészlet lehetőségeitől. Éppen a robotvezérlési alkalmazások esetében a szakemberek hajlamosak elfogadni, hogy az *utasításkészlet* játssza a fontosabb szerepet. Éppen emiatt könyvünk egyik fontos célkitűzése, hogy megmutassuk egy programozási nyelvnek a robotok intelligens és flexibilis vezérlésére való alkalmazásainál az adatszervezés egész komplexuma legalább olyan fontos, mint a jó utasításkészlet, tekintve az érzékelők által szolgáltatott információk nagy tömegét.

Az érthetőség kedvéért a témát számos programozási példa kapcsán fejtjük ki, majd összehasonlítjuk a különféle programnyelvek egymástól eltérő megoldásait és ábrázolási módjait. Ezt a célkitűzést sajnálatos módon nem mindig sikerült következetesen végigvinni, mivel gyakorlati kísérletek céljára csak az AL és a VAL állt rendelkezésünkre, a többi nyelv kézikönyvei pedig egyes kérdésekben nem elég részletesek, vagy pontatlanok voltak.

3.1. Az adatobjektumok

Adatobjektumokon egy meghatározott *adattípus* megnevezhető memóriaegységeit értjük. Az adattípus fogalmára még később visszatérünk. Az említett memóriaegység különböző nagyságú területet jelenthet (szót, bájtot stb.). Ha egy ilyen adatobjektumot a programozó valamilyen névvel lát el, akkor *változóról* beszélünk. Mint a 2.2.1. pontban bemutattuk, a változók egy vagy több memóriarekeszt foglalnak el, amelyekbe a program adatokat írhat, vagy onnan adatokat olvashat ki. A program a tárterületet annak címe segítségével éri el. A programozó számára azonban a címek kezelése igen körülményes és nehezen áttekinthető, ami a programot végső soron olvashatatlanná teszi. Ezért a fordítóprogram (a *compiler* vagy az *assembler*) lehetővé teszi, hogy a programozó szabadon választható *fantázianévvel* – mnemonikus névvel – szimbolikusan nevezhessen meg egy-egy tárterületet.

A programozási nyelvek fejlődésének kezdetén gazdasági okokból még korlátozták a választható változónevek karaktereinek számát. A FORTRAN-ban változónévként pl. hat karakterből álló jelsorozat használható.

A modern programozási nyelvek tetszőleges hosszúságú nevek, azonosítók használatát teszik lehetővé, azonban sokszor balról számítva csak bizonyos számú karaktert tekintenek értékelhetőnek. A többi karaktert figyelmen kívül hagyják, és így ezek nem használhatók az egyes azonosítók megkülönböztetésére, tehát a compiler, ill. az assemb-

ler sem különbözteti meg az olyan azonosítókat, amelyek csak az értékelt karaktereken túl különböznek egymástól. Ez persze kellemetlen programhibákra vezethet.

Az AL nyelv mintájául szolgáló ALGOL-ban az implementációtól függően az első 6–12 karakter értékelhető, a PASCAL-nál pedig legalább az első nyolc, de egyes fordítóprogramok többet is megengednek.

Egyes nyelvekben így az AL-ben és a legtöbb PASCAL-implementációban, mind nagy-, mind pedig kisbetűk használata megengedett, a nagy- vagy kisbetűkkel írt azonosítókat azonban nem különböztetik meg egymástól. Tehát abc, ABC vagy aBc ugyanazt a változót jelöli. Ez a szabály igen áttekinthető írásmódot tesz lehetővé. A változónevekben számjegyek használata is megengedett, a változónévnek azonban betűvel kell kezdődnie. A *3.1. táblázat* felsorolja az egyes robotvezérlési programnyelvekre érvényes megkötéseket.

3.1. táblázat. Azonosítóként használható karaktersorozatok megengedett, ill. értékelt hossza (A SIGLA nyelvben használt speciális nevek jelentésére a 3.1.3. pontban még visszatérünk)

Nyelv	Értékelt karakterek száma	Megengedett hossz
AL (karlsruhei implementáció, kisbetű megengedett)	30	tetszőleges
VAL csak nagybetűk	tetszőleges	tetszőleges
HELP csak nagybetűk	6	tetszőleges
SIGLA	csupán az I1...I16, P1...P16 és M1...M1023 azonosítók használata megengedett	
ROBEX csak nagybetűk	6	6

Az imént említett megkötések jelentőségének illusztrálására példaként bemutatunk különféle nyelvekben egy-egy matematikai képlettel felírt értékadási műveletet.

FORTRAN: RAKTKE = REGKES + BEERKZ − KISZAL
ALGOL: RAKTARKESZLET = REGIRAKTARKESZLET + BEERKEZES −
− KISZALLITAS
PASCAL: Raktarkeszlet = Regiraktarkeszlet + Beerkezes
− Kiszallitas
AL: Raktar_keszlet = Regi_Raktar_keszlet + Beerke_zes −
− Kiszal_litas
VAL: RAKTARKESZLET = REGIRAKTARKESZLET + BEERKEZES
RAKTARKESZLET = RAKTARKESZLET − KISZALLITAS
HELP: RAKTARKESZLET = REGIRAKTARKESZLET + BESZALLITAS −
KISZALLITAS
SIGLA: képletek, algebrai kifejezések kiértékelése közvetlenül nem lehetséges, hanem speciális számlálóregisztereket kell használni
ROBEX: A jelenleg kialakult verzió szerint képletek használata nem lehetséges

Az AL nyelvben a változónevekben használt „_" aláhúzásjel használata adja a legáttekinthetőbb megoldást. Mindegyik nyelvben lehet persze olyan lerövidített változóneveket alkalmazni, mint amilyet a FORTRAN-nál bemutattuk, ezzel azonban a változónevek használatában elveszítenénk az öndokumentálás lehetőségét. A későbbi programkarbantartás során, tehát a hibakeresések, programváltoztatások, vagy bővítések alkalmával az ilyen értelmes jelentésű változónevek jelentősen megkönnyítik a munkát, és ez indokolttá teszi a program kódolásakor és a fordítóprogram elkészítésekor felmerülő többletmunkát.

Az adattípusok korábban bevezetett fogalmát a következő példával világíthatnánk meg:

Tegyük fel, hogy egy 16 bites memóriarekeszben a következő bitkombináció van elhelyezve

0000000000110101

Felmerül a kérdés, hogy mit ábrázol ez a bitsorozat? Jelentheti ugyanis az „53"-as szám bináris ábrázolását, vagy az ASCII kódban ugyanezzel a bitsorozattal ábrázolható „5"-ös számot. Ennek a kérdésnek az egyértelmű megválaszolásához hiányzik egy fontos információ, ez pedig az adat típusának ismerete. A fenti első esetben pl. az egész számtípus (az ún. INTEGER), a második esetben a CHARACTER („jel") adattípus lenne a kérdéses típusinformáció.

Ezen a főleg technikai okon kívül egy legalább ugyanilyen fontos és a felhasználó szempontjából jelentős oka is van az adattípusok bevezetésének. A felhasználó ugyanis nem szeret túl sok energiát pazarolni ilyen adatfeldolgozás-technikai részletekre, és az adattípusok bevezetése ezt a kérdést tökéletesen megoldja. Az egyik olyan adattípus, amelyre valamennyi alkalmazásnál szükségünk van, az INTEGER típus. Ez az egész számok halmazának egy részhalmazát foglalja magában. Azért csak egy részhalmazát, mert az egész számok halmazának végtelen sok elemét egyetlen számítógép sem képes ábrázolni, hisz erre memóriája sem lenne elegendő. Ezzel az általánosan használt, problémafüggetlen adattípussal ellentétben létezik egy sor különféle speciális probléma kezelésére orientált adattípus, így pl. a robotvezérlési alkalmazásoknál használt VECTOR típus, amely a legegyszerűbb esetben három INTEGER típusú elemből áll, és ezek segítségével a háromdimenziós tér pontjai egy részhalmazának ábrázolására alkalmas.

Könnyen belátható, hogy az adattípus fogalma arra is alkalmas, hogy segítségükkel problémaorientált adatszerkezeteket adhassunk meg. A különféle adattípusokat a 3.1.3.-tól 3.1.6.-ig terjedő pontokban részletesen is ismertetjük.

3.1.1. A típusdeklaráció

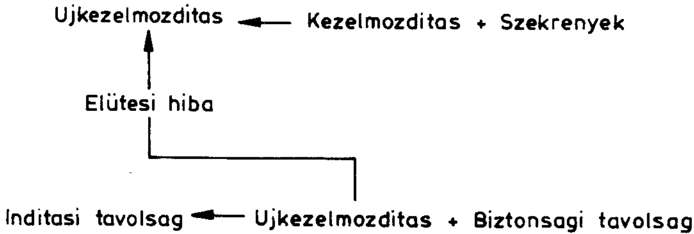
Egy változónevet egy típusdeklarációs utasítás segítségével rendelhetjük hozzá valamilyen adattípushoz. A típusdeklarációs utasításban a programozó változókat rendel hozzá egy meghatározott adattípushoz. Az AL nyelvben a típusdeklaráció a következőképpen írható:

VECTOR kezelmozditas, szekrenyele, biztonsagitavolsag;

Ezután a fordítóprogram már ismeri ezt a három változót a hozzá tartozó adattípussal együtt, meg tudja pl. vizsgálni, hogy a változóhoz tartozó adattípus egyáltalán lehetővé teszi-e valamilyen kívánt művelet végrehajtását. Vektorokat össze lehet pl. adni egymással, de vektort egész számmal már nem. A típusdeklaráció hatására a fordítóprogram a megadott adattípusnak megfelelően a szükséges tárterületet is lefoglalja. Az ilyen explicit formában megadott deklaráción kívül ismeretes a változók automatikus vagy

implicit típusdeklarációja is, amely bizonyos utasításokkal kapcsolatban jöhet létre. Például az ujkezelmozditas ← kezelmozditas + VECTOR (20, 5, 3); értékadó utasítás egyértelmű, mivel a jobb oldali kifejezés eredményeként egy vektor keletkezik, minek következtében a bal oldali újonnan bevezetett változónak (az „ujkezelmozditas"-nak) szintén VECTOR típusúnak kell lennie. A típusdeklarációnak ez a módja a programozó számára első pillantásra talán kényelmesnek tűnik, hiszen a program egyszerűbben írható, és valahányszor új változóra van szükségünk, az automatikusan definiálható.

Felmerülhet a kérdés, hogy mi célt szolgál akkor az explicit típusdeklaráció, amely sok jól bevált és elterjedt nagyszámítógépes programnyelvben az egyedüli deklarációs lehetőség. Erre ad választ a *3.1. ábrán* bemutatott példa.



3.1. ábra. Egy elütési hiba hatása automatikus típusdeklaráció esetén

Ha az ábra első sorában az ujkezelmozditas változónévnél a programozó elütési hibát vét, ez automatikus típusdeklarációnál a hiba helyén nem okoz észrevehető bajt, tehát nem kapunk hibaüzenetet, csak egy későbbi utasítássorban, ahol a biztonságitávolságvektorhoz egy ismeretlen, addig nem definiált változót akarunk hozzáadni. Ha most tekintetbe vesszük, hogy a két utasítássor között általában több utasítás is állhat, sőt egész terjedelmes programrészletek is elhelyezkedhetnek, akkor beláthatjuk, hogy ez a hiba milyen fáradságos hibakereséshez vezet. Még rosszabb a helyzet, ha az **„ujkezelmozditás"** változónevet egy korábbi közbülső utasítás már automatikusan deklarálja. Ekkor a fordítóprogram az elütési hibát már egyáltalán nem képes felismerni és így az már csak a program futása közben derülhet ki.

Ha azonban a változókat explicit kell deklarálni, akkor a fordítóprogram azonnal felismer minden elütési hibát (sokszor még akkor is, ha az elütéssel egy másik, már deklarált változónév áll elő), ezért a fenti hibajelenség fel sem léphet.

Az explicit deklaráció nagyobb programbiztonságot nyújt. Ezenkívül hozzájárul a robotfejlesztésben a rendszeresség betartásához is, mivel ez kényszeríti a programozót, hogy már a program leírása előtt tisztában legyen mindazokkal az adategységekkel, amelyeket programjában fel kíván használni. Az explicit deklarációk mellett szól az is, hogy a programszöveg áttekinthetőbbé válik, ami annak következménye, hogy valamennyi a programban, vagy valamely programrészletben használt adategyüttest deklarálni kell, és ez elsősorban a program fejrészében történik.

A változó deklarációknak az egyes nyelvekben megengedett formáiról a *3.2. táblázat* nyújt áttekintést.

A változók deklarálása mellett az egyes nyelvekben még különféle egyéb programozástechnikai objektumok is deklarálhatók. Erről a *3.3. táblázatban* közlünk áttekinthető összeállítást, és ezek jelentését a következőkben kissé részletesebben is megvilágítjuk.

3.2. táblázat. A változók típusdeklarációjának módjai

Deklaráció módja	PASCAL	AL	VAL	HELP	SIGLA	ROBEX
Explicit deklaráció a programfejrészben	igen	lehetséges	./.	lehetséges	./.	./.
Explicit deklaráció a változó használatát megelőzően	./.	igen	./.	igen	./.	./.
automatikus (implicit)	./.	igen	igen	igen	igen	igen

3.3. táblázat. Megengedett típusdeklarációk

	PASCAL	AL	VAL	HELP	SIGLA	ROBEX
Konstansok	expl/aut.	aut.	aut.	aut.	aut.	•
Mértékegységek deklarálása	./.	expl.	megelőző	megelőző	./.	••
Típusdeklaráció	expl.	./.	./.	./.	./.	./.
Változódeklaráció	expl.	•••	aut.	expl/aut.	aut.	•
Makródeklaráció	./.	expl.	./.	./.	./.	expl.
Címdeklaráció	expl.	••••	aut.	aut.	aut.	aut.
Alprogramok definíciója	expl.	expl.	aut.	aut.	aut.	./.
Eljárások definíciója	expl.	expl.	./.	./.	aut.	./.
Függvényeljárás deklarációja	expl.	expl.	./.	./.	./.	./.
Megszakításkezelő rutinok	./.	expl.	expl.	./.	./.	./.
Taskok	./.	./.	./.	•••••	./.	./.

Megjegyzések: • A ROBEX-ben az eddig ismert publikációk szerint különleges konstansok használata megengedett, ezt a következő fejezetben tárgyaljuk.

•• Csak metrikus és angolszász mértékegység között lehet választani.

••• Az amerikai verzióban csak explicit deklarációra van mód. A karlsruhei változat az automatikus deklarációt is megengedi, mivel ez tartalmazza a POINTY nevű interaktív nyelvi komponenst is.

•••• Az amerikai változatban a címkéket csak az ún. condition monitorok számára tartják fenn, és mivel a karlsruhei AL nyelvben ezt a nyelvi szerkezetet általánosították, címkék használatára nincs szükség (l. a 4.5.3. pontot).

••••• A HELP-ben kezelhető processek nagyon hasonlítanak a Task koncepcióra (l. a 7.2. szakaszt).

Konstansok deklarálása

A konstansok deklarálásának kétféle módja van. Automatikus deklaráció jön létre, ha egyszerűen hozzáírunk egy számot egy változónévhez, mint pl. az AL nyelvben a következő képlet írásakor:

forint←filler/100;

Explicit módon is deklarálhatunk konstansnevet.

PASCAL-ban pl. a következőképpen:

CONST divisor = 100;

Az előbbi példa PASCAL-ban ezekután a következőképpen írható:

forint←filler/divisor;

(Vegyük észre, hogy az AL-ban és a PASCAL-ban formailag különböző értékadási jeleket használunk!)

A konstansok explicit deklarálása által nyújtott lehetőségekre a későbbiekben még visszatérünk.

A mértékegységek deklarálása

A robotvezérlő programnyelvekben a különféle adattípusok a környezet leírására szolgálnak, ezért a változók mértékegységeit is meg kell adnunk. Ez vagy úgy történik, mint pl. a VAL vagy a ROBEX nyelvben, ahol ti. eleve rögzített, hogy valamennyi hosszadat milliméterben (ill. inch-ben), a szögadatok pedig fokokban (ill. degree-ben) értendők, vagy pedig lehetőség van új, összetett mértékegységek kifejezésére és deklarálására. Erre megvan a lehetőség pl. az AL-nyelvben, ahol a következő előre definiált dimenziókból és mértékegységekből kell kiindulni:

– távolság **(distance)**	**cm**-ben
– idő **(time)**	**sec**-ben
– erő **(force)**	**gm**-ben
– szög **(angle)**	**deg**-ben

Ezen alapszavak használatával azután a programban tetszés szerinti újabb mértékegységeket képezhetünk. A gyorsulás deklarálására pl. a

DIMENSION gyorsulás = distance/(time * time);

deklarációs utasítást szerkeszthetjük.

Új mértékegységeket állíthatunk elő makrodefiníciók segítségével is (l. a 2.2.4. pontot). Például:

DEFINE meter = # 100 * cm #

Makrodefiníció használatával definiálhatjuk az amerikai AL-változatban egyébként meglévő **inch** (hüvelyk), **oz** (uncia), **lbs** (font) és **radians** angolszász mértékegységeket is.

Mindezek mellett még a következő mértékegységek is értelmezve vannak:

– forgatónyomaték	**(torque)**	**gm/cm**-ben
– sebesség	**(velocity)**	**cm/sec**-ben
– szögsebesség	**(angular_ velocity)**	**1/sec**-ben
– dimenziónélküli szám	**(dimensionless)**	

Ha képletben dimenzióval rendelkező változók fordulnak elő, akkor az AL-fordítóprogram először ellenőrzi, hogy a dimenzionált kifejezés egyáltalán helyes-e, vagyis az értékadó utasításban a jobb és a bal oldal mértékegységei megegyeznek-e, és hogy a kapott dimenzió egyáltalán megengedett-e az aktuális utasításban.

Ez a már megismert frame koordináta-rendszerrel kapcsolatban (l. a 2.4.1. és a 3.1.4.), amelynek vektoros része lehet dimenzionált is, már problémát okozhat. Létezik ugyanis egy különleges frame-rendszer, az „@”, amely a robotkar pillanatnyi pozícióját adja meg. Hogyan értelmezhető e mennyiség dimenziója? Emiatt és még más egyéb itt nem részletezett problémák miatt a karlsruhei AL-változatban lehetővé tették a dimenzió nélküli és a hosszúság **(distance)** dimenziójú frame-rendszerek egy relációban történő használatát. Dimenzionált változóhoz úgy jutunk, hogy a deklarációs részben az adattí-

pus megadása előtt feltüntetjük a dimenziót jelölő alapszót. Az automatikus típusdeklarációnál az is elegendő, ha az újonnan bevezetett változónak dimenzióval rendelkező értéket adunk.

Példa:
distance VECTOR távolsag, elteres;
szog←30 * deg

Példánkban a **szög** változó automatikus típusdeklarációja **angle** dimenziójú SCALAR lett.

Az NSZK-ban eladott VAL rendszereknél és a HELP-ben valamennyi koordináta milliméterben, a szög pedig fokokban értendő, míg a ROBEX-ben választani lehet a metrikus mértékegységek (MM, ill. DEGREES) és az angolszász mértékegységek (INCHES, ill. DEGREES) között.

A SIGLA nyelv ezzel szemben nem kezel dimenzionált változókat és a vezérlés számára meghatározott számértékeket úgy tekinti, mint amelyek a robotkar helyzetének megváltoztatásához a léptetőmotornak adott számadatok (l. a 4.2.3. pontot).

Az adattípusok

Eddig három adattípussal ismerkedtünk meg, az INTEGER, a CHARACTER és a VECTOR típusokkal. A VECTOR típus három INTEGER típusú adatot fog össze egyetlen adatban. A legtöbb nyelvben – és ez alól sajnos a robotvezérlő nyelvek sem kivételek – bizonyos adattípusok a nyelv elemeiként előre definiáltak, és ezeken túlmenően a programozónak már nincs lehetősége arra, hogy újabb, problémájához szorosan

```
TYPE vektortípus = RECORD
                     x, y, z: INTEGER
                   END;
```

3.1. programrészlet. Vektortípus deklarálása a PASCAL nyelvben

```
TYPE szakasztípus = RECORD
                      startpont,
                      irany              : vektortípus
                      hossz              : INTEGER
                    END;
```

3.2. programrészlet. A „szakasztípus" nevű új adattípus deklarálása a PASCAL nyelvben

```
VAR  xmax    ,
     ymax    ,
     zmax    : INTEGER;
     xvect   ,
     yvect   ,
     zvect   : vektortípus;
     kockax  ,
     kockay  ,
     kockaz  : szakasztípus;
```

3.3. programrészlet. A PASCAL nyelvből vett példa deklarációk írására

illeszkedő adattípust definiáljon. Erre csak néhány nagyszámítógépes programnyelvben van lehetőség, így pl. a PASCAL-ban. A *3.1. programrészletben* bemutatott deklaráció segítségével egy **„vektortípus"** nevű adattípust vezettünk be, amely az x, y és z komponensekből áll. Ezt az adattípust egy későbbi deklarációban már ismét felhasználhatjuk, ahogy a *3.2. programrészlet* mutatja. A programszöveg tagolása – ebben a programrészletben, ill. a későbbiekben is – és az alapszavak nagybetűs írása csak a jobb olvashatóságot biztosítja, a PASCAL nyelv ezt nem írja elő.

Az előre definiált és a programozó által szabadon választott adattípusok segítségével azután különféle változók deklarálhatók (pl. a *3.3. programrészletbeli* példát). Az adattípusok deklarálásának módjait és lehetőségeit a 3.1.5. pontban még tovább részletezzük.

Változók deklarációja

A változók deklarálását az előzőekben már részletesen tárgyaltuk, ezért itt csak a teljesség kedvéért említjük.

Makrodefiníciók

Minden nyelvre általánosan érvényes, hogy a leírt programszöveget egy általános makrogenerátor segítségével lehet feldolgozni. Ez a makrogenerátor nem szükségképpen része a nyelvnek. Egyes nyelveknél (az AL-nál és a ROBEX-nél) már a fordítóprogramban megvan a makrogenerátor, mint azt a 2.2.4.3. pontban már említettük, a makrók hívásakor tulajdonképpen az adott esetben paraméterezett makrohívás helyébe egy előre definiált szöveget helyettesítünk be. (Ezt az előre definiált szöveget a makrodefiníció tartalmazza.)

Címkék, ugrási címek

Az utasítások általában megjelölhetők egy-egy címkével is (angolul label). Ez a legtöbb nyelvben úgy történik, hogy a kérdéses utasítás elé egy kettősponttal lezárt címkét írunk. Például ha a **cel2** címkével kívánunk megjelölni egy értékadó utasítást, ez a HELP nyelvben a következőképpen írható:

```
cel2:xmax := 2 * zmax;
```

Később a 4.7. szakaszban látni fogjuk, hogy ez az értékadó utasítás elérhető egy ún. *ugróutasítással* is. Egyes nyelveknél – így a Pascal-nál is – módszertani okokból megkövetelik a címkék explicit deklarálását, többnyire azonban elegendő az előbbi példában megismert automatikus deklaráció.

Szubrutinok vagy alprogramok

A 2.2.4.1. pontban tárgyalt szubrutinokat az egyszerűbb nyelvekben nem deklaráljuk explicit módon, hanem az eléjük írt címke segítségével megjelöljük, majd az egészet RETURN utasítással zárjuk le. Ez a címke azután akár szubrutinhívás segítségével, akár pedig egy feltételes vagy feltétel nélküli ugróutasítással is elérhető. A fordítóprogram számára kizárólag a szubrutin címkével megjelölt kezdetének van jelentősége, amelyet a program ugyanúgy kezel, mint bármely más címkével jelölt utasítást (l. még a 6.1. szakaszt).

Eljárások és függvényeljárások

A 2.2.4. pontban már szó volt róla, hogy az eljárások és a függvényeljárások olyan szubrutinok, amelyek lokális adatterülettel is dolgozhatnak, vagyis olyan adatokkal,

amelyek csakis az eljáráson (ill. a függvényeljáráson) belül hozzáférhetők. Egy további jellemzőjük az, hogy az eljárás (ill. a függvényeljárás) hívásakor lehetőség van paraméterátadásokra is. E kétféle eljárást mindig explicit módon kell deklarálni (l. a 6. fejezetet).

Megszakításkezelő rutinok

Az AL-ban és a VAL-ban lehetőség van különleges programrészleteknek megszakításkezelő programként való deklarálására. Ezek a megszakításkezelő rutinok egy-egy megadott érzékelő jelének meghatározott küszöbértéke elérésekor aktivizálódnak. A megszakítást kezelő alprogram vagy azonnal, vagy az éppen végrehajtott utasítás befejezését követően aktiválódik.

A 4.2.6., ill. a 4.5.3. pontokban erre még visszatérünk.

Taskok deklarációja

A 2.2.6. pontban bemutatott elkülönülő feldolgozási folyamatok, másképpen taskok a könyvünkben tárgyalt nyelvek egyikével sem definiálhatók. Egyedül a HELP tesz lehetővé a nyelv nyújtotta feldolgozási koncepció (unprocessz-koncepció) révén olyan programszervezést, amely sokban hasonlít a task-feldolgozáshoz. A karlsruhei egyetemen kidolgozott AL-változatban létezik néhány fontos utasítás, amely a task-feldolgozás vezérlésére szolgál, azonban csak lezárt programok kezelhetők taskokként. A kérdést a 7.2. szakaszban még részletesebben is tárgyaljuk.

3.1.2. Konstansnevek deklarálása

Minden nyelvben megvan a számkonstansok automatikus deklarálásának a lehetősége, amely egyszerűen a szám felírásával, tehát annak használatakor automatikusan létrejön. Emellett létezik az a lehetőség, hogy egy konstansnevet egy név (azonosító) explicit deklarálásával is definiálhatunk. Például a PASCAL-ban

```
CONST divisor = 100;
```

vagy

```
CONST targyakszama = 10.
```

Ha egy meghatározott feladatot a programban mondjuk 10-szer akarunk végrehajtani, pl. egy tálcáról tíz tárgyat kell leemelnünk, akkor célszerű olyan programciklust készíteni, amelyben a ciklusváltozó tízig növekszik (l. a 4.7.2.1. pontot). Ha programunkban többször is szerepelnek olyan ciklusok, melyek ciklusváltozója a tárgyak darabszámának eléréséig növekszik, akkor a 10-es konstans helyett mint ismétlési faktor helyett célszerű a *„targyakszama"* konstansnevet használni, amely előzőleg a deklaráció során a 10-es értéket kapja. Ha később újabb tálcákat kell kiszolgálnunk, melyeken azonban már pl. 12 tárgy van, akkor nem kell az egész programban a 10-es ismétlési faktort 12-re javítani, hanem elegendő a **targyakszama"** konstansnév deklarációjánál elvégezni a módosítást. A tárgyak száma természetesen változónévként is deklarálható, és ennek példánkban értékadó utasítással a 10-es értéket kellene adnunk.

A konstansnév deklarálása egyértelműen megmutatja, hogy a **„targyakszama"** esetében a program egy állandó értéket használ, és ezt a program futása közben már nem változtatjuk. Így egyértelműen megkülönböztethetők a programban az állandó és a változó adatok, ami növeli a programbiztonságot. A konstansnevek definiálásának egy további előnye az, hogy lehetőségünk nyílik olyan áttekinthető adatszerkezetek definiálására, amelyek adott esetben könnyen változtathatók. A kérdéssel a 3.1.5.1. pontban még bővebben foglalkozunk.

A konstansnevek deklarálása csak a PASCAL-ban megengedett, a robotvezérlő nyelvekben a szerzők tudomása szerint ilyen lehetőség nincs.

Mint már említettük, a ROBEX jelenlegi változatában nem használhatók olyan értelemben vett változók, mint a többi említett nyelvnél. A ROBEX-ben különféle geometriai alakzatok definiálhatók. Az alakzat lehet egy pont, vagy tetszőleges akár több geometriai objektumból (pl. felületekből) összetett test is (l. a 3.1.4. pontot).

Ezeknek az alakzatoknak a definiálása konstans értékekkel történik, esetleg a már korábban definiált konstansokkal. Ezekből a fordítóprogram egy rögzített értékekből álló numerikus kódot állít elő a robotot vezérlő interpreter számára, amely már nem tartalmaz további számítási utasításokat (l. a 2.4.6. pontot és a 8. fejezetet).

Ilyen értelemben mondjuk, hogy a ROBEX-ben csakis konstansnevek deklarálása megengedett, bár igaz, hogy ezek igen bonyolult konstansok, lehetnek pl. mátrixok is. Ezt az eljárást az NC-technológiában igen elterjedt nyelvekből, az APT-ből és az EXAPT-ból vették át. Alapvető hátránya, hogy a program futása során már semmilyen számérték, vagy adat sem változtatható – pl. egy érzékelő adatától függően –, és emiatt a vezérlés sem módosítható.

3.1.3. A standard adattípusok

A *standard adattípusok* közé soroljuk azokat az egyszerűbb adattípusokat, amelyek a legtöbb nyelvben megvannak. Ezek az

INTEGER:	a pozitív és negatív egész számok
REAL:	a törtszámok *lebegő tizedesvesszős ábrázolásban*
BOOLEAN:	a TRUE (igaz) és FALSE (hamis) logikai értékek
CHARACTER:	jelek, ún. karakterek. A karakterkészlet mindig gépfüggő (példa: ASCII karakterkészlet)
EVENT:	(eseménytípus) Ezek különleges számlálók *(szemaforváltozók)*, amelyek bizonyos programok szinkronizálására használhatók (l. a 4.7.3. fejezetet)

Eseményként értelmezhetünk bizonyos digitális bemeneteket vagy kimeneteket, még akkor is, ha ez az adattípus nem deklarálható, és műveletvégzés céljából sem hozzáférhető (l. a 4.7.3. pontot).

A lebegő tizedesvesszős ábrázolásnál – ugyanúgy mint a jobb zsebszámológépeknél – a számot mantisszára és kitevőre bontják fel, és a számot az ún. normált alakban ábrázolják.

Példák:

5,3 ábrázolása:	0,53000000 E+01
−100 ábrázolása:	−0,10000000 E+03
0,0513 ábrázolása:	0,51300000 E−01.

Valós (real típusú) számokkal történő számolásnál figyelembe kell venni, hogy a véges számú helyiérték és az ezzel járó pontatlanság miatt a megszokott algebrai összefüggések nem minden esetben használhatók a programban. Például az algebrában fennáll, hogy

$$\frac{1}{3}3 = 1$$

Ezzel szemben az **(1/3) * 3** kifejezésnek számítógépen történő kifejtésekor 1/3 × 3 csak 0,99999999 E00-nak adódnék. Ha tehát az 1/3 × 3-at egy logikai kifejezésben 1-gyel hasonlítjuk össze, akkor a FALSE (hamis) eredményt kapjuk. Az ilyen *kerekítési hibák* nemcsak logikai összehasonlításoknál hamisítják meg az eredményt, hanem a hosszadalmas, összetett számításoknál is igen tetemes hibát okozhatnak. Ezekkel a kérdésekkel a numerikus matematika foglalkozik. Mivel nem áll módunkban ennek részleteibe bocsátkozni, így csak az idevonatkozó szakirodalomra utalunk (l. [3.1], [3.2]). A robotvezérléseknél általában elegendő a rendelkezésre álló pontosság, a gyakorlati programozáshoz pedig elegendő figyelembe venni, hogy valós számok egyenlőségére vonatkozó logikai ellenőrzéseknél ésszerűbb az egyenlőnek feltételezett számok különbségét vizsgálni, hogy az kisebb-e egy előre felvett kicsiny toleranciaértéknél. Az előbbi példánkban szereplő logikai ellenőrzést tehát

(1/3) * 3 − 1 < 0.0001

alakban írhatjuk, és így az összehasonlítás már helyes eredményt fog adni.

16 bites, tehát két 8 bites bájtból álló szóhosszúságú számítógépet véve alapul, a standard adattípusok ábrázolható tartományait és az ábrázoláshoz igénybe vett szóhosszakat (a bájtok számát) a *3.4. táblázatban* felsorolt összeállítás mutatja. Ebből látható, hogy az adattípusok megválasztása nem csak a tárolt tartalom interpretálása szempontjából lényeges kérdés, hanem a változó tárolására igénybe vett tárterületet is meghatározza.

A *3.5. táblázat* áttekintést közöl a különféle nyelvekben meglévő standard adattípusokról.

3.4. táblázat. A standard adattípusok ábrázolási hosszai és ábrázolási tartományai

Adattípus	Ábrázolási hossz bájtokban	Ábrázolási tartomány
INTEGER	2	−32768 ... +32767
REAL	4	0.15224277 E−39 ... 0.17014111 E+38**
BOOLEAN	min 1*	FALSE, TRUE
CHARACTER	min 1*	⟨pl. az összes ASCII-karakter⟩
EVENT	2	−32768 ... +32767

Megjegyzések: * Ha a számítógépen az egyes bájtok külön is megcímezhetők, akkor elég egyetlen bájt is az adattípus ábrázolásához.
** Az ábrázolási tartomány itt nemcsak a 16 bites szóhossztól függ, hanem attól is, hogy a 16 bitet hogyan osztják fel a karakterisztika és a mantissza között, tehát az ábrázolási pontosságtól is.

3.5. táblázat. A különböző nyelvekben használható standard adattípusok és alapszavaik

Adattípus	PASCAL	AL	VAL	HELP	SIGLA	ROBEX
INTEGER	INTEGER	SCALAR	Integer	Scalar	Zähler	./.
REAL	REAL	SCALAR	./.	Scalar	./.	./.
BOOLEAN	BOOLEAN	SCALAR	./.	./.	./.	./.
CHARACTER	CHAR	./.	./.	./.	./.	./.
EVENT	./.	EVENT	./.	Semaphor	Flags	./.

Az alapszavak a nyelv részét képezik.

A HELP nyelv az INTEGER és a REAL típusokat egyetlen változótípusban, a SCALAR típusban foglalja össze, míg az AL nyelv az INTEGER, a REAL és a BOOLEAN típusokat ábrázolja a SCALAR változótípus segítségével. Ekkor az Integer értékek tört rész nélküli valós (real) típusú számok, a nullánál nagyobb számok pedig a logikai TRUE értékét jelentik.

A SIGLA nyelvben számlálóként használt egész típusú változókra előre adott változónevek egy külön csoportja van kijelölve. Ezek ún. számlálótípusú változók, amire a „Zähler" alapszót használják. Globális memóriaszámlálóra **M1**-től **M1023**-ig, alprogramok paraméterátadásaihoz **P1**-től **P16**-ig, indirekt címzéshez pedig **I1**-től **I16**-ig terjedő változónevek használhatók. Indirekt címzésen SIGLA-ban a következőt értjük: pl. **I1** megadásával az ennek megfelelő **P2** paraméter tartalmát kapjuk meg, ami egy memóriaszámláló sorszámát jelenti. Ennek tartalma kerül **I2** helyére. Így ha pl. az *a* programban az **M100**-as memóriaszámlálóba **705**-öt írunk, és az *a* programból az

EX/b, 100

utasítással aktiváljuk a *b* programot, akkor ez azt jelenti, hogy egyrészt az első, vagyis **P1** paraméter értéke **100**, másrészt, hogy nincs is több paraméter. Ezután ha a *b* program valamelyik utasításában **I1**-et használjuk, akkor ezen **P1** tartalmát értjük, vagyis a **100**-as számot, amit a program számlálótípusúnak értelmez, és az ilyen sorszámú memóriaszámláló tartalmát veszi, tehát **M100** tartalmát, azaz **705**-öt.

3.1.4. A geometriai adattípusok

A pályavezérlés megvalósításához a legtöbb robotvezérlő nyelv speciális adattípusokat is tud kezelni. Ezeknek az új adattípusoknak a segítségével a programozó geometriai összefüggéseket és frame koordináta-rendszereket írhat fel egyszerű eszközökkel. Kár, hogy csak a programnyelvek egy kisebb hányada engedi meg a geometriai adattípusoknak a standard adattípusokkal való együttes használatát, sőt többnyire még olyan operátorok sem állnak rendelkezésre, amelyek ezekkel az új adattípusokkal műveleteket tudnak végezni (l. a 3.2.1.2. pontot). Az explicit pályavezérlés támogatására bevezetett geometriai adattípusok a következők (l. még a 4.2. szakaszt is):

- VECTOR (háromdimenziós vektor)
- ROT (egy vektor körüli adott szöggel történő elforgatás)
- FRAME (az elforgatási és vektoradatok segítségével definiált frame-rendszer)
- TRANS (transzformáció, melynek definíciója azonos a frame-ével).

A vektorok segítségével megadható valamely pozíció, a rotáció segítségével az orientáció írható le, a frame segítségével pedig egy pályapontbeli állapot adható meg (l. még a 2.4.1. pontot is).

A strukturált adattípusok (3.1.5. pont) segítségével a geometriai adattípusok még a PASCAL-ban is explicit módon definiálhatók (l. a *3.4. programrészletet*).

A **rotmatrix** nevű rotációs mátrixnak három háromdimenziós vektorral való deklarálása csak a rotációnak a 3 × 3-as mátrix segítségével történő belső ábrázolásához szükséges, ami a programozó szempontjából érdektelen. A vektor és a rotációs mátrix természetesen tömbök segítségével is definiálható (l. a *3.5. programrészletet*).

Az AL nyelv az itt bemutatott nyelvek közül az egyetlen, amely valamennyi geometriai adattípust tudja kezelni, éspedig az explicit pályavezérlés céljából is *(3.2. ábra)*.

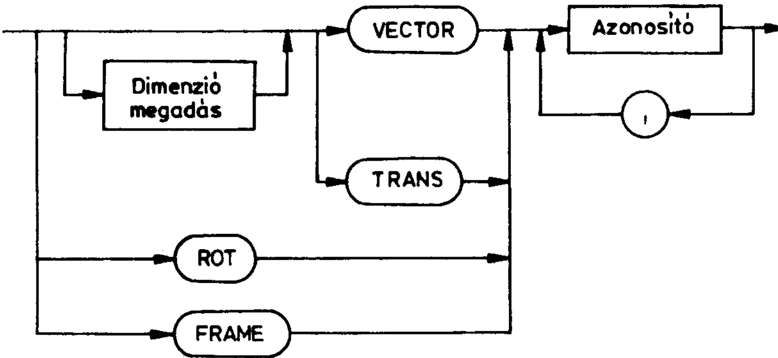
A mértékegységek megadásának módját a 3.1.1. fejezetben taglaltuk, a geometriai változók nevét a programozó szabadon választhatja meg.

```
TYPE
  vector        = RECORD
                      x, y, z: REAL
                  END;
  romatrix      = RECORD
                      t, o, a: vektor
                      END;
  rot           = RECORD
                      tengely : vektor;
                      szoeg : REAL;
                      matrix : rotmatrix
                  END;
frame           = RECORD
                      orient: rot;
                      pozic : vektor
                  END;
transz          = frame;
```

3.4. programrészlet. Geometriai adattípus deklarálása a PASCAL nyelvben

```
TYPE
  vektor        = ARRAY [1...3] OF REAL;
  rotmatrix     = ARRAY [1...3, 1...3] OF REAL;
```

3.5. programrészlet. Blokkok segítségével definiált vektor és rotációs mátrix



3.2. ábra. Geometriai adattípusok deklarálása AL nyelvben

AL:

DISTANCE VECTOR szállítószalag;
ROT alkatrészelfordulás;
FRAME megkoezelites, felvetel, lerakas:
szallitoszalag ← VECTOR (0,35,0) * CM;
A programban egy távolságvektort, egy rotációt és három frame-rendszert deklaráltunk. A távolságvektornak az x = 0 cm, y = 35 cm és z = 0 cm értéket adjuk.

A VAL nyelvben nincsenek vektorok és nincs rotáció; a frame egy SET utasítással kaphatja meg egy másik frame értékét. A frame értéke megadható ezenkívül a programfutáson kívül POINT-kommanddal is. A frame *transzformációját* közvetett, implicit módon lehet megadni. Ehhez először kijelölünk egy bázisframe-et, majd ezután ettől kettősponttal elválasztva kiírjuk a transzformációs frame-et.

Egy példán bemutatjuk ezt a megoldást, de az egyszerűség kedvéért a frame-nek csak a pozíciót jelölő vektorát tüntetjük fel.

VAL:

POINT bazis

300, 550, 250, 0, 0, 0

Itt a bázis nevű frame-et definiáltuk x, y, és z értékeivel, valamint az orientációt megadó *Euler*-féle szögek segítségével

POINT transzform

0, 0, 150, 0, 0, 0

Ezzel transzformációs frame-et definiáltunk

A későbbi programszövegben:

MOVE bázis: transzform

Hatására a robotkar beáll az X = 300 mm, Y = 550 mm és Z = 400 mm koordinátákkal jellemzett pontba.

Egymástól kettősponttal elválasztva több transzformációs frame is megadható.

A HELP nyelvben a vektor tömbként is deklarálható, azonban nem definiálható pályavezérlő utasítás pozícióvektoraként (4.2. szakasz).

A frame-mátrix elemei (a három koordinátaadat és a három szögadat) ugyancsak elhelyezhetők egy hatelemű tömbben. Ennek kezelésére azonban ugyanúgy nincs a nyelvben megfelelő utasítás.

A SIGLA nyelv nem kezel geometriai adattípusokat. A ROBEX-ben a vektorokat egy ún. POINT deklarációval definiáljuk, a rotáció mint adattípus teljesen hiányzik. A frame-eket és a transzformációkat mátrix-deklarációs utasítással lehet megadni, míg a tulajdonképpeni frame adatai egy pontot ábrázoló változóban tárolhatók. Ezt megelőzően egy TRASYS utasítás segítségével létre kell hozni magát a frame-et, ill. a transzformációt.

ROBEX:

frame = MATRIX/XYROT, −180, TRANSL, 300, 550, 250

TRASYS/frame

cel = POINT/0, 0, 150

GOTO/cel

Ez a programrészlet ugyanannak a frame-rendszernek a beállítását idézi elő, mint az előbbi VAL nyelvben felírt példa.

Az előbbi ún. *explicit pályavezérlést* lehetővé tevő adattípusok mellett a ROBEX-ben még további geometriai adattípusok is ismeretesek, melyek a pálya bejárásának *implicit* leírásához és tárgyak definiálásához szükségesek. Ezeket az adattípusokat nagyrészt az EXAPT-ból, ill. az APT nyelvből vették át. Ezek a következők:

- POINT (pont)
- LINE (egyenes)
- CIRCLE (kör)
- PATTERN (ponthalmaz)
- PLANE (sík felület)
- BEAM (téglatest)
- CYLNDER (henger)
- SPHERE (gömb)
- CONE (körkúp)
- BODY (előző elemekből összetett test)
- PART (testekből összeállított alkatrész)

Megemlítjük még a VECTOR típust, amely azonban nem használható fel explicit pályavezérléshez.

3.1.5. Strukturált adattípusok (Adatszerkezetek)

Többször utaltunk már rá, hogy célszerűbb, ha az összetett adattípusok – mint pl. a frame, vagy a rotáció – egyszerűbbekből is összetehetők és egyben az egyes komponensek közvetlenül is hozzáférhetők.

A kérdés a strukturált adattípusokkal, vagy más néven az adatszerkezetekkel kapcsolatos. A PASCAL-ban pl. már mód van ilyen strukturált adatszerkezetek felépítésére. A következőkben strukturált adattípusokkal kapcsolatban három fontos fogalmat, a *tömbök*, a *rekordok* és a *fájlok* fogalmát mutatjuk be.

3.1.5.1. Az adattömbök

Amikor azonos adattípushoz tartozó elemeket egyetlen adathalmazzá foglalunk össze, melyben az elemeket indexük segítségével tudjuk azonosítani, akkor az így létrejött adatobjektumot adattömbnek nevezzük. Egy **v** vektort pl. egy három komponenesből álló adattömb segítségével is ábrázolhatunk:

SCALAR ARRAY v [1 : 3];

Ezzel az AL nyelvben egy három skalárértékből álló v nevű tömböt (angolul array) deklaráltunk (vö.) a *3.5.* táblázattal). A skalár komponensek az 1, 2 és a 3-as indexekkel érhetők el. Ezzel a módszerrel a 2.4.1. pontban megismert *Denavit-Hartenberg* mátrix is egyszerűen deklarálható:

SCALAR ARRAY dhm [1 : 4, 1 : 4];

Az egyes komponensek eléréséhez egyszerűen felírjuk a

v[2]

vagy a

dhm[2,4]

tömbelemet, ami a **v** vektor y-komponensét, ill. a DH-mátrix eltolási vektorának y-komponensét jelenti. A kétdimenziós **dhm**-mátrixot a következőképpen képzelhetjük el:

dhm[1,1] dhm[1,2] dhm[1,3] dhm[1,4]
dhm[2,1] dhm[2,2] dhm[2,3] dhm[2,4]
dhm[3,1] dhm[3,2] dhm[3,3] dhm[3,4]
dhm[4,1] dhm[4,2] dhm[4,3] dhm[4,4]

A PASCAL-ban, az AL-ban és a HELP-ben lehet tömböket használni, a VAL-ban a SIGLA-ban és a ROBEX-ben nem. A PASCAL, az AL és a HELP tetszőleges számú index használatát megengedi (n-dimenziós mátrixok). A PASCAL és az AL nem korlátozza az indextartományt, az index lehet akár negatív szám is, de az alsó index csak kisebb vagy legfeljebb egyenlő lehet a felső indexszel. Az AL-ban pl. deklarálható a következő:

SCALAR ARRAY tömb1[−1 : 4, −10 : −5];

Ezzel szemben a

SCALAR ARRAY tömb2 [−1 : −3, −2 : 7];

deklaráció hibás, mert az első index −1-es alsó határa nagyobb, mint a megfelelő felső határ −3.

HELP-ben ezzel szemben az indexelés csak 1-gyel kezdődhet, ez azonban a robotvezérlések esetében nem jelent semmilyen lényeges megkötést.

Más a helyzet a tömbindexek megadási módjával kapcsolatban. A HELP-ben az index megadására csak numerikus konstans használható, a PASCAL-ban explicit definiált konstans is írható, az AL pedig a tömbdeklarációban az indexhatárok megadására változókat, sőt aritmetikai kifejezéseket is megenged. Ezért PASCAL-ban és AL-ban igen jól kezelhető és áttekinthető adatszerkezetek konstruálhatók. Tegyük fel pl., hogy egy program csőkötések szerelését vezérli, a csőkötések csavarjainak száma pedig tetszőleges lehet, de a program számára kívülről meg kell adni az egyes kötőelemek számát. Ekkor egy AL programban a kötőelemeket a következőképpen definiálhatjuk:

FRAME ARRAY kotoelemek [1: furatszam];

Ha a csavarok beszerelésére a programban egy ciklust létesítünk, amelyben a ciklusváltozó a felső indexig, a **furatszam**-ig növekszik (l. még 4.7.2.1.), akkor a különböző csőkötések szereléséhez a programban csak a **furatszam** nevű változót kell megváltoztatni. A program ezt a változót egy input utasítás segítségével terminálról is bekérheti, miáltal a program igen egyszerűen és kényelmesen használható.

Az egyes tömbelemek elérésére mindhárom nyelvben használhatunk konstansokat, változókat, sőt aritmetikai kifejezéseket is. Az index tehát kiszámítható, ami az ilyen adatszerkezeteket sokoldalúan kihasználhatóvá teszi.

A tömbök méreteit mindhárom nyelvben csak a mindenkori központi tár kapacitása korlátozza.

3.1.5.2. Rekordok (adategyüttesek, kapcsolt adatmezők)

Ha különböző típusú adatokat kell egy adatszerkezetben összefognunk, arra az adattömb nem alkalmas. Ekkor egy általánosabb szerkezethez, a **rekordok** (angolul record) használatához kell folyamodnunk.

A jelenleg használatos robotvezérlő nyelvek ezt a programelemet nem tudják kezelni – csak a PASCAL –, így e helyen csak arról számolhatunk be, ami a robotvezérlések terén is nagyon hasznos és érdekes lehetne. Egy rekordban előzőleg már deklarált, vagy pedig standard adattípushoz tartozó különböző típusú adatok foglalhatók össze. Egy rekordnak más rekordok és tömbök is részét képezhetik.

A 3.1.1. pontban bevezettük a három Integer típusú **x, y, z** komponensből álló **vektortyp** adattípust. A *3.6. programrészletben* az AL nyelv adattípusait a PASCAL nyelvben definiáljuk real típusú adatokból álló rekordok és tömbök segítségével.

```
TYPE
  vektortyp         = ARRAY [1...3] OF REAL
  rotaciostyp       = RECORD
                        tengely : vektortyp;
                        szog : REAL
                      END;
  transzformaciostyp = RECORD
                        orient : rotaciostyp;
                        pozicio : vektortyp
                       END;
         frametyp = transzformaciostyp;
```

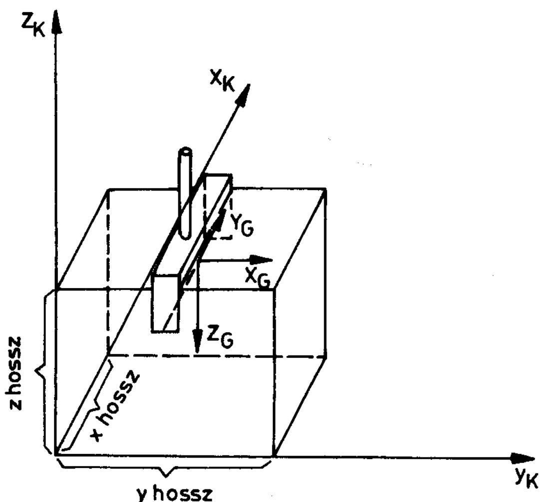
3.6. programrészlet. Az AL adattípusainak deklarálása PASCAL-ban

```
TYPE
  formaszekrenytipus = RECORD
                         koordrendsz         : frametyp
                         xhossz              ,
                         yhossz              ,
                         zhossz              : REAL;
                         megfogasipozicio
                                             : frametyp;
                         tele                : BOOLEAN
                       END;
```

3.7. programrészlet. Egy formaszekrénytípus deklarálása a 3.6. programrészletben felírt adattípusok segítségével PASCAL-ban

Ennek segítségével a *3.7. programrészlet* szerinti téglatest alakú tárgy definiálható. A *3.3. ábra* szemlélteti a frame-ek koordináta-rendszerei és a rekord változói közötti összefüggést.

Ha egy ennek megfelelően definiált formaszekrény pl. felülről nyitott, akkor a szekrény valamilyen anyaggal meg is tölthető. Eszerint a rekord egyik elemének a **„tele"** változónak az értéke lehet TRUE vagy FALSE. A **„megfogasipozicio"** szóval egy olyan frame-

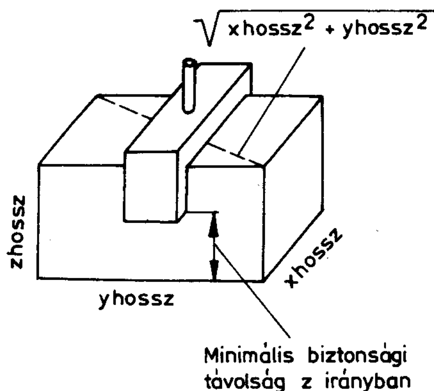


3.3. ábra. A formaszekrény a megfogószerkezettel együtt ábrázolva megfogási pozícióban. A formaszekrény frame-rendszerének „koordrendsz" nevű koordináta-rendszerét K index-szel, a megfogási pozíció rendszerét G-vel jelöltük

rendszert jelölünk, amely meghatározza a robotkéz számára a megfogási pozíciót. A hosszadatok az akadályok közötti manőverezéshez a távolságok becslését könnyítik meg. A pályától jobbra és balra a formaszekrény tetszőleges irányú forgatásához legalább

SQRT (SQR (xhossz) + SQR(yhossz))/2;

méretű helynek kell maradni, lefelé pedig a megfogószerkezet egy részének levonásával **zhossz**-nyi távolságnak (l. a *3.4. ábrát;* SQRT és SQR a négyzetgyökvonás, ill. a négyzetreemelés ismert standardfüggvényei).



3.4. ábra. Minimális biztonsági távolságok egy hasáb alakú tárgy mozgatásakor

A rekord egyes komponenseit a változó nevének, majd ettől ponttal elválasztva a komponens nevének megadásával lehet elérni. A **formaszekrény** változó deklarálása a következő:

VAR formaszekrény: formaszekrénytípus;

Ezután már a **formaszekrény** tele változónév a formaszekrény töltöttségi állapotát jelenti, a

formaszekreny · koordrendsz · pozicio

pedig a **formaszekrény** koordináta-rendszerének origóját, vagyis az elülső bal alsó sarkot. A

formaszekreny · koordrendsz · pozicio[3]

a **formaszekrénynek** a báziskoordináta-rendszer feletti magasságát adja meg.

3.1.5.3. Fájlok

Az eddig megismert adatszervezési eszközök (a tömb és a rekord) problémaorientált nyelvi elemek, ezzel szemben a fájl (ang.: file) az adatfeldolgozás területéről származik. A fájlok nagy adatmennyiségek tárolására valók, melyek a tömbökhöz hasonlóan azonos elemekből épülnek fel. A fájlokat általában nem az *operatív tárban* tárolják, hanem *háttértárakon,* így *mágneslemezen, mágnesszalagon,* esetleg *floppy-diszken.* A fájl szerkezete az egyes fájl-elemek szekvenciális elrendezésének felel meg, és az egydimenziós tömböktől annyiban különbözik, hogy a fájloknál a fájl hosszát a deklaráció során nem kell megadni.

Megkülönböztetünk kulcsos (indexelt) és szekvenciális (soros) hozzáférésű fájlokat. A kulcsos hozzáférés a tömbelemek elérési módjához hasonlítható. A fájl elemei többnyire 1-től kezdődően számozottak a tömbindexekéhez hasonlóan. A kulcsos elérés azt is

jelenti, hogy az olvasáshoz vagy íráshoz igénybe vett idő nem, vagy csak nagyon kis mértékben függ a fájl-indextől. A kulcsos elérésnek csak mágneslemezes vagy floppydiszkes tárolás esetén van értelme. Szalagon való tárolásnál először előre vagy hátra kell tekercselni, így a pozicionálás adott esetben már nagyon sok időt vehet igénybe (akár másodperces nagyságrendben). Lemez esetén 20...50 ms-os pozicionálási idő átlagosnak tekinthető.

A szokásos szekvenciális hozzáférés a fájl megnyitása után a legelső fájl-elem olvasásával/írásával kezdődik, majd ezt követően az egyik elem a másik után olvasható vagy írható. Az egyszer már kezelt elemhez csak úgy férhetünk hozzá még egyszer, ha visszaállunk a fájl elejére és a keresett elemig sorban kiolvassuk az összes elemet.

A tárgyalt nyelvek közül a fájlok deklarálásával kapcsolatban a PASCAL nyújtja a legtágabb lehetőségeket. Itt az elemek tetszőleges típusúak lehetnek. Tekintsük a következő két deklarációt:

```
TYPE text = FILE OF CHAR;
Szekrenyek = FILE OF formaszekrenytipus;
```

Az első deklaráció segítségével *szövegfájlt* határoztunk meg, a második esetben pedig hasáb alakú tárgyak leírására olyan fájlt deklaráltunk, amely a 3.1.5.2. pontban bemutatott rekordok tárolására alkalmas.

A szövegfájlban elhelyezhetők pl. programszövegek is, a *szekrenyek* nevű fájlban pedig különféle problémafüggő adatokat tárolhatunk.

A tárgyalt nyelvek közül csak AL-ban és HELP-ben lehet fájlt kezelni. Az AL-rendszerben fájl létesíthető frame-ek tárolására is. Ezekben a programozó által meghatározott frame-azonosítók alatt mindazok a frame-ek tárolhatók, amelyeket az ún. betanítási üzemmódban a felvett pozíció és orientációadatok közül le kell tárolni (l. az 5. fejezetet).

PASCAL-ban a frame-rendszerek tárolására alkalmas fájl az előbbiekhez hasonlóan a *3.8. programrészletben* bemutatott módon deklarálható, itt a deklaráláshoz felhasználtuk a *3.6. programrészlet* deklarációit.

```
TYPE
  framefileelem      = RECORD
                          nev : ARRAY [1...30] OF CHAR;
                          frame : frametyp
                       END;
framefiletyp         = FILE OF framefileelem;
VAR
                 framefile: framefiletyp;
```

3.8. programrészlet. Egy frame-fájl deklarálása PASCAL-ban a 3.6. programrészletben előforduló adattípusok felhasználásával

Az AL nyelv karlsruhei változatában ez a fájltípus előre definiált formában van jelen. A fájlkezelő rendszer szükséges funkciói a következők:

- Egy vagy több változó írása a fájlba. Ehhez adott esetben a fájl megnyitása, vagy új fájl létesítése is hozzátartozik.
- A teljes fájl olvasása. Ennek következtében a fájlban szereplő és a programban azonos névvel előforduló frame-rendszerek azzal az értékkel inicializálódnak, amellyel a fájlban szerepelnek;
- A fájlok lezárása.

A HELP csak szövegfájlok használatát ismeri. Ez arra alkalmas, hogy az öntanuló program futása során HELP utasításokkal fájlt lehessen létrehehozni. Ugyanis a HELP betanítási eljárás részében, az ún. öntanuló programban nyert adatokat a program szövegszerűen, értékadási utasítások formájában írja a fájlba, majd ezután a compiler a tulajdonképpeni program fordítása során azzal együtt lefordítja. Ehhez az eljáráshoz HELP-ben a következő fájlkezelő utasítások állnak rendelkezésre:

- CREATE : a fájl létrehozására;
- OPEN : egy már létező fájl megnyitására;
- RECORD : egy fájl leírására;
- CLOSE : a fájl lezárására.

A HELP-ben természetesen nemcsak ilyen HELP-utasításokból álló szövegből, hanem tetszőleges szövegből álló fájlok is létrehozhatók. További fájl-típusok kezelését nem tervezik a HELP-ben.

3.1.6. A pointer-típus és a környezetleíró modell

Mielőtt a környezetleíró modell és a pointer-típusú változók közti összefüggésre rátérnénk, ez utóbbi változótípus értelmezését közelebbről is megvilágítjuk.

Az eddig bemutatott adattípusokkal olyan változók deklarálhatók, amelyek – mint a 2.2.3. pontban láttuk – egy blokkhoz statikusan hozzárendeltek, azaz a blokkba való belépéskor a változók számára fenntartott tárterületet teljesen lefoglaljuk, majd a blokkból való kilépéskor ismét szabaddá tesszük. Elképzelhető azonban az a megoldás is, hogy a változókat a statikus blokkszervezéstől függetlenül tetszőleges időpontban hozzuk létre úgy, hogy később ezek a változók bármikor megszüntethetők legyenek. Az ilyen változókat *dinamikus változóknak* nevezzük. Mivel a program írásának időpontjában nem ismert, hogy hány változó lesz dinamikus, ezért ezek a változók statikus deklarációval nem jelölhetők ki.

A dinamikus változókat egy olyan eljárás hozza létre (PASCAL-ban a NEW nevű eljárás), amely a változóra mutató pointert ad eredményül. Ez a pointer nem más, mint az új változó számára fenntartott tárterület címe, amely egy pointer típusú változóban tárolható le. A *3.9. programrészlet* (PASCAL példaprogram) bemutatja, milyen új lehetőségek nyílnak meg pointer használatával.

A pointer típusú változókat PASCAL-ban a deklaráció során a változónév elé írt ∧ jellel jelöljük meg. A pointer által mutatott objektumhoz úgy férhetünk hozzá, hogy megadjuk a pointer nevét, ami után kitesszük a ∧ jelet. Mivel példánkban rekordokról van szó, ezért a pointer név után még egy pont következik, majd a rekord komponensének a neve áll. Így tehát a

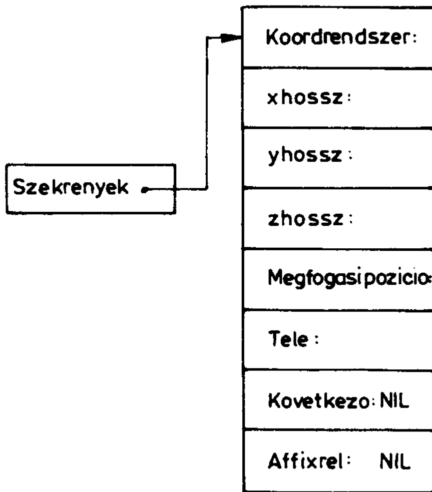
szekrenyek^ .kovetkezo

annak a rekordnak a **„kovetkezo"** nevű komponense, amelyre a **„szekrenyek"** nevű változó éppen mutat.

Programunkban két problémára is láthatunk egy-egy példát. Az első példában formaszekreny-rekordokból egy lineáris listát akarunk felépíteni, éspedig annyi rekordból álló listát, ahányat a felhasználó egy adatbevitellel meghatároz. Ez az érték lesz a **„formaszekrenyekszama"** nevű változó értéke. Ehhez csak a már ismert adattípusokat és a **formaszekrenytyp** változótípust kell használni. Mutasson a lista elejére a **szekrenyek** nevű pointer, a formaszekreny rekordok számát pedig a program a futás közben fogja bekérni a kezelőtől.

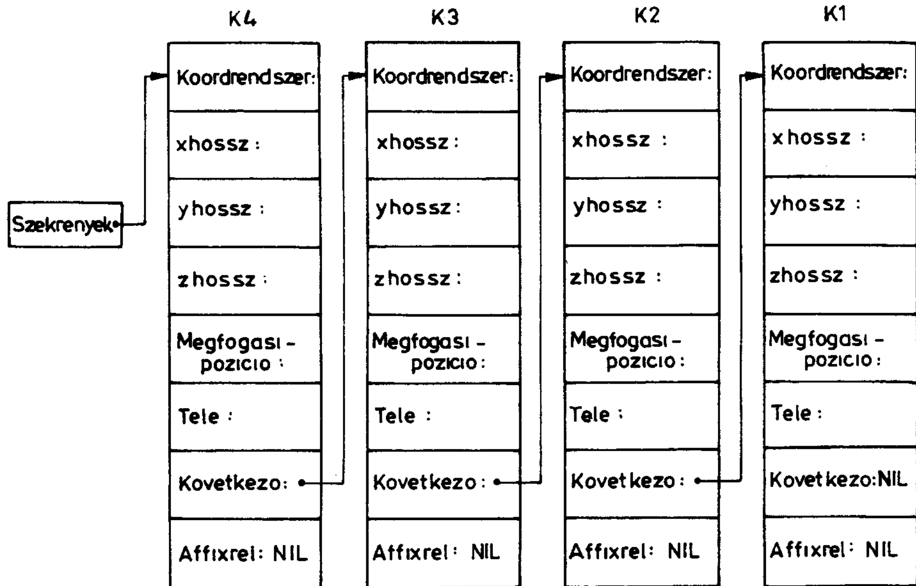
A program először beolvassa a szekrények számát, majd előállít egy rekordot a *3.5. ábrának* megfelelően. Az áttekinthetőség kedvéért csak az itt szóba jövő pointer típusú változóknak adunk értéket: a **szekrenyek^ .affixrel, szekrenyek^ .kovetkezo,** ill. a **k^ .affixrel** és a **k^ .kovetkezo** változóknak. A többi változót figyelmen kívül hagyjuk.

Ezután egy ciklusban annyiszor futunk végig, amennyi a **szekrenyekszama** tartalma, és ebben a ciklusban előállítunk ugyanennyi rekordot, ezeket a lista elejére fűzzük, aminek következtében a *3.6. ábrának* megfelelő szerkezetű lista keletkezik. Az ábrán a keletkezési sorrendet K1–K4 jelöli.



3.5. ábra. Egyetlen "formaszekrenytyp" típusú elemből álló szekrények listája. A koordrendsz, xhossz, yhossz, zhossz, a tele és a megfogasipozíció nevű rekordkomponensek nem kaptak értéket

3.6. ábra. „Formaszekrénytyp" típusú négy elemből álló egyszeresen pointerezett lista. A koordrendszer, xhossz, yhossz, zhossz, a tele és megfogasipozicio nevű rekordkomponenseknek nem adtunk értéket



Pointer típusú változók használata nélkül ezt a problémát csak úgy oldhatnánk meg, ha a nyelv ugyanúgy, mint az AL-ban a tömbök hosszára is megengedné a dinamikus kezelést, és ha az előállítandó rekordok száma a programfutás során már nem változna. Ugyanis a program futása során már a dinamikus méretű tömb is változatlan hosszúságú marad, és ezért a példabeli pointerezett listával ellentétben további tömbelemeket már nem fűzhetünk a tömbhöz. Mivel az ismert robotvezérlő nyelvek egyikében sem megoldott a pointer típusú változók kezelése, ezért a dinamikus adatszerkezetek felépítésének problémája sajnos nem megoldható.

AL-ban ugyan lehetővé van téve, hogy a program írásakor ismeretlen számú objektumot dinamikus méretű tömbökkel kezelhessünk, de e lehetőség kihasználásakor is hamar akadályba ütközünk. Dinamikus méretű tömbök használatánál ugyanis az ezzel együtt járó lineáris elrendezés jelent korlátozást és az elemek egyszer már deklarált száma is megváltoztathatatlan.

Lássuk ezek után a *3.9. programrészlet* második példáját. A *3.7. ábra* szerint a formaszekrényeken végrehajtott műveletek eredményeképpen a robot kettőt már összecsavarozott, egyet ezekre ráhelyezett, a negyedik pedig még érintetlenül ott áll mellettük. Ha tehát K1-et vagy K2-t mozgatjuk, akkor azzal együtt elmozdítjuk K3-at és K2-t, ill. K1-et is. Ha viszont K3-at mozdítjuk el, ettől K1 és K2 még érintetlen marad. Felmerül a kérdés, hogy hogyan lehetne ezeket a kapcsolatokat egyetlen adatszerkezetben ábrázolni, arra is gondolva, hogy ezek a kapcsolatok a program végrehajtása közben gyakran megváltozhatnak?! Egy ilyen környezetleíró modellben nemcsak az egyes tárgyak egy-

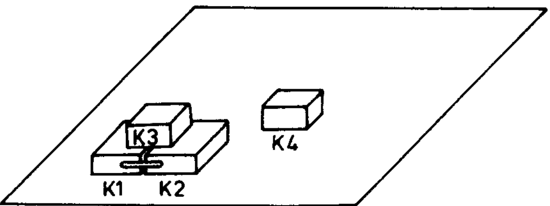
```
PROGRAM pointerespelda (input);
TYPE
   formaszekrenypointer        = ∧ formaszekrenytyp;
framepointer                   = ∧ frametyp;
affixpointer                   = ∧ affixelem;
transzpointer                  = ∧ transzformaciostyp;
vektortípus                    = ARRAY [1..3] OF REAL;
rotaciostipus                  = RECORD
                                   tengely: vektortípus;
                                   szög: REAL
                                 END;
   transzformaciostipus        = RECORD
                                   orient: rotaciostipus;
                                   pozicio: vektortipus
                                 END;
   frametipus                  = transzformaciotipus;
affixelem                      = RECORD
                                   affixexfszekreny:formaszekrenypointer;
                                   transzrelacio: transzpointer;
                                   kovetkezo: affixpointer
                                 END;
formaszekrenytyp               = RECORD
                                   koordrendszer:     frametipus;
                                   xhossz             :
                                   yhossz             :
                                   zhossz             : REAL;
                                   megfogasipozicio: frametipus;
                                   tele               : BOOLEAN;
                                   kovetkezo          : formaszekrenypointer;
                                   affixrel           : affixpointer
                                 END;
   PROCEDURE affix (kx, ky: formaszekrenypointer);
***************************************************
(*                                                *)
```

```
(*        A kx és ky közötti affix-kapcsolatokat úgy állítja elő,                      *)
(*        hogy kx függ ky-tól                                                          *)
(*        Hasonlít az AFFIX kx TO ky NONRIGIDLY AL-utasításhoz                          *)
(*                                                                                     *)
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
VAR
  affixelem: affixpointer;     (* uj affixelem *)
  affixhelp: affixpointer;     (* segedpointer *)
BEGIN (* affix *)
  NEW (affixelem);                            (* Előallit egy affixelemet *)
  affixelem∧.kovetkezo: = NIL                 (* Nem lesz kovetkezo elem          *)
  affixelem∧.affixedfszekreny: =kx;           (* Ez kx-re mutat                   *)
  IF ky∧.affixrel = NIL                       (* Megvizsgalja, hogy ky mar        *)
                                              (* bekerült-e egy affix megneve-    *)
                                              (* zésbe                            *)
  THEN ky∧. affixrel: = affixelem             (* Az előállított affixelemet       *)
   ELSE                                       (* közvetlenül ky után csatolja     *)
  BEGIN (* affixelem feltűzése *)
    affixhelp : = ky∧.affixrel;               (* A segedpointert az               *)
                                              (* első affixelemre állítja         *)
    WHILE affixhelp∧.kovetkezo< >NIL DO       (* utolsó affixelem                 *)
                                              (* keresése segedpointer            *)
      affixhelp : = affixhelp∧.kovetkezo      (* erre mutat                       *)
    affixhelp∧.kovetkezo: = affixelem         (* Az előállított affix-            *)
                                              (* elemet a lista végére            *)
                                              (* illeszti                         *)
    END (* affixelem felfutas vége *)
END (* affix vége *)

VAR
  Szekrenyek          : formaszekrenypointer;        (* listakezdet-mutató           *)
  szekrenyekszama     ;                              (* A listaelemek szama          *)
  i                   : INTEGER;                     (* seged                        *)
  k                   : formaszekrenypointer         (* valtozok                     *)
BEGIN (* peldaprogram *)
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
(*                                                                                     *)
(*        PROGRAM_RÉSZLET 1:                                                           *)
(*                                                                                     *)
(*        A 3.6. ábra szerinti listafelépítés                                          *)
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
  READ (szekrenyekszama);                     (* A szükséges darabszam            *)
                                              (* beolvasása terminalról           *)
  NEW (szekrenyek);                           (* Előállítja az első f.szekrenyt   *)
  Szekrények∧. kovetkezo: =NIL;               (*A szekrenynek nincs következő     *)
                                              (* eleme                            *)
  Szekrenyek∧. affixrel: =NIL;                (* A szekrenyeknek nincs            *)
                                              (* affix-kapcsolata                 *)
                                              (* Eredményt lásd a 3.5. ábrán      *)
  FOR i: = 2 TO szekrenyekszama DO            (* A DO ciklus 2-től                *)
                                              (* a szekrenyekszama-ig fut, fela-  *)
                                              (* data (szekre.szama-1) számú      *)
                                              (* szekrenyek rekord előállítása    *)
                                              (* és láncolása                     *)
  BEGIN (* ciklus *)                          (* A ciklus kezdete                 *)
    NEW (k);                                  (* Új szekrenyt állít elő           *)
    k∧.kovetkezo : = szekrenyek;              (* Az új szekreny a lista           *)
                                              (* legelső szekreny-elemére         *)
                                              (* mutat                            *)
    k∧.affixrel: = NIL;                       (* A szekrenyek-nek nincsenek       *)
                                              (* affix-kapcsolatai                *)
    Szekrenyek : = k;                         (* A lista elejére mutató szekreny  *)
                                              / a legelső új listaelemre          *)
                                              (* mutat                            *)
  END (* ciklus *)                            (* ciklusvege                       *)
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
(*                                                                  *)
(*      A 3.6. ábra szerinti listafelépítés befejeződött            *)
(*      PROGRAM_RÉSZLET 2:                                          *)
(*      Az affix-kapcsolatok felépítése 3.7. ábra szerint           *)
(*                                                                  *)
*****************************************************************
   k:= Szekrenyek^. következo^. kovetkezo;    (* k a k2 szekreny-     *)
                                              (* -rekordra mutat      *)
   affix (k^.kovetkezo, k);                   (* Affix kl függ k2-től *)
   affix (szekrenyek^.kovetkezo,k);           (* Affix k3 függ k2-től *)
   affix(k, k^. kovetkezo);                   (* Affix k2 függ kl-től *)
   affix(szekrenyek^.kovetkezo, k^.kovetkezo); (* affix k3 függ kl-től *)
   END (* a peldaprogram vege *)
```

3.9. programrészlet. Példaprogram pointer típusú változók alkalmazására PASCAL-ban



3.7. ábra. A formaszekrények elrendezése a munkaasztalon

máshoz viszonyított kapcsolatait kell kifejezésre juttatni, hanem az egymáshoz rögzített formaszekrények egymáshoz viszonyított helyzetét is le kell tárolni, aminek az egyes mozdulatok szempontjából van jelentősége. Ezeknek az AL-nyelvben *Affix-kapcsolatoknak* (németül Affix-*Beziehung*) nevezett relációknak az ábrázolására alkalmas az **affix-elem** adategyüttes, melynek komponensei a következők:

affixszekreny (arra a „szekrenyek"-re mutat, amellyel a kapcsolat fennáll, tehát amelyet együtt kell mozgatnunk);

transrelation (egy „transzformacio"-t pointerez, amely a két szekrény egymáshoz viszonyított helyzetét adja meg);

kovetkezo (a következő affixelemre mutat, ha több kapcsolat is fennáll).

A *3.9. programrészletben* definiált adatstruktúra segítségével felépíthető az affix-kapcsolatok ábrázolására alkalmas adatmodell *(3.8. ábra)*. Erre való a második programrészlet az **affix** eljárással kapcsolatban (l. a 2.2.4.2. pontot és a 4.1. szakaszt). Legyen **kl** és **k2** két formaszekreny-típusú pointerváltozó. Ekkor az

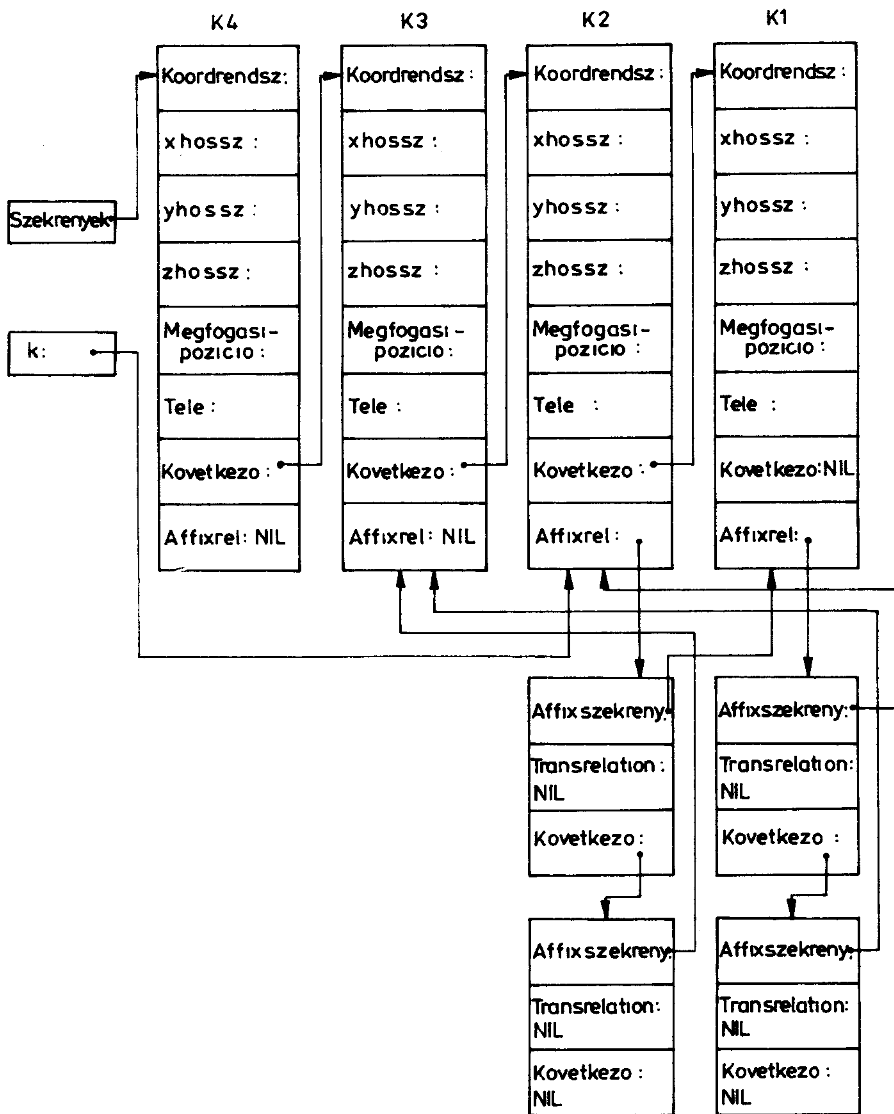
affix (kl, k2);

szubrutinhívás **kl**-et és **k2**-t egymástól teszi függővé, azaz **k2** mozgása **kl**-et is változtatja, fordítva azonban ez nem áll. Az ennek megfelelő AL-utasítás a következő:

AFFIX kl to k2 NONRIGIDLY;

Itt **kl**-nek és **k2**-nek frame-típusúnak kell lennie.

Míg az AL nyelvben csak frame-rendszereket hozhatunk egymással affix-kapcsolatba, a PASCAL-ban dinamikusan kezelt objektumok is affix-relációba hozhatók. A program könnyen bővíthető úgy, hogy tetszőleges dinamikusan előállított rekordok is affix-relációba hozhatók legyenek.



3.8. ábra. A *3.6. ábra* listaelemei közötti affix-relációk ábrázolása a *3.9. programrészlet* és a *3.7. ábrának* megfelelően

Ha az AL-ban a NONRIGIDLY kitételt RIGIDLY-vel helyettesítjük, akkor szimmetrikus affix-reláció jön létre. A *3.9. programrészletben* ennek az

affix (k1, k2);
affix (k2, k1);

utasításpár felelne meg.

E két szubrutinhívó utasításnak a sorrendje lényegtelen. A **k1^.koordrendsz** és **k2^.koordrendsz** kifejezésekkel kifejezhető **k1** és **k2** bázisframe-rendszerek közötti transzformáció meglehetősen bonyolult. Ennek az **affix** eljárásban történő kiszámításától az áttekint-

hetőség kedvéért eltekintettünk, mivel ez amúgy sem segíti elő a pointer-szerkezet tökéletesebb ábrázolását. Csupán utalni szeretnénk rá, hogy a számítás menete a DH-mátrixok használatával a következő

$$\mathbf{DH}_{\mathrm{transzf}} = \begin{pmatrix} \mathbf{R}_1 & l_1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{R}_2 & l_2 \\ 0 & 1 \end{pmatrix}^{-1}$$

Ez AL-ban a következő képletnek felelne meg:

DH_transf←k2^.koordrendsz→k1^.koordrendsz;

(l. még a 2.4.1. pontot). A példaprogram második része azzal kezdődik, hogy a **k** segédpointert úgy állítjuk be, hogy az a **K2** „szekrenyek” rekordra mutasson. Ezután négy szubrutinhívással előállítjuk a **K1, K2** és **K3** közötti affix-relációkat (vö. *3.8. ábrával*).

Bár az AL-ban az ilyen kapcsolatok megfogalmazásához rendelkezésünkre áll az AFFIX-utasítás (vö. 4.1. szakasszal), mégis bizonyára hasznos lenne, ha a PASCAL nyelvhez hasonlóan az AL nyelvben is lehetőség lenne egy sokoldalú és egyszerűen megvalósítható pointerezésre.

3.2. Az adatkezelés

A különböző adatobjektumok kezelésére a legkülönfélébb *műveletek* és *függvények* állnak rendelkezésünkre, amelyek segítségével *képleteket* írhatunk fel és értékelhetünk ki. Egy nyelv használhatósága egyaránt függ a megengedett adatszerkezetektől és az adatok kezelési lehetőségeitől.

3.2.1. Műveletek

Műveleteken a szokásos monadikus (egyoperandusú) és diadikus (kétoperandusú) műveleteket értjük, ellentétben az AL nyelv olyan függvényeljárásaival, mint a SIN(x), vagy pl. egy frame pozícióvektorát szolgáltató POS(f). A fordítóprogram (compiler) megvizsgálja, hogy a műveletben megengedhető-e a megadott típusú operandusok használata. Tehát pl. az **5 + 7** értelmezhető művelet, ellenben **5 + VECTOR (1,0,3)** nem az. Az egyszerűbb esetek könnyen beláthatók, de vannak nehezebb kérdések is. Például hogyan viselkedik a fordítóprogram egy integer és egy real típusú változó összeadásakor, mondjuk az

5 + 7.1

utasítás esetén?

Ebben az esetben *típusillesztés* történik, a gép az integer számértéket real típusúvá alakítja át. Az ilyen jellegű kérdésekre az egyes műveletek tárgyalásakor még részletesebben is kitérünk.

3.2.1.1. Aritmetikai műveletek

Az aritmetikai műveletek az INTEGER és a REAL adattípusokra vonatkoznak. Ezeket a változótípusokat az AL és a HELP nyelvekben a SCALAR változótípuson belül ábrázolhatjuk. A *3.6. táblázat* bemutatja az egyes nyelvekben megengedett aritmetikai

3.6. táblázat. Az egyes nyelvekben megengedett aritmetikai műveletek

<table>
<tr><th rowspan="2">Művelet</th><th rowspan="2">Az eredmény típusa PASCAL-ban</th><th colspan="5">Ábrázolás az egyes nyelvekben</th></tr>
<tr><th>PASCAL</th><th>AL</th><th>VAL</th><th>HELP</th><th>SIGLA</th></tr>
<tr><td>Összeadás</td><td>INTEGER
REAL
REAL
REAL</td><td>i+i
r+r
i+r
r+i</td><td>s+s</td><td>i+i</td><td>s+s</td><td>IC/i,i</td></tr>
<tr><td>Kivonás</td><td>INTEGER
REAL
REAL
REAL</td><td>i−i
r−r
i−r
r−i</td><td>s−s</td><td>i−i</td><td>s−s</td><td>IC/i,−i</td></tr>
<tr><td>Szorzás</td><td>INTEGER
REAL
REAL
REAL</td><td>i*i
r*r
i*r
r*i</td><td>s*s</td><td>i*i</td><td>s*s</td><td>./.</td></tr>
<tr><td>Osztás</td><td>REAL
REAL
REAL
REAL</td><td>i/i
r/r
i/r
r/i</td><td>s/s</td><td>./.</td><td>s/s</td><td>./.</td></tr>
<tr><td>Egészosztás</td><td>INTEGER i</td><td>DIV i</td><td>s DIVs</td><td>i/i</td><td>./.</td><td>./.</td></tr>
<tr><td>Modulo</td><td>INTEGER i</td><td>MOD i</td><td>s MODs</td><td>i%i</td><td>./.</td><td>./.</td></tr>
<tr><td>Hatványozás</td><td></td><td>./.</td><td>s^s</td><td>./.</td><td>./.</td><td>./.</td></tr>
<tr><td>Abszolútérték-képzés</td><td>INTEGER REAL</td><td>ABS |i|
ABS |r|</td><td>|s|</td><td>./.</td><td>./.</td><td>./.</td></tr>
<tr><td>Negálás</td><td>INTEGER REAL</td><td>−i
−r</td><td>−s</td><td>−i</td><td>−s</td><td>NE/−i</td></tr>
</table>

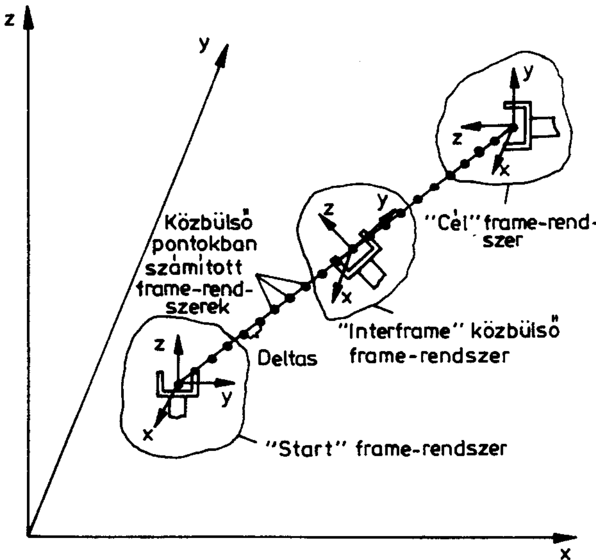
Megjegyzések: AL-ban és HELP-ben az adattípusok mindig SCALAR-ban adottak, míg az egészosztás és modulo esetén a tört rész elesik. VAL-ban és SIGLA-ban az adat mindig integer típusú.

műveleteket (i az integer, r a real és s a skalar típus rövidítése). A ROBEX-et ebben a táblázatban nem tüntettük fel, mivel a jelenlegi változatban a ROBEX nem kezel változókat és így a program futása során nem is hajt végre aritmetikai műveleteket a számítások elvégzésére. Ha az operandusok eltérő adattípushoz tartoznak, a PASCAL a négy alapműveletnél úgy hajtja végre a típusátalakítást, hogy az integer változót alakítja át real típusúvá. Egész számok osztásánál az eredmény törtrészét levágja. Sem az AL, sem a VAL nyelv nem rendelkezik afelől, hogy az egészosztás eredményét kerekíteni kell-e, vagy egyszerűen elhagyni a törtrészt, így ez a nyelv mindenkori gépi reprezentációjától függ. Amint a táblázat mutatja, a PASCAL és az AL a legtöbb műveletet értelmezni tudja, ez megkönnyíti a programozó munkáját, mivel az aritmetikai számítások könnyen kódolhatók. Ez a sokoldalúság már nem jellemzi a HELP-et, és a VAL nyelvben is csak INTEGER típusú változókkal végezhetők el a legfontosabb műveletek. A műveletekkel kapcsolatban a SIGLA nyelv nyújtja a legszűkösebb lehetőségeket.

3.2.1.2. Geometriai műveletek

Ahhoz, hogy a 3.1.4. pontban megismert geometriai adattípusokat egymással, vagy egyszerű adattípusokkal, továbbá strukturált adatszerkezetek elemeivel relációba hozhassuk, speciális geometriai műveletekre van szükségünk. A tárgyalt nyelvek közül csak az AL kínál elég széles választékot a különféle geometriai műveletekből, ezért ezt részletesen tárgyaljuk, majd egy példa kapcsán megvilágítjuk az alkalmazási lehetőségeket. A legtöbb gépen a billentyűzet csak korlátozott számban tartalmaz különleges jeleket. Emiatt, de más praktikus okokból is több geometriai művelet jelölésére az aritmetikai műveleteknél már megszokott műveleti jeleket kell használni. A *3.7. táblázat* az AL nyelv geometriai műveleteit mutatja be az egyoperandusú és kétoperandusú műveletek, valamint a művelet eredményének típusa szerint csoportosítva.

Az AL nyelv geometriai adattípusaira értelmezett standard függvényekkel – amelyeket a 3.2.2. pontban tárgyalunk – és az aritmetikai standard függvényekkel együtt már igen bonyolult pályavezérlési, illetőleg keresési műveleteket lehet explicit módon beprogramozni. Álljon itt ennek illusztrálására egy egyszerű AL nyelven írt program *(3. 10. programrészlet)*. Az AL nyelvet több robotból álló rendszer programozására is implementálták. Ebben a rendszerben egy-egy robot nem pályavezérlés szerinti mozgást valósít meg, hanem csak egy ún. ponttól pontig vezérlést (vö. 2.4.2. pont). Ilyen esetben a robotmanipulátor bizonyos mozgásoknál biztonsági okokból a kiindulási és a véghelyzet között egy egyenes mentén halad, és az orientációt is lineáris folyamat szerint változtatja. Az ilyen jellegű vezérlés nem gyakori, ezért a vezérlési eljárás kibővítése nem lenne gazdaságos, ezért megengedhető a vezérlés lassabb végrehajtása is. Ez a probléma az itt bemutatott AL nyelven írt programeljárással oldható meg *(l. 3.10. programrészlet)*. A program derékszögű koordináták szerinti *lineáris interpolációt* valósít meg. A közbülső pályapontok távolságát 0,5 cm-nek vettük fel, ezekre számítja ki a program a közbülső frame-rendszerek sorozatát *(3.9. ábra)*.



3.9. ábra. A *Descartes* koordináta-rendszerben végrehajtott lineáris interpoláció AL nyelven írt eljárásának szemléltetése

3.7. táblázat. Geometriai műveletek az AL nyelvben

A művelet jelölése	A művelet neve	Eredmény
$s \leftarrow \lvert v \rvert$	Vektor hosszának előállítása	$s = \sqrt{x^2+y^2+z^2}$
$s \leftarrow \lvert r \rvert$	A rotáció szöge	$s = \alpha$
$s \leftarrow v_1 \cdot v_2$	Skaláris szorzat	$s = x_1 * x_2 + y_1 * y_2 + + z_1 * z_2$
$v \leftarrow s * v$	Skalárral szorzás (nyújtás)	$v = (s * x, s * y, s * z)$
$v \leftarrow v/s$	Vektorok zsugorítása	$v = (x/s, y/s, z/s)$
$v \leftarrow v_1 + v_2$	Vektorok összeadása	$v = (x_1 + x_2, y_1 + y_2, z_1 + z_2)$
$v \leftarrow v_1 - v_2$	Vektorok kivonása	$v = (x_1 - x_2, y_1 - y_2, z_1 - z_2)$
$v \leftarrow v_1 * v_2$	Vektoriális szorzat	$v = (y_1 * z_2 - z_1 * y_2, z_1 * x_2 - x_1 * z_2, x_1 * y_2 - y_1 * x_2)$
$v \leftarrow r * v$	Vektor rotációja	v vektort a v_r rotációs vektor körül α szöggel elforgatjuk
$v \leftarrow t * v$	Vektortranszformáció	$v = r_t * (v_t + v)$
$v \leftarrow f * v$	Vektortranszformáció	$v = r_f * (v_f + v)$
$v \leftarrow v$ WRT f	A frame orientációjára vonatkoztatott vektortranszformáció (WRT = With Respect To)	$v = r_f * v$
$r \leftarrow r_1 * r_2$	Két rotáció láncolása	a rotációmátrixok szorzata
$f \leftarrow f + v$	Egy frame-rendszer transzformációja	$f = \text{FRAME}\ (r_f, v_f + v)$
$f \leftarrow f - v$	Egy frame-rendszer transzformációja	$f = \text{FRAME}\ (r_f, v_f - v)$
$f \leftarrow t * f$	Két transzformáció láncolása	$f = \text{FRAME}\ (r_t * r_f, v_t + v_f)$
$f \leftarrow f * f$	Egy frame-rendszer transzlációja	$f = \text{FRAME}\ (r_f * r_f, v_f + v_f)$
$f \leftarrow t * t$	Egy frame-rendszer transzlációja	$f = \text{TRANS}\ (r_t * r_t, v_t + v_t)$
$t \leftarrow f_1 \rightarrow f_2$	Transzformáció szerkesztése	Az $f_2 \leftarrow t * f_1$ egyenlet megoldása

A rövidítések jelentése a következő:
s = SCALAR s
v = VECTOR (x, y, z)
r = ROT (v_r, α)
f = FRAME (r_f, v_f)
t = TRANS (r_t, v_t)

```
PROCEDURE smove (VALUE FRAME cél);
BEGIN "smove"
  SCALAR     deltas, szög, posanz, deltaszög számláló;
  VECTOR     startpos, translácio, elfordvektor, deltav;
  ROT        startorient, orientváltás;
  FRAME      start, transframe, interframe;
  TRANS      startcéltrans;

  start←ARM2;                          (* A robot aktuális pozícióját     *)
                                       (* a start nevű frame-rendszer-    *)
                                       (* ben tároljuk                    *)
  deltas←0.5;                          (* Az interpoláció lépésköze       *)
  startpos←POS (start);                (* A kiinduló helyzet pozícióadata *)
  startorient←ORIENT (start);          (* A kiinduló helyzet orientációja *)
  startceltrans←start→cél;             (* A start-tól a cél-ig végre-     *)
                                       (* hajtott elmozdulás és orien-    *)
                                       (* táció-megváltozás               *)
  transframe←startcéltrans *           (* Úgy transzformáljuk mint egy    *)
              FRAME (NIL ROT           (* frame-et                        *)
                VECTOR (0,0,0) * cm);
  orient change←ORIENT (transframe);   (* Orientáció-megváltozás          *)
  translacio←POS (transframe);         (* Elmozdulási vektor              *)
  szog←orientchange |;                 (* Az orientáció-megválto-         *)
                                       (* zás elfordulási szöge           *)
  elfordvektor←AXIS(orientchange);     (* Az orientáció-megválto-         *)
                                       (* zás elfordulási szöge           *)
  posanz←(|translacio|DIV deltas) −1;  (* A közbülső pontok száma         *)
  deltav←translacio/posanz;            (* A közbülső frame-helyzetek      *)
                                       (* közötti elmozdulas-adat         *)
  deltaszog←szog/posanz;               (* A lépésközönkenti szög-         *)
                                       (* változás                        *)
  FOR szamlalo←1 STEP 1 UNTIL posanz DO
  BEGIN                                (* A közbülső frame-rendsze-       *)
                                       (* rek kiszamitasa                 *)
      interframe←FRAME (startorient * ROT (elfordvektor, deltaszog * szamlalo);
                          startpos + deltav * szamlalo);
    MOVE ARM2 TO interframe;           (* A közbülső frame-rendszerek     *)
                                       (* szerinti mozgás-vezérlés        *)
  END;
  MOVE ARM2 TO cel;                    (* A vezérlés vége                 *)
END "smove";
```

3.10. programrészlet. Descartes koordinátákban végrehajtott lineáris interpoláció

Ha a programrészlet felhasználásával a robotkart úgy akarjuk vezérelni, hogy a megfogóeszköz egy „lerakóhely"-et közelítsen meg, akkor a következő eljáráshívást adhatjuk ki:

SMOVE (lerakóhely);

de előzőleg a **„lerakóhely"** nevű frame-rendszert már definiálnunk kell akár explicit módon, akár betanítási (teach-in) módszerrel.

Ebben az AL nyelven készült megoldásban a tömör, problémaorientált fogalmazáson kívül az a körülmény is igen figyelemre méltó, hogy az orientáció lineáris interpolációját is elvégzi. Vegyük figyelembe, hogy olyankor, amikor a programozó több különböző forgástengely körüli rotáció segítségével adja meg az orientáció megváltozását, a program csak egy rotációs mátrixot számít ki. Ebből mindig egyetlen egyértelműen meghatá-

rozott forgástengely, ill. egyetlen forgatási szöggel meghatározott egyértelmű forgatási vektor vezethető le. Ekörül a forgástengely körül kell lépésenként sorozatosan elforgatni a kiinduló frame-rendszert, míg el nem érjük a célként megadott frame-rendszer orientációértékét. Ezzel az eljárással egyébként megoldható az is, hogy a célértékként megadott frame-rendszert interpolált mozgással közelítsük meg, ahol a céladatokat előzőleg ugyanúgy betanítási eljárással vagy érzékelők segítségével kell definiálnunk.

VAL nyelv esetében – jóllehet megvan a frame-rendszer kezelési lehetősége – a programozó nem definiálhat geometriai műveleteket.

VAL-ban még egy adott frame-rendszer egyes koordinátaadatai sem kérdezhetők le közvetlenül. Például olyan esetben, amikor egy kontaktusérzékelő miatt a robotkar mozgása megáll, és le szeretnénk kérdezni a z-koordináta értékét. A frame-rendszer eltolásának kiszámításához azonban csak a SHIFT utasítást hívhatjuk segítségül. Ha pl. valamennyi frame-rendszert el kell tolnunk, vagy a báziskoordináta-rendszer z tengelye körül el kell forgatnunk amiatt, mert mondjuk a robotkar egy másik helyzetbe került, akkor ezt a BASE utasítással hajthatjuk végre *(3.10. ábra)*.

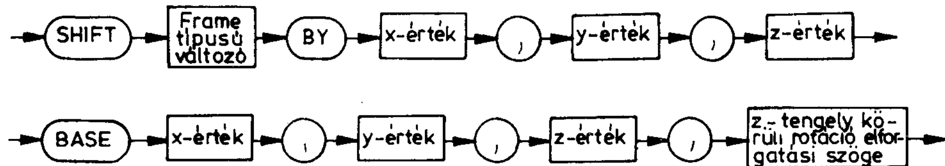
Minthogy a SHIFT és a BASE utasításokban az x-, y- és z-értékeinek csak konstansok engedélyezettek, változók nem, minden programozói ügyességre szükségünk van, ha pl. a frame-rendszer eltolásának értéke egy *érzékelő jelétől* függ, azaz nem állandó. A feladatot AL nyelven a következő egyszerű utasítással oldhatjuk meg:

célframe←célframe + VECTOR (xertek, yertek, 0);

Ezzel szemben VAL-ban erre a problémára csak egy eléggé nehézkes megoldás kínálkozik. Ezt mutatjuk be a *3.11. programrészletben.*

Ezen a téren a VAL nyelv egy voltaképpen felesleges korlátozást állít a programozó elé, hiszen a SHIFT utasításban a változók kezelésének implementálása csekély ráfordítással megoldható lenne *(3.11. programrészlet)*.

A HELP, a SIGLA és a ROBEX nyelvekben geometriai művelet nincs értelmezve.



3.10. ábra. A frame-rendszer lineáris eltolására (transzlációjára) alkalmas VAL-utasítások

```
    SETI szamlalo = 0
100 SHIFT celframe BY 1, 0, 0
    SETI szamlalo = szamlalo + 1
    IF szamlalo LE xértek THEN 100
    SETI szamlalo = 0
200 SHIFT celframe BY 0, 1, 0
    SETI szamlalo = szamlalo + 1
    IF szamlalo LE yérték THEN 200
```

3.11. programrészlet. VAL nyelven írt példaprogram a frame-rendszer transzlációjának végrehajtására

3.2.1.3. Összehasonlítási műveletek

Az egyes nyelvekben az integer, a real és a skalár típusú változók egymásközti összehasonlítására a *3.8. táblázatban* felsorolt műveletek ismeretesek.

Az összehasonlítási műveletek segítségével egyszerű logikai kifejezések írhatók fel, pl. s<s is logikai kifejezés. A VAL-ban és a HELP-ben logikai kifejezéseket csak vezérlésátadó utasításokban lehet használni (4.7. szakasz). Csupán az AL és a PASCAL engedi meg, hogy logikai kifejezések eredményét (értékét) változónak is átadhassuk. Az AL-ban, HELP-ben és a PASCAL-ban a műveletek konstansokra, változókra és aritmetikai kifejezésekre egyaránt értelmezve vannak, a VAL-ban ezzel szemben csak konstansokra és változókra.

REAL típusú változók között egyenlőségre vagy egyenlőtlenségre történő vizsgálat általában nem célszerű, mert az összehasonlítandó számértékek kiszámításakor keletke-

3.8. táblázat. Az összehasonlítási műveletek

Összehasonlítási művelet	PASCAL	AL	VAL	HELP	SIGLA	ROBEX
kisebb mint	x<x	s<s	i LT i	s<s	*	LT
nagyobb mint	x>x	s>s	i GT i	s>s	*	GT
egyenlő	x=x	s=s	i EQ i	s=s	*	EQ
kisebb egyenlő	x<=x	s<=s	i LE i	./.	./.	LE
nagyobb egyenlő	x>=x	s>=s	i GE i	./.	./.	GE
nem egyenlő	x<>x	s<>s	i NE i	./.	./.	NE

Jelölések: x real típusú vagy integer típusú számot jelöl
s scalar típusú számot jelöl
i integer típusú számot jelöl.

* A SIGLA nyelvben kizárólag egészen speciális feltételes vezérlésátadási utasításokon belül végezhetünk logikai összehasonlítást (4.7.1.1 pont).

** A ROBEX nyelvben a logikai összehasonlítások használatát a feltételes vezérlésátadó utasításoknak csak egy különleges típusánál tervezik bevezetni.

ző kerekítési hibák miatt algebrailag egyébként azonosan egyenlő kifejezések is eltérő számértékeket vehetnek fel (skalár típusú változóknál ugyanez a helyzet) (vö. a 3.1.3. ponttal).

3.2.1.4. Logikai műveletek

A logikai műveletek segítségével logikai típusú változók, konstansok és olyan egyszerű logikai kifejezések közötti logikai természetű kapcsolatokat tudunk kiértékelni, amelyeket összehasonlítási műveletekkel fejezünk ki. Például az

a>5 AND a<7

logikai műveletnek csak akkor lesz az eredménye igaz („true"), ha **„a"** értéke **5** és **7** közötti szám. Ilyen kifejezéseket csak PASCAL-ban, AL-ban és HELP-ben írhatunk fel. A *3.9. táblázat* bemutatja az egyes nyelvekben felírható műveleteket. A HELP-ben csak

3.9. táblázat. Logikai műveletek

Művelet	PASCAL	AL	HELP
Logikai ÉS	b AND b	s AND s	expr AND expr
Logikai VAGY	b OR b	s OR s	expr OR expr
Logikai NEM	NOT b	NOT s	NOT expr
Kizáró VAGY	b< >b	s XOR s	./.
Ekvivalencia (egyenlőség)	b=b	s EQV s	./.

Jelölések:
b boolean típusú konstans, változó vagy egyszerű logikai kifejezés,
s skalár típusú konstans, változó ill. kifejezés vagy logikai konstans,
expr egyszerű logikai kifejezés.

olyan logikai kifejezések írhatók fel, amelyeket programunkban vezérlésátadások szervezésére használunk (l. még a 4.7. szakaszt is).

Az AL nyelv nem kezel logikai típusú változókat, ehelyett azonban minden nullánál nagyobb skalár-típusú számérték olyan értelemben is használható, mint a logikai TRUE érték.

3.2.2. Standard függvények

A 3.2.1. pontban a műveletek fogalmát elhatároltuk azoktól a jellegzetes függvényektől, amelyek az egyes programnyelvekben mint előre definiált függvények állnak rendelkezésünkre. Ezeket az előre definiált függvényeket standard függvényeknek nevezzük. A leggyakrabban használt függvényeknek és műveleteknek kialakult egy meghatározott köre, amelyek a különféle programnyelvekben közvetlenül hívhatók. Ezek közt általában a legfontosabb trigonometriai függvények, legalább egy REAL-ből INTEGER-re konvertáló típusátalakítási művelet, az exponenciális függvény, a természetes alapú logaritmus és a négyzetgyökvonás szerepelnek.

A HELP-ben az *aritmetikai standard függvényeknek* két változata van meg; egyrészt egy alapváltozat, melyben a szögfüggvényeknél a szögeket ívmértékben kell megadni, de ebben a változatban a gyökfüggvények nem szerepelnek. Másrészt létezik egy bővített változat, amely az SQRT-függvény mellett olyan szögfüggvényeket tartalmaz, melyeknél a szögértékeket közönséges fokban kell megadni.

A *3.10. táblázatban* felsorolt standard függvényeken kívül az AL nyelvben még egy sor olyan függvény is van, amelyek geometriai típusú adatokat kezelnek *(3.11. táblázat)*.

A VAL nyelv nem kezel függvényeket, ezért pl. egy frame-rendszert külön utasítás segítségével kell definiálnunk és egy másik utasítással végezhetjük az invertálást:

FRAME⟨argumentumok⟩ = ⟨frame1⟩, ⟨frame2⟩, ⟨frame3⟩
INVERSE⟨argumentumok⟩ = ⟨frame⟩

A *3.11. táblázatban* felsorolt standard függvények és a 3.2.1.2. pontban tárgyalt műveletek együttesen széles körű lehetőségeket nyújtanak az AL nyelvben a geometriai adattípusok kezelésére. Ebből a szempontból az AL a leghatékonyabban és legkényelmesebben használható robotvezérlési programnyelvnek tekinthető.

Minthogy a ROBEX rendszer jelenlegi verziójában változók használata nincs megengedve, a ROBEX-ben meglevő standard függvények arra valók, hogy a fordítás ideje alatt a négy alapművelettel együtt számításokat lehessen végezni velük. Írhatjuk tehát pl. a következőt:

3.10. táblázat. Aritmetikai standard függvények

Függvény	PASCAL	AL	HELP	ROBEX
Szinusz	SIN	SIN	SIN	SIN
Koszinusz	COS	COS	COS	COS
Tangens	./.	TAN	./.	./.
Árkusz tangens	ARCTAN	ATAN2	ARCTG	ATAN
Árkusz szinusz	./.	ASIN	./.	./.
Árkusz koszinusz	./.	ACOS	./.	./.
REAL INTEGER típusátalakítás – a törtrész levágásával	TRUNC	INT	./.	./.
– kerekítéssel	ROUND	./.	./.	./.
Exponenciális függvény	EXP	EXP	./.	EXP
Természetes alapú logaritmus	LIN	LOG	./.	NLOG
Logaritmus	./.	./.	./.	LOG
Négyzetgyökvonás	SQRT	SQRT	SQRT	SQRT
Négyzetreemelés	SQR	*	./.	./.
Abszolútérték	ABS	**	ABS	ABS
Beolvasás – skalár értékké	***	INSCALAR	ASK	./.
– logikai típusú értékké	***	QUERY	ASKN	./.

* A négyzetreemelés az AL nyelvben az általános hatványraemelési művelet segítségével végezhető el, melynek műveleti jele ∧.
** AL-ben az abszolútérték-képzés az |s| művelettel történik.
*** PASCAL-ban a beolvasási, ill. kiviteli műveletet standard beviteli-kiviteli eljárások segítségével végezhetjük el.

3.11. táblázat. Az AL nyelv geometriai típusú változókra értelmezett standard függvényei

A függvény hívása	Az eredmény típusa	Elvégzett művelet leírása
VECTOR (s,s,s)	VECTOR	Vektort állít elő
UNIT (v)	VECTOR	A v vektor egységvektorát állítja elő
POS (f) POS (t)	VECTOR	Előállítja egy frame vagy egy transzformáció helyvektorát
AXIS (r)	VECTOR	Az r rotáció rotációs tengelyét határozza meg
ROT (v,s)	ROT	A v tengely körül s szöggel történő rotációt állítja elő
ORIENT (f) ORIENT (t)	ROT	Egy f frame vagy t transzformáció orientációját adja meg
FRAME (r,v)	FRAME	Az r orientációval és v helyvektorral jellemezhető frame-rendszert állítja elő
CONSTRUCT (v1,v2,v3) CONSTRUCT (f1,f2,f3)	FRAME	Előállít egy v1-gyel, ill. POS (f1) helyvektorral jellemezhető frame-rendszert v2 ill. POS (f2) x-irányába mutat, v3 ill. POS (f3) pedig az xy sík egy pontja
TRANS (r,v)	TRANS	Előállítja az r elforgatással a v eltolással jellemzett transzformációt
INV (t)	TRANS	A t inverz transzformációját határozza meg

Jelölések: s skalár
v vektor
r rotáció
f frame
t transzformáció típusú változók

```
PONT = POINT/100 * COS(30), 100 * SIN(30),0
```

amivel egy **PONT** nevű pontot állítunk elő az xy-síkban, melynek távolsága az origótól 100 mm és a pontot az origóval összekötő szakasz az x tengellyel 30°-os szöget zár be. Ezeket az adatokat természetesen konstans értékek formájában is megadhatnánk, ha a felírt számításokat előzetesen zsebszámológépen elvégeznénk. Az, hogy egy zsebszámológép nyújtotta lehetőségeket a compiler-fordítóprogram szolgáltatásai közé sorolnak, ma már nem tekinthető különösen hasznosnak vagy kényelmesnek. Figyelembe kell azonban venni, hogy a numerikus vezérlésekhez kifejlesztett EXAPT nyelv – amelynek alapján a ROBEX-et fejlesztették – abban az időszakban alakult ki, amikor zsebszámológépről még csak nem is álmodhattunk.

A PASCAL és az AL nyelvben a felhasználónak megvan a lehetősége arra, hogy a mindenkori alkalmazási feladat tipikus képleteinek kiszámításához tetszőleges függvényeket definiáljon. Erről már tettünk említést a 2.2.4. pontban, de még egyszer visszatérünk a kérdésre a 6.3. szakaszban is.

3.2.3. Komplex kifejezések

A korábbi fejezetekben bemutatott műveletek és standard függvények felhasználásával képletek, ill. egészen komplex kifejezések is felírhatók. Ezzel kapcsolatban rögtön felmerül a műveletek prioritási sorrendjének kérdése. Ez természetesen csak a PASCAL, az AL és a HELP nyelvekre vonatkozik, hiszen csak ezekben a nyelvekben lehet egyáltalán képleteket felírni. A VAL és a SIGLA csupán azt teszi lehetővé, hogy két integer szám között olyan egyszerűbb műveleteket el lehessen végezni, mint pl.

```
SETI A = B⟨op⟩C
```

ill.

```
IC/a,b
```

Itt **B** és **C** integer számérték vagy változó lehet, **⟨op⟩** pedig a *3.6. táblázatban* feltüntetett műveletek egyikét jelenti. **IC/a,b** tehát hozzáadja **b** értékét az **a** változó értékéhez. Az összetettebb számításokat ezeknél a nyelveknél kétoperandusú műveletek sorozatára kell felbontanunk. Tehát pl. PASCAL-ban felírhatjuk, hogy

```
x := (a+b) DIV c; (* Egészosztás *)
```

(ahol **a**, **b**, **c**, **x** legyenek integer típusú változók). Ez az utasítás VAL-ban a következőképp fejezhető ki:

```
SETI X = A + B
SETI X = X / C
```

A SIGLA nyelvben még az osztási műveletet is explicit programozással kell felírnunk. (Erre vonatkozóan l. a *3.12. programrészletet!*)

A ROBEX nyelv jelenleg érvényes és ismert megfogalmazásában nem rendelkezik a műveletek prioritási sorrendjéről és csak feltételezhető, hogy legelőször a standard függvények kerülnek kiértékelésre, majd ezt követően a szorzási és osztási műveletek, legvégül pedig az összeadási és kivonási műveletek.

A *3.12. táblázat* áttekintést nyújt az egyes programnyelvek esetében a műveletek prioritási sorrendjéről. Megjegyezzük, hogy a PASCAL-nál a +, − műveleti jelek egyaránt jelentik az illető művelet kétoperandusú és monadikus (ún. egyoperandusú) változatát. A PASCAL-ban és a VAL-ban a program az egyes műveleteket azonos prioritásúaknak tekintve balról jobbra haladva dolgozza fel, míg a HELP nyelvben ezzel kapcsolatban nincs lerögzített szabály. Ennek a kérdésnek különösen az AL-nál van nagy jelentősége, amint ezt a következő példa is szemlélteti:

Jelentése:

SE/M1, M2;	x := A
IC/M1, M3;	x := A + B
SE/M5, 0;	Töröld a ciklusszámlálót
NU/10;	A ciklus címkéje
IC/M5, 1;	Inkrementáld a ciklusszámlálót
IC/M1, -M4;	a + b értékéből c levonása
BG/M1, 0, 10;	Vizsgálat: véget ért-e a ciklus (M1 < = 0)
BE/M1, 0, 11;	Vizsgálat: az osztás végrehajtható-e? (azaz az adott helyértéken való kivonás eredménye ≧0?)
IC/M5, -1;	Az osztás eredményét korrigálja, ha az osztási (kivonási) művelet negatív számra vezet
NU/11;	Az ugrás címkéje, ha az osztás eredménye ≧0, azaz végrehajtható
SE/M1, M5;	Az osztási művelet eredményét X-nek adja át

A programrészletben M1 tartalmazza X-et
M2 tartalmazza A-t
M3 tartalmazza B-t
M4 tartalmazza C-t
M5 tartalmazza az osztás eredményét

3.12. programrészlet. (a + b)/c programozása SIGLA nyelven (egészosztás)

3.12. táblázat. A műveletek prioritási sorrendje PASCAL-ban, AL-ban és HELP-ben

Prioritás	PASCAL	AL	HELP
1	Függvényeljárás hívása ( ), NOT	Függvényeljárás hívása ( ), ||, NOT +, −(csak mint előjelek)	Standard függvények hívása () NOT
2	*,/,DIV,MOD,AND	WRT,→, 1	*,/ AND
3	+, −, OR	*,/, .,DIV,MOD,MAX,MIN	+, − OR
4	=,<,>,< =, > =,< >	+, −	=,<,>
5		=,<,>,< =, > =,< >	
6		AND	
7		OR, XOR	
8		EQV	

A legmagasabb prioritás az 1-es.

A

v1. v2 * v3

(v1. v2) * v3

vagy a v1.(v2 * v3) kifejezések a műveletek elvégzésének sorrendjétől függően vezetnek más és más eredményre.

A PASCAL, az AL és a HELP nyelvekben az a lehetőség, hogy komplex aritmetikai kifejezéseket a matematikában megszokott módon írhatunk fel, nagyon megkönnyíti a program olvashatóságát. Nézzük pl. a következő AL programrészletet:

IF | POS(kar).zhat − POS(munkadarab).zhat | <zbiztonsági távolság
THEN {Utasítás az összeütközés elkerülésére}.

Ezzel a programrészlettel ellenőrizhetjük a robotkarnak z-irányban való pozicionálá-

sát. (Itt **zhat** egy a z tengely irányában előre definiált egységvektor.) A z-irányban végzett relatív karmozgatás ekkor kifejezhető a

MOVE kar TO kar + distanz * zhat

utasítással, ahol **distanz** egy SCALAR típusú változó.

Végezetül álljon itt egy az AL nyelvben felírt – és nem is egészen komolynak szánt – programpélda. A 3.2.1.2. pontban megadott adatok felhasználásával érdemes elgondolkozni a példabeli művelet végeredményén. (A megoldás a fejezet végén található meg.)

```
| v * INV(f2 → (v−f1)) | = | (FRAME(ORIENT(f1),
v − POS(f1)) → f2) * v | AND v.v * s = | v | ^2 * s
```

3.3. Értékadó utasítások

Értékadó utasítás segítségével egy képlettel kiszámított, vagy standardfüggvénnyel előállított számértéket átadhatunk egy változónak, amely művelet során a változó korábbi értéke törlődik.

3.3.1. Konstans értékadás

Az értékadó utasítás eredeti és legegyszerűbb formája a konstans értékadás, amelynél valamely változónak egy vele azonos típusú konstans értéket adunk. Ennek az utasításnak a különféle nyelvekben a legkülönbözőbb felírási módjai ismeretesek *(3.13. táblázat)*.

PASCAL-ban, AL-ban és HELP-ben az értékadó utasítás jobb oldalán tetszőleges összetett kifejezés is állhat. A „tetszőleges" jelző természetesen bizonyos korlátozásokkal értendő, hiszen a compiler-program az egyes implementációktól függően csak bizonyos mérvű bonyolultságot enged meg.

ROBEX-ben az értékadás úgy értendő, hogy a compiler-program az értékadást követően az összes utasításban az A változónév helyett pl. az 5-ös számértéket veszi.

Valamennyi nyelv megköveteli, hogy a bal oldali változó típusa egyezzen meg a jobb oldali adat típusával. Egyedüli kivétel a magasabb szintű programnyelvekben az a lehetőség, hogy egy real típusú változónak egy integer típusú számérték segítségével is lehet értéket adni, de ennek során is végbemegy egy ún. implicit típusátalakítás. Ugyanezt az eljárást követi a PASCAL is, a könyvünkben tárgyalt robotvezérlő programozási nyelvekben pedig fel sem vetődik ez a kérdés, mivel az AL és a HELP nyelvek nem tesznek különbséget integer és real típusú változók között, a többi nyelvben pedig nem is ábrázolhatunk real típusú változót.

Az AL nyelv az adattípusok megegyezésén kívül még a mértékegység – az ún. dimenzió

3.13. táblázat. Az értékadási utasítások írásmódja az egyes nyelvekben

PASCAL:	a := 5;
AL:	a ←5;
VAL:	SETI A=5
HELP:	A := 5;
SIGLA:	SE/M1,5
ROBEX:	A=5

– egyezését is megköveteli, ahol a dimenzionált változókkal végzett aritmetikai műveletek során adott esetben igen bonyolult dimenziójú kifejezések jöhetnek létre. Az AL nyelv mindezeken túlmenően még az összeadás és a kivonás műveleteinél is ellenőrzi a dimenziók azonosságát. A típusazonosság ellenőrzésének ez a rendszere a program *megbízhatóságát* hivatott fokozni. Ugyanígy a SCALAR és a VECTOR adattípusokra végrehajtott ellenőrzések is indokoltnak mondhatók. Kétségesnek tűnik azonban, hogy a dimenzionált rotációs, frame- és transzformációs kifejezések növelnék a program áttekinthetőségét és megbízhatóságát. Ezekkel a kérdésekkel kapcsolatban azonban majd csak azután lehet megalapozott véleményt kialakítani, ha az AL nyelv a gyakorlati ipari alkalmazások fázisába jut.

A *3.10. programrészlet* az értékadó utasítások használatára mutat példát. Az itt bemutatott értékadási típusok mellett létezik még az ún. implicit értékadás is. Ilyennek tekinthetők pl.: számértékek beolvasása (3.3.2. pont), az AL pozicionáló utasítása (3.3.2.2. pont), vagy egyéb olyan utasítások, amelyek megváltoztatják valamely változó értékét anélkül, hogy ez valamilyen közvetlenül látható utasításban felismerhető lenne. Példa erre az **AFFIX**-utasítás egy speciális alakja az AL nyelvben (**AFFIX...AT⟨trans⟩**), amelyet a 4.1. szakaszban tárgyalunk részletesebben.

3.3.2. Beviteli utasítások

A program futása közben végrehajtott beviteli (input) műveletek különféle eredetűek lehetnek: előfordul a felhasználóval létrejövő párbeszéd (ún. dialógus) során végrehajtott adatbevitel – pl. képernyőn keresztül –, másrészt létrejöhet adatbevitel műszaki berendezések, periferiális eszközök lekérdezése során is, így pl. a robotkar helyzetérzékelője, vagy a robot környezetében elhelyezett végálláskapcsolók lekérdezése során. Az egyes robotvezérlő nyelvek hajlékonysága és kezelhetősége erősen függ a beviteli műveletek sokrétűségétől és természetesen ugyanígy a lehetséges beviteli műveletektől is. A program ugyanis csak olyan mértékben illeszkedhet a technológiai környezethez, amilyen mértékben a nyelvben megvan a lehetőség a műszaki és kezelői környezettel megvalósítható kommunikációra.

A továbbiakban a kezelői adatbevitellel és a robotkar-pozíció, ill. térbeli koordináta bevitelével kapcsolatos utasítások fajtáit tárgyaljuk. A technológiai környezetből érkező megszakításkérésekkel, ill. az érzékelők adatainak bevitelével kapcsolatos utasításokra a 4. fejezetben kerül sor.

3.3.2.1. Adatbevitel párbeszédes technikával (interaktív módszer)

Numerikus értékek, ill. logikai típusú értékek bevitelére csak az AL és HELP nyelvekben van lehetőség standard függvények segítségével. Ezekről a *3.14. táblázat* közöl áttekintő összefoglalást.

A **⟨text⟩**, ill. a **⟨printlist⟩** tartalma egy rendszerüzenettel együtt megjelenik a terminálon, amely megadja, hogy milyen adattípust vár a rendszer.

Boolean (logikai) típusú adatok bevitelekor (QUERY, ASK) a vezérlőrendszer választ kér a felhasználótól. A válasz lehet Y (YES rövidítése a **TRUE** érték esetén) vagy N (NO rövidítése **FALSE** érték esetén). Ezt követően a felhasználó beviheti a kívánt adatokat. A program futása mindaddig szünetel, amíg az adatok bevitele tart. Az INSCALAR típus definiálásakor sajnos nem gondoskodtak a **⟨printlist⟩** létrehozásáról. Ehelyett az INSCALAR a „Scalar, please" üzenettel jelentkezik a képernyőn, így a programozónak

3.14. táblázat. Adatbeviteli eljárások

Az eredmény típusa	AL (utasítás)	HELP (utasítás)
SCALAR*	QUERY (⟨printlist⟩)	ASK ('⟨text⟩')
SCALAR	INSCALAR	ASKN ('⟨text⟩')

⟨text⟩ tetszőleges karaktersorozat a ' (felsővessző) karakter kivételével
⟨printlist⟩ egymástól vesszővel elválasztott paraméterek sorozata. A paraméterek algebrai kifejezések, változók vagy "⟨text⟩" típusúak lehetnek
* logikai típusú értékekre is skalár értéket kezel

kell jóelőre gondoskodnia arról, hogy ezt megelőzően egy PRINT (⟨**printlist**⟩) utasítással kiadjon egy ⟨printlist⟩-et arról, hogy tulajdonképpen mit is vár a program a kezelőtől. Mivel a beviteli rutinoknál tulajdonképpen függvényeljárásokról van szó, az eredményt azonnal ki kell értékelni, amint azt a *3.13. programrészlet* mutatja.

Itt a **magassag** nevű változó deklarációja DISTANCE SCALAR (l. a 3.1.1. pontot).

Az adatbeviteli eljárások ellenőrzik a beadott adatokat. Tehát pl. egy számérték csak előre meghatározott jelekből állítható össze: számjegyekből és egyetlen tizedespontból, és természetesen a számítógép ábrázolási tartományában felírhatónak kell lennie. Az adatbeviteli hibákra adott válaszok már az implementációtól függnek, a hibaüzenetet követően pl. megismételhető az adatbeviteli kérelem stb.

Célszerű, ha a programon belül is gondoskodunk a beolvasott adat hihetőségvizsgálatának elvégzéséről, hiszen pl. egy 10 000 cm-es nagyságrendű hosszúságadat a számítógépben ábrázolható ugyan, azonban a szóban forgó technikát tekintve minden bizonnyal hibás.

A többi robotvezérlő programozási nyelvben nincs lehetőség rá, hogy a program futása közben a felhasználóval dialógust folytassunk.

A PASCAL nyelv READ-utasítása az adatbeviteli eljárás egy általánosabb formáját kínálja. A READ utasítás tetszőleges számú paramétert tartalmazhat. Ezeknek karakter, integer vagy real típusú változóknak kell lenniük, amelyeknek az érték átadódik. Ezzel a READ-utasítással akár fájlokból, akár pedig terminálról (mint speciális fájlról) kérhetünk adatot. A WRITE-utasítással ennek a fordítottja végezhető el: dialógus esetén adatokat adhatunk ki a terminálra (ill. általános esetben egy fájlba írhatjuk azokat).

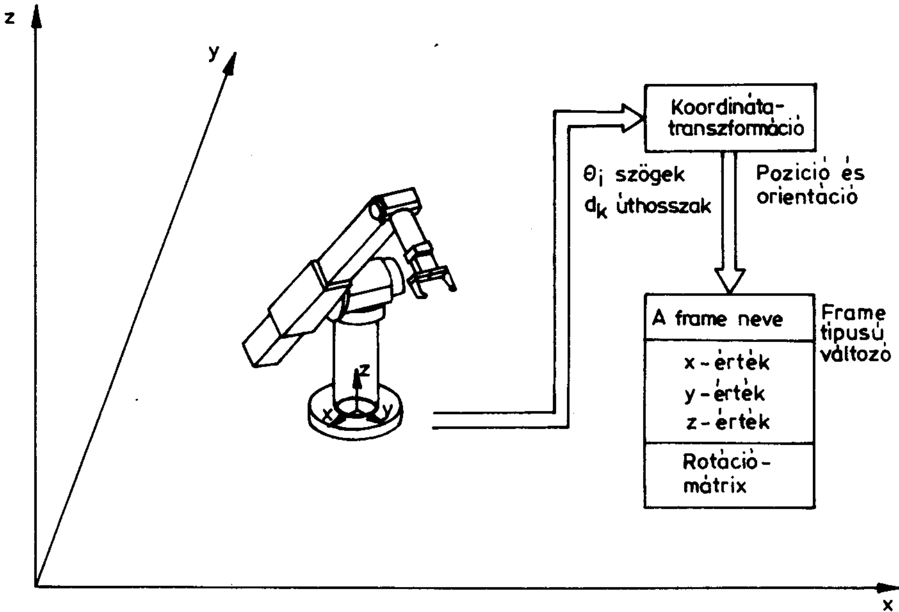
```
z←0;
WHILE QUERY (z, "Szekrények feldolgozva. Folytassuk?") DO
BEGIN
  PRINT ("Milyen magas a szekrény?");
  magassag←INSCALAR * cm;
  z←z + 1;
  :
END;
```

3.13. programrészlet. Példa adatbeviteli eljárásra AL nyelven

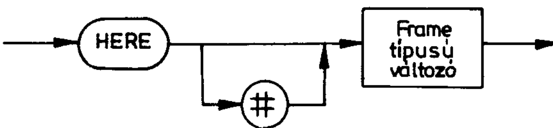
3.3.2.2. A robot aktuális pozíció- és orientációadatainak bevitele

A robotberendezések – mindenekelőtt a szerelőrobotok – többnyire egy mérőrendszerrel vannak felszerelve, amely a megtett utat méri. Ez a mérőrendszer a robot aktuális helyzetének, más szóval a robot pillanatnyi koordinátaértékeinek lekérdezését teszi lehetővé. A legtöbb robotvezérlő nyelvben megvannak azok az utasítások, amelyek elindítják az egyes tengelyek szöghelyzetét és az elmozdulási úthosszakat megadó adatok beolvasását. Ezeket az adatokat azután a legtöbb esetben derékszögű koordináta-rendszerbe transzformálják, majd egy frame típusú változónak való értékadással tárolják *(3.11. ábra)*.

Az AL nyelvben a robotberendezés mindenkori pozíció- és orientációadatait az ARM, ARM1, ... ARM7 nevű rendszerváltozóban tároljuk. A felhasználó így mindazokban az utasításokban vagy kifejezésekben megadhatja a robot állapotát leíró frame-rendszert, ahol a „szokványos" frame-ek használata egyáltalán megengedett. Ha a megadott robotkar nincs mozgásban, akkor az aktuális pozíció- és orientációadat már a robotkar helyzetét ábrázoló frame-változóban van, egyébként pedig sor kerül a robotkar koordinátaadatainak beolvasására, transzformálására és a robot helyzetét leíró frame-be való betöltésére.



3.11. ábra. A robot pillanatnyi pozíció- és orientáció-adatainak beolvasása



3.12. ábra. A robot pozíció- és orientációadatainak beolvasása, VAL-ban

AL:

```
(* ARM:robotkar *)
aktuális _ pozíció ← POS(ARM); (* lefoglalt változónév *)
AL:
IF POS(ARM).ZHAT > 50 THEN
  PRINT ("tul magas!");
```

A VAL nyelvben van egy speciális utasítás a robotkar pozíció- és orientációadatainak beolvasására. A *3.12. ábrán* alkalmazott HERE-utasítás hatására az adatok beolvasását követően a koordinátaadatok átadódnak a megfelelő frame-változónak.

VAL:

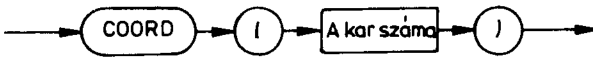
HERE célframe

Ezzel a pillanatnyi robotpozíció- és -orientációadatokat a **célframe** nevű változóba visszük be.

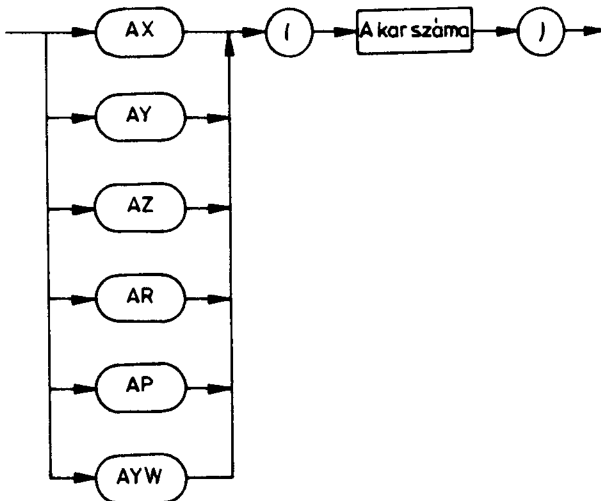
HERE#precizpoz

Ez az utasítás a robot helyzetét tartalmazó frame-et ún. precíz frame-ként tárolja le, azaz nem a derékszögű koordinátákat, hanem az ízületi (csukló-) szögek és eltolási úthosszak pillanatnyi értékeit őrzi meg.

A robot-frame beolvasását HELP nyelvben a COORD-utasítás kezdeményezi *(3.13. ábra).* A beolvasott pozíció- és orientációadatokhoz különféle aritmetikai függvények segítségével férhetünk hozzá. Ezek az AX, AY, AZ, AR, AP, AYW *(3.14. ábra).*



3.13. ábra. A robotberendezés frame-adatainak beolvasása HELP-ben



3.14. ábra. A beolvasott pozíció- és orientációadatok kezelése a HELP nyelvben

Jelentésük a következő:

AX X-koordinátatengely
AY Y-koordinátatengely
AZ Z-koordinátatengely
AR Forgatási (elcsavarodási) szög (Roll)
AP Hajlítási (bólintási) szög (Pitch)
AYW Tengely körüli fordulási (irányváltoztatási) szög (Yaw)

Miután a HELP nyelv nem kezel frame típusú változót, így a koordinátaadatokat közvetlenül – explicit módon – egy tömbbeli értékadással kell megadni.

HELP:

```
COORD (1);
framef (1) := AX (1);
framef (2) := AY (1);
framef (3) := AZ (1);
framef (4) := AR (1);
framef (5) := AP (1);
framef (6) := AYW (1);
```

Az 1. robotpozíció- és orientációadatainak beolvasását követően az adatokat beírjuk a **framef** tömbbe.

A SIGLA nyelv elmozdulásmérés nélküli, léptetőmotoros rendszerekre orientált nyelv. Emiatt a SIGLÁ-ban – valamint a ROBEX-ben, amely változókat egyáltalán nem kezel – nincsenek is meg azok az utasítások, amelyek a robot aktuális pozíció- és orientációadatainak beolvasását hajtanák végre.

3.3.3. Értékadás pointer típusú változónak

A *3.9. programrészletben* pointer típusú változókkal végeztünk műveletet, így többek között pointernek való értékadást is végrehajtottunk. Az ilyen típusú értékadás során átadott érték egy adatobjektumra való utalás. Az utasítás bal oldalán álló változóban korábban már meglevő utalás (azaz rámutatás, pointerezés) ennek során felülíródik. Emiatt egy adatobjektumra vonatkozó valamennyi utalás elveszhet, miáltal a memóriában olyan adatterületek jöhetnek létre, amelyeket többé nem tudunk elérni. Ezek ismételt hozzáférhetővé tétele a PASCAL-implementációknak csak egy részénél megoldott kérdés.

Az eddig tárgyalt értékadási típusokhoz hasonlóan a compiler-program a pointer típusú értékadásnál is ellenőrzi, hogy a jobb és a bal oldalon levő változók típusai megegyeznek-e. Ily módon csakis olyan értékadás megengedett, amelyeknél az érintett pointerek azonos típusú adatokra utalnak. Az egyetlen kivételt az üres pointer jelenti, ez a PASCAL-ban a NIL-nek nevezett pointer. A *3.14. programrészlet* ilyen esetet szemléltet. Itt felhasználjuk a *3.9. programrészletben* alkalmazott típusdeklarációkat:

Az első sorban **„szekrenytipus"** típusú adatobjektumot állítunk elő, a **szekreny** nevű változó pedig erre az objektumra mutató utalást kap értékként. A második sorban álló érvényes, végrehajtható utasítást követően azonban a harmadik sorban hibás utasítás áll, mivel az ott levő változók különböző típusúak, hiszen különböző típusú objektumokra mutatnak. A program utolsó sora arra ad példát, hogy egy adatobjektum elérhetetlenné válik, a memóriában azonban továbbra is létezik és tovább foglalja a helyet.

```
VAR szekrények, aktuális szekrény   : szekrénypointer;
    framepointer                    : framemutató;
BEGIN
  NEW (szekrenyek);                 (* Szekrenytipus típusu       *)
                                    (* adat-objektum előállítása  *)
  aktualisszekreny := NIL;          (* Érvényes értékadás         *)
  framemutató := szekrenyek;        (* Hibás értékadás            *)
  szekrények := NIL;                (* Többé nem elérhető         *)
END                                 (* az adatobjektumon          *)
```

3.14. programrészlet. Értékadás pointer típusú változónak (PASCAL)

3.4. Szövegek és számadatok kiviteli utasításai

A kiviteli utasítások a beviteli utasításokhoz hasonlóan a programnak a környezettel folytatott kommunikációját, kapcsolattartását szolgálják. Formái:

- a kezelői kommunikáció érdekében olvasható formában, képernyőn vagy nyomtatón megjelenő adatok (esetleg szövegfájlba való írás);
- adatkivitel külső tárolóegységre adatállományok tárolása céljából, fájlokba, nem közvetlenül olvasható formában (l. a 3.1.5.3. pontot);
- robotberendezések, ill. a műszaki környezet egyéb berendezései felé kiadott információk (l. a 4.2., 4.3., 4.4. szakaszt).

Az adatkiviteli utasítások első típusát részletesebben is tárgyaljuk. A második típus a tárgyalt robotnyelvekben legfeljebb problémaorientált esetekben juthat szerephez, ugyanakkor programtechnikai segédeszközként a programozó ezt általában nem tudja úgy felhasználni, mint a PASCAL-ban. Példa erre az AL nyelvbeli frame-fájlja. A kérdésre az 5. fejezetben még visszatérünk.

A ROBEX nyelvet kivéve a program futása közben kiírathatunk szöveg-, vagy számadatokat. Az AL-ban és a VAL-ban ezenkívül azzal a lehetőséggel is élhetünk, hogy a szöveg kiíratásakor a programot megállíthatjuk és a várakozást valamilyen speciális adatbeviteltől tesszük függővé. Részleteiben a következő utasítások állnak rendelkezésünkre:

AL:

PRINT (⟨paraméterlista⟩)
PROMPT (⟨paraméterlista⟩)

A ⟨**paraméterlista**2 tartalmazhat skalár változót, skalár kifejezéseket vagy jelsorozatokat. Ez utóbbiakat idézőjelek ("-jelek) közé tesszük.

A program a PRINT-utasítás végrehajtását követően folytatódik, a PROMPT-utasítás hatására ezzel szemben a paraméterlista kiíratása után megjelenik a „Type P to proceed" (üsse le a P billentyűt, ha folytatni kívánja) üzenet. A program ezután mindaddig várakozik, míg a kezelő le nem üti a p betűt.

VAL:

TYPE [⟨text⟩]
TYPEI ⟨intvar⟩
PAUSE [⟨text⟩]

Itt a TYPE utasítás a **Pause** utasításhoz hasonlóan kiíratja a ⟨text⟩ szöveget, azonban a **Pause** megszakítja a programot, és az csak PROCEED-del folytatható. A TYPEI-utasításnak van egy különleges tulajdonsága: egy integer típusú változó értékével együtt kiírja a változó nevét is.

HELP:

PRINT(⟨paraméterlista⟩)

Ez az utasítás megegyezik az AL PRINT-utasításával azzal a különbséggel, hogy HELP-ben nincs megengedve skaláris kifejezések használata, csupán változóké, a karaktersorozatot pedig idézőjelek helyett '-jelek (aposztróf jelek) közé zárjuk. Lehetőség van további speciális nyomtatási utasítások konstruálására, így soremelésre, akusztikai jelzések kiadására stb.

SIGLA:

NT/⟨text⟩, ⟨szamlalo⟩

A **⟨text⟩** szöveg nyolc karakterre korlátozódik, a **⟨szamlalo⟩** pedig megmutatja azt a memóriaszámlálót, amelynek tartalmát a gép kinyomtatja. A PASCAL nyelv általánosabb WRITE-utasításáról a 3.3.2.1. pontban már volt szó. (A 3.2.3. pont példájának megoldása: TRUE.)

4. Utasítások

A robotvezérlő nyelveknél utasításokon olyan műveleteket értünk, amelyeket a program adataival, vagy ipari robotberendezésekkel és végrehajtószervekkel hajtunk végre. Alapesetben az utasításokat az utasítás kiadásának, tehát leírásának sorrendjében kell végrehajtani. Kivételt képeznek az ugró utasítások, valamint olyan speciális utasítások, amelyek hatására ún. párhuzamos programfeldolgozás jön létre.

Azokat az utasításokat, amelyek a „közönséges" programnyelvekben is megvannak a 3.3., ill. a 3.4. szakaszokban már áttekintettük (értékadó utasítások, beviteli/kiviteli utasítások), éspedig elsősorban adatok kezelésével kapcsolatban. E fejezeten belül a 4.7. szakaszban bemutatjuk a program végrehajtását vezérlő általános utasításokat. Ezektől eltérő, új problémakör speciálisan a robotok programozásához szükséges utasítástípusok rendszeres és részletes tárgyalása.

Ilyen utasítások a környezeti modell leírásához szükséges utasítások, a működtető (pozicionáló), végrehajtó és érzékelő utasítások, valamint rendszerkapcsolók beállítására és visszaállítására vonatkozó utasítások és a kivételes helyzetek kezelésére szolgáló utasítások.

Az előző fejezethez hasonlóan számos programrészleten keresztül mutatjuk be a különböző nyelvek legkülönfélébb problémamegoldási és ábrázolási lehetőségeit. A szintaktikai szerkezetet a 2.3. szakaszban bemutatott és a 2.3.3. pontban megadott szintakszisdiagramok felhasználásával kívánjuk bemutatni. Az eddig tárgyalt nyelveken (AL, VAL, HELP, SIGLA és ROBEX) kívül a 4.5.4. pontban a vizuális (megjelenítő) rendszerekre alkalmas RAIL nyelv utasításait is tárgyaljuk, mivel ez igen alapos, és jól használható támpontot nyújt az érzékelők adatait kezelő utasításkészlet megértéséhez, ami vizuális rendszerekkel megvalósított kommunikáció esetén nélkülözhetetlen.

A 4.9. szakaszban a kivételes üzemállapotok kezelésére a PEARL nyelv megoldásait mutatjuk be, mivel a jelenleg ismert robotvezérlő nyelvek egyikében sincsenek meg ezek a lehetőségek.

4.1. A környezeti modellt kezelő utasítások

A 2.4.8. pont alatt leírt – off-line módon létrehozott és kezelt – környezeti modell típuson kívül, amely a compiler, ill. a processzor szempontjából fontos, olyan on-line típusú környezetleíró modell használatára is szükség van, amely az alkalmazói program futása során dinamikusan jön létre, és a robot munkaterében bekövetkező tényleges változásokhoz is képes illeszkedni.

Különböző operációk a környezeti modellen az érzékelők jeleinek felhasználásával, vagy pedig az alkalmazói program speciális utasításainak segítségével automatikusan végrehajthatók. Meg kell azonban jegyezni, hogy az első eset – amikor a környezeti modell az érzékelők jeleinek segítségével folyamatosan a pillanatnyi állapotnak megfelelően aktualizálódik – a technika jelenlegi fejlettségi fokán még nem tekinthető megoldott-

nak. Így csak azt a kérdést tárgyaljuk, hogy hogyan lehet programbeli utasítások segítségével explicit változtatásokat végrehajtani a munkatér belső ábrázolású modelljén.

A tárgyalt robotvezérlő nyelvek közül csak az AL nyelv rendelkezik on-line-környezetleíró modell kezelésére alkalmas eszközökkel (l. pl. PAUL [4.13]). A ROBEX-ben a geometriai modell csak off-line módon kezelhető (l. a 3.1.4. fejezetet).

A 3.1.6. pontban a pointer típusú változók használatával kapcsolatban bemutattunk egy példát arra vonatkozóan, hogy hogyan építhető fel a környezeti modellt meghatározó adatstruktúra rekordok segítségével a PASCAL nyelvben. Ezért minden további bevezetés nélkül PASCAL-ban is felírhatjuk az AL nyelvből már ismert bővített frame-rendszereket, majd bemutatjuk a megfelelő AL-utasításokat. A *4.1. programrészlet* a kibővített frame-rendszer felhasználásával bemutatja a geometriai adattípusok deklarálását, majd példát mutat a környezeti modellen végrehajtható műveletekre.

A 3.1.4. pontban bemutatott *3.4. programrészlet*tel összehasonlítva megállapíthatjuk, hogy az itt látható frame-rendszer rendelkezik egy, a frame-listára mutató **next** nevű pointerrel, valamint az objektum megközelítését és elhagyását meghatározó frame-rendszerekre mutató **approach** és **departure** nevű pointerekkel (4.2.4. pont), valamint egy olyan résszel, amely arra szolgál, hogy a frame kiegészíthető legyen az ún. affix hozzárendeléssel. AL nyelvben affix kiegészítésen egy frame-rendszernek egy másik frame-hez való hozzáfűzését, hozzáragasztását értjük, amely a két frame-rendszer egymáshoz viszonyított térbeli viszonyait fejezi ki. Két frame-rendszer térbeli relációja egy transzlációból (eltolásból) és egy rotációból (forgatásból) áll, amelyek megmutatják, hogy milyen transzformáció viszi át az egyik frame-rendszert a másikba (2.4.1. pont). Ha tehát egy A frame-rendszerhez hozzáfűzünk egy B frame-et, akkor a B értékeinek megváltozásakor automatikusan megváltoznak az A frame-rendszer adatai is, éspedig oly módon, hogy a hozzáfűzéskor megadott reláció állandó marad. Ebben a példában az affix kiegészítést az egyszerűség kedvéért csak a hozzáfűzendő frame-rendszer **affixframe** nevű pointerének segítségével végezzük el, majd a frame-rendszerek relációját az **affixrel** nevű transzformáció (trans) típusú változóban rögzítjük. Az affix kiegészítés tényének későbbi lekérdezése érdekében az **affixedon** nevű logikai változó jelzi ezt a kiegészítést. Az **újframe** nevű eljárás felállít egy frame-et, majd üres hivatkozásokkal előre lefoglalja a szükséges helyeket. Ha ezután egy frame-rendszert hozzá kívánunk fűzni egy másikhoz, akkor hívhatjuk az **affix** nevű eljárást, melynek paraméterei:

1. a hozzáfűzendő frame-rendszer;
2. az a frame-rendszer, amelyhez hozzá kívánjuk fűzni a másikat;
3. egy transzformáció, amelyben a frame-rendszerek közt fennálló relációt leírjuk.

Az affix (lyuk, szekrény, dummytrans) eljáráshívó utasítás hatására beállítódik a **lyuk** nevű frame-rendszer **affixedon** változója, az **affixframe** nevű pointer rámutat a **szekrény** frame-re, majd kiszámítódik a **szekrény** nevű frame-nek a **lyuk** frame-be történő transzformálását leíró frame-reláció értéke. Az egyszerűség kedvéért csak a transzlációs elmozdulást számítottuk ki. A transzláció értékét, valamint a „zérusrotáció" értékét bejegyeztük a **lyuk** nevű frame **affixrel** nevű transzformáció típusú változójába, éspedig a következő értékekkel:

affixrel.transzláció.x = 45 − 55 = −10
affixrel.transzláció.y = 30 − 30 = 0
affixrel.transzláció.z = 28 − 22 = 6

Feltéve, hogy a szekrény mozgatásakor a **szekrény** nevű frame új értékként x = 25, y = 35 és z = 22 értékeket vesz fel, a programvezérlő rendszer az **affixrel**-nek megfelelően automatikusan kiszámítja a **lyuk** nevű frame új értékeit is. Így

lyuk.poz.x = szekrény.poz.x + lyuk.affixrel.transzláció.x
= 25 − 10 = 15
lyuk.poz.y = szekrény.poz.y + lyuk.affixrel.transzláció.y
= 35 + 0 = 35
lyuk.poz.z = szekrény.poz.z + lyuk.affixrel.transzláció.z
= 22 + 6 = 28

A **lyuk** nevű frame pozíció-adatai tehát x = 15, y = 35 és z = 28. A frame-rendszerek közötti reláció a második eljáráshívásban, az

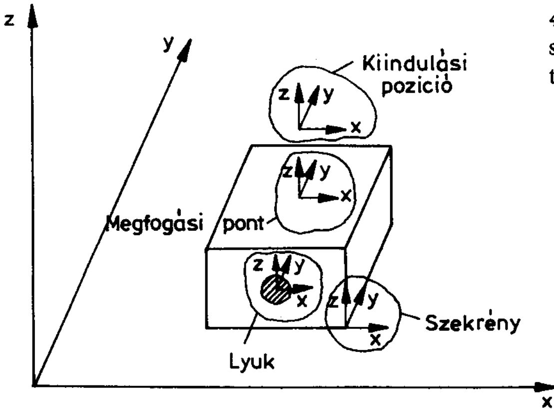
affix (megfogpont, szekreny, affixvaltozo);

utasítással aktivált eljárásban átadódik az **affixvaltozo** nevű transzformáció típusú változónak, miáltal az egyszerűbben lekérdezhetővé válik. Az **unfix** nevű eljárás a frame-rendszerek közötti reláció felbontását végzi, melyben az **affixedon** FALSE értéket, az **affixframe** pedig NIL értéket kap. A *4.1. ábra* a környezeti modellnek a *4.1. programrészletben* előállított frame-ek közötti relációját szemlélteti. Ennek során, mint már említettük, a rotációkomponenst elhagytuk, mivel ez jelentősen növelné a program terjedelmét, áttekinthetőségét pedig rontaná.

```
TYPE
  vector              = RECORD
                          x, y, z: REAL
                          END;
  rotmatrix           = ARRAY [1..3, 1..3] OF REAL;
  rot                 = RECORD
                            tengely     : vector;
                            szog        : REAL;
                            matrix      : rotmatrix
                          END;
  trans               = RECORD
                            transzláció    : vector;
                            rotació        : rot
  END;
  framepointer        = ∧ frame;
  frame               = RECORD
                            next             : framepointer;
                            orient           : rot;
                            poz              : vector;
                            approach,
                            departure        : framepointer;
                            affixedon        : BOOLEAN;
                            affixframe       : framepointer;
                            affixrel         : trans
                          END;
VAR
  nilvector          : vector;
  nilrot             : rot;
  affixvaltozo : dummytrans : trans;
  szekrény, lyuk, megfogpont, kiindpoz : framepointer;
PROCEDURE ujframe (VAR framevaltozo: framepointer;
                           xf, yf, zf : REAL);
BEGIN (* ujframe *)
  NEW (framevaltozó);
  WITH framevaltozo∧ DO
  BEGIN
```

```
      next              := NIL;
      orient            := nilrot;            (* Egyszerűség kedvéért                    *)
      poz. x            := xf;
      poz. y            := yf;
      poz. z            := zf;
      approach          := NIL;
      departure         := NIL;
      affixedon         := FALSE;
    END;
  END (* ujframe *);
  PROCEDURE affix (VAR anframe:                 framepointer;
                           celframe           : framepointer;
                       VAR affixtrans         : trans );
  BEGIN (* affix *)
    WITH anframe∧ DO
    BEGIN
      affixedon             := TRUE;
      affixframe            := celframe;
      affixrel, transzláció.x                  := poz.x - célframe∧ .poz.x;
      affixtrans.transzláció.x                 := affixrel.transzláció.x;
      affixrel.transzláció.y                   := poz.y - célframe∧. poz.y;
      affixtrans.transzlació.z                 : affixrel.transzlacio.z;
      affixrel.rotacio                         := nilrot; (* Egyszerűség            *)
      affixtrans.rotacio                       := nilrot; (* kedvéért               *)
    END;
  END (* affix *);
    PROCEDURE unfix (VAR anframe               : framepointer ;
                         celframe              : framepointer);
  BEGIN (* unfix *)
    anframe∧ . affixedon      := FALSE;
    anframe∧ . affixframe     := NIL;
  END (* unfix *);
  BEGIN (* Főprogram *)
    WITH nilvector DO
    BEGIN
      x := 0;
      y := 0;
      z := 0;
    END;
    nilrot.tengely            := nilvector;
    nilrot.szog               := 0;
       :
    ujframe (szekreny ,              55, 30, 22 );
    ujframe (lyuk,                   45, 30, 28 );
    ujframe (megfogpont,             45, 50, 34 );
    ujframe (kiindpoz,               45, 50, 40 );
       :
    affix (lyuk,               szekreny, dummytrans);
    affix (megfogpont,         szekreny, affixvaltozo);
    affix (kiindpoz,           megfogpont, dummytrans);
       :
    unfix (megfogpont, szekreny);
  END (* Főprogram *).
```

4.1. programrészlet. A környezeti modell definiálása és a modellel végzett műveletek PASCAL nyelven írt program segítségével



4.1. ábra. A környezeti modell frame-rendszerei közti relációk geometriai szemléltetése

A *4.2. programrészlet* ugyanezt a feladatot oldja meg AL nyelven. Ez természetesen sokkal rövidebb, hiszen elmaradnak a VECTOR, ROT, FRAME és TRANS geometriai adattípusok deklarációi, valamint az **ujframe,** az **affix** és az **unfix** eljárások deklarációi. Az AL ezeket az adattípusokat standard változótípusként képes kezelni, az eljárásdeklarációkat pedig a frame típusú változónak történő közvetlen értékadással, valamint az AFFIX és az UNFIX-utasításokkal oldja meg. A *4.2. ábra* a környezeti modellre vonatkozó AL-utasítások szintaktikai struktúráját szemlélteti.

Az AFFIX lyuk TO szekrény utasítás – ugyanúgy, mint a PASCAL nyelven írt példában – azt jelenti, hogy egy pointer a **lyuk** nevű frame-ről a **szekrény** nevű frame-re fog mutatni és a két frame egymás közti relációját megadó transzformációs mátrix kiszámításra kerül. A PASCAL-ban írt programrészlettől eltérően azonban itt a programozónak nincs közvetlen, explicit hozzáférési lehetősége a pointerhez, ill. a frame-ek relációját leíró transzformációs adatokhoz. Ez utóbbi úgy lehetséges, hogy a BY alapszót követően megadunk egy transzformáció típusú változót. Például a következőképpen:

AFFIX megfogpoz TO szekrény BY affixtrans;

Az utasítást követően az **affixtrans** vektoros komponensének felhasználásával már hozzáférhetővé válnak az x = – 10, = 20, z = 12 transzformációs adatok.

A PASCAL-példában az affix-kiegészítés mindkét frame-rendszerét előzetesen definiálnunk kellett. Erre az AL-ban nincs szükség. Az AT alapszó hatására a hozzáfűzött frame-rendszert az elsődleges frame-hez viszonyítva számítja ki a számítógép a megadott

AFFIX kiindpoz TO megfogpont AT TRANS(NILROT, VECTOR (0,0, 6) * CM);

transzformációs kifejezés segítségével.

A felírt utasítás utána **kiindpoz** vektorváltozó komponensei az x = 45, y = 50, z = 40 aktuális értékeket veszik fel a **megfogpoz** aktuális értékének és az x = 0, y = 0, z = 6 transzformációs vektornak megfelelően. Az UNFIX utasítás ismét feloldja az affix kiegészítést. A *4.3. ábra* szemlélteti az affix hozzárendeléseknek az

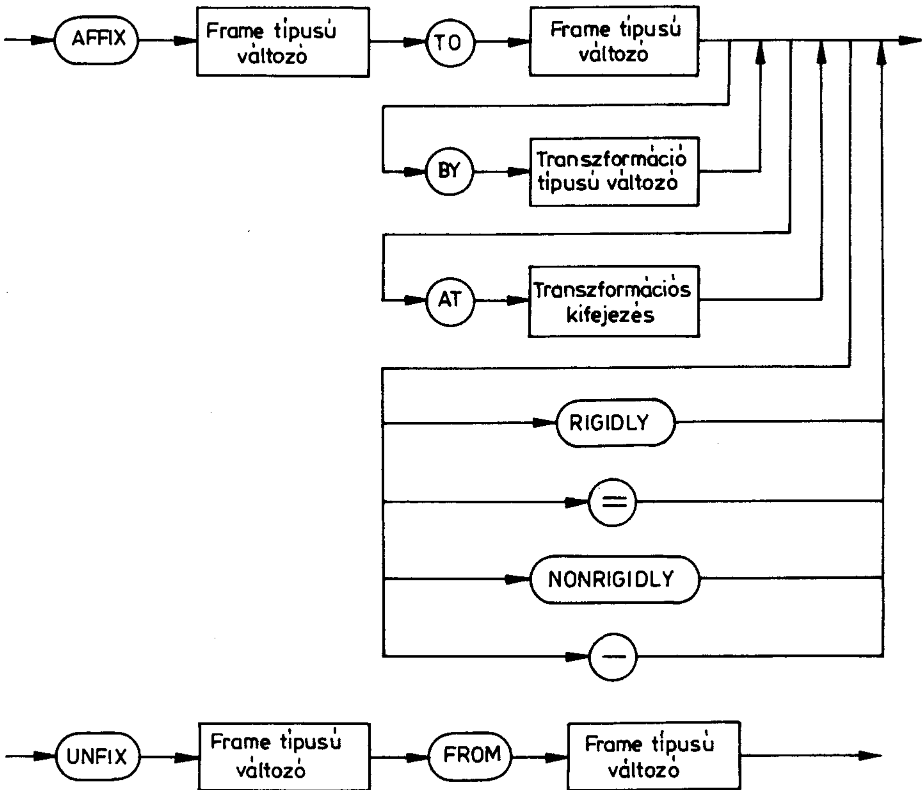
UNFIX megfogpoz FROM szekreny;

utasítás előtti és utáni struktúráját.

Az AL nyelvben valamely kísérőframe-rendszert nemcsak más frame-rendszerekhez, hanem magához a robot, vagy a robotkar frame-rendszeréhez is hozzáfűzhetjük, vagyis ezekkel affix-relációba hozhatjuk. Ennek az a következménye, hogy a robot minden mozdulatánál nemcsak a robotkar frame-rendszere változik, hanem a frame-rendszerek közt rögzített relációknak megfelelően a hozzáfűzött frame-rendszer is. Ennek akkor van

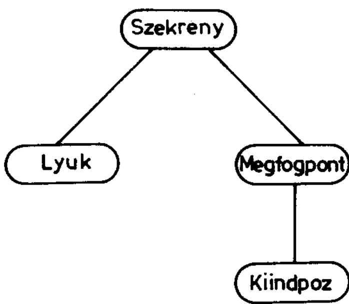
```
FRAME szekreny, lyuk, megfogpont, kiindpoz;
TRANS affixtrans;
szekreny←FRAME (NILROT, VECTOR (55, 30, 22) * CM);
lyuk←FRAME (NILROT, VECTOR (45, 30, 28) * CM);
megfogpont←FRAME (NILROT, VECTOR (45, 50, 34) * CM);
    :
AFFIX lyuk TO szekreny;
AFFIX megfogpont TO szekreny BY affixtrans;
AFFIX kiindpoz TO megfogpont AT TRANS (NILROT, VECTOR (0,0,6) * CM);
    :
UNFIX megfogp FROM szekreny;
```

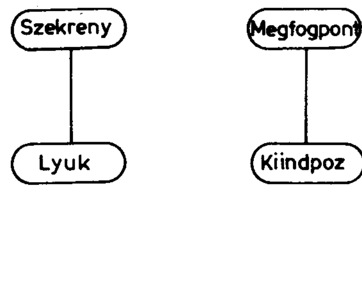
4.2. programrészlet. A környezeti modellre vonatkozó utasítások AL-ben



4.2. ábra. A környezeti modellre vonatkozó AFFIX- és UNFIX-utasítások AL-ban

értelme, amikor a robotkéz megfog egy alkatrészt, majd valahová átrakja. Ekkor a program a megfogott alkatrész megfogási pontjához tartozó frame-rendszert módosítja. Így az alkatrész ismételt megfogásakor a robot megtalálja az alkatrészt. A programozónak ugyanakkor arról sem szabad megfeledkeznie, hogy az alkatrész lerakásakor UNFIX-utasítással ismét fel kell oldania az affix kapcsolatot, mert különben a robot minden további mozdulatánál az alkatrész kísérőframe-rendszere is megváltozik, annak ellenére, hogy a valóságban a gép már letette az alkatrészt a munkapadra.





4.3. ábra. Az AL nyelvben felírt programpéldában használt affix hozzárendelés struktúrája

---

AL:

```
MOVE ARM TO doboz;               (* ARM: robotkar *)
CLOSE HAND TO 5 * CM;            (* HAND: robotkéz *)
AFFIX doboz TO HAND RIGIDLY;     (* lefoglalt változónevek *)
MOVE doboz TO lerakohely;
OPEN HAND TO 8 * CM;
UNFIX doboz FROM ARM;
MOVE ARM TO tovább;
:
```

Miután a robotkar elérte a **dobozt,** vagyis a frame-rendszer által meghatározott helyzetet, a kéz pedig megfogja az alkatrészt, a **doboz** nevű változót hozzáfűzzük a kar frame-rendszeréhez, vagyis affix-relációba hozzuk vele. Miután a **doboz** a **lerakóhely**-re kerül és ott lerakjuk, az UNFIX utasítás ismét feloldja az affix relációt. A **doboz** nevű frame-ben most már a doboznak a továbbítást követő új megfogási koordinátaadatai vannak.

A RIGIDLY alapszóval (esetenként az "=" jellel) azt állítjuk be, hogy az affix-relációba hozást kölcsönösen végrehajtjuk, azaz az egyik frame-rendszer megváltozása automatikusan maga után vonja a másik frame megváltozását és fordítva. A NONRIGIDLY alapszó megadásával (esetenként a "–" jellel) azt írjuk elő, hogy csakis a hozzáfűzött frame-rendszer kövesse a fölérendelt frame-rendszer változásait, megfordítva azonban ne.

A fenti példa középső pozicionáló utasításában a robotkar helyett a robotkarral együtt mozgó **doboz** frame-rendszerét adtuk meg. (Azt mondhatjuk, hogy az ilyen frame elmozdítható – más néven együtt mozgó –, mert a kar frame-rendszeréhez csatlakozik, azzal együtt mozog.) A pozicionáló utasítások szintaktikájával kapcsolatban l. a *4.14., 4.17.* és *4.19. ábrákat.*

4.2. Mozgásvezérlő utasítások

A mozgásvezérlő utasítások azok az utasítások – még ha a különböző nyelvek eltérő felfogást tükröznek is –, amelyek a robotvezérlő nyelvek igazi arculatát meghatározzák. Ezt az utasítástípust valamennyi általunk ismert robotvezérlő nyelvben megtalálhatjuk (l. BLUME [2.5]), sőt az utasítás alapszava is csaknem kizárólag a MOVE alapszó.

A programozó a mozgásvezérlő utasítások segítségével irányítja a robottal végzett műveleteket. A MOVE-utasítások nagyvonalakban explicit és implicit mozgásvezérlő utasításokra oszthatók fel. Explicit utasítással a programozó meghatározza a mozgás pozíciójának koordinátaadatait, valamint a megfogóeszköz vagy a szerszám orientációadatait. (Ez többnyire a frame-rendszer segítségével történik.) A frame-et nem szükséges a számadatok konkrét megadásával definiálnunk, e definíció teach-in (ún. betanítási) eljárással is történhet (5. fejezet). Az implicit mozgásvezérlő utasításnak ezzel szemben az a feltétele, hogy a programrendszer a rendelkezésre álló adatbázis alapján önállóan kiszámíthassa a mozgás végrehajtásához szükséges koordinátaértékeket.

A program futása során a mozgásvezérlő utasítások végrehajtásáról a robotvezérlés gondoskodik. Attól függően, hogy a vezérlés a 2.4.3. pontban tárgyalt vezérlési módok melyikét tudja megvalósítani, a végrehajtószerv pályája lehet pl. egyenes (lineáris *Descartes*-féle interpoláció esetén), vagy köríves szakaszokból álló (lineáris tengelyinterpoláció esetén), sőt függhet a robot kinematikai viszonyaitól is. Ilyenkor aligha láthatók előre a részletek (ponttól pontig vezérlés). A robotvezérlő rendszer tulajdonságai miatt egy egyszerű

⟨vezesd a robotkart x pontba⟩

alakú mozgásvezérlő (pozicionálási) utasítás hatása az alkalmazott programvezérlő rendszertől függ. Ugyanis ma még alig van olyan nyelv, amelyben a vezérlés módját a programozó írhatná elő. Ennek az az oka, hogy a legtöbb nyelv lényegében egy-egy adott vezérlési rendszerre épülve fejlődött ki, más rendszerekre való átültetésüket nem is tervezték. Emiatt az egyes nyelvekben implicit módon már eleve meg van határozva a pályavezérlési mód és magának a pályának a felépítése. Ezért ez a programozás oldaláról nem is specifikálható. Még a ROBEX-ben sincs olyan utasítás, amely a vezérlés módját adná meg, jóllehet ezt a nyelvet a vezérlőrendszerektől független nyelvnek szánták, bár programvezérlő rendszerre még sehol sem implementálták.

4.2.1. Implicit mozgásvezérlő utasítások

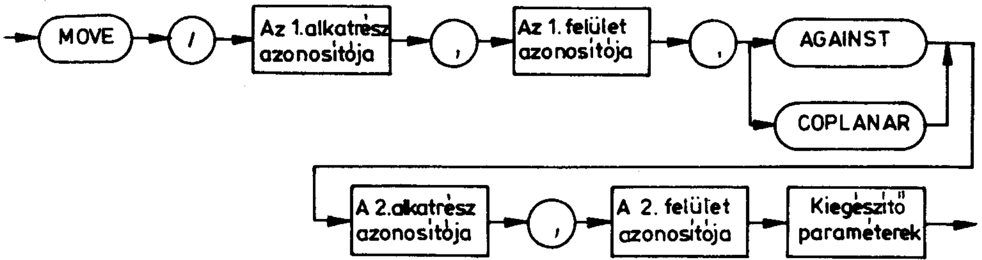
A tárgyalt nyelvek közül egyedül a ROBEX rendelkezik implicit mozgásvezérlő utasítással, éspedig a meglevő explicit utasítástípus mellett, azt kiegészítve (l. a 4.2.3.). Implicitnek nevezzük az utasítást amiatt, mert nem kell közvetlenül megadni a frame-rendszer koordinátaértékeit. A programozó csak az alkatrészek geometriai leírását, valamint az alkatrészek pályáját definiáló geometriai relációkat adja meg. Például ilyeneket: „1. alkatrészt a 2. alkatrésszel párhuzamosan". A ROBEX implicit mozgásvezérlő utasításának struktúráját a *4.4. ábra* szemlélteti (l. még AMBLER [4.2]).

Az alkatrészeket és felületeket explicit koordinátaadatok segítségével előzetesen deklarálni kell (3.1.4. pont). A compiler-program a kezelt alkatrész, a robot, valamint a munkaterület geometriai adatainak segítségével meghatározza az alkatrész megfogási pontját és kiszámítja a mozgás pályáját. Ezzel előállít egy végrehajtható programot, és az így generált kód már az ütközések elkerüléséhez szükséges kitérések figyelembevételével összeállított explicit működtető utasításokat tartalmazza. A kiegészítő paraméterek azokat az adatokat tartalmazzák, amelyek a cél frame-rendszerére történő exakt pozicionáláshoz és a külső jelek figyelembevételéhez szükségesek.

Példa ROBEX nyelven:

MOVE/box, alsó, AGAINST,szerkezet,teteje,EVENT,2,ELSE,üzenet

Ennek az utasításnak a hatására a compiler-program úgy állítja elő a szükséges mozgásvezérlő és megfogási utasításokat, hogy a robot a boxot alsó lapjával helyezze a szerkezetre. Ha a művelet alatt a 2. csatornáról nem fut be jelzés, akkor a program az „üzenet" címkére ugrik. (A 2. jelzést kiválthatja pl. egy erőmérő cella a doboz lerakásakor.)



4.4. ábra. Egy implicit mozgásvezérlő utasítás szerkezete ROBEX-ben

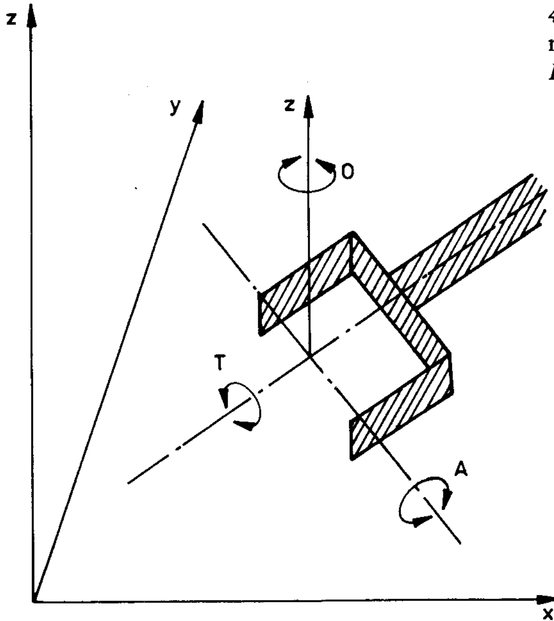
4.2.2. Explicit mozgásvezérlő utasítások

Ha a pálya pozíció- és orientációadatainak koordinátáit explicit módon kívánjuk megadni, azt megtehetjük szövegszerűen leírt adatokkal, vagy egy betanítási (teach-in) eljárás segítségével (5. fejezet). A művelet szövegszerű definiálásakor különbözőképpen járunk el aszerint, hogy milyen koordináta-rendszert vettünk alapul és aszerint, hogy a műveletet az érzékelők aadatainak figyelembevételével tehát szabályozottan, vagy anélkül hajtjuk-e végre. A programozónak azonban mindegyik esetben pontosan ismernie kell a kezelt objektumok pozícióját, és ehhez a helyzethez a robotkéz, vagy a szerszám alkalmas orientációját. Tisztában kell lennie ezenkívül az egyes objektumok geometriai viszonyaival, hogy a beprogramozott útvonal mentén ne fordulhasson elő ütközés, és ugyanakkor a vezérlőrendszer képes legyen meghatározni az útvonal megtételéhez szükséges lépéseket.

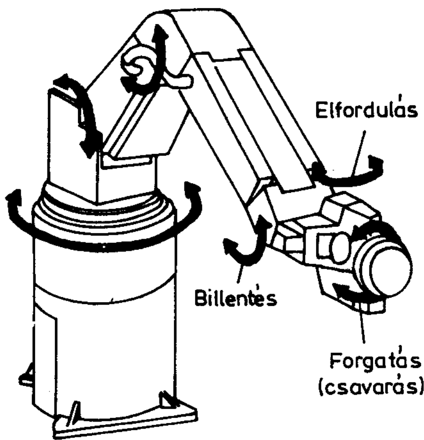
Az explicit koordinátaadatok többnyire a *Descartes*-féle koordinátákra vonatkoznak. Emellett használatosak még a hengerkoordináták, polárkoordináták, ill. az ízületi koordináták (csuklókoordináták, csuklószögek). *Descartes*-koordináták használatakor csaknem kizárólag a frame-rendszerrel dolgozunk (vö. a 2.4.1. ponttal). Ilyenkor az orientációadatot vagy a koordinátatengelyek körüli elfordulás szögével fejezzük ki, mint pl. az AL-ban, vagy az *Euler*-féle szögekkel (mint pl. a VAL nyelvben). Használatosak még a repüléstechnika és a hajózás területéről kölcsönzött „csavarás" (ném: Rollen, ang: Roll), „bólintás" (ném: Nicken, ang: Pitch) és „fordulás" (ném: Gieren, ang: Yaw) fogalmak is (pl. a HELP-ben) *(4.6. ábra)*. (Ezzel a kérdéssel foglalkozik még a VDI egy szabványtervezete, a VDI-Gründruck 2864.)

A VAL nyelvben használt *Euler*-féle szögeket, az O, A és T szögeket – formailag úgy definiáljuk, hogy azok rendre megfelelnek a *Descartes*-féle koordináta-rendszer z tengelye körüli elforgatásnak, az új y tengely körüli elforgatásnak és végül az így előállított új z tengely körüli forgatásnak. Ez megegyezik a megfogószerszám tengelyének a z koordináta körüli elforgatásával, a kézkoordináta-rendszernek a két megfogási ponton áthaladó y tengelye körüli forgatásával és végül magának a megfogóeszköznek saját tengelye körüli elforgatásával, amely egyben a kézkoordináta-rendszer z koordinátatengelyét is jelenti *(4.5. ábra)*.

A csavarás (a repüléstechnikában csűrés) megfelel annak a mozdulatnak, ahogyan az emberi kéz a csavarhúzót forgatja. A bólintás a csukló hajlításával, a fordulás pedig oldalirányú kézlendítéssel szemléltethető. A *4.6. ábra* ezeket a mozdulatokat mutatja be a Cincinnati Milacron egyik robotberendezésén.



4.5. ábra. Az orientáció megadása a VAL nyelvben az O, A és T szögekkel, az ún. *Euler*-féle szögekkel



4.6. ábra. Az orientáció meghatározása a forgatás (csavarás), a billentés és az elfordulás szögeinek segítségével

4.2.3. Egyszerű mozgásvezérlő utasítások

Egyszerű mozgásvezérlő utasításon egy olyan működtető parancs kiadását értjük, amelyben meghatározzuk a mozgás célpontját anélkül, hogy a közbülső pontokat, sebességeket vagy a mozgás időtartamát előírnánk. Természetesen végrehajtható egy egyszerű mozgásvezérlő utasítás az érzékelők jeleinek figyelembevételével, eseményfigyeléssel esetleg időzítésfelügyelettel is, ezeket a lehetőségeket azonban a 4.2.5.–4.2.7. pontokban tárgyaljuk. Ha a programozó a célpont pozícióját a motor azon **lépéseinek** számával adja meg, amennyit a robot egyes tengelyeinek el kell fordulniuk az adott pont eléréséhez, akkor a robot vezérlésének „paraméterezéséről" beszélünk. A SIGLA nyelvben pl.

MO/1,1317,3,M2,P1,158

azt jelenti, hogy az 1. motort a nullpozíciótól számított 1317. lépésre levő abszolút pozícióba léptetjük, míg a 3. motort az M2 számlálóváltozó értékének megfelelően léptetjük, a P1 számmal jelzett motort pedig az 518-as léptetési pozícióba vezéreljük. A *4.7. ábra* bemutatja a SIGLA mozgásvezérlő utasítását.

Lehetőség van arra is, hogy a léptetések számát a robot pillanatnyi pozíciójához viszonyítva relatív módon adjuk meg. Ezt a SIGLA-ban egy előzetesen kiadott II-utasítással jelezzük, tehát pl. a

```
II
MO/1,357,2,-182
```

utasítás az 1. motort 357 lépéssel pozitív irányba lépteti, a 2. motort pedig 182 lépéssel negatív irányba. Bár a SIGLA nyelvben az egyes motorok mindig valamelyik *Descartes*-féle koordinátatengelyhez vannak hozzárendelve (pl. az 1. motor az x tengely pozitív irányához), a pozíció pontos beállítása a léptetőmotor lépéseinek száma alapján mégis eléggé bizonytalan megoldásnak tűnik.

Az AL, a ROBEX és a VAL nyelvek ismerik az ún. DRIVE-utasítást, amely igen jó robotorientált programozási segédeszköz. Ennek segítségével az egyes tengelyek úgy működtethetők, hogy a forgómozgásra képes csuklók a megadott szögértékkel, a transzlációs mozgást végző csuklók pedig a megadott úthosszal mozdulnak el. Az AL-nyelvből és a ROBEX-ből vett következő utasítások a második robotízület elforgatását hajtják végre úgy, hogy a csukló véghelyzetében a 110°-os pozícióra áll rá.

AL:

```
DRIVE JOINT (2) OF ARM TO 110;
```

ROBEX:

```
DRIVE/2,110
```

A VAL nyelvben csak relatív rotációadat (tehát pl. 20°) és egy százalékosan felírható sebességadat megadására van mód (l. 4.10., 4.11. és 4.12. ábrák).

AL:

```
DRIVE JOINT (2) OF ARM1 BY 20;
```

VAL:

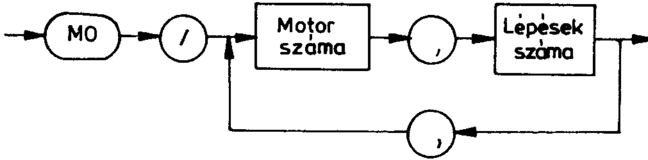
```
DRIVE 2,20,100
```

ROBEX:

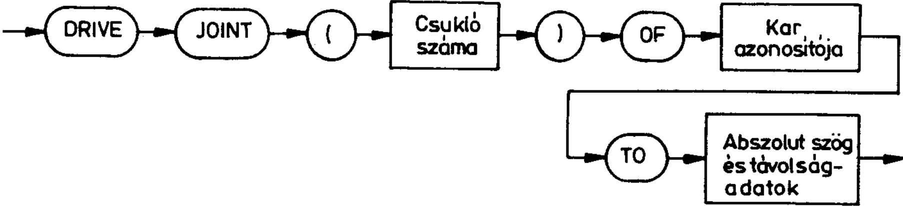
```
DRIDLT/2,20
```

A HELP-ben nincs mód arra, hogy a robot egyes tengelyeit közvetlen vezérlő utasítással akár abszolút, akár relatív értékadással működtessük. Egyes nyelvek lehetővé teszik, hogy a mozgásvezérlési utasításban robotkoordinátákban fejezhessük ki a célpont adatait. A VAL-ban ez az ún. *precíz* pont[4.1.]. A precíz pontot az adat elé kitett # jel jelöli. Az ilyen pont definiálása betanítási eljárással, vagy explicit módon történik (l. a 4.2.2. pontot). A programozó a robotot a kívánt pozícióval és orientációval jellemezhető helyzetbe állítja. Ezután a precíz pont rendszerparanccsal definiálható, majd ennek szög- és távolságadatait egy szimbolikus név (azonosító) alatt a frame-listában tároljuk le. Ez a precíz pont ezután egyszerű MOVE-utasítással már programból is beállítható.

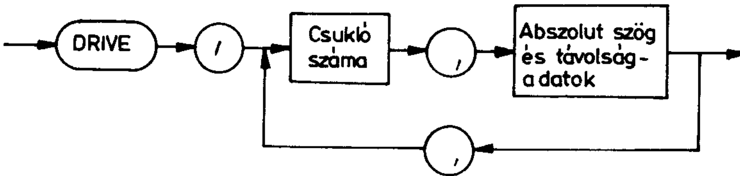
4.1. Az ún. *precíz* pont tulajdonképpen egyfajta referenciapont.



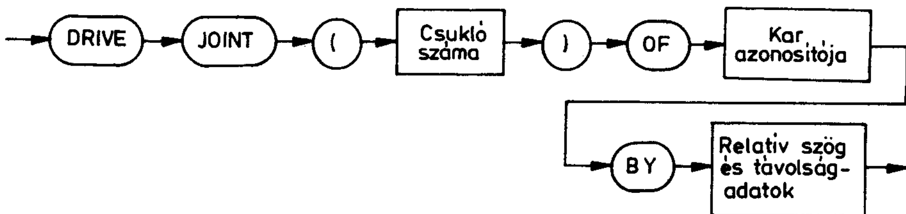
4.7. ábra. Egyszerű mozgásvezérlő utasítás SIGLA-ban



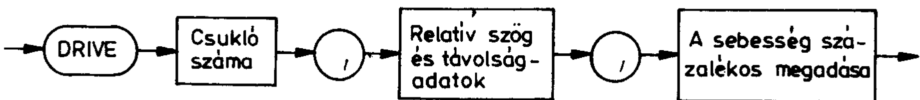
4.8. ábra. Az AL nyelv DRIVE-utasítása (abszolút szög- és távolságadatok használatakor)



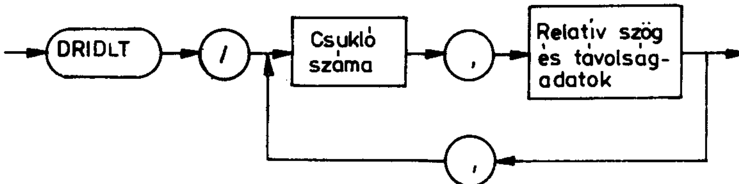
4.9. ábra. A ROBEX nyelv DRIVE-utasítása (abszolút szög- és távolságadatok használatakor)



4.10. ábra. Az AL nyelv DRIVE-utasítása relatív szög- és távolságadatok használatakor



4.11. ábra. DRIVE-utasítás a VAL-ban relatív szög- és távolságadatok használata esetén



4.12. ábra. DRIVE-utasítás a ROBEX nyelvben relatív szög- és távolságadatok használatakor

VAL:

```
1. lépés: POINT PP1
          0,110,30,0,60,0          (Rendszerparancs)
2. lépés: MOVE PP1
```

Térbeli elképzelő képességünknek jobban megfelel, ha a célpontot, ill. a cél frame-rendszerét derékszögű koordinátákban adhatjuk meg. Erre az AL-ban, a HELP-ben és a ROBEX-ban közvetlenül is lehetőségünk van. Az AL-ban az orientációt a koordinátatengelyek körüli rotáció segítségével adhatjuk meg, a HELP-ben ezzel szemben a robotkéz ízületére vonatkoztatott csavarási, bólintási és elfordítási műveletekkel (az ún. Roll, Pitch és Yaw műveletekkel). A ROBEX-ban az orientációt a tér három egymásra merőleges síkjában végrehajtott rotációval írjuk le. Példaképpen mindhárom nyelvben felírjuk a robotnak az $x = 95$ cm, $y = 20$ cm és $z = 40$ cm-es pontba (ún. pozícióba) való beállításához szükséges utasításokat. Az orientációt pedig úgy állítjuk be, hogy a megfogószerkezet az x koordinátatengely irányába mutasson, feltéve, hogy a kiindulási helyzet a z irány volt.

AL:

```
MOVE ARM1 TO FRAME (ROT (YHAT,90 * DEG),VECTOR (95,20,40) * CM;
```

HELP:

```
MOVE (1,#1,950,#2,200#3,400,#5,90);
```

ROBEX:

```
GOTO/95,20,40,ZXROT,90
```

Érdekes, hogy a ROBEX az explicit mozgásvezérlő utasítás alapszavaként a GOTO-t használja, holott ez a szó általában a programbeli feltétel nélküli vezérlésátadás (ugrás) alapszava szokott lenni. (A ROBEX-ben azonban erre a JUMP alapszó használatos.)

A *4.13.* ábra explicit koordinátával megadott pozíció- és orientáció beállítását szemlélteti.

A *4.14.*, *4.15.* és *4.16. ábrák* olyan egyszerű mozgásvezérlő utasítások szintaxisdiagramjait mutatják, amelyeknél a frame-rendszereket adjuk meg.

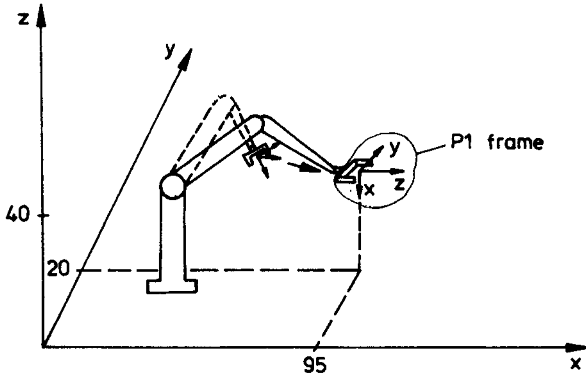
A VAL-ban ilyen explicit adatot nem lehet közvetlenül beírni a programutasításba. Rendszerparancsal azonban definiálni lehet a pálya egy pontját, vagy annak frame-rendszerét. Ez a frame-rendszer automatikusan bekerül egy globális frame-listába (l. 5. fejezet). Ennek a frame-nek a szimbolikus neve (azonosítója) ezután már felhasználható MOVE-utasításban, és így az azonosító által jelölt pozícióra és orientációra pozicionálhatjuk a robotberendezést. Ez a példa VAL-ban a következőképpen írható:

VAL:

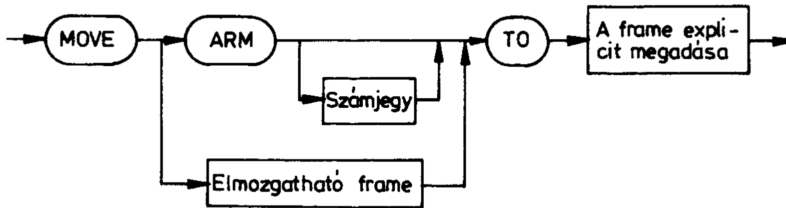
```
1. lépés: POINT P1
          95,20,40,0,90,0          (Rendszerparancs)
2. lépés: MOVE P1                  (Utasítás)
```

Az orientációt a VAL-ban az Euler-féle szögekkel adjuk meg.

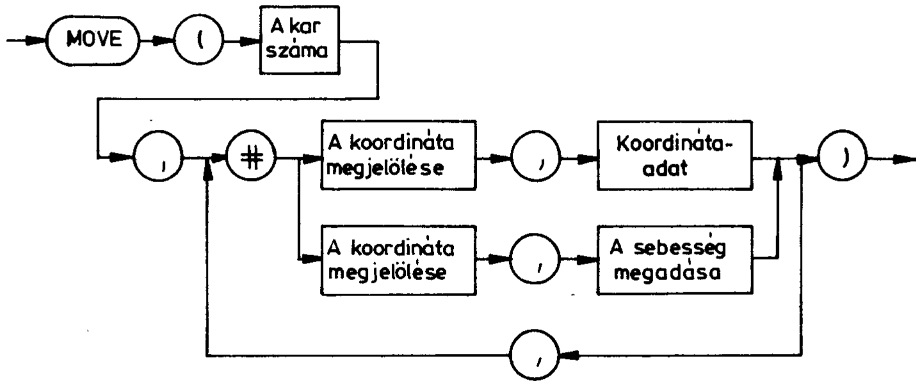
Az AL, a VAL és a ROBEX lehetőséget ad arra is, hogy a pálya adott pontját, ill. annak frame-rendszerét betanítási eljárással definiálhassuk (5. fejezet). Mivel az öt tárgyalt nyelv közül egyedül az AL nyelv kezeli a frame-eket önálló adattípusként, így



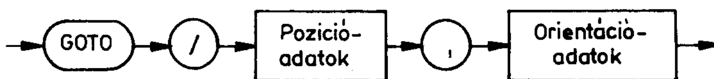
4.13. ábra. Helyzet beállítása explicit koordináták megadásakor



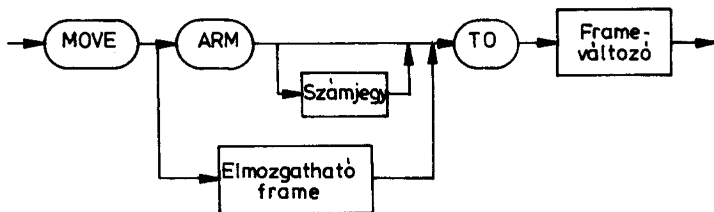
4.14. ábra. Mozgásvezérlő utasítás az AL nyelvben a frame-rendszer explicit megadásakor



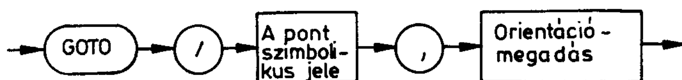
4.15. ábra. Mozgásvezérlő utasítás a HELP-ben



4.16. ábra. Explicit mozgásvezérlő utasítás a ROBEX-ban



4.17. ábra. Frame-változó segítségével felírt mozgásvezérlő utasítás az AL nyelvben



4.18. ábra. A pont szimbolikus nevének segítségével felírt mozgásvezérlő utasítás ROBEX-ben

csak AL-ban lehet olyan programutasítást felírni, amely egy frame-változó által jellemzett helyzetbe pozicionálja a robotot (l. *4.17. ábra.*)

Ez lehetőséget ad arra, hogy ha a későbbi műveletek során a beállítani kívánt célpozíció külső jeltől függ, akkor a cél frame-rendszerét ettől függően definiálhassuk.

A *4.3. programrészletben* ha a külső jel értéket kap, akkor a cél frame-rendszerében a z-koordináta értéke 8 cm lesz, ellenkező esetben pedig 20 cm.

```
SCALAR externsignal;
FRAME celframe;
      :
IF externsignal THEN
  celframe←FRAME (NILROT, VECTOR (30, 50, 8) * CM)
ELSE
  celframe←FRAME (NILROT, VECTOR (30, 50, 20) * CM);
      :
MOVE ARM TO celframe;
      :
MOVE ARM TO celframe;
      :
```

4.3. programrészlet. Frame-változó segítségével felírt mozgásvezérlő utasítás AL-ban

A HELP-ben szimulálni lehet a frame-változókat, ha az x-, y- és z-értékeket, valamint a csavarás, a bólintás és az elfordulás szögértékeit skalár változókban helyezzük el. Eltekintve attól, hogy a ROBEX-ben változót nem használhatunk, a fordításhoz is egy pont csak szimbolikusan adható meg, azaz ez egy változó pozíció (l. a *4.18. ábrát*). AL-ban ezzel szemben változó rotációadat is előírható. Ez csak a program futása alatt kerül kiértékelésre. ROBEX-ben először egy P1 pontot kell definiálnunk, és ezután adhatjuk ki a mozgásvezérlő utasítást.

```
ROBEX:
P1 = POINT/17,25,15
:
GOTO/P1
```

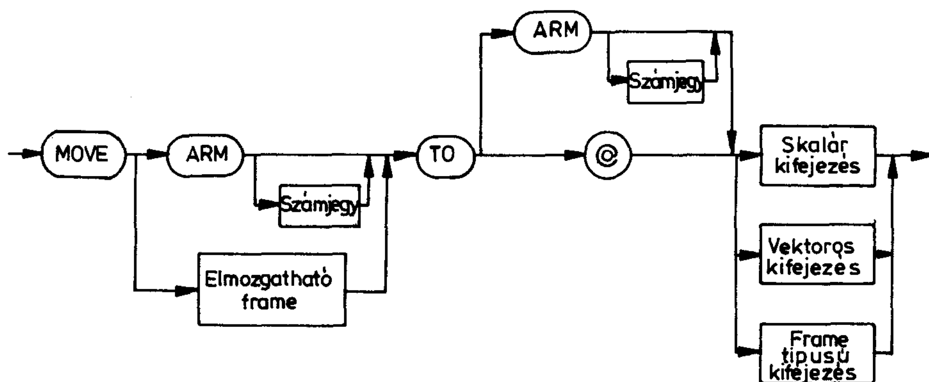
Ha megadjuk az EX („exakt") kulcsszót, akkor a robot pontosan fog ráállni az előírt pozíció- és orientációértékekre (l. a 4.2.4. pontot). Amellett, hogy ilyen abszolút frame-rendszerek szerinti elmozdulásokat is meg tudunk valósítani, az AL, a VAL, a HELP és a ROBEX nyelvekben lehetőség van derékszögű koordinátákban relatív elmozdulások megtételére is. Például az aktuális robotpozícióhoz képest az x irányba 20 cm-es, és z irányba 10 cm-es relatív elmozdulást a következőképpen írhatunk elő (l. a *4.19.*, *4.20.* és *4.21. ábrákat*).

AL:
```
MOVE ARM TO@ + VECTOR(20,0,10) * CM;
```
VAL:
```
DRAW 20,,10
```
HELP:
```
MOVE (1,#1,X+200,#2,Y#3,Z + 100);
```
ROBEX:
```
GODLTA/20,0,10,0,0
```

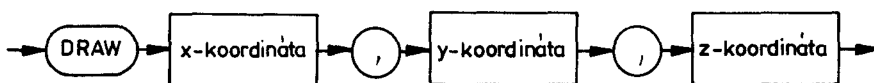
Az AL nyelvben a robotkar frame-rendszerével kapcsolatban akár frame típusú, akár vektor akár pedig skalár típusú kifejezéssel végezhetünk műveletet, azonban vektoros kifejezés használatakor csak a kar pozícióját, skalárkifejezés használatakor pedig csak a pozícióvektor z koordinátáját tudjuk megváltoztatni.

AL-ban relatív elmozdulások végrehajtásához nincsenek speciális utasítások, erre a célra a pillanatnyi robotpozícióhoz képest egyszerűen csak relatív ábrázolásban fejezzük ki a célként megadott frame-rendszert (az „@" vagy az ARM szimbólumok segítségével).

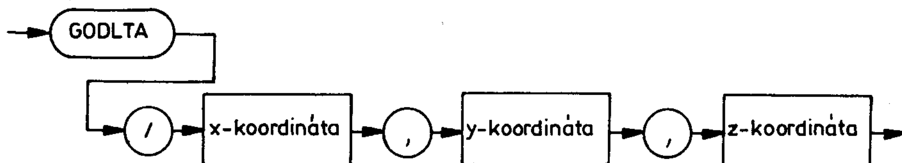
Ugyanígy a HELP nyelvben sincs ilyen célra speciális utasítás, először le kell kérdeznünk a pillanatnyi pozíciót, és ennek számértékeit kell letárolnunk az x, y és z skaláris változókban.



4.19. ábra. Az AL nyelv utasítása relatív elmozdulás végrehajtására



4.20. ábra. A VAL nyelv utasítása relatív elmozdulás végrehajtására



4.21. ábra. A ROBEX nyelv utasítása relatív elmozdulás végrehajtására

4.2.4. Mozgásvezérlő utasítások paraméterezéssel

A robot mozgásának céladatain kívül a programozáskor további paramétereket is meg szeretnénk adni, amelyekkel a művelet végrehajtása mellett még a vezérlés módját is befolyásolni kívánjuk. Ilyen paraméterek segítségével a következő jellemzőket írhatjuk elő:

- a művelet sebességét;
- a pozicionálás időtartamát;
- a robot helyzetét, végállásban;
- a végpont megközelítésekor a pozicionáló szabályozás pontosságát;
- a vezérlés módját (pálya-, szakasz- stb.);
- az egyidejűleg végrehajtható műveleteket;
- egy a robot mozgására szuperponált vibráló mozgást;
- közbülső pozíciók, ill. frame-helyzetek érintését;
- az oda- és visszautak frame-rendszerét.

Ezeknek a paramétereknek a segítségével igen összetett és dinamikai tulajdonságaiban tökéletesített mozgatási pályákat (trajektóriákat) lehet biztosítani. A legkülönfélébb paraméterekkel kiegészített vezérlési utasításokat a robot vezérlőrendszere csak kisebb-nagyobb pontatlansággal képes megvalósítani.

Az egyes nyelvekben a legkülönfélébb paraméterek, ill. paraméterkombinációk használata megengedett. Ezzel kapcsolatban a *4.1. táblázat* nyújt áttekintést. A közbülső pozíciókat, ill. frame-rendszereket előíró, valamint a cél megközelítését és elhagyását előíró paramétereket nem tartalmazó egyszerű pozicionáló utasítások esetében a paraméterek automatikusan a teljes útvonalra vonatkoznak. Először ezt az esetet tekintjük át.

A sebesség vagy globálisan adható meg, vagy csakis a soron következő utasításra vonatkoztatva. A legtöbb esetben a sebességet nem fizikai mértékegységekben kifejezve adjuk meg, hanem a rendszer által kiszámított maximális vagy közepes sebességre vonatkoztatva, tehát százalékosan kifejezve.

AL:

```
SPEED_FACTOR←2
```

Erre az utasításra a robot minden ezután következő mozgását a maximális sebesség felével fogja végrehajtani.

```
MOVE ARM TO célframe
WITH SPEED_FACTOR = 3;
```

Ekkor a mozgást a maximális sebesség harmadával fogja végrehajtani.

4.1. táblázat. Az AL, VAL, HELP, SIGLA és a ROBEX mozgásvezérlő utasításainak összefoglaló táblázata

	AL	VAL	HELP	SIGLA	ROBEX
Egyszerű mozgásvezérlő utasítások					
– A léptető motor lépésszámának megadása	–	–	–	MO/	–
– Szög- és távolságértékek megadása az egyes tengelyekre	DRIVE	DRIVE precíz pont	–	–	DRIVE
– Explicit derékszögű koordinátaadatok	Frame	(Frame)	Skalár	–	Frame
– Frame-típusú változó kezelése	igen	igen	Skalárok segítségével szimulálja	–	csak a fordítás során
– Relatív elmozdulás megadása	Frame + Frame, Frame + vektor	DRAW	Skalár típusú kifejezéssel szimulálja	–	GODLTA
Paraméterezett mozgásvezérlő utasítások;					
– Sebesség	SPEED-FACOR	SPEED	SPEED	–	RAPID FEDRAT
– Időtartam	DURATION	–	–	–	–
– Robothelyzet	–	RIGHTY LEFTY ABOVE BELOW	–	–	–
– Szabályozási pontosság	NULLING, NO-NULLING	COARSE NONULL NULL INTOFF INTON	–	–	EX
– Időben paralel művelet-végrehajtás	igen, általában lehetséges	MOVE MOVES MOVET MOVEST (csak a megfogás)	igen	igen taskok segítségével	–
– Szuperponált rezgőmozgás	WOBBLE	WEAVE	–	–	–
– Közbenső frame-helyzetek megadási módja	VIA	CP	SMOVE	–	–
a) sebességmódosítással	VELOCITY	SPEED	igen	–	–
b) időtartam-megadással	DURATION	–	–	–	–
– Megközelítési frame	APPROACH	APPRO	skalár kifejezésekkel szimulálja	–	szimulálja
– Elhagyási frame	DEPARTURE	DEPART	skalár kifejezésekkel szimulálja	–	szimulálja

A 4.1. táblázat folytatása

	AL	VAL	HELP	SIGLA	ROBEX
ellenőrzött mozgásvezérlés (határérték-felügyelet mellett) – Erő (megfogószerszám) – Nyomaték (megfogószerszám) – Ellenőrzött paraméterek	 FORCE TORQUE FORCE TORQUE	 – – –	 FORCE – –	 RP – –	 – – –
Szabályozott paraméterű mozgásvezérlés – Erő (megfogószerszám) – Nyomaték (megfogószerszám)	 FORCE TORQUE	 –	 –	 –	 –
Mozgásvezérlés eseményfelügyelettel – Megszakítások	 EVENT típusú változók	 REACT (IGNORE)	 explicit lekérdezés	 –	 EVENT-ek specifikálása
Mozgásvezérlés időzítési feltételekkel	DURATION	–	explicit lekérdezés	–	–

VAL:

SPEED 80 ALWAYS

A robot minden ezt követő mozgatási műveletet a normál sebesség 80%-ával fogja végrehajtani.

SPEED 200

MOVE célframe

Erre a robot a pozicionálási mozdulatot a normál sebesség kétszeresével fogja végrehajtani.

HELP:

SPEED(1,80)

Az 1. kar minden ezután következő mozdulatát a maximális sebesség 80%-ával fogja végrehajtani.

MOVE (1,#1, 30, #3, 10, #11, 80)

A robot X irányban a maximális sebesség 80%-ával fogja végrehajtani a műveletet.

SIGLA:

Sebességparaméter nem adható meg.

ROBEX:

RAPID

A következő műveletet a „gyors" fokozatban hajtja végre.

FEDRAT/80

A normál műveleti sebesség 80%-át állítja be.

A művelet időtartama csak az AL-nyelvben adható meg.

AL:

```
MOVE ARM TO célframe
  WITH DURATION = 5 * SEC;
```

Ezzel a művelet időtartama kb. 5 másodperc lesz.

Egyes ipari robotoknál a kinematikai adottságok olyanok, hogy a robot különböző állapotban is rá tud pozicionálni a céltárgy frame-rendszerére. Az ilyen nem egyértelmű beállítási lehetőséget adott esetben maga a rendszer oldhatja meg, ill. iktathatja ki, vagy a megkívánt robothelyzetet a programozó határozza meg, ill. választja ki. A tárgyalt nyelvek közül csak a VAL nyújtja ezt a programtechnikai szolgáltatást, amely speciálisan az UNIMATION cég PUMA elnevezésű robotjainak vezérléséhez készült. Ennek a robotnak a konstrukciója az emberi test főbb mozgáslehetőségeit igyekszik lemásolni a felsőtest, a váll és a kar mozgáslehetőségének egyszerűsített utánzásával (l. a *4.22. ábrát*).

VAL:

RIGHTY

Az ember jobb kezének megfelelő helyzet beállítása.

LEFTY

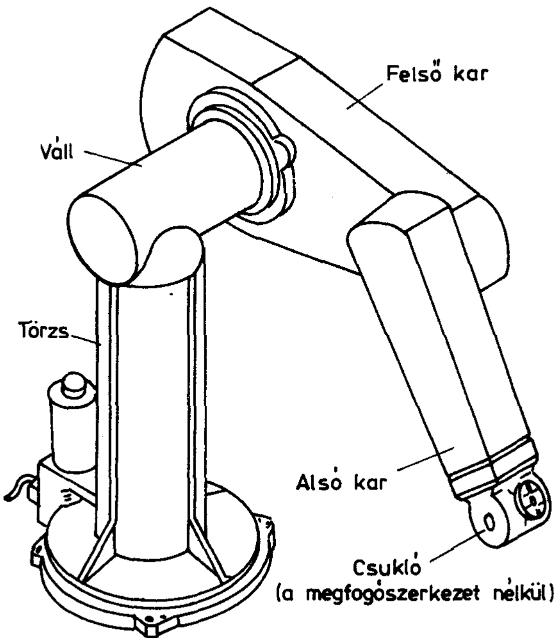
Az ember bal kezének megfelelő helyzet beállítása.

ABOVE

A PUMA-robot könyökét felső helyzetbe fordítja.

BELOW

A PUMA-robot könyökét alsó helyzetbe fordítja.



4.22. ábra. A PUMA nevű robot, amely részleteiben az ember törzsének és kezének mozgását utánozza

Az, hogy egy megadott célpont frame-rendszerére pontosan ráálljon a robot (a közbülső frame-ek érintésével), sokkal több időt vesz igénybe, mint egy tűréssel végrehajtott pozicionálás. Emiatt a programban megadható, hogy *exakt szabályozást* kívánunk-e végrehajtani, vagy sem.

AL:

```
MOVE ARM TO célframe
  WITH NULLING;
```

A művelet végén megmaradó kicsiny pozicionálási hibát kinullázza.

VAL:

COARSE

Durvább tűrésmező beállítása a szervoszabályozáshoz.

FINE

Kisebb tűrésmezőt írunk elő.

NONULL

A mozgást akkor is leállítja, ha nem érkezik visszajelzés valamennyi tengelytől a célpozíció elérésével kapcsolatban.

NULL

Valamennyi tengelynek el kell érnie a célpozíciót és erről visszajelzésnek kell befutnia.

INTOFF

A pályavezérlés során a hibaintegrálást mellőzi.

INTON

A hibaintegrálást elvégzi.

HELP:

Paraméterezés nem lehetséges.

ROBEX:

GÖTO/PI,EX

A PI nevű frame-rendszer exakt beállítása.

A VAL-rendszerben mind a tengelyinterpoláció, mind pedig a derékszögű koordináták lineáris interpolációja megvan, ezért a megfelelő MOVE-utasítás kiválasztásával a programban előírhatjuk a vezérlés módját.

VAL:

MOVE célframe

A pozicionálást tengelyinterpolációval hajtja végre.

MOVES célframe

A pozicionálást a derékszögű koordináták lineáris interpolációjával hajtja végre.

Az arra alkalmas vezérlési rendszerre épülő nyelveknél meg lehet adni, hogy a kar mozgatásával egyidejűleg a végrehajtó szerv is hajtson végre műveletet. Ezt VAL-ban közvetlenül a mozgásvezérlő utasításban, AL-ban pedig közvetett módon tehetjük meg.

AL:

```
COBEGIN
MOVE ARM TO célframe;
OPEN HAND TO 1 * CM
```

```
COEND
```

A COBEGIN és COEND között álló két utasítást egymással párhuzamosan hajtja végre.

VAL:

```
MOVET célframe, 10
```

Pozicionálási művelet tengelyinterpolációval, miközben a megfogószerszám 10 mm-nyire záródik.

```
MOVEST célframe, 10
```

Pozicionálási művelet a derékszögű koordináták lineáris interpolációjával, miközben a megfogószerszám 10 mm-nyire záródik.

Ha a kiindulási és a célállapot frame-rendszerei között végrehajtott mozgás példája nem lehet egyenesvonalú, de definiálatlan pályát sem szeretnénk megengedni, akkor *közbenső* frame-rendszereket kell megadnunk. A programrendszerek többsége nem biztosítja a pontos áthaladást ezeken a közbenső frame-eken. Ez egyébként a frame-eket összekötő egyenes pályaszakaszokon közbenső megállás nélkül nem is lenne lehetséges. A robot ezért lekerekíti a pálya töréspontjait. (Ez a pályaszakaszok ún. „simítása".)

AL:

```
MOVE ARM TO célframe
  VIA közbframe1, közbframe2;
```

Közbülső frame-eket tartalmazó mozgásvezérlő utasítás esetén az AL-nyelv úgy definiálja a pálya kialakulását, hogy a közbenső frame-ek érintésekor a végrehajtó szerv orientációja egyezzen meg a frame orientációadatával. A pályát polinomok segítségével kell kiszámítani, melyek az egyes tengelyek állásszögeit az idő függvényében kifejezve adják meg. Ennek a számítási eljárásnak a következtében a végrehajtó szerv pályájában hurkok is képződhetnek egyik vagy másik közbenső frame elérése előtt. Ez az ún. túllendülés (l. *4.23. ábra*). A rendszer ezeket üzenetek formájában képes ugyan jelezni, programozáskor azonban az ilyen túllendülések általában nem láthatók előre.

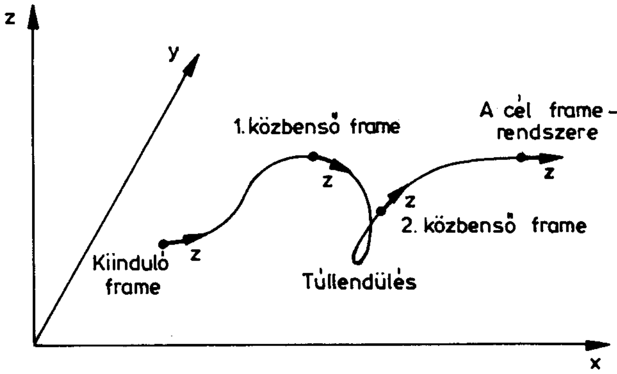
VAL:

```
ENABLE CP
MOVES közbframe1
MOVES közbframe2
MOVES célframe
DISABLE CP
```

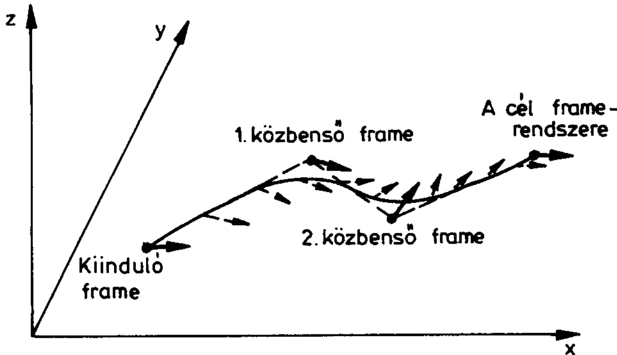
A *4.24. ábrán* az az eset látható, amikor a pálya kialakításakor a soron következő közbenső frame figyelembevételét a CP rendszerkapcsoló segítségével ki lehet iktatni, ill. ismét be lehet vonni az útvonal számításába. Ilyenkor a pályaszakaszok „egybesimítása" áll elő. A végrehajtó szerv orientációja a pályán való haladás közben a közbenső frame-ek megfelelő értékei szerint egyenletesen változik.

HELP:

```
SOME(1,#1, xz1, #2, yz1, #3, zz1, #4, rz1, #5, pz1, #6, yawz1, #7, 400);
SMOVE(1, #1, xz2, #2, yz2, #3, zz2, #4, rz2, #5, pz2, #6, yawz2, #7, 300);
MOVE(1, #1, x, #2, y, #3, z, #4, r, #5, p, #6, yaw);
```



4.23. ábra. Közbenső frame-helyzetekkel megadott pálya az AL nyelvben



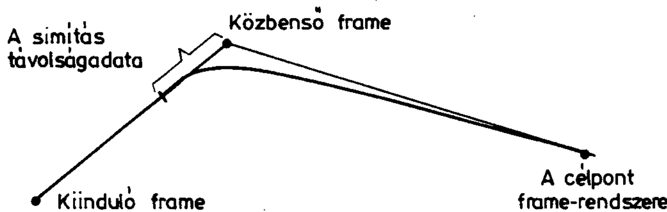
4.24. ábra. Közbenső frame-helyzetekkel megadott pálya „egybesimítása" a VAL nyelvben

A HELP nem kezel frame típusú változókat, ezért az x-, y-, z-koordinátákat, valamint a csavarás, a bólintás és az elforgatás szögértékeit skalár-változókban kell letárolni. A pálya számítása hasonlóképpen történik, mint a VAL-ban, azzal a különbséggel, hogy az SMOVE utasítással még egy #7-tel jelölt paramétert is megadunk, amely a simítás távolságadatát rögzíti (l. *4.25. ábra*).

SIGLA: közbenső frame használatára itt nincs mód.

ROBEX: közbenső frame kitűzésére itt sincs lehetőség.

Annak érdekében, hogy a robotkar által bejárt pálya dinamikailag is módosítható legyen, ezért az eddig tárgyalt paraméterek közül többnek – így az időnek, sebességnek stb. – közbenső frame-ekre történő megadására is lehetőség van némelyik nyelvben.



4.25. ábra. A pálya simítása a HELP nyelvben

AL:

```
MOVE ARM TO célframe
  VIA közbframe WHERE VELOCITY = 15 * CM/SEC,
    DURATION = 4* SEC;
```

A vezérlésnek gondoskodnia kell arról, hogy a közbenső frame érintésekor a végrehajtó szerv sebessége kb. 15 cm/s legyen, és hogy a közbenső frame elérésének időtartama 4 s legyen.

VAL:

SPEED 100	névleges sebesség
Enable CP	
MOVE közbframe1	
SPEED 20	A névleges sebesség 20%-a
MOVES közbframe2	
MOVEST célframe,5	A megfogóeszköz zárása
DISABLE CP	

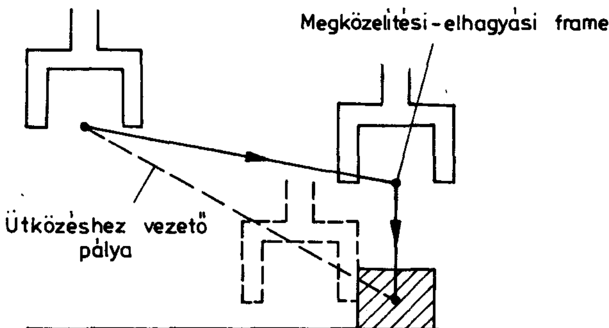
VAL-ban egy közbenső frame-től kezdve meg lehet változtatni a sebességet, vagy pedig a kar mozgatásával egyidejűleg működtetni lehet a megfogóeszköz nyitását, ill. zárását.

HELP:

```
SMOVE(1, #1, x, #2, y, #3, z, #4, roll, #5, pitch, #6, yaw, #11, xspeed, #14, rollspeed, #16, yawspeed, #7, 300);
```

A HELP SMOVE utasításában a sebesség megadásával egy közbenső frame-től kezdve meg lehet változtatni az egyes tengelyek vezérlésének sebességét.

A *megközelítési*, ill. *elhagyási frame*-ek a közbenső frame-ek megkülönböztetett alakjai[4.2.]. Ezeket a frame-eket cél-, ill. start-frame-hez viszonyítva, relatív módon definiáljuk annak érdekében, hogy a robot végrehajtó szerve a célt, annak közvetlen közelében meghatározott irányból közelítse meg, és hogy a kiindulási pontot egy adott irányban hagyja el. Ezáltal elkerülhető pl. az, hogy a megfogóeszköz a munkadarab megközelítésekor vagy annak elhagyásakor nekiütődjön a munkadarabnak vagy eltolhassa azt (l. *4.26. ábra*).



4.26. ábra. Megközelítési, ill. elhagyási frame alkalmazásakor elkerülhető, hogy a szerszám ütközzön a munkadarabbal

4.2. Ezek tulajdonképpen a cél egy adott környezetében, az oda-, ill. visszavezető utat definiálják.

Az AL és a VAL nyelvekben a megközelítési, ill. elhagyási frame-eket a célpont, ill. a kiindulási pont frame-rendszereihez viszonyítva lehet definiálni. Az AL-ban a megközelítési, ill. elhagyási frame kifejezhető egy frame-ek közötti reláció (azaz transzláció és rotáció), valamint egy eltolási vektor segítségével (ami tisztán transzláció), vagy pedig a cél, ill. a kiindulási pontbeli frame z-koordinátatengelye irányában értelmezett távolság segítségével.

AL:

```
MOVE ARM TO célframe
  WITH APPROACH = FRAME(rotáció, vektor)
  WITH DEPARTURE = FRAME(rotáció, vektor);
```

A *4.27. ábra* bemutatja a kiindulási pont és a célpont frame-rendszerei, valamint a megközelítési és az elhagyási frame-rendszerek geometriai viszonyait. Az AL nyelvben explicit utasítással is kiszámítható a megközelítési, ill. elhagyási frame-rendszer. Az alábbi AL utasításban ezt **alframe** változónévvel jelöljük.

```
alframe ← célframe * FRAME (rotáció, vektor);
```

AL:

```
MOVE ARM TO célframe
  WITH APPROACH = VECTOR(25,9,0)
  WITH DEPARTURE = VECTOR(0,20,12);
```

A *4.28. ábra* szerint a megközelítési, ill. elhagyási frame ebben az esetben megőrzi a célpont, ill. a kiindulási pont frame-rendszerének orientációját. Ennek explicit kiszámítására alkalmas utasítás a következő:

```
alframe ← célframe + VECTOR(25,9,0)WRT célframe;
```

Az eltolási vektor koordinátaadatai a célpont, ill. a kiindulási pont frame-koordinátarendszerére vonatkoznak.

AL:

```
MOVE ARM TO célframe
  WITH APPROACH = 20
  WITH DEPARTURE = 20;
```

Az aframe a *4.29. ábra* szerint a cél-, ill. kiindulási frame z irányú eltolásának felhasználásával számítható ki:

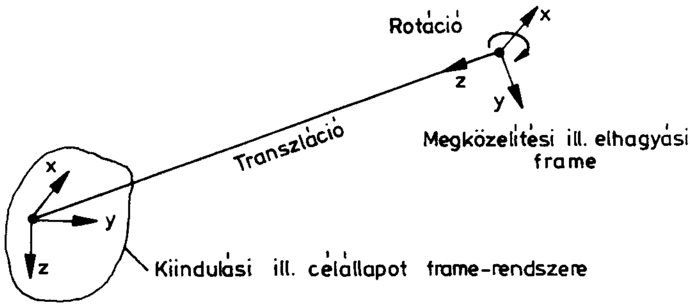
```
aframe ← célframe + (20 * ZHAT)WRT célframe;
```

A megközelítési, ill. elhagyási frame-ek megközelítésére, ill. elérésére a VAL nyelvben speciális utasítások állnak rendelkezésre. Ezek az APPRO és az APPROS utasítások. Az előbbi tengelyinterpolációs eljárást, az utóbbi derékszögű koordináták szerinti lineáris interpolációs eljárást használ.

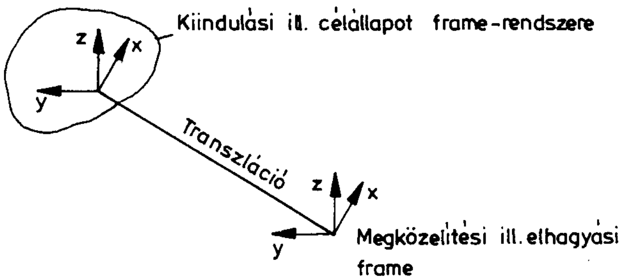
VAL:

```
APPRO célframe, 50
APPROS célframe, 50
MOVE célframe
```

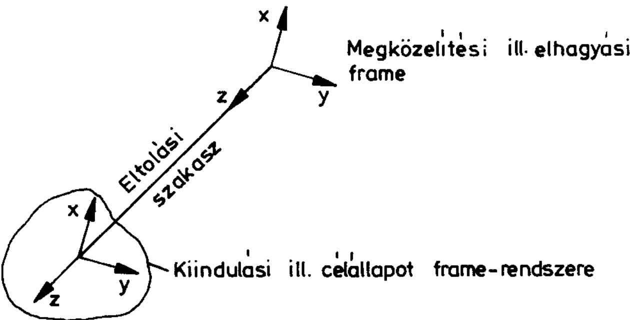
A megadott 50 mm-es távolság a célframe z irányú eltolását jelenti, az eltolást azonban csak a MOVE utasítás hajtja végre.



4.27. ábra. A megközelítési, ill. elhagyási frame definíciója a frame-rendszerek közötti reláció segítségével



4.28. ábra. A megközelítési, ill. elhagyási frame definíciója eltolási vektor segítségével



4.29. ábra. A megközelítési, ill. elhagyási frame definíciója egy z irányú eltolási szakasz megadásával

4.30. ábra. Az AL nyelvben definiálható (vibrációs) mozgás

VAL:

DEPART 50

DEPARTS 50

A kiindulási frame-től z irányba 50 mm-es eltolást ír elő a megfelelő interpolációs eljárással együtt.

Az AL és a VAL nyelvekben arra is meg van a lehetőség, hogy a robot mozgására egy rezgő mozgást szuperponáljunk. Ennek olyankor vehetjük hasznát, ha pl. az alkatrészeket a robotra szerelt mágnessel emeljük fel, egy lerakóhelyre visszük, majd ott elengedjük. Ilyenkor az a veszély fenyeget, hogy az alkatrészek a remanens mágnesség miatt a mágnesen maradnak, de kismértékű rázással ez elkerülhető.

AL:

MOVE ARM TO célframe
  WITH WOBBLE = 2;

Az eredeti mozgáshoz ilyenkor a *4.30. ábrán* látható módon egy szinuszos mozgás is hozzáadódik.

VAL:

WEAVE 25,5,2

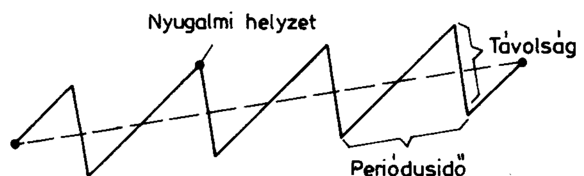
MOVES célframe

Amint a *4.31. ábra* mutatja, a WEAVE-utasítással olyan fűrészrezgést definiáltunk, amelynek amplitúdója 25 mm, periódusideje 5 s a fűrészfogmintázat csúcsain pedig a robotkar 2 s-ig nyugalmi helyzetben marad. A fűrészrezgés szerinti robotmozgást csak a definíciót követő MOVE vagy MOVES utasítások indítják el.

4.2.5. Az érzékelő jeleinek felhasználása a robot mozgásvezérlő utasításainál

A robot mozgásának ellenőrzésére és vezérlésére erő- és nyomatékérzékelőket, valamint megközelítést és érintést érzékelő eszközöket alkalmaznak. Némelyik nyelvben az érzékelők jeleire határértékek adhatók meg, ezeket a mozgás során a rendszer figyelemmel kíséri. Az érzékelők jeleinek felhasználási lehetőségei ezzel sajnos ki is merültek, vagyis az érzékelőkre orientált speciális utasításokban jelentős hiány mutatkozik. Ennek a mulasztásnak az az oka, hogy még a fejlesztőrendszereknél is csak a meglevő hardver, ill. érzékelőkonfigurációra fordítanak figyelmet és kizárólag a meglevő speciális rendszerkonfiguráció működtetésére hoznak létre nyelvi eszközöket. Az AL amerikai változatában pl. egy mozgásvezérlő utasításon belül le lehet kérdezni az erő- és nyomatékérzékelőket, vizuális eszközök azonban már nem kezelhetők, míg egy VAL-nyelvi változatban éppen fordított a helyzet: vizuális eszköz kezelhető, de más nem.

Két esetet különböztethetünk meg aszerint, hogy az érzékelők jeleire előírt határértékeket hogyan használjuk fel. Az egyik esetben ezek csak felügyeleti célt látnak el. Ilyenkor a határértékek elérése esetén a program egy külön ágra ugrik, vagy a főprog-



4.31. ábra. A VAL nyelvben definiálható (vibrációs) mozgás

rammal párhuzamosan egy külön task-ot aktivál. A másik esetben az érzékelő jelét paraméterként kezeljük, és a vezérlés eszerint módosítja a robot mozgását. Példa az utóbbi esetre, amikor a robot végrehajtó szervét valamelyik irányban egy, a programban előírt adott erővel kívánjuk működtetni.

4.2.5.1. A mozgásvezérlő utasítások végrehajtásának ellenőrzése az érzékelők jeleinek segítségével

A legtöbb nyelvben megvan a lehetőség arra, hogy a robot mozgását az érzékelők segítségével ellenőrizzük. Ha a kar helyzete egy előírt határértéket túllép, akkor bekövetkezik egy programelágazás, vagy a mozgás megáll. AL-ban határértéket adhatunk meg a megfogószerkezetben elhelyezett *erőmérő* és *nyomatékmérő cellák* jeleire. A mérőcella a kar frame-jének (azaz a megfogószerkezet frame-rendszerének) x, y és z tengelyei irányában ébredő erőket méri, amelyek tetszőleges frame koordinátatengelyeire is átszámíthatók. Azt a frame-et, amelyre az erőhatásokat vonatkoztatni kívánjuk, megfelelő paraméterezéssel jelölhetjük ki (l. *4.32. ábra*).

Az erők felbontásának frame-rendszerét – az ún. erő-frame-et – kijelölhetjük a báziskoordináta-rendszerhez viszonyítva (alapszava: IN WORLD), vagy a megfogóeszköz frame-rendszeréhez, az ún. kézkoordinátákhoz viszonyítva (alapszava: IN HAND). Ebben az utóbbi esetben a mérési irányok a mozgás tartama alatt a robotkéz orientációváltozásának megfelelően módosulnak (l. a *4.34. ábrát*). A tulajdonképpeni ellenőrzési műveletet (az ún. érzékelővezérlést) a *4.33. ábrán* látható szintaxisdiagram mutatja. A kissé bonyolult szintaxis néhány példán keresztül könnyebben megérthető.

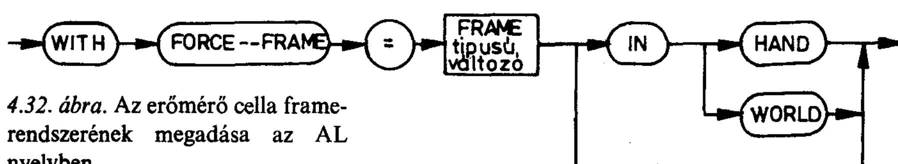
AL:

```
MOVE ARM1 TO célframe
  ON FORCE(ZHAT) ≧ 100 * GM DO
    STOP &;
```

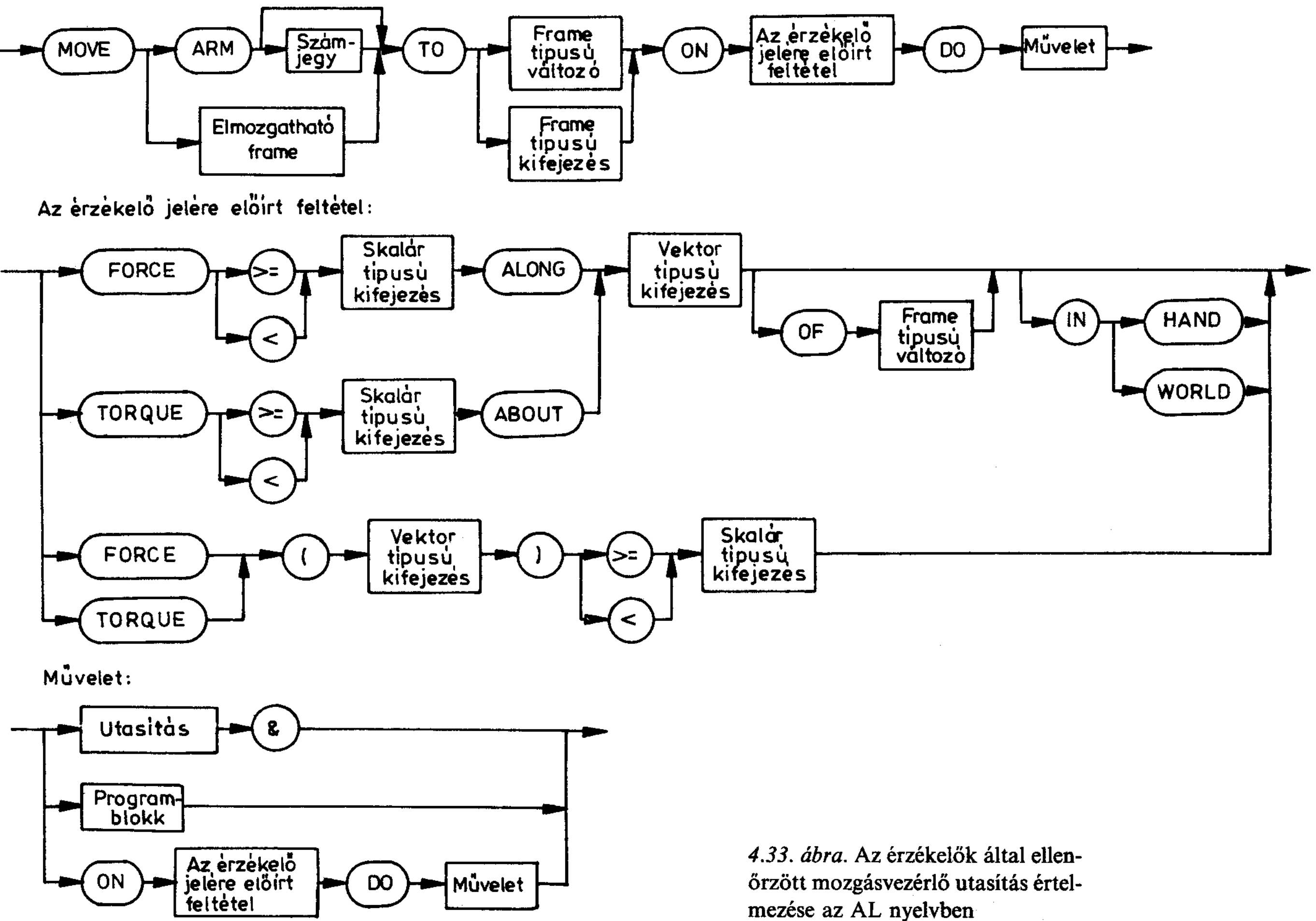
Az & jel értelmezését a negyedik példában mutatjuk be. Az utasítás hatására az 1. robotkart a **célframe** által meghatározott helyzetbe hozzuk. Ha eközben a z tengely irányában 100 g vagy ennél nagyobb erő ébred, akkor a mozgás azonnal megáll. Ebben az utasításban nincs specifikálva, hogy melyik az a koordináta-rendszer, ill. frame, amelynek z-koordinátatengelyére az adatok vonatkoznak. Ezt egy korábbi Force-Frame előírás határozza meg.

AL:

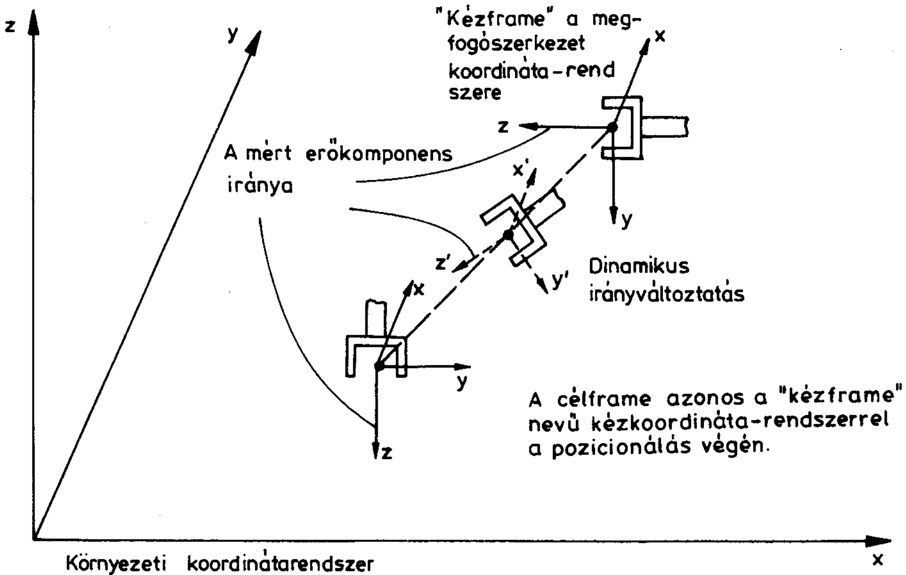
```
MOVE ARM 2 TO célframe
  WITH FORCE FRAME = kezframe IN HAND
  ON FORCE ≧ 200 * GM ALONG ZHAT DO
    BEGIN
    STOP;
    PRINT ("Elérte az asztalt"),
```



4.32. ábra. Az erőmérő cella frame-rendszerének megadása az AL nyelvben



4.33. ábra. Az érzékelők által ellenőrzött mozgásvezérlő utasítás értelmezése az AL nyelvben



4.34. ábra. A megfogószerkezet rendszerének z irányába eső erőkomponens ellenőrzése az AL nyelvben

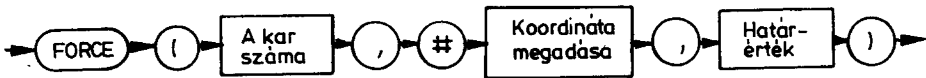
```
    asztal ← ARM2;
    jelzőbit ← TRUE
    END;
```

Legelőször a WITH FORCE FRAME kifejezéssel definiáljuk a **"kezframe"** nevű frame-et. Ez lesz az a koordináta-rendszer, amelyre vonatkozóan a következő utasításokban (a későbbi MOVE utasításokban is) az erő vektorkomponenseit értelmezzük. A **"kezframe"** frame-et az IN HAND kifejezés megadásával a robotkéz (azaz a mozgó végrehajtó szerv) mindenkori orientációjához rendeltük hozzá, és így a mért erőkomponens iránya a robotkéz mozgása során változik *(4.34. ábra).* Ha a művelet során egy 200 grammnál nagyobb, vagy azzal egyenlő erő érzékelhető a z koordináta irányában, akkor a mozgás automatikusan leáll, megjelenik az „Elérte az asztalt" üzenet, majd az elért pozíció- és orientációérték átadódik az **"asztal"** nevű framenek, és a program beállít egy **jelzőbitet.**

AL:

```
MOVE ARM3 TO célframe
  ON FORCE < 30 * GM ALONG XHAT OF kantni IN WORLD DO
    ON FORCE ≧ 500 * GM ALONG YHAT DO
      PRINT ("Vigyázat") &;
```

E példában a program azt figyeli, hogy a mért erőnek a **kantni** nevű frame-változó x irányába eső komponense elérte-e az előírt alsó határértéket. Eközben az erő mérési iránya rögzített marad, mivel a frame-et a környezeti koordináta-rendszerre vonatkoztatva definiáltuk. Ha az alsó határértéket túlléptük, akkor még azt is ellenőrizzük, hogy az előzőleg WITH FORCE__ FRAME-mel definiált frame-rendszer y irányába eső erő nagyobb–egyenlő-e 500 g-nál. Csak ha ez is teljesül, akkor jelenik meg a "Vigyázat" üzenet. Ez a programrészlet két különböző irányba eső erőkomponens szerinti ellenőrzésre mutat példát.



4.35. ábra. Erőmérő cella jelének határérték beállítása a HELP nyelvben

AL:

MOVE ARM4 TO célframe
  ON FORCE ≧ 150 * GM ALONG ZHAT DO
    MOVE ARM2 TO lerakóhely &
  WITH DURATION = 5 * SEC;

E legutolsó programrészlettel most azt szeretnénk bemutatni, hogy miért kell az & jellel lezárni a MOVE-utasítást az ON FORCE DO kifejezés után. Ha ugyanis elmaradna az &-jel, akkor nem lehetne megállapítani, hogy a WITH DURATION = 5 * SEC kifejezés az ARM4 karra vagy az ARM2 karra vonatkozik-e. Ebben a példában a **célframe** eléréséig tartó pozicionálás időtartana 5 s. Ezzel szemben, ha a **„lerakóhely"** elérése tartana 5 s-ig, akkor az **&** jelnek a pontosvessző előtt kellene állnia.

A VAL nyelvben nincs lehetőség arra, hogy a mozgásvezérlő utasításban explicit módon adjuk meg a határértékek figyelését. Alkalmas hardver-eszköz használatával azonban segíthetünk magunkon. Az előírt alsó és felső határértékek túllépésekor le-, ill. felfutó élvezérléssel bináris kódot nyerhetünk, amely bitek felhasználásával – megfelelő programszervezés esetén – olyan programágra futhatunk rá, ahol egy adott alprogramot aktivizálhatunk (l. a 4.2.6. pontot).

A HELP nyelvben FORCE-utasítással beállítható egy-egy alkalmas határérték az erőmérő cellák jeleire. Ezek túllépésekor az éppen végrehajtott robotmozdulat leállítható. Ennek az utasításnak a szintaktikai szerkezete a *4.35. ábrán* látható. A FORCE-utasítás a programban az SMOVE-utasítást követi, és végrehajtása attól az időponttól kezdődik, amikor a simítási művelet elkezdődik, ill. a robotkar a célpont egy adott környezetébe ér (l. a 4.2.4. pontot). Ha az SMOVE-utasításban nem adunk meg távolságértéket, akkor az erőmérési jel figyelése azonnal elkezdődik.

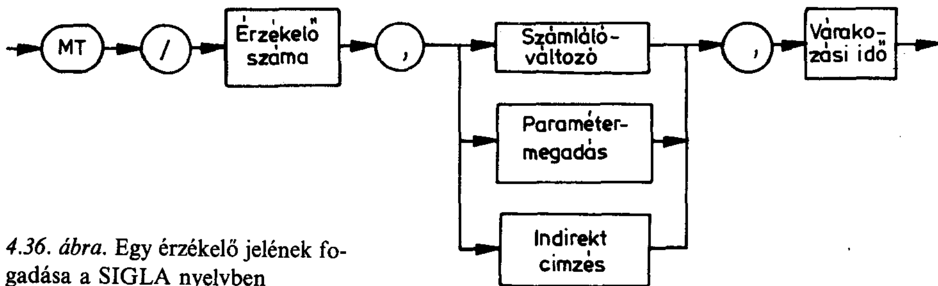
HELP:

SMOVE (1, #3, 150);
FORCE (1, #3, 60);

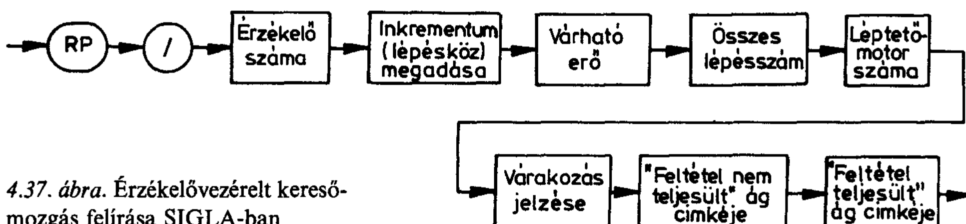
A robot pozicionálását ezzel úgy irányítjuk, hogy a pozíció z irányban 150 mm-re eltolódik. Ha eközben az erő eléri a 60-nal jelölt fokozatot (a 0-tól 255-ig felosztott teljes értéktartományban), akkor a kar mozgása megáll. Ha ezután lekérdezzük, hogy a kar elérte-e a z = 150 mm-es pozíciót, akkor kiértékelhető az erőhatásra megadott határérték túllépése és ennek függvényében a program megfelelő ágára ugorhatunk.

SIGLA-nál annak ellenére, hogy ebben a nyelvben nem lehet deklarálni sem frame típusú, sem pedig szabadon használható változókat, fogadni lehet az erőmérő cellák jeleit és meg lehet valósítani olyan robotpozicionálást, amikor az ébredő erőkre előírt határértékeket figyeli a rendszer, és annak megfelelően szabályozza a robotmozgást. Az erőmérő cella lekérdezését végrehajtó utasítás szerkezetét a *4.36. ábra* szemlélteti.

Az érzékelő beállása és annak lekérdezése közötti várakozási időt csak akkor kell megadni, ha a robotberendezés több érzékelővel rendelkezik. Az érzékelő jelét (0-tól



4.36. ábra. Egy érzékelő jelének fogadása a SIGLA nyelvben



4.37. ábra. Érzékelővezérelt keresőmozgás felírása SIGLA-ban

255-ig terjedő értéktartományban) vagy egy számlálóban, paraméterben vagy pedig indirekt címzés segítségével meghatározott memória rekeszben tároljuk le (l. még a 3.1.3. pontot is). Ennek a rekesznek a tartalmát lekérdezzük, és ettől függően hajthatunk végre valamilyen alkalmas programelágazást.

SIGLA:

MT/3, M5, 1;

Itt a 3-as érzékelő jelét fogadjuk és az M5 számlálóváltozóban tároljuk.

A *4.37. ábrán* bemutatott RP-utasítás hatására az adott léptetőmotor az összes előírt lépésszámot végrehajtja, közben a program minden lépés után ellenőrzi, hogy a megadott erőmérő cella jele elérte-e az előírt értéket. Ha ez bekövetkezik, akkor a program a teljesülő feltétel címkéjénél folytatódik, ellenkező esetben pedig a nem teljesülő feltétel címkéjénél. Ez az utasítás elsősorban arra való, hogy a SIGMA-robot karját z irányba függőlegesen lefelé lehessen mozgatni, vagy pl. a robot végrehajtó szervét addig lehessen forgatni, amíg a fellépő erőhatás el nem éri a megadott értéket.

SIGLA:

RP/2,7,120,70,3, −10,4,5;

A 3. léptetőmotorral 70 lépést végeztetünk. Minden hetedik lépésnél megvizsgáljuk, hogy a 2. erőmérő cella jele eléri-e a 120. fokozatot (értéktartományuk 0-tól 255-ig van beosztva). Ezekre a pontokra várakozást is előírtunk. Ha az erő még nem éri el a megadott értéket, akkor a program a 4. címkén folytatódik, ellenkező esetben az 5. címkén.

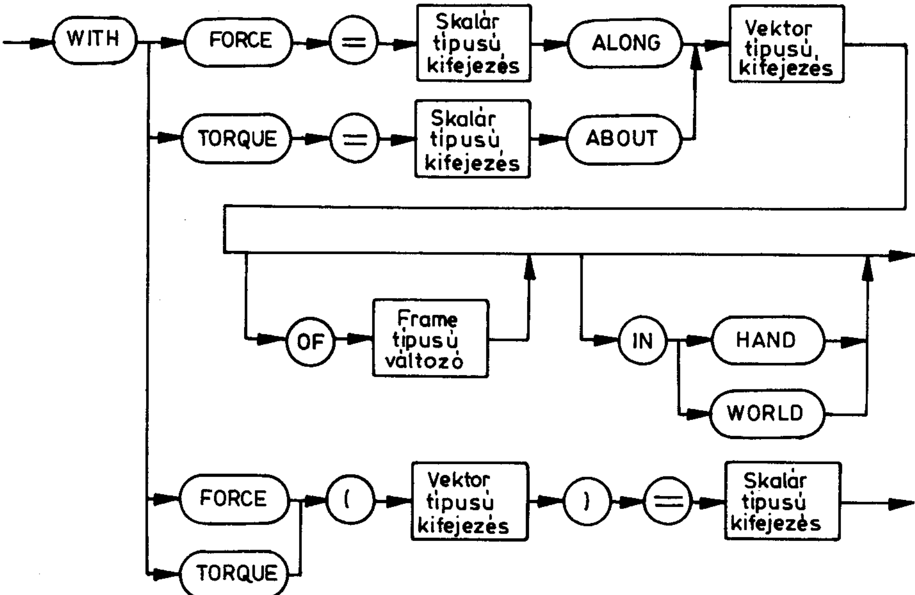
Az érzékelők segítségével végrehajtott ún. érzékelővezérelt irányítás paramétereit a ROBEX nyelvben nem lehet a pozicionáló utasításon belül explicit módon megadni. A VAL nyelv adottságaihoz hasonlóan azonban alkalmas hardver-eszközzel itt is lehet programmegszakítást szervezni, amelynek hatására a kar mozgása leállítható (l. a 4.2.6. pontot).

4.2.5.2. A mozgásvezérlő utasítások paramétereinek szabályozása az érzékelők jelei alapján

Különféle megmunkálási feladatoknál, így pl. köszörülésnél, sorjázásnál, fúrásnál, ragasztásnál szükségünk lehet arra, hogy a szerszám vagy a megfogóeszköz mozgását úgy vezéreljük, hogy éppen az előre megadott erő ébredjen a végrehajtó szerv és a munkadarab, ill. az alátámasztás között. A végrehajtó szerven elhelyezett erőmérő cella jelének figyelembevételével a vezérlés annak megfelelően gyorsítja vagy lassítja a robot mozgását, hogy éppen az előírt erőhatás lépjen fel, és a mért erő e körül az érték körül hozzávetőlegesen állandó maradjon.

A tárgyalt nyelvek közül csak az AL-ban létezik olyan mozgásvezérlő utasítás, amelyben erő- és nyomaték-alapjelek is megadhatók. Hasznos lenne továbbá – pl. hegesztési feladatok esetén –, ha egy megadott irányban a munkadarabtól mérve állandó távolságot lehetne tartani. Ehhez egy távolságmérő eszköz által szolgáltatott jeleket kellene felhasználni. Ilyen megoldás esetén akármilyen görbült felület mentén végig lehet haladni.

A robotkarnak az erő- és nyomatékalapjelek segítségével, valamint érzékelők jelei alapján szabályozott vezetése azonos szintaktikai szerkezetű utasítással történik, mint a 4.2.5.1. alatt tárgyalt mozgásvezérlő utasítás (l. *4.33. ábra*). Az a frame-rendszer, amelyre az erőkomponensek irányait vonatkoztatjuk, ismét csak a Force-Frame-paraméterrel definiálható (l. a *4.38. ábrát*). A HAND/WORD, azaz a kézkoordináta-rendszer/környezeti koordináta-rendszer kijelölésével kapcsolatban a 4.2.5.1.-ben mondottak érvényesek.



4.38. ábra. Erő- és nyomatékparaméterek megadása az AL nyelvben

AL:

```
MOVE köszörű TO bal oldal
  WITH FORCE(XHAT) = 500 * GM;
```

A munkadarab felületének csiszolására alkalmas csiszolóberendezés a korábban definiált Force-Frame x irányában 500 g nyomóerőt gyakorolva jobbról balra halad.

AL:

```
MOVE furo TO lefele
  WITH FORCE = 700 * GM ALONG ZHAT
    OF furohegye IN HAND
  WITH FORCE = 0 * GM ALONG XHAT
  WITH FORCE = 0 * GM ALONG YHAT;
```

A *4.39. ábrán* bemutatott viszonyoknak megfelelően a robot úgy vezeti az előre befogott speciális fúrót, hogy a fúrás 700 g nyomóerővel történjen. Az erő irányát a fúróhoz viszonyítva definiáltuk, így mindig a fúró irányába eső erőkomponenst vesszük figyelembe.

4.2.6. Mozgásvezérlő utasítások eseményfelügyelet mellett

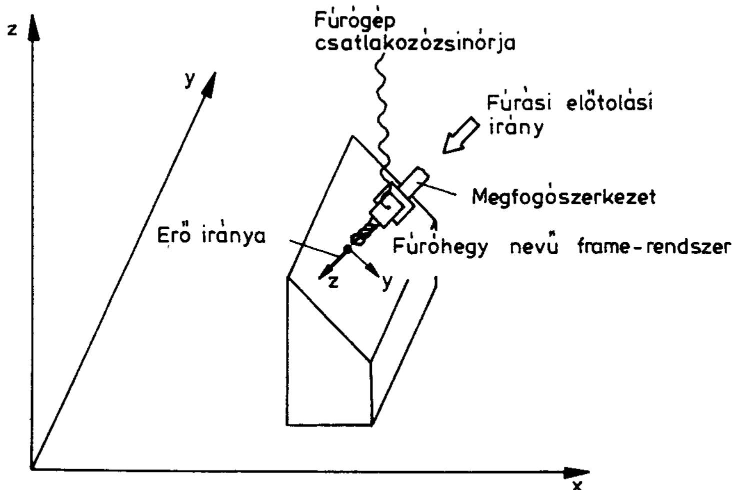
Az ipari robotokkal többnyire egy adott körzeten belül végeztetünk műveleteket. Ezen a területen még további robotok, gépek, szállítóberendezések is működhetnek, így szükség lehet rá, hogy bizonyos külső eseményeket ne csak lekérdezni tudjunk, hanem azok közvetlenül a robot mozgása közben befolyásolhassák a program alakulását. Tehát pl. adott esetben meg kell állítani a robotkart, mihelyt a megfogószerkezet valamelyik gép fénysorompóját takarja, ugyanakkor várnia kell a gépre, míg az kiadja a munkadarabot. A mozgás megszakítását valamilyen adott esemény bekövetkezésekor több nyelvben meg lehet valósítani, azonban egyes nyelveken – így az AL-ban és a HELP-ben – lehetőség van arra is, hogy az esemény bekövetkezésekor bizonyos programrészleteket időben párhuzamosan futtathassunk a mozgásvezérléssel.

Az AL nyelv karlsruhei változatában a mozgásvezérlő utasításon belül az eseményfelügyelet szervezését a *4.40. ábra* szemlélteti. Itt az eseményeket esemény típusú változókkal reprezentáljuk (alapszava: EVENT, l. a 3.1. és 4.5.3. szakaszt, ill. pontot). Az ilyen esemény típusú változót egy párhuzamos szervezésű blokk SIGNAL-utasításának segítségével állítjuk be (l. a 7.1. szakaszt). A külső eseményeket előre rögzített, definiált eseményváltozó-névvel ábrázoljuk (pl. kapcsoló). Az esemény bekövetkezésekor elvégzendő műveletre fennáll a 4.2.5.1. pontban leírt és a *4.33. ábrán* bemutatott szervezési struktúra.

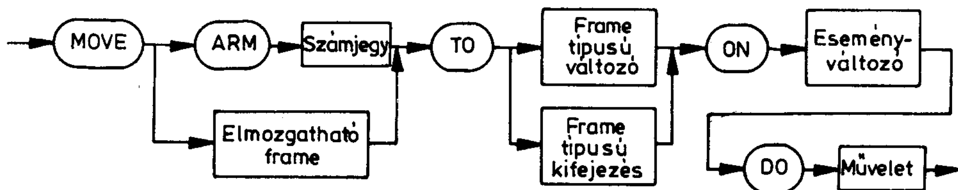
A *4.4. programrészlet* az eseményfelügyelet technikáját világítja meg. Itt alkalmaztuk a 7.1. szakaszban részletesen tárgyalt párhuzamos szervezésű blokkokat és ezek szinkronizálását.

A COBEGIN és a COEND attribútumok között felsorolt blokkok időben párhuzamosan vagy legalábbis látszólag párhuzamosan kerülnek végrehajtásra. A blokkok legelején ARM1 megfogja egy kötél bal oldali, ARM2 a jobb oldali végét. Ezután elkezdenek egymástól távolodni. Mihelyt az ARM1 megfogószerkezetén elhelyezett erőmérő eszköz 300 g határértéknek megfelelő erőt érzékel, akkor beállítja az első blokk **„arm2halt"** nevű eseményváltozójának értékét, és megállítja ARM1-et.

Ha az ARM2-n nem helyezünk el erőmérő eszközt, akkor ennek megállításáról az **„arm2halt"** nevű eseményváltozó segítségével kell gondoskodnunk. Mindkét robotkar teljes megállása után lehet folytatni a programot. Mármost a második kar beállításáról



4.39. ábra. Fúrási művelet előírt nyomóerő alkalmazásával AL nyelvben



4.40. ábra. Mozgásvezérlő utasítás az AL-ban eseményfelügyelet mellett

```
COBEGIN
  BEGIN "ARM1 mozgásai"
        :                         (* Megfogja a kötél                  *)
                          (* bal oldali végét                          *)
    MOVE ARM1 TO bal felé húz
      ON FORCE (XHAT) > = 300 * GM DO
      BEGIN
        SIGNAL arm2halt;
        STOP ARM1;
      END;
    SIGNAL arm1kész;
    WAIT arm2kész;
        :
    END "ARM1 mozgásai"
    BEGIN "ARM2 mozgásai"
        :                         (* Megfogja a kötél                  *)
                          (* jobb oldali végét                         *)
      MOVE ARM2 TO jobb felé húz
        ON arm2halt DO STOP ARM2 ;
      SIGNAL arm2kész;
      WAIT arm1kész ;
        :
    END "ARM2 mozgásai"
COEND
```

4.4. programrészlet. Mozgásvezérlő utasítások az AL nyelvben eseményfelügyelet mellett

az ARM1 működtető utasításának műveletleíró részében is lehetne rendelkezni, pl. a következőképpen:

```
MOVE ARM1 TO balfeléhúz
  ON FORCE(XHAT) ≧ 300 * GM DO
    BEGIN
    STOP ARM1;
    STOP ARM2
    END;
```

Ekkor azonban a két robotkart nem tudnánk egyidejűleg leállítani, és így ARM2 tovább feszíthetné a kötelet, miközben ARM1-et már lefékeztük.

Ez a megoldás egyébként programszervezés és stílus szempontjából sem elég jó, hiszen a második blokk mozgásvezérlő utasításából nem lehet megállapítani, hogy azt az első blokk felfüggesztheti, ill. a mozgást befejezteti.

Az amerikai Stanford University-n alkalmazott AL változatban a logikai kifejezések is ugyanúgy kezelhetők, mint az események.

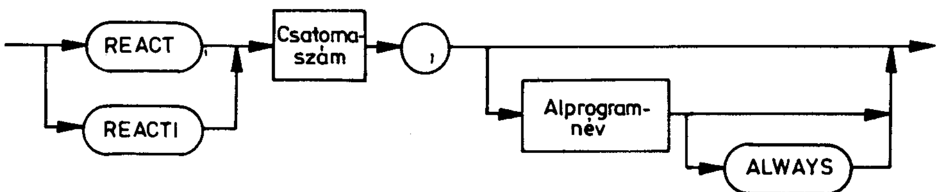
AL (Stanford Uni):

```
MOVE YARM TO célframe
  ON A>5.2 DO
    BEGIN
    PRINT ("elérte a szintet");
    STOP
    END;
```

Itt az YARM = Yellow Arm (azaz „sárga kar"-t jelent). Ennek az utasításnak az a hatása, hogy az A>5.2 kifejezés 100 ms-onként mindaddig kiértékelésre kerül, amíg értéke FALSE. Mihelyt A nagyobbá válik, mint 5.2, a program kiadja az „elérte a szintet" üzenetet, majd megállítja a kar mozgását.

Mármost az az időpont, amikor A éppen nagyobbá válik 5.2-nél a paralel folyamatok utasításainak végrehajtási időpontjaitól függ. Tehát attól, hogy éppen hol változtatjuk A értékét, és így függ a fordítóprogram implementációjától, a számítógép típusától (pl. az aritmetikától stb.). Ezért az utasítás tényleges hatása a program írásakor sokszor nem látható előre. Ez az az ok, ami miatt az eseményként kezelhető logikai kifejezések használatának lehetősége nincs beépítve a karlsruhei AL-implementációba.

A VAL nyelvben le lehet kezelni olyan külső jeleket, amelyek a robot mozgása közben futnak be. Ezt a REACT és a REACTI utasításokkal oldjuk meg. Az utasítások szintaktikai sémája a *4.41. ábrán* látható. A REACT utasítás hatására a gép a robot soron következő művelete alatt folyamatosan lekérdezi a megadott bemenőcsatornát, hogy a logikai „igen" feszültségszint észlelhető-e. Ha ez a kérdéses művelet időtartama alatt



4.41. ábra. Külső megszakításkérelemre végrehajtott rutinhívás deklarációja VAL nyelvben

bekövetkezik, akkor aktivizálódik a megadott alprogram, majd ennek befejeztével a mozgásvezérlő utasítást követő utasítással folytatódik a program. Ha alprogramot nem adunk meg, akkor az éppen aktuális utasítást követő utasításra ugrik a program. Az ALWAYS kiegészítő megjegyzés hatására a megadott bemeneti csatornát a gép nemcsak a soron következő robotművelet időtartama alatt, hanem a következő IGNORE-utasítás megjelenéséig folyamatosan lekérdezi, és a logikai „igen" feszültségszint beérkezésekor aktivizálja is a megadott alprogramot.

VAL:
REACT 2, megszakítás
MOVE lerakóhely
Ha az alatt az idő alatt, amíg a robot a **„lerakóhely"** frame-rendszeréhez eljut, a 2-es bemeneti csatornán megfelelő jelzés érkezik, akkor a **„megszakítás"** nevű alprogram aktivizálódik.

A REACTI-utasítás annyiban különbözik a REACT-utasítástól, hogy azt az utasítást, amelynek végrehajtása közben a megszakítást előidéző jel beérkezik, a gép azonnal megszakítja és a kijelölt alprogramot azonnal elvégzi.

VAL:
REACTI2, megszakitjelz ALWAYS
Amikor a 2-es csatornán megszakításkérelem fut be, akkor a gép felfüggeszti az éppen futó parancs végrehajtását, és aktivizálja a **"megszakitjelz"** nevű alprogramot. Ez az eljárás az adott csatornán minden egyes megszakításkérelem beérkezésekor végrehajtásra kerül egészen az IGNORE-utasítás megjelenéséig.

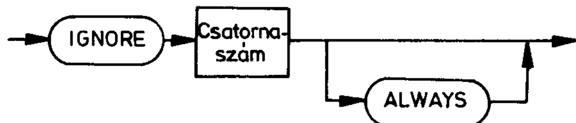
Az IGNORE-utasítás struktúráját a *4.42. ábra* szemlélteti. Ez megszünteti a REACT-, ill. a REACTI-utasítás hatását. Az ALWAYS attribútum megadásakor az IGNORE-utasítás hatása folyamatosan megmarad, egyébként csak a következő mozgásvezérlő utasítás végéig maradna érvényben.

VAL:
IGNORE2 ALWAYS
Hatására a 2-es csatornát többé nem kérdezi le a program.

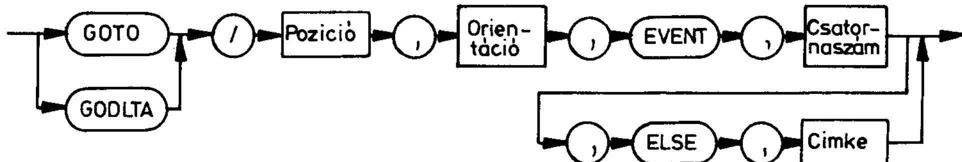
Ha tehát egy külső esemény miatt meg kell állítanunk a robotkart, akkor ezt a *4.5. programrészletben* bemutatott módon tehetjük meg.

Az IGNORE-utasítás a 2-es csatorna lekérdezését a MOVE-utasításra korlátozza. A „dummyprogram" csak egy komment-sorból és a visszatérési utasításból áll. A RETURN, ill. RETURN0 helyett álló RETURN1-gyel azt érjük el, hogy nem a megszakított mozgásvezérlő utasítást folytatjuk, hanem a visszatérési pont utáni IGNORE-utasítást hajtjuk végre.

A HELP-ben lehetőség van rá, hogy egy éppen futó mozgásvezérlő utasítás közben időben paralel módon további utasításokat hajtsunk végre. Ezért explicit lekérdezési ciklus és programelágaztatás segítségével eseményfelügyeletet szervezhetünk. Az események programból is beállíthatók más taskokban – ilyenkor szemaforoknak tekinthetők –, de külső megszakításkérések is lehetnek. A *4.6. programrészletben* ennek programozástechnikai megoldását mutatjuk be.



4.42. ábra. Egy eseményfelügyelet letiltása VAL-ban



4.43. ábra. Eseményfelügyelettel szervezett mozgatás ROBEX-ben

```
REACT 2, dummyprogramm
MOVE célframe
IGNORE 2 ALWAYS
     :
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
REM dummyprogram
RETURN 1 ;
```

4.5. programrészlet. Mozgásvezérlő utasítás eseményfelügyelettel VAL-ban

```
                SMOVE (1, 2, 150);
ciklus: IF      IF TESTB (5) THEN
                  :     (* Az 5-ös csatornán beérkező eseményt       *)
                        (* követően a robotmozgatással para-         *)
                        (* lel végrehajtott utasítások               *)
                  EOM (1)
                END;
                COORD (1);
                IF AY (1) > 149 THEN
                  GOTO ciklus
                END;
                EOM (1);
```

4.6. programrészlet. A robot mozgása közben végrehajtott eseményfigyelés HELP nyelvben

Az 5. csatorna lekérdezése után beolvassuk az aktuális robotkoordinátákat, majd megvizsgáljuk, hogy a kar elérte-e az előírt végpontot. Ha nem, akkor feltétel nélküli ugrás következik a **„ciklus"** nevű címkére, majd az egész eljárás elölről kezdődik. Közvetlenül a pozicionálás befejezése előtt az EOM-utasítás hatására a program várakozni kezd. Megvárja, hogy a robot valamennyi tengelyéről visszajelzés érkezzen, és csak ezután halad tovább. Ha az 5. csatornán eseményjelzés érkezik, akkor az az eset fordulhatna elő, hogy az IF-utasításon belül a megfelelő utasításokon többször is végigfut a program. Ezt ugyancsak egy EOM-utasítás akadályozza meg az 1 jelű robotra.

A SIGLA-ban nem lehet eseményfelügyeletet megvalósítani a mozgatás közben. A WZL Aachen Intézet tájékoztatása szerint még nem dőlt el véglegesen, hogy a ROBEX-ben milyen legyen az eseményfelügyelettel szervezendő mozgásvezérlő utasítás szemantikája. A szintaxist azonban a *4.43. ábrának* megfelelően már definiálták.

Ha az ELSE-ág hiányzik, akkor a robot működtetése közben azt ellenőrzi, hogy fellépett-e esemény a megadott input-csatornán. Ha ez bekövetkezik, leállítja a robot mozgását. Ha az utasításban az ELSE-ágat is megadjuk, akkor a mozgás befejeztével a program a megadott címkére ugrik, ha egyébként esemény nem volt.

ROBEX:

```
GOTO/p3,XYROT,180,EVANT,2,ELSE,nobox
```

Az ipari robot először beáll a p3 pozícióra, a megfelelő orientációt is tartva. Eközben figyeli a 2-es csatornát, hogy fellép-e valamilyen esemény, ill. befut-e megszakításkérelem. Ha ez bekövetkezik, akkor a robot megáll, és a program következő utasítását hajtja végre. Ha pedig nem, akkor a program a nobox címkétől folytatódik.

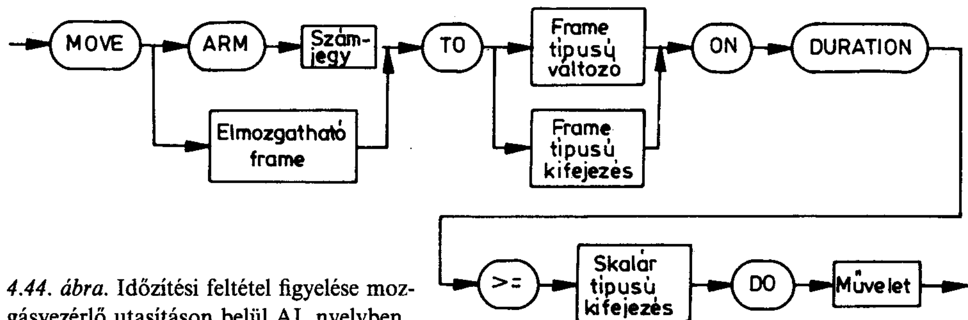
4.2.7. Mozgásvezérlő utasítások időellenőrzés (időfelügyelet) mellett

Bizonyos robotmozdulatoknál fontos szempont, hogy a mozgás adott időtartamon belül befejeződjön, mert ellenkező esetben valamilyen szállító vagy egyéb berendezéssel már nem lehet koordinálni az egymással összefüggő műveleteket. Ilyenkor a robot mozgását vezérlő utasítást egy időzítési feltétel ellenőrzésével köthetjük össze. AL-ban ezt arra is fel lehet használni, hogy a mozgatás kezdetétől számítva, adott idő elteltével valamilyen programrészletet a robot mozgásával párhuzamosan hajtassunk végre. Az időzítési feltétel figyelésének szintaktikai struktúráját a *4.44. ábra* mutatja. A robot mozgásának kezdetekor a program beállítja egy visszaszámláló kezdő értékét, és indítja azt. A visszaszámlálás befejezése kivált egy előírt akciót, hacsak a robot mozgása már korábban be nem fejeződik.

AL:

```
MOVE ARM1 TO célframe
  ON DURATION ≧ 3 * SEC DO
    BEGIN
      MOVE ARM2 TO centrum
        WITH DURATION = 2 * SEC;
      PRINT ("Folyamat kezdete");
      alfa←xrotacio * SIN(30 * szamlalo)
    END;
```

Az ARM1 kar mozgásának megkezdése után 3 s-mal a második kart is elindítjuk 2 s időtartamra. A második kart a **"centrum"** nevű frame-re pozicionáljuk. Megjelenítünk még ezenkívül egy üzenetet, és kiszámítunk egy képletet.



4.44. ábra. Időzítési feltétel figyelése mozgásvezérlő utasításon belül AL nyelvben

A *4.7. programrészletben* bemutatott szervezési struktúrának megfelelően a HELP mozgásvezérlő utasításában explicit módon megadható az időzítési feltétel. A TIME segítségével lekérdezhető a rendszer belső órája (másodpercben kifejezve). Ennek segítségével a robot mozgásával párhuzamosan egy ciklusban állandóan összehasonlítható az indulási időpont és a pillanatnyi időpont d*t* különbsége az előre megadott időtartammal.

Mihelyt ez az időkülönbség nagyobb vagy egyenlő az előre adott időtartammal, az IF THEN ág utasításai kerülnek végrehajtásra és a szervezéstől függően megállítható a robot mozgása is.

```
            időtartam := 3;
            SMOVE (1, 2, 120);
            kezdidőpont := TIME ( );
            ciklus:           := TIME( )- kezdidőpont;
            IF dt = időtartam OR dt > időtartam THEN
            :       (* Itt jönnek azok az utasítások,              *)
                    (* amelyek a 3 sec eltelte után esedékesek,    *)
                    (* pl. HALT (1), a kar megállítása cél-        *)
                    (* jából                                       *)
              EOM(1) (* várakozik a pozicionálás befejezésére      *)
            ELSE
              COORD (1)
              IF AY (1) < 119.9 THEN
               GOTO ciklus
            END
END;
```

4.7. programrészlet. Egy pozicionálás időzítési feltételének explicit ellenőrzése HELP-ben

4.2.8. A robot alaphelyzetbe állítása[4.3.]

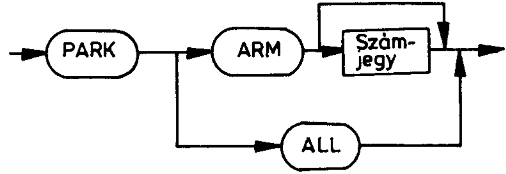
Több robotrendszernél létezik egy kitüntetett pozíció (esetenként orientáció is), amelybe a robotot egyetlen rövid utasítással tetszőleges pozícióból és orientációból vissza lehet állítani. Ez a nyugalmi helyzet hardver-eszközökkel is beállítható: pl. ha a robot minden tengelyénél végálláskapcsolók vannak felszerelve. Szoftveres úton a robot tengelykoordinátáinak rögzített értéktáblázat szerinti beállításával érhetjük el a nyugalmi helyzetet.

Nyugalmi (vagy alap-) helyzetben a kar csaknem minden rendszernél felfelé mutat. Így a munkaterületen dolgozó szerelők jobban hozzáférnek a berendezésekhez és ütközés veszélye nélkül lehet működtetni más gépeket is. A tárgyalt nyelvek közül csak az AL, a VAL és a ROBEX rendelkezik a nyugalmi helyzet gyors beállítására alkalmas utasításokkal. Az AL-nál és a VAL-nál a kar mindig felfelé mutat. A beállító utasítások a *4.45.*, ill. *4.46. ábra* szerinti egyszerű felépítésűek.

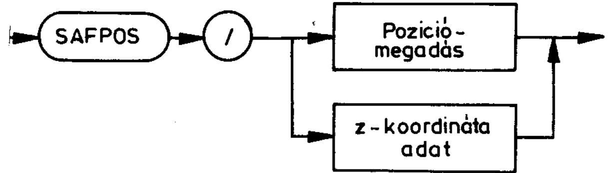
AL:
```
PARK ARM1;
```
VAL:
```
READY
```

4.3. Tulajdonképpen itt egy ún. nyugalmi helyzetbe való állításról van szó, ez azonban az esetek döntő többségében megegyezik az alaphelyzettel.

4.45. ábra. A nyugalmi helyzet beállítása AL-utasítással



4.46. ábra. A robot nyugalmi helyzetbe állítása VAL-utasítással



4.47. ábra. A robot nyugalmi helyzetének definiálása ROBEX-ben

ROBEX-ben definiálni lehet tetszőleges nyugalmi helyzetet (l. *4.47. ábra*). Ilyenkor vagy a kívánt pozíció x, y és z koordinátáit adjuk meg, vagy pedig csak a z koordinátáját. Ez utóbbi esetben a robot működési terében egy bizonyos biztonsági sávot határozunk meg, ahova a robot visszahúzódik. Eközben a robot megtartja pillanatnyi x és y koordinátáját.

A robotot egy egyszerű GOTO utasítással pozicionálhatjuk a korábban definiált SAFP nyugalmi helyzetbe.

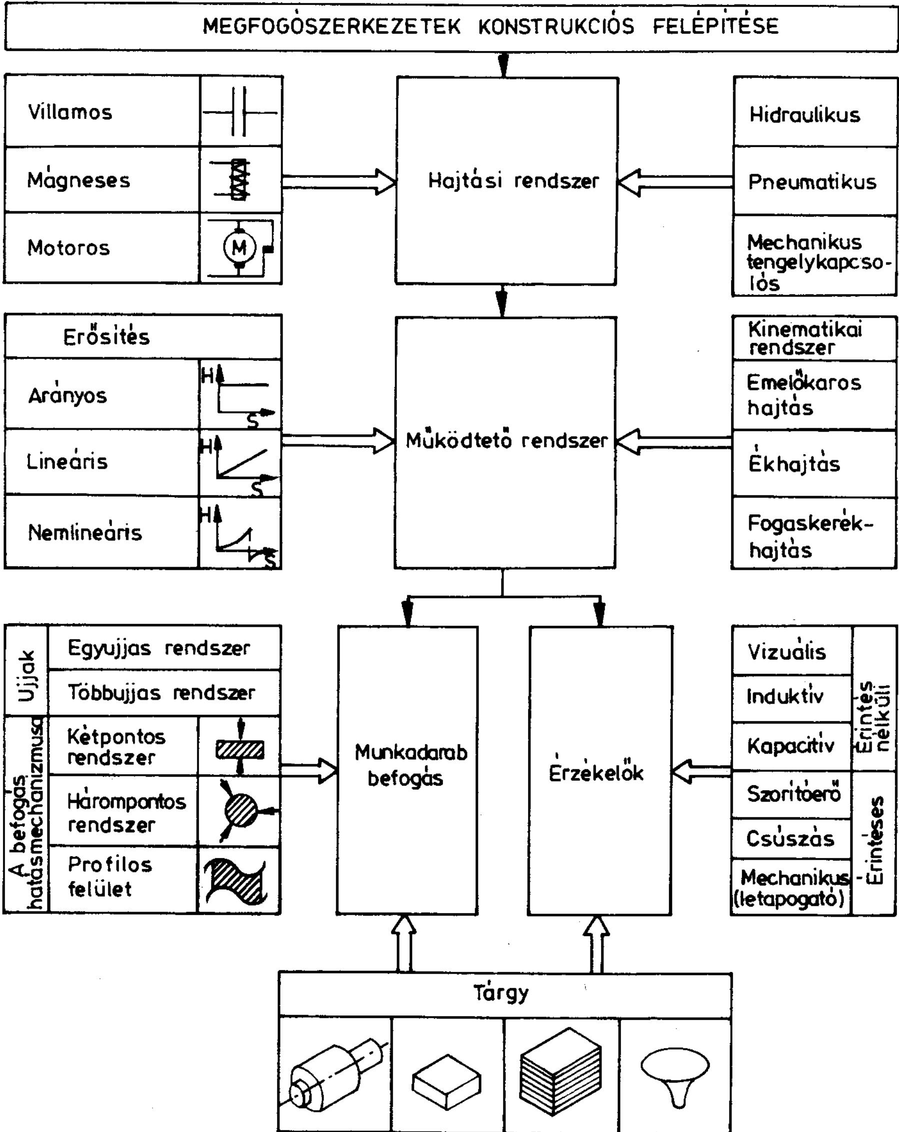
ROBEX:

```
SAFPOS/50,20,35
:
GOTO/SAFP
```

4.3. A végrehajtó szervekre vonatkozó utasítások

A robotkar végén megfogószerkezet, szerszám vagy valamilyen megmunkáló- vagy gyártóeszköz helyezhető el. Ezeket összefoglaló néven végrehajtó szerveknek, más néven effektoroknak nevezzük. Bár a végrehajtó eszköz a robotberendezés szerves része, önmagában mégis ismét csak olyan egységekből áll, mint pl. a mechanikai rendszer, a hajtás, a kinematikai és a vezérlőrendszer (l. *4.48. ábra,* és LUNDSTRÖM [4.3.]). A szerelőrobotoknál általában megfogószerkezetet használnak. Ezek működtetésére a robotvezérlő nyelvek egy részének – így az AL-nak, a VAL-nak és a ROBEX-nek speciális utasításai vannak. Más nyelveknél – így a HELP-nél és a SIGLA-nál – a megfogószerkezet – ugyanúgy, mint a szerszámok és megmunkálóeszközök – a periferiális segédberendezések közé számít, így ezeket a kiadott vezérlőjelekkel irányítják. A végrehajtó eszközökkel elvégzendő műveletek programozásához pontosan ismernünk kell az eszköz kinematikai

és hajtási rendszerét, valamint az eszköz működési módját, hiszen pl. egy pneumatikus megfogószerkezet csak nyitni vagy zárni képes, míg a villamos eszközök nyitását tetszőleges mértékben lehet állítani, sőt adott esetben meghatározott erővel is lehet fogni velük a munkadarabot. Más példa: egy háromujjas megfogószerkezettel hengeres alkatrészeket biztosabban meg lehet fogni, mint kétujjassal.



4.48. ábra. A különféle megfogószerkezetek szerkezeti kialakítása AUER [4.4] nyomán

4.3.1. Egyszerű effektorvezérlő utasítások

Az egyszerű effektorvezérlő utasítások segítségével a megfogószerkezet nyitását vagy zárását, a tartómágnes ki- és bekapcsolását vagy más ezekhez hasonló egyszerű műveleteket hajthatunk végre. Ezek az egyszerű utasítások nem alkalmasak arra, hogy speciális paramétereket is előírhassunk velük (pl. a megfogóeszközzel kifejtett szorítóerőt, a nyitás vagy zárás mértékét stb.). Ilyen egyszerű utasításokat természetesen olyankor alkalmazunk, amikor maga a végrehajtó eszköz is valamilyen egyszerűbb készülék, másképp kihasználatlanok maradnának az effektor-rendszer lehetőségei.

Az AL nyelvben nem létezik olyan egyszerű effektorvezérlő utasítás, amelyben ne kellene valamilyen paramétert megadni. Legalább a megfogószerkezet nyitásának vagy zárásának mértékét elő kell írnunk (l. a 4.3.2. pontot). Ez természetesen villamos hajtású megfogóeszközre vonatkozik. Pneumatikus működtetésű szerszám esetén az interpreternek, ill. a vezérlőrendszernek kell egy olyan moduljának lennie, amely a megfogószerkezet nyitását, ill. zárását irányítja, ill. ennek irányításáról dönt. Programozástechnikai oldalról nem lehet ezt a döntést egyszerűen azzal elintézni, hogy az OPEN utasítás a szerszám nyitását, a CLOSE pedig a zárást jelenti, hiszen pl.:

az OPEN HAND TO 5 * CM;

és a CLOSE HAND TO 5 * CM;

utasítások hatása teljesen azonos, ha megelőzően a befogószerszám teljesen zárt állapotban volt. A szerszám ugyanis mindkét esetben nyit.

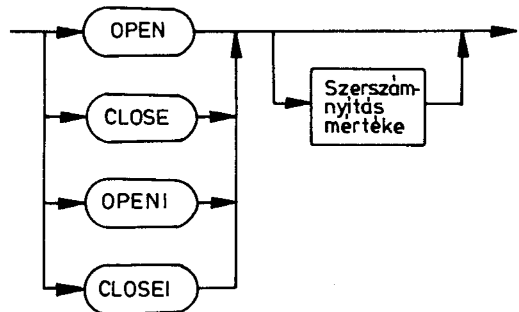
A VAL nyelvben az OPEN- és a CLOSE-utasítások mind villamos, mind pedig pneumatikus hajtású megfogószerkezet működtetésére alkalmasak (l. *4.49. ábra*). Az egyszerű effektorvezérlő utasításoknál elhagyható a szerszámnyitás mértékének megadása.

Az OPEN- és CLOSE-utasításokat a robot csak az éppen soron következő pozicionálási művelet során hajtja végre, míg az OPENI- és CLOSEI-utasításokat azonnal végrehajtja.

VAL:

```
OPEN
MOVE megfogframe
CLOSEI
DEPART 80
```

A "megfogframe"-re történő pozicionálás közben a penumatikus megfogószerkezet kinyit, majd z irányú eltávolodása előtt bezár.



4.49. ábra. A megfogószerkezet nyitása és zárása VAL-ban

A RELAX-utasítással a pneumatikus megfogószerkezetről lekapcsolható a működtető sűrített levegő.

HELP-ben nincsenek speciális effektorvezérlő utasítások. Itt a végrehajtó eszközöket a SET, RESET és PULSE output-utasítások segítségével lehet működtetni.

HELP:

```
SET(5);
DELAY(20);
MOVE(1,#1, 500, #3, 40);
RESET(5);
```

Itt az 5. csatornán bekapcsoljuk a robotra felszerelt elektromágnest. A mágnes bekapcsolása után 4 s-ot várunk, hogy a mágnes biztonságosan tartsa az alkatrészt. Ezután a robotkart az x = 500 mm és z = 40 mm koordinátájú pontba visszük, majd ott elengedjük az alkatrészt.

HELP:

```
PULSE(1);                 ! Zárja a megfogószerkezetet
DELAY(effektoridő);       ! Várakozik a zárás befejezésére
MOVE(1,...);              ! A robotkar pozicionálása
PULSE(2);                 ! A szerszám nyitása
DELAY(effektoridő);       ! Megvárja a szerszámnyitás befejezését
```

A PULSE-utasítás a megadott számú csatornára vezérlőimpulzust ad ki.

A SIGLA nyelvben nincsenek effektorvezérlő utasítások. A megfogószerkezet vagy befogópofák kinyitásáról és összezárásáról az AX-utasítás segítségével kell gondoskodnunk (l. *4.50. ábra*).

Itt „ – ”-jellel a készülék bekapcsolását, a „ + ”-jellel pedig a készüléknek a működtető erőforrásról való lekapcsolását jelöljük.

SIGLA:

```
AX/–1,–12;      A megfogószerkezet nyitása
MO/...;         A robot pozicionálása
AX/1;           A megfogószerkezet összezárása
```

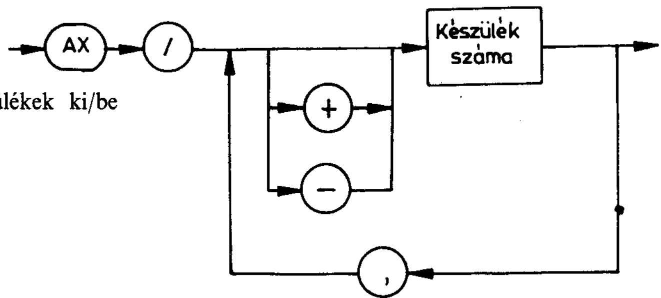
A ROBEX-ben előre ki lehet jelölni azt az effektort, amelyre a következő effektorvezérlő utasítások vonatkoznak (l. *4.51. ábra*). A csatorna számával megadott megfogóeszköz, ill. szerszám az OPENGR, ill. a CLOSEGR-utasítással vezérelhető, de csak akkor, ha ezekben az utasításokban nem adunk meg külön csatornaszámot (l. *4.52. ábra*).

Pontos szemantikai értelmezés hiányában csak azt mondhatjuk, hogy a pneumatikus megfogószerszámokat OPENGR-utasítással nyitjuk és CLOSEGR-utasítással zárjuk.

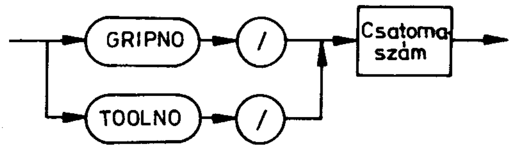
ROBEX:

```
GRIPNO/5
OPENGR
CLOSEGR/3
```

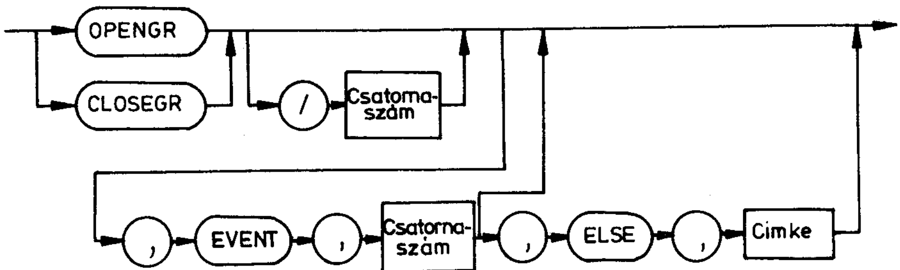
Az 5. output-csatornán levő szerszámot nyitjuk és a 3.-on levőt zárjuk.



4.50. ábra. Perifériális készülékek ki/be kapcsolása SIGLA nyelvben



4.51. ábra. A megfogószerkezet vagy szerszám előzetes beállítása ROBEX-ben



4.52. ábra. Effektor nyitása és zárása ROBEX-ben

4.3.2. Paraméteres effektorvezérlő utasítások

A mozgásvezérlő utasításokhoz hasonlóan (l. a 4.2.4. pontot) a végrehajtó eszközök (ún. effektorok) vezérlésekor is olyan paraméterekkel szeretnénk kiegészíteni a programbeli utasításokat, amelyekkel a végrehajtó eszköz vezérlésének sok részletét befolyásolni tudjuk. Ezek a paraméterek meghatározhatják

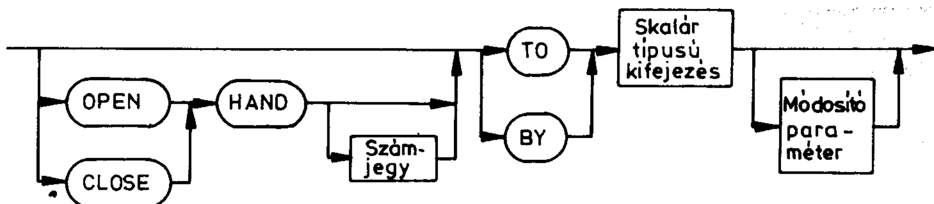
- a megfogószerkezet helyzetét;
- az effektor működési időtartamát;
- az effektor működésének sebességét;
- a megmunkálási tűréseket;
- technikai segédberendezések paramétereit, pl. a hegesztési áramerősséget stb.

Ezeket a paramétereket többnyire a vezérlőrendszer állítja be, nem a programozó.

Néhány robotvezérlő programozási nyelvben azonban lehetőség van arra, hogy kizárólag a megfogószerkezetekre megadhassuk a beállítani kívánt nyílásméretet két (esetleg több) befogópofa között.

A *4.53. ábrán* bemutatjuk, hogy hogyan adhatjuk meg AL-utasításokkal villamos, ill. hidraulikus hajtású megfogószerkezetek abszolút, ill. relatív nyílásának méretét. (Az abszolút nyílásméretre TO szóval, a relatív nyílásméretre BY szóval utalunk.)

Ha az OPEN, ill. a CLOSE HAND előírást elhagyjuk, akkor a TO, ill. BY megjelölés a legutolsó OPEN, ill. CLOSE-utasításra vonatkozik.



4.53. ábra. Effektorvezérlő utasítás AL-ban

AL:

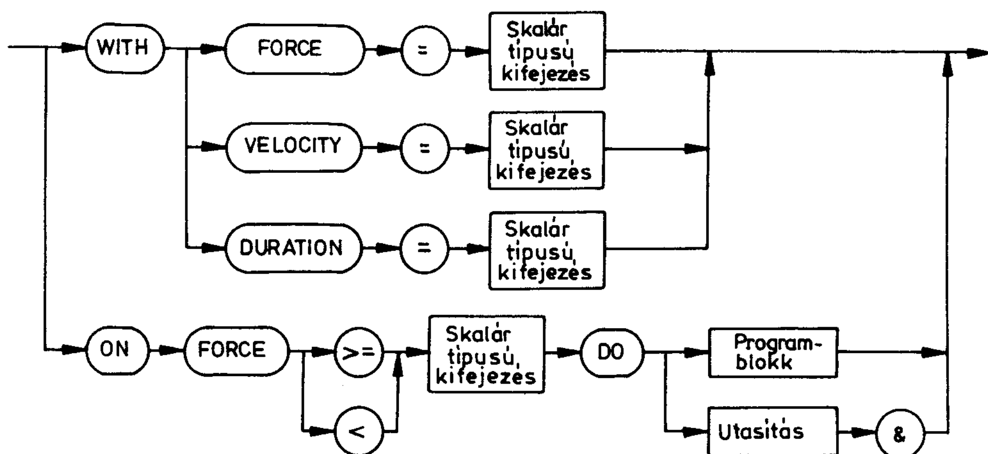
OPEN HAND1 TO 8 * CM;	(* Nyitás 8 cm-re *)
MOVE ARM1 TO célframe;	
CLOSE HAND1 BY 5 * CM;	(* A szerszám nyílásmérete *) (* most 3 cm lesz *)
MOVE ARM1 TO @ + ZHAT;	
TO 2 * CM;	(* HAND1-et 2 cm-re zárja össze *)

A 4.49. ábra VAL nyelvű utasítást mutat be effektor vezérlésére. Milliméterben kifejezve megadtuk a megfogószerkezet nyitásának mértékét, az ún. *nyílásméretet.*

VAL:
OPEN 45
A megfogószerkezet szétnyitása 45 mm-re. VAL-ban csak abszolút nyílásméret adható meg.

A HELP, a SIGLA és a ROBEX nem biztosít paramétermegadási lehetőséget.

Az AL nyelvben elő lehet még írni a szerszám nyitási és zárási sebességét, valamint a nyitás és az összezárás időtartamát is. (Megfogáskor, a szorítóerő megadásával a következő pontokban foglalkozunk részletesen.) A *4.54. ábrán* bemutatott színtaxisdiag-



4.54. ábra. A szerszám által kifejtett szorítóerő megadása és szabályozása AL nyelven felírt effektorvezérlő utasításban

ram az AL nyelv karlsruhei implementációjának egy bővített változatából való. E téren az újonnan kifejlesztett effektorvezérlő rendszerrel bővített karlsruhei hardver sok tekintetben új programozási lehetőségeket nyújt.

AL:

OPEN HAND TO 5 * COM
  WITH VELOCITY = 0.5 * CM/SEC;
A megfogószerkezet nyitása adott sebességgel BY 2 * CM
  WITH DURATION = 5 * SEC;
Az eszköz nyitása/zárása 2 cm-nyivel 5 s alatt.

4.3.3. Az érzékelők jeleinek felhasználása effektorvezérlő utasításoknál

Egyes nyelvekben előírhatjuk, hogy a rendszer úgy hajtsa végre az effektor vezérlését, hogy az érzékelők jeleinek felhasználásával ellenőrizze is annak működését. Szigorúan véve ide sorolhatnánk az előző pontban tárgyaltak közül azt a vezérlési módot is, amikor a megfogószerkezet nyílásméretét is előírtuk. Ez ugyanis általában csak úgy oldható meg, ha a megfogószerkezeten helyzetérzékelő van elhelyezve és ez jelzi vissza a megfogónyílás tényleges mértékét.

AL-ban a mozgásvezérlő utasításokhoz hasonlóan az effektorvezérlő utasítások is megadhatók olyan formában, melyben előírjuk, hogy a rendszer a művelet végrehajtását az erőmérő érzékelők jeleinek határérték-figyelésével, tehát ellenőrzötten hajtsa végre (l. *4.54. ábra*). Az ellenőrzés alapelve az, hogy ha a mért szorítóerő az előírt határértéket túllépi, akkor a vezérlőrendszer a programban a kijelölt utasítássorozatnak megfelelő akciókat hajtja végre.

AL:

CLOSE HAND TO 2 * CM
  ON FORCE ≧ 100 * GM DO
    STOP HAND &;
Ha a szorítóerő nagyobb mint 100 gm, akkor a robotkéz összezárása megáll. Az „&"-jel hatását a 4.2.5.1. pontban említettük.

A VAL nyelvben meg lehet oldani azt a feladatot, hogy a vezérlőrendszer ellenőrizze az alkatrész pontos megfogását. Ehhez a megfogó összezárását egy szabályozott szervomechanizmussal kell működtetni. A pofák a GRASP-utasítás hatására addig mozognak, míg a reakcióerőt érzékelve a rendszer meg nem állítja a motort. Ekkor a rendszer megméri a megfogónyílást, és összehasonlítja a megadott értékkel. Ha a mért érték kisebb mint az utasításban szereplő határérték, akkor a program a megadott címkére ugrik (l. a *4.55. ábrát*).

4.55. ábra. A megfogószerkezet érzékelővezérelt működtetése a VAL nyelv utasításával

VAL:

```
GRASP 14,8,200
```

Ha a 15 mm átmérőjű munkadarabot a megfogószerkezet nem tudta megfogni, akkor a program elágazik és a 200-as címkén folytatódik.

A HELP-ben, a SIGLA-ban és a ROBEX-ben nincs periferiális készülékek ellenőrzött működtetésére alkalmas utasítás.

Az AL nyelv CENTER-utasítása egy különleges effektorvezérlő, ill. pozicionáló utasítás (l. *4.56. ábra*). Ennek az utasításnak a hatására a robotkéz addig mozdul el a kézkoordináta-rendszer (megfogó-frame) Y irányába, amíg az egyik befogópofába épített érzékelő nem jelez. Ekkor a robotkéz elmozdulási iránya megfordul, és az addig megtett szakasz felével visszaáll, majd a robotkézen levő befogópofák közel ugyanakkora elmozdulással összébb zárnak. Ezután ugyanez a folyamat megismétlődik a szemben levő befogópofával is, és így tovább. Ha ez az iteratív összezárási folyamat már odáig jut, hogy az elmozdulás kisebbé válik egy adott alsó tűréshatárnál, akkor a robot mindkét befogópofát egyidejűleg működtetve megfogja a munkadarabot. A CENTER-utasítás tehát azt a feladatot oldja meg, hogy a robot úgy fogjon meg egy tárgyat a robotkézre szerelt megfogószerkezet pofáinak egyidejű zárásával, hogy a tárgyat még akkor se tolhassa el oldalra, ha a kéz nem tökéletesen központosan állt volna rá a tárgyra.

4.3.4. Szabályozott paraméterű effektorvezérlő utasítások

Ritkán, de lehet olyan alkalmazási problémát is találni, amikor az effektorvezérlés mellett még bizonyos paraméterek vezérlését is érzékelők segítségével, szabályozottan kell végrehajtanunk. Ilyen feladat lehet mondjuk egy automatizált csavarhúzó, amelynél a forgatónyomatékot kell pontosan szabályoznunk.

A karlsruhei AL-változatban mégis van ilyen lehetőség: felírható olyan utasítás, amely a megfogószerkezet pofáinak nyitását, ill. összezárását a megadott szorítóerő pontos betartásával végezteti el (l. *4.54. ábra*).

AL:

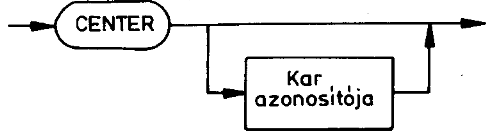
```
CLOSE HAND BY 0.5 * CM
  WITH FORCE = 50 * GM;
```

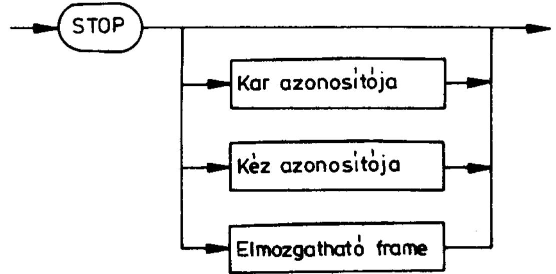
4.3.5. Effektorvezérlő utasítások eseményfelügyelettel

A mozgásvezérlő utasításokhoz hasonlóan vannak olyan effektorvezérlő utasítások is, amelyeknél meghatározott események befolyásolják az utasítás, ill. a program végrehajtását. Ilyen eljárással befejezhető pl. a megfogószerkezet összezárása, mihelyt a szerkezet eltakarja a fénysorompót. A ROBEX jelenlegi formájában külső jelek (események) segítségével meg lehet állítani a megfogószerkezet szétnyitását, ill. összezárását. A *4.52. ábra* tanúsága szerint az utasítás szintaxisa hasonlít a 4.2.6. pontban tárgyalt eseményfelügyelet mellett végrehajtott, tehát eseményvezérelt mozgásvezérlő utasítás szintaxisához.

Ha az ELSE-ágat is megadjuk, akkor a program abban az esetben ugrik a megadott címkére, ha a megfogó nyitása, ill. zárása közben nem lép fel esemény. Eseményfelügyelettel a ROBEX-ben ellenőrizni lehet az előírt szorítóerő betartását is a megfogószerkezet működtetésekor.

4.56. ábra. Az AL nyelv CENTER-utasítása





4.57. ábra. A robot, ill. végrehajtó szerve működtetésének megállítása AL-ban

---

ROBEX:

CLOSE/5, EVENT, 1,ELSE,hibajel

Az 5. output-csatornára csatlakozó megfogószerkezet záródik. Ha az 1. csatornán az összezáródás közben nem lép fel esemény, vagy interrupt (megszakításkérelem), akkor a program a „hibajel" címkéjű ágra ugrik.

A többi nyelvben nem lehet eseményfelügyeletet előírni.

4.4. A robotmozgás, ill. az effektorműködtetés leállítása

Az érzékelők mérési jeleire épülő szabályozásokkal kapcsolatban igen hasznosnak, esetenként nélkülözhetetlennek bizonyult, ha a robot, ill. végrehajtó eszközének pillanatnyi működése a programban elhelyezett STOP-utasítással megállítható. Így a művelet nem igényel kezelői beavatkozást (l. a 4.2.5., ill. 4.3.3. pontokat).

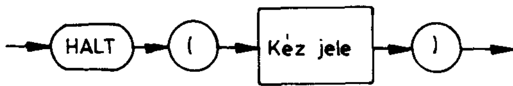
A robot működését leállító utasítás használatának elvileg csak olyankor van értelme, ha az adott nyelvekben valamilyen formában az utasítások párhuzamos feldolgozása is megengedett. STOP-utasításra ugyanis éppen egy mozgásvezérlő utasítás végrehajtása közben kell hogy sor kerüljön, és így a robot pozíciója és orientációja a megállítást követően meghatározatlan marad, ezért ezt újból be kell olvasni.

Az AL nyelvben a STOP-utasítás a robotkarra, valamelyik megfogószerkezetre, vagy a robotkarral összeköttetésben levő rendszerre vonatkozhat (l. *4.57. ábra*). A megfogószerkezetre kiadott STOP a befogópofák szétnyitását, ill. összezárását állítja meg. Ha csak a STOP alapszót adjuk meg, akkor ez azt idézi elő, hogy valamennyi működésben levő robot, ill. effektor mozgása leáll.

AL:

STOP doboz;

Az a robotkar áll meg, amelyikkel a **"doboz"** nevű frame éppen összeköttetésben van.



4.58. ábra. A robotmozgás megállítása HELP-ben

Minthogy a VAL-ban nincs párhuzamos feldolgozás, nincs STOP-utasítás sem.

A HELP-ben csak a robot állítható meg, külön az effektor nem (l. *4.58. ábra*). A HELP-ben van párhuzamos feldolgozás, megoldható, hogy valamilyen bejövő jelre történjen a leállítás.

HELP:

HALT(2)

A 2. jelű robotot állítja meg.

A SIGLA-ban is van STOP-utasítás, azonban ez programmegszakítást idéz elő, ami pedig már csak a kezelőpultról szüntethető meg kézi beavatkozással.

SIGLA:

HL

A robotkar és a program leáll. A CYCLE START gomb lenyomásával a rendszer újra indítható.

A ROBEX-ben nincs explicit STOP-utasítás.

4.5. Érzékelőkkel kapcsolatos utasítások

Az érzékelők alkalmazása sok műszaki feladatnál, de elsősorban az anyagmozgatás és szerelés területén ma már nélkülözhetetlen a műveletek gyors és biztonságos végrehajtásához. Az utóbbi időben valamelyest előtérbe kerültek a végrehajtás és a látás – azaz a kéz és a szem – koordinálásának kérdései. Küszöbön áll jó néhány piacra kész robottal összekapcsolt vizuális rendszer széles körű ipari alkalmazása (l. pl. LIEBERMAN [4.5]). Ugyanakkor az érzékelők közé soroljuk a különféle végálláskapcsolókat, melyeknél a kapcsolás villamos érintkezők, esetleg fénysorompó, vagy ultrahangos érzékelők révén jön létre. Ezek lényegesen egyszerűbbek és olcsóbbak, mint a vizuális rendszerek, használatukkal – a program szempontjából – mégis sok esetben a bonyolult optikai érzékelők szolgáltatásaival egyenértékű jelekhez juthatunk. Ezeket az optikai érzékelőket ugyanakkor mind ez ideig csaknem kizárólag fejlesztés alatt álló rendszerekben alkalmazták, ipari körülmények között csak elvétve, ezért még nem alakultak ki azok a programozástechnikai eszközök, amelyek biztosítanák az optikai érzékelők adatainak és jelzéseinek általános felhasználását a robotvezérlési programrendszerekben. (Gondolunk itt mind a fizikai, mind a logikai, mind pedig a programozási lehetőségek sokoldalú felhasználására.) A legtöbb nyelvben ezeket a kérdéseket úgy oldják meg, hogy kialakítanak olyan utasításokat is, amelyek a számítógépes folyamatirányítás technikájának mintájára kezelni tudják a kétállapotú, ill. az analóg ki- és bemeneteket. A jelenlegi alkalmazások szintjén ez a megoldás még elfogadhatónak mondható, azonban a strukturált programozás, az öndokumentációs képesség és a kompatibilitás követelményeit már semmiképpen sem elégíti ki. Ezért különös tekintettel a vizuális rendszerek alkalmazására létrehozták egyes nyelvek bővített változatait. Ezek között a VAL 11V verziója mellett (itt V a

vizuális eszközre utal) meg kell még említenünk az Automatix cég RAIL nyelvében a vizuális rendszerek kezelésére alkalmas utasításokat. A RAIL egy magasabb szintű programozási nyelv, amely elsősorban a PASCAL-lal és az ALGOL-lal rokon, és ipari folyamatok programozására, rugalmas gyártórendszeri egységek, valamint vizuális rendszerekkel összekapcsolt robotrendszerek programozására alkalmas. A RAIL nyelv vizuális eszközöket kezelő utasításai csaknem kivétel nélkül átfogják azokat a funkciókat, amelyek az optikai érzékelőknek a robotvezérlő programrendszerekbe történő integrációjához szükségesek. Így pl.:

- az érzékelőket olyan paraméterekkel működtetik, amelyek az üzemállapot kiválasztásához és a felvételek pontosságának biztosításához szükségesek;
- érzékelőket működtetnek;
- az érzékelők adatait meghatározott szempontok szerint fogadják és kezelik;
- az érzékelők állapotát lekérdezik;
- az érzékelőket kikapcsolhatják.

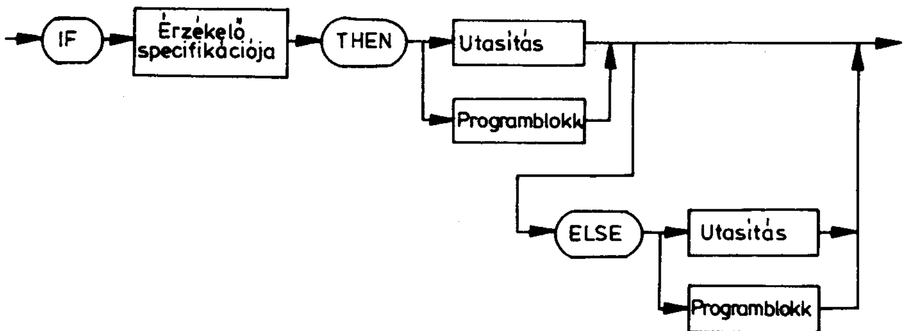
Ami a RAIL-ből hiányzik, az a programmegszakítások definiálási lehetősége, aminek segítségével az érzékelő minden programmegszakításnál elindíthatna egy a főprogrammal párhuzamosan futó alprogramot, és ezzel az adott érzékelő megfigyelési adatainak feldolgozását is vezérelhetné.

Ezeket a kérdéseket megelőzően azonban először a programelágaztatás technikáját, az érzékelők adatainak fogadását és a robotműveleteknek az érzékelők adataira épülő szabályozását és ezek programozástechnikai szervezését tárgyaljuk.

4.5.1. Érzékelők jeleitől függő programelágaztatás

Érzékelők jeleitől függő (érzékelővezérelt) programelágaztatáson azt a (statikus) lekérdezési eljárást értjük, ahogy egy kétállapotú (bináris) jelzés beérkezéséről, ill. valamelyik mérési adatra előírt alsó/felső határérték túllépéséről a program tudomást szerez, és ezektől függően különféle programelágazásokat hajt végre.

Az AL nyelv amerikai változata kizárólag az erő- és nyomatékmérési jelek kezelésére volt alkalmas, ezért a karlsruhei változat fejlesztésekor a későbbi bővítési lehetőségekre gondolva egy általánosabb mérésiadat-lekérdezési utasítással egészítették ki a nyelvet. A *4.59. ábra* szemlélteti ezt a lekérdező utasítást, amely formailag feltételes utasítás, és amely a mérési adattól függően hajt végre programelágazást.



4.59. ábra. Érzékelő jelétől függő feltételes utasítás alakja AL-ban

Az utasítást megelőzően az érzékelőt – ill. a rá vonatkozó információkat – előzetesen deklarálni kell. Ennek specifikációját mutatja a *4.60. ábra.*

Ezzel az utasítástípussal mind a kétállapotú jeleket, mind pedig a mérési adatokat le lehet kérdezni. (A bináris jeleknél az aktív állapot egyes, a passzív állapot zérus.)

AL:

```
IF SENSOR(3)THEN
  PAUSE(1);
```

Ha a 3-as érzékelő bejelez, akkor a program egy másodpercig várakozik.

AL:

```
IF SENSOR(1) ≧ 75 THEN
  BEGIN
  MOVE ARM TO felvétel
  CENTER ARM;
  MOVE ARM TO lerakóhely
      WITH DEPARTURE = 5 * CM
  END
ELSE
  PROMPT ("színezés nem kielégítő");
```

Az 1. érzékelő a munkadarabok festése után színmérést végez; ha a szín megfelelő, akkor a munkadarabokat a lerakóhelyre kell vinni. Ellenkező esetben kiadja a "színezés nem kielégítő" üzenetet és kezelői beavatkozásra vár.

A VAL nyelvben csak a kétállapotú jelzések kezelhetők. A programelágaztatást megelőző vizsgálatban maximálisan négy kétállapotú bemenet logikai ÉS kapcsolatát lehet megvizsgálni (l. *4.61. ábra*). Kissé szokatlan, hogy a csatornaszámokat elválasztó vesszőket minden esetben ki kell tenni. Ha a csatornaszám előtt pozitív előjel áll, vagy egyszerűen előjel nélküli, akkor a lekérdezéskor a gép felső feszültségszintre végez ellenőrzést, ha pedig az előjel negatív, akkor alacsony feszültségszintre történik az ellenőrzés.

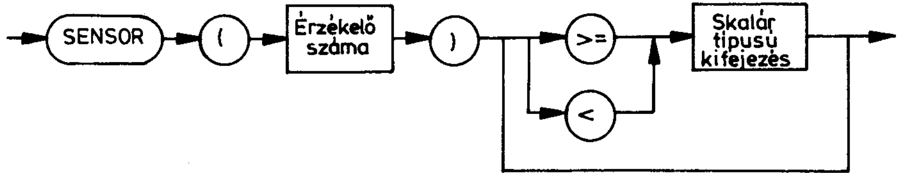
VAL:

```
IFSIG 1,−5,,THEN 150
```

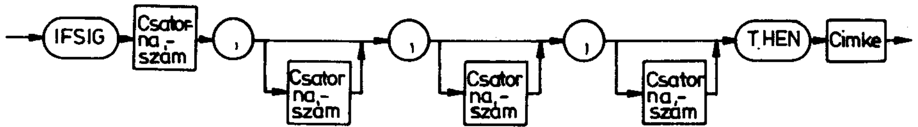
Ha az 1. bemeneten jelzés érkezik, és az 5. bemeneten korábban meglevő aktív jelzés megszűnik, akkor a program a 150-es címkén folytatódik.

A mérési adatok lekérdezési eljárása a HELP-ben lényegében megegyezik az AL-éval (l. *4.62. ábra*).

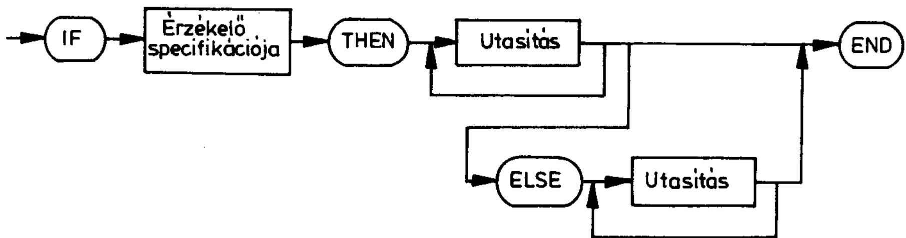
A TESTB alapszó egy logikai típusú függvényeljárást takar, amelynek eredménye aktív bemenet esetén TRUE, egyébként FALSE. A VALUE és az IVALUE lényegében véve adatbeviteli utasítások, amelyek a megadott érzékelőről érkező mérési adatot – annak aritmetikai értékét – olvassák be. A programvezérlő rendszer az érzékelő által szolgáltatott adatot megadott időközönként ciklikusan beolvassa. Az IVALUE eljárással a mérési adat pillanatnyi értékét lehet beolvasni, a VALUE eljárással pedig egy stabilizálódott mérési adatot. Ezt az eljárás úgy állítja elő, hogy addig vár az adatbevitellel, amíg a két utolsónak mért érték már csak a legalacsonyabb helyi értékű bitben különbözik.

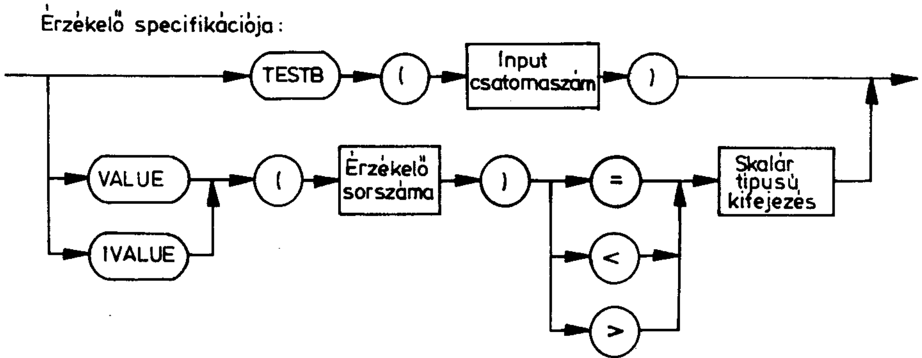


4.60. *ábra.* Érzékelő specifikációja AL-ban



4.61. *ábra.* Kétállapotú jelek lekérdezésének technikája a VAL nyelvben programelágaztatással





4.62. *ábra.* Az érzékelő jelétől függő feltételes utasítás alakja a HELP nyelvben

HELP:

```
IF TESTB(3) THEN
  DELAY(50)
END;
```
Ha a 3-as érzékelő aktív, akkor 1 s-ot vár.

```
IF VALUE(1) > 74,9 THEN
  MOVE(1,#1,140,#3,50);
  PULSE(2);
  MOVE(1,#1,20,#2,260,#3,150)
ELSE
  PRINT ("színezés nem kielégítő");
  HOLD
END;
```
Hatása megegyezik az előző AL példaprograméval.

A SIGLA nyelvben – annak ellenére, hogy viszonylag egyszerűbb nyelvről van szó – lehetőség van mikrokapcsolókról és más érzékelőkről beolvasott jelektől, ill. adatoktól függő programelágazások szervezésére. A *4.63. ábra* egy mikrokapcsoló jelének fogadására alkalmas utasítás szerkezetét mutatja be.

SIGLA:

```
PP/-1,12
```
Ha az 1-es mikrokapcsoló nyitott állapotban van, akkor a program a 12. címkére ugrik.

A *4.65. ábrán* látható MT-utasítás használatával egyébként SIGLA-ban is fogadhatjuk és tárolhatjuk az érzékelőről beérkező mérési adatot. Az így letárolt adatot azután már felhasználhatjuk az ismert feltételes ugróutasításban (l. a 4.5.1.1. pontot).

SIGLA:

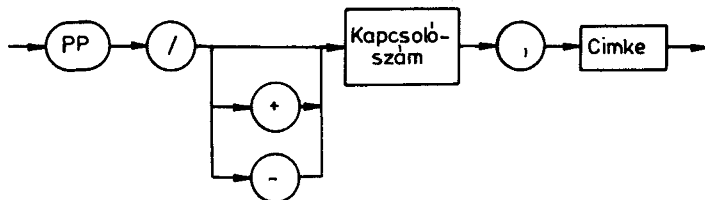
```
MT/ 1,M8,2;
BL/M8,75,3
:            Az itt álló utasításokra akkor jut a vezérlés, ha a mért érték
             nagyobb mint 75.
JU/4;
NU/3;
:            Az itt álló utasításokra akkor kerül sor, ha a mért érték túl
             kicsi.
NU/4;
```
Az 1. érzékelő mérési adatát beolvassa az MB számlálóba. Ettől az értéktől függően hajtja végre a program a megfelelő utasíttássorozatot.

A ROBEX-ben a VAL-hoz hasonlóan csak kétállapotú jelbemenetek kérdezhetők le (l. *4.64. ábra*).

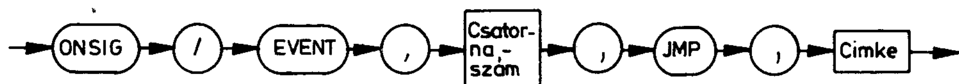
Ha a megadott bemeneti csatornán aktív jel érkezik, akkor a programban a megadott címkére adódik át a vezérlés.

ROBEX:

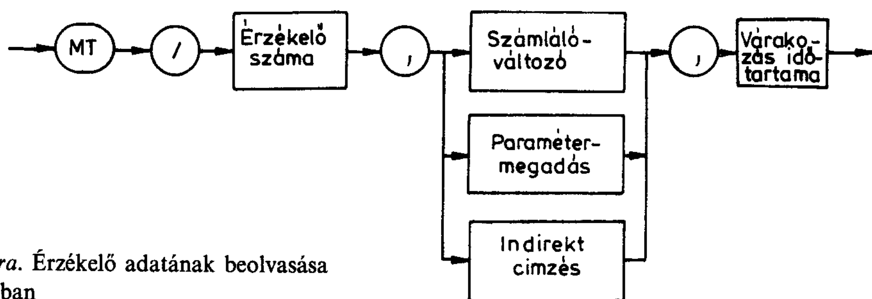
```
ONSIG/EVENT,3,JMP, várj
```
Ha a 3. bemenet aktív, akkor a program a **"várj"** címkére ugrik.



4.63. ábra. Programelágaztatás mikrokapcsoló lekérdezésekor SIGLA-ban



4.64. ábra. Kétállapotú jeltől függő programelágaztatás ROBEX-ben



4.65. ábra. Érzékelő adatának beolvasása SIGLA-ban

4.5.2. Mérési adatok beolvasása

Mint korábban megállapítottuk, az érzékelőkről érkező mérési adatok fogadását többnyire a folyamatperifériáknak szóló utasítások segítségével oldhatjuk meg. Sajnos azonban még ezt a lehetőséget sem nyújtja mindegyik robotprogramozási nyelv. A könyvünkben tárgyalt nyelvek közül csak a HELP és a SIGLA azok, amelyekben megvannak az érzékelők mérési adatainak beolvasására alkalmas utasítások.

A HELP nyelv TESTB, VALUE és IVALUE nevű függvényeljárásairól az előző pontban már szó volt (l. *4.62. ábra*). Ezeket az eljárásokat nemcsak mérési adatok fogadására lehet használni, hanem értéket is adhatunk velük skalár típusú változóknak. Ezzel az érzékelőről beolvasott adatok rendelkezésünkre állnak, és a további műveleteknél már felhasználhatók.

HELP:

sensl := VALUE(1);

Az 1. érzékelőről jövő adatot letárolja a **"sensl"** skalár változóban.

Mint már említettük, a SIGLA nyelv MT-utasítása alkalmas arra, hogy valamely érzékelő mérési adatát egy számlálóba vagy paraméterbe helyezze el. Az MT utasítás szerkezete a *4.65. ábrán* látható (l. még a 4.2.5.1. pontot). A megadott várakozási idő a hardver adottságainak megfelelően a címzéshez és a beolvasáshoz szükséges időt jelenti.

4.5.3. Határértékfigyelés

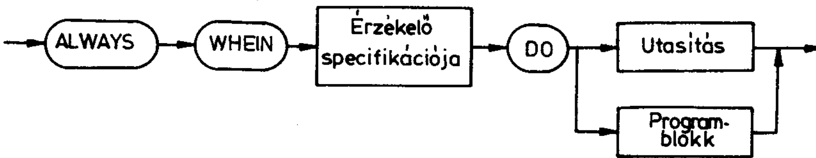
Az érzékelőkről jövő jelek, valamint a mérési adatok alsó/felső határértékeinek túllépése kiválthatja valamilyen adott utasítássorozat végrehajtását. Az utasítások végrehajtásának időpontja, azaz a többi utasításhoz viszonyított helye nem ismert, ezért a főprogram bárhol megszakadhat, majd ismét folytatódhat.

Több nyelvben megfelelő utasítások állnak rendelkezésre, melyek főleg a mozgásvezérlő utasításoknál alkalmazhatók határértékfigyelésre (4.2.5.1. pont).

Alig találni azonban olyan utasítást, amellyel a jelzések, ill. a határértéktúllépések hozzárendelhetők lennének valamilyen programrészlethez, esetleg egy általánosabb taskkezelési rendszer keretein belül. Ehelyett bizonyos feladatok kezelését az alsó vezérlési szinten oldják meg, és ilyenkor a programozónak már nincs beleszólási lehetősége.

Az AL nyelv karlsruhei változatában a határérték-ellenőrzési feladat megoldására az ALWAYS-WHEN utasítást lehet használni (l. *4.66. ábra*).

Ez az utasítás csak annak a blokknak a végéig marad érvényben, amelyben az utasítást elhelyeztük. Ha bekövetkezik az adott érzékelőre specifikált állapot (l. *4.60. ábra*), akkor az interpreter felfüggeszti a főprogram futását, végrehajtja a DO alapszót követő utasítást vagy blokkot, majd ezután ismét folytatja a főprogramot.



4.66. ábra. Érzékelő jelétől függő feltételes AL-utasítás

AL:

```
ALWAYS WHEN SENSOR(3) ≧ 5 DO
  BEGIN
  PRINT ("A gép elindult!");
  gépindul ← TRUE
  END;
```

Mihelyt a 3. érzékelő mérési adata nagyobb vagy egyenlő 5-tel, megjelenik az üzenet és a **"gépindul"** nevű változó TRUE értéket kap. Ennek a változónak a tartalma a főprogramból is hozzáférhető.

VAL-ban nincs általános megoldás az érzékelőkről érkező mérési adatok határérték-felügyeletére, így a feladatot a 4.2.6. pontban megismert eseményfelügyelettel oldhatjuk meg.

HELP-ben sincs közvetlen határérték-felügyeletet megvalósító utasítástípus, a taskkezelő eljárás segítségével azonban az érzékelőket ciklikusan lekérdező programrészletek valósíthatók meg. A felügyeletet végző task végrehajtja a ciklikus lekérdezést, és olyan utasításokat is tartalmaz, amelyek végrehajtására akkor kerül sor, ha az érzékelők jeleivel kapcsolatban vizsgált feltételek teljesülnek. A task ezenkívül tartalmazza még a várakozást vezérlő és a lekérdezési ciklusra visszaugrató utasításokat is (l. a *4.8. programrészletet*).

A SIGLA-ban és a ROBEX-ben a mérési adatokkal kapcsolatban nem lehet határérték-felügyeletet megvalósítani.

```
suetask: sens3 := VALUE (3);
         IF sens3>5 OR sens3 = 5 THEN
           PRINT (0, "gép elindult");
           mian := 1
         END;
         DELAY (10);
         GOTO suetask;
```

4.8. programrészlet. Mérési adattól függő feltételt kezelő task a HELP nyelvben

4.5.4. Vezérlő utasítások vizuális rendszerek esetén

A legtöbb *vizuális rendszer* rendelkezik a következő jellemző tulajdonságokkal:

1. A felismerendő munkadarabok a programban szabadon választható ASCII kódban felírt szimbolikus azonosítókkal ábrázolhatók.
2. A tárgy pozícióját és orientációját derékszögű koordináta-rendszerben ábrázolva veszik fel.
3. A felismerendő munkadarabok alakjára a számítógép általában úgy tanítható meg, hogy a tárgyat egyszerűen megmutatják a rendszernek és ehhez kapcsolódóan megadják szimbolikus azonosítóját is.

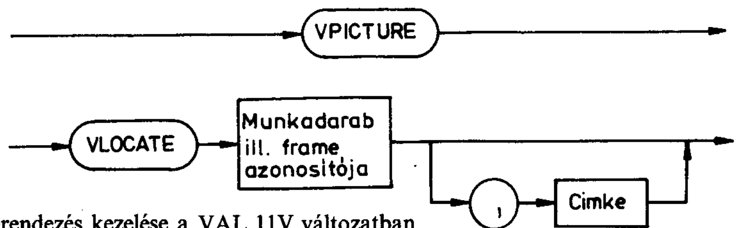
E három tulajdonság igen hasznos a vizuális rendszerek és a robotvezérlő programrendszerek összekapcsolása szempontjából.

Az alkatrészek *azonosítása* – más szóval az *alakfelismerés* – a fő jellemvonásokat összefoglaló vektorok (az ún. karakterisztikus *alakfelismerő-vektorok*) segítségével történik. E vektor komponensei olyan jellemző adatok, mint pl. a tárgy jellemző felszín-, ill. területadatai, a furatok száma, a legkisebb és a legnagyobb átmérő vagy a kerület mérőszáma stb. Ezeknek az adatoknak a beállításával a felhasználónak általában nem kell törődnie, mivel a vizuális rendszer a betanuláskor automatikusan előállítja az alkatrész karakterisztikus vektorát, majd a tárolt értékeket a program futása közben összehasonlítja az aktuális képről készített vektor komponenseivel.

A bővített VAL 11V változatban a kezelőprogram különféle parancsokkal vezérelhető. Ezek a felvevőkamera beállítását, és a különféle alkatrészek jellemzőinek betanulását vezérlik, ill. a tárgy szimbolikus azonosítójának neve alatt tárolják a tárgy karakterisztikus vektorának elemeit.

Az alkalmazói programból a VPICTURE- és a VLOCATE-VAL-utasításokkal fordulhatunk a vizuális rendszerhez *(4.67. ábra)*.

A VPICTURE-utasítás hatására a vizuális rendszer felvételt készít a tárgyról, a képpontokat pedig saját memóriájában tárolja le. A VLOCATE-utasítás ezzel szemben a megadott tárgynak megfelelő tárolt képet keresi vissza. Ha a keresés sikertelen és van megadott címke, akkor a vezérlés erre ugrik, vagy pedig hibaüzenettel leáll a program.



4.67. ábra. Vizuális berendezés kezelése a VAL 11V változatban

Ha azonban a keresés sikeres, akkor a vizuális rendszer ugyanolyan frame-azonosító alatt jegyzi be a pozíció- és orientációadatokat, mint amilyen a tárgy azonosítója volt.

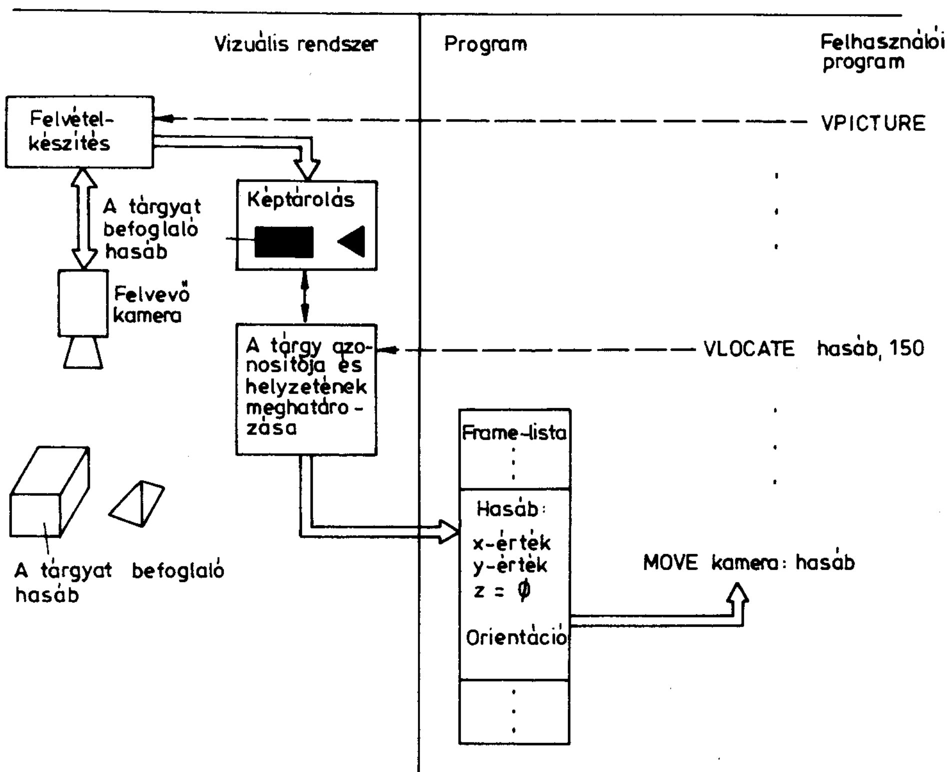
A tárgy frame-rendszerének koordinátaadatait a *felvevőkamera derékszögű koordináta-rendszerében* vesszük fel, nem pedig a robot báziskoordináta-rendszerében. A *4.68. ábra* vázlatosan mutatja a vizuális rendszer kapcsolatát a VAL 11V nyelven programozott robotvezérléssel.

A felhasználó ezek után a 2.4.1. pontban tárgyalt *relatív frame*-ek segítségével már könnyen átszámíthatja a felvevőkamera koordinátáit robotkoordinátákra. Ehhez nem kell mást tennie, mint a kamera látómezejében elhelyezett korong- vagy gyűrű alakú tárgy pozíció- és orientációadatait betanítási eljárással fel kell vennie és egyszerűen ezt a pozíciót **"calibrobot"** nevű frame-ként deklarálnia. Az adatok felvétele úgy történik, hogy a robotkart a tárgy középpontjába állítjuk és a betanítást végző rutin ezt a helyzetet egyszerűen beméri.

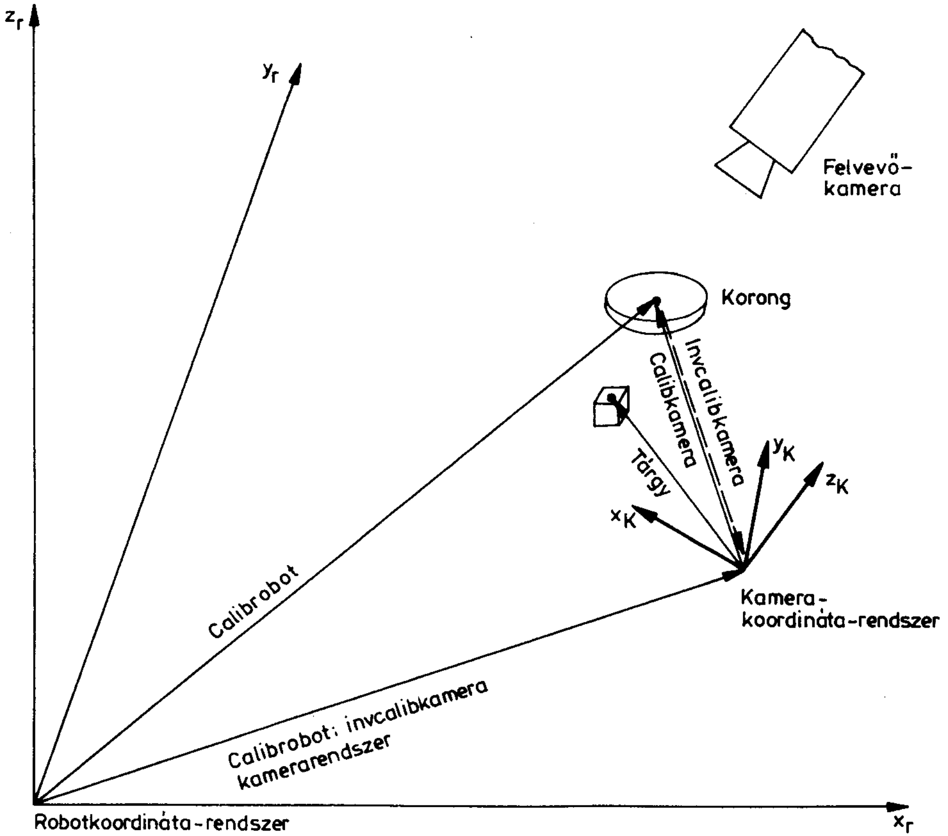
Miután a robotkart a felvevőkamera látómezejéből eltávolítjuk, a *4.9. programrészlet* segítségével (kissé leegyszerűsítve a dolgot) a robot báziskoordináta-rendszeréhez viszonyítva is meghatározhatjuk a kamera koordináta-rendszerének origóját és orientációját.

A *4.69. ábra* bemutatja a két rendszert leíró ún. relatív frame-rendszer geometriai viszonyait.

Az egyszerűség kedvéért itt csak a pozícióvektorokat mutatjuk, az orientációadatok átszámítása hasonlóképpen történik.



4.68. ábra. A vizuális berendezés és a robot programozása közti kapcsolat a VAL 11V változatban



4.69. ábra. A kamera koordináta-rendszerének robotkoordinátákba való átszámításához használatos relatív frame geometriai viszonyai

```
HERE calibrobot
MOVE nyugalmi helyzet            (* a robot nincs a látómezőben          *)
VPICTURE
VLOCATE calibkamera, 150
INV invcalibkamera = calibcamera
SET kamerarendszer = calibrobot : invcalibkamera
        :                        (* tetszőleges programrészlet           *)
VPICTURE
VLOCATE tárgy, 150
MOVE kamerarendszer: tárgy
```

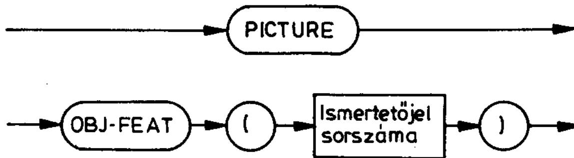
4.9. programrészlet. A felvevőkamera koordináta-rendszerének meghatározása a robot saját koordináta-rendszerében a VAL 11V változatban

A HERE-utasítással a korong középpontját a robot koordináta-rendszerében definiált **"calibrobot"** nevű frame-ként vehetjük fel. A VLOCATE-utasítással ezzel szemben ugyanazt a pontot a kamera koordináta-rendszerében kifejezett ún. **"calibkamera"** frame-ként definiáljuk. Ha most a **"calibkamera"** frame-et invertáljuk, kapjuk az **"invcalibkamera"** pozícióvektort, amely a korong középpontjából a kamera koordináta-rendszerének origójába mutat. Mármost a **calibrobot: invcalibkamera** relatív frame, amely a

kamerakoordináta-rendszer pozíció- és orientációadatait a robot koordináta-rendszerében fejezi ki, vagyis a frame pozícióvektora a robot origójából a kamera koordináta-rendszerének origójába mutat. Egyszerűsítésképpen felvettük a **"kamerarendszer"** nevű frame-et és értékadó utasítással ennek adtuk a relatív frame tartalmát. Ha a későbbiekben a VLOCATE-utasítással bármilyen **tárgy** frame-rendszerét felvesszük, akkor ezeket a fenti számítás elvégzése után a **"kamerarendszer"** nevű relatív frame segítségével MOVE-utasításban is használhatjuk.

A példa rámutat arra, hogy mennyire nélkülözhetetlen a frame-rendszerek és a geometriai típusú műveletek használata vizuális eszközök alkalmazásakor.

Az AUTOMATIX cégnél kifejlesztett RAIL nyelv a robotvezérlés és a vizuális eszközök közötti adatforgalom lebonyolítására speciális utasítások széles választékát kínálja. Lehetővé teszik többek között, hogy a felhasználó saját maga programozza az alkatrészek alakfelismerését végző eljárást. Ehhez a felvétel PICTURE utasítással történő elkészítését követően az OBJ-FEAT függvényeljárás meghatározza a tárgy különféle jellemző adatait (l. *4.70. ábra*).



4.70. ábra. Kép felvételét és feldolgozását vezérlő RAIL-utasítások

Maximálisan 45 jellemző emelhető ki. Ilyenek pl. a következők:

- pozíció (OBJ_XMIN, OBJ_XMAX, OBJ_YMIN, OBJ_YMAX),
- szöghelyzet (OBJ_ANGLE),
- furatok száma (OBJ_NHOLES),
- legnagyobb/legkisebb átmérő (OBJ_RMAX, OBJ_RMIN),
- szín (OBJ_COLOR),
- felület (OBJ_TOTALAREA),
- az alakzat súlypontja (OBJ_XCENT, OBJ_YCENT).

Ezek a jellemzők előre definiált karakterváltozó azonosításával is lehívhatók.

RAIL:

```
IF OBJ_NHOLES = = 3 THEN
  alkrész = 1
ELSE
  BEGIN
  raktar = 1
  alkrész = 2
  END
```

Ha a megfigyelt alkatrészen 3 furat látható, akkor az az 1. alkatrész, egyébként pedig a 2. Ilyenkor a „raktár" nevű változót is beállítjuk.

Az alakfelismeréshez rendszerint nincs szükség mind a 45 jellemző meghatározására. Az eljárás gyorsítása érdekében a jellemzők egy részének kiszámítását elhagyhatjuk. (Ez arra a 24 jellemző paraméterre vonatkozik, amelyet minden felvételre kiszámítunk.) Az ún. rendszerkapcsolókkal beállíthatjuk, hogy mely paraméterek maradjanak ki a számításból. Képzésük a REC attribútum megadásával történik (az előbb megismert OBJ előtagok helyett) és ezt követi a kijelölt jellemző paraméter lefoglalt neve. Példa:

RAIL:

```
REC_NHOLES = OFF
```

Hatására a furatok számát nem határozza meg az eljárás.

Fontos a REC__FEAT rendszerkapcsoló. Ennek hatására valamennyi standard paraméter meghatározását letiltjuk, majd ezután közülük a néhány legfontosabbat külön kijelöljük. Az alkatrészek jellemző paramétereinek vizsgálata előtt analizálni kell a teljes felvételt. Erre valók a következő utasítások:

FIRSTPART	Vedd a felvételen az első alkatrészt
NEXTPART	Vedd a következő alkatrészt (korábban FIRSTPART-nak már szerepelnie kellett!)
FIRSTBLOB	Vedd a képen az első alkatrészt vagy furatot
NEXTBLOB	Vedd a következő alkatrészt vagy furatot (korábban FIRSTBLOB-nak már szerepelnie kellett!)

Fontos még megemlíteni ezzel kapcsolatban az előre lefoglalt VIS__NPARTS változót, amely az aktuálisan vizsgált felvételen előforduló *alkatrészek számát* tartalmazza. Ennek segítségével szisztematikusan kielemezhetők a felvételen levő tárgyak.

RAIL:

```
PICTURE
FOR darabszam = 1 TO VIS__NPARTS DO
  BEGIN
  IF darabszam = = 1 THEN
    FIRSTPART
  ELSE
    NEXTPART
  IF OBJ__NHOLES = = 3 THEN
    EXITLOOP
  END
```

A felvétel elkészítése után a képen levő alkatrészeket sorban megvizsgáljuk, hogy a rajtuk látható furatok száma egyenlő-e hárommal. Ha ez teljesült, akkor a ciklus megszakad és az alkatrész tovább elemezhető.

A rendszer speciális rendszerkapcsolók, paraméterek és függvényeljárások segítségével jól illeszthető a felhasználó birtokában levő vizuális rendszerhez, és az adott feladathoz is. Minthogy a rendszer különböző szürkeségi fokok feldolgozására alkalmas, ezért az egyes küszöbértékeket táblázatok használatával veszi figyelembe, így kapcsolódik a meglevő hardver-rendszerhez. A rendszernek ezt a sokféle szolgáltatását valójában csak egyetlenegyszer használjuk, a rendszer installálásakor. Általában ezt is egy erre specializálódott szakember végzi, így ez csak egészen kivételes esetekben terheli a felhasználót.

RAIL:
VIS_DISPLAY = OFF
Hatására a memóriában tárolt képet nem jeleníti meg a monitoron.
VIS_CAMERANO = 2
A további felvételekhez a 2-es kamerát működteti.

4.6. A blokkszervezés és utasítássorozatok

A 2.2.3. pontban már szó volt a blokkszervezésről és a változók élettartamának és érvényességi tartományának kérdéseiről. Ezek szerint blokknak nevezzük a meghatározott jelekkel (pl. BEGIN és END segítségével) elhatárolt összetartozó utasítássorozatokat, amelyeken belül változókat külön is lehet deklarálni. Ez a meghatározás az AL nyelvre minden megszorítás nélkül igaz. Ezt szemlélteti a *4.10. programrészlet.*

Az 1. és a 2. blokkban új változókat is deklaráltunk, míg a 3. blokk csak egy utasítássorozatot foglal összetartozó egységbe. Ennek a szerkezetnek a következő fejezetben tárgyalt programvezérlési utasításokkal kapcsolatban van jelentősége.

PASCAL-ban a BEGIN és az END kizárólag az utasítássorozatok elhatárolására való. PASCAL-ban blokkok szervezése abban az értelemben, hogy bennük a változókat elkülönítetten deklarálhatjuk és kezelhetjük, a főprogramban, az eljárásokban és a függvényeljárásokban történik. Ezt szemlélteti a *4.11. programrészlet* (l. még a 6. fejezetet!).

A **„főblokk"** nevű főprogramban az **a**, **b**, és **c** változókat deklaráltuk. Az **„ezegyblokk"** nevű eljárásban az **i**, az **x** és az **y** változókat deklaráltuk, ahol **x** és **y** egyben eljárásparaméterek is.

A HELP-ben *dinamikus, statikus* és *init-blokk* deklarálható. A statikus blokkal ellentétben az utasítások a dinamikus és az init blokkon belül végrehajtásukat követően törlődnek. Ugyanakkor csakis a statikus és az init blokkon belül deklarált változók maradnak érvényben a blokk végrehajtását követően is. Így tehát az init blokkok – mint nevük is mutatja – a változók inicializálására használhatók. A statikus blokk alprogramok deklarálására való, a dinamikus blokk pedig a program szegmentálására alkalmas – segítségével a program olyan részekre osztható, amelyek, ha végrehajtásuk után többé nem használjuk őket, törölhetők a memóriából. Ezt az esetet szemlélteti a *4.13. programrészlet.*

```
BEGIN "blokk1"

SCALAR határ;
      :
IF SENSOR (3)<határ THEN
BEGIN "blokk2"
  SCALAR b;
      :
END "blokk2"
ELSE
BEGIN "blokk3"
      :
END "blokk3";
      :
END "blokk1"
```

4.10. programrészlet. Blokkszervezés AL-ban

```
PROGRAM főblokk;
VAR, a, b, c : INTEGER;
PROCEDURE ezegyblokk (x, y: REAL);
VAR i: INTEGER;
BEGIN (* ezegyblokk utasítássorozatának kezdete *);
  :
END (* ezegyblokk utasítássorozatának vége *)
BEGIN (* a főblokk utasítássorozatának kezdete *)
  :
END (* a főblokk utasítássorozatának vége *).
```

4.11. programrészlet. Blokkok és utasítássorozatok PASCAL-ban

```
STATICBLOCK
alprog1:
        :
        RETURN;
ENDBLOCK;
INIT BLOCK
  DEFINE v (3);
  v[1] := 1;
  v[2] := 3;
  v[3] := 0;
  ENDBLOCK;
  BLOCK                : 1-es programszegmens
        :
  ENDBLOCK;            : 2-es programszegmens
        :
  BLOCK                : 3-as programszegmens
        :
  ENDBLOCK;
```

4.12. programrészlet. Blokkszervezés HELP-ben

```
IF x > = a
THEN    IF x < = b
        THEN
        BEGIN
          IF x = (a + b)/2
          THEN PRINT („egyenlő")
        END
        ELSE PRINT („nagyobb")
ELSE PRINT („túl kicsi")
```

4.13. programrészlet. A *4.88. ábrának* megfelelő AL nyelven írt programrészlet

Az init blokk végrehajtását követően (először ezeket hajtja végre a program) a változó a memóriában van. A "**subr1**" nevű alprogramot a többi blokkból is meghívhatjuk, mivel statikus blokkban van. A program fennmaradó része három szegmensre bomlik, ezek egymás után futnak, és kódjuk a feldolgozás után törlődik a memóriából. Az egész szervezés azt a célt hivatott szolgálni, hogy a többé már nem használt utasításokat és változókat eltávolítsuk a memóriából. Tekintve azonban, hogy a tárolóberendezések ára állandóan csökken, az ilyen jellegű szervezés pedig elbonyolítja a programozási megoldásokat, a memóriatakarékos szervezés létjogosultsága megkérdőjelezhető.

4.7. Futtatásvezérlő utasítások

A szakirodalom tanúsága szerint a 60-as évek végén, de főleg a 70-es évek elején a *strukturált programozás* és a *top-down tervezés* előnyei igen élénken foglalkoztatták a szakembereket. A kérdés részleteibe itt nem kívánunk elmélyedni, csupán a szakirodalomra utalunk: lásd [4.6, 4.7, 4.8, 4.9, 4.10].

Sajnos annak ellenére, hogy a publikációk egész sora foglalkozik a témával, még egy általános definíció sem található arra vonatkozóan, hogy mi is voltaképpen a „struktu-

rált programozás", lásd pl. [4.10, 4.11, 4.12, 4.13, 4.14]. Az egyik legelső ilyen témájú publikáció a „DIJKSTRA GOTO-levele"-ként ismertté vált cikk volt [4.15], melyben a szerző az *ugróutasítás* használatától óvta az olvasót, mivel ez az egyik leggyakoribb hibaforrás. A kérdés jobb megértése céljából hasonlítsuk össze a következő két utasítást:

eredmény ← SIN (szög);

és

GOTO 500;

Az első utasítás hatása azonnal világosan érthető, a másodiké nem. Ennek az az oka, hogy az ugróutasítás alakja olyan, hogy semmiféle támpontot nem nyújt arra vonatkozóan, hogy miért hajtjuk végre az ugrást és a vezérlésátadás milyen szerkezeti összefüggések szerint megy végbe. Amint a *4.71. ábrán* látható folyamatábra bizonyítja, a statikus folyamatábrán rendkívül nehezen lehet követni nem strukturált program esetén az algoritmus működése közben befutott programágak változatait. A feltétel nélküli ugróutasítás gyakori használata épp emiatt hordoz magában sok hibalehetőséget.

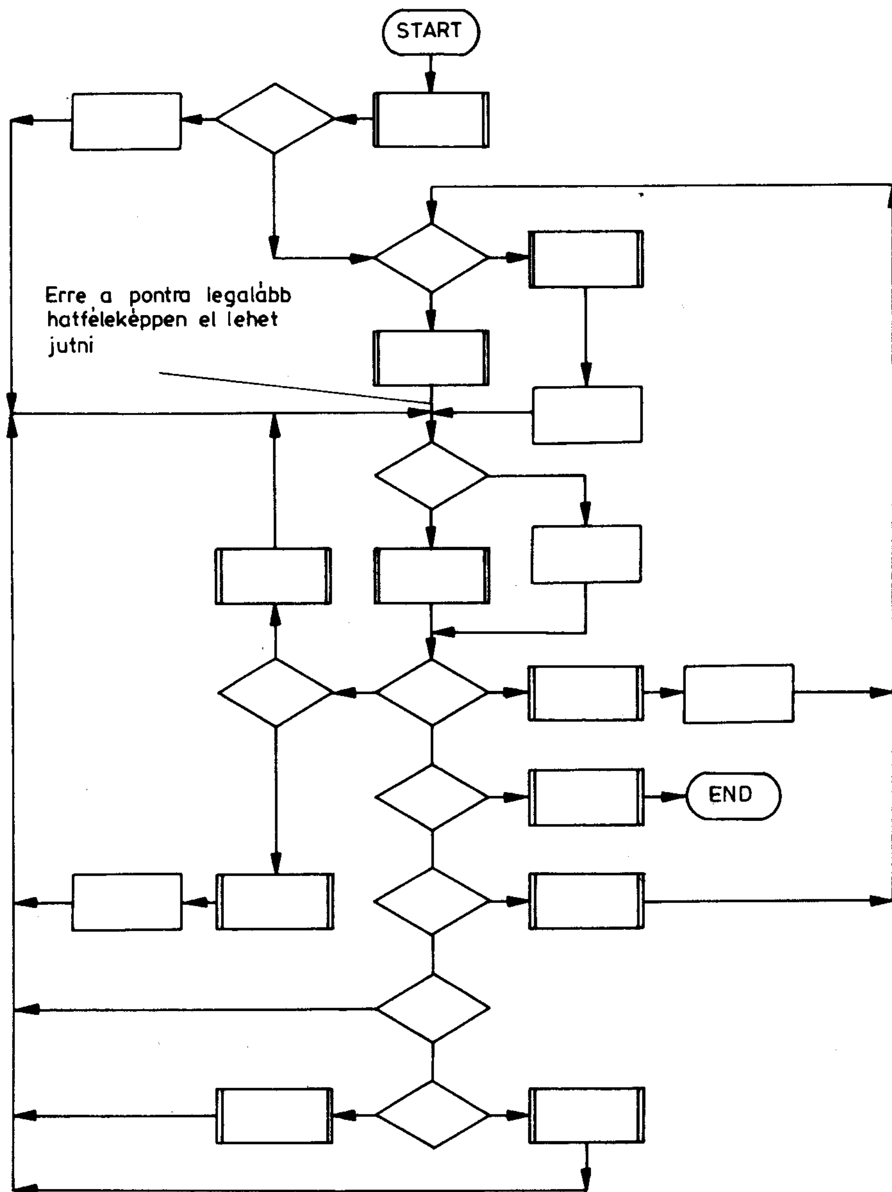
A strukturált programozás módszere éppen arra irányul, hogy a programok dinamikus futásának olyan alaptípusaira dolgozzon ki problémaorientált nyelvi megoldásokat, amelyek használatával a statikusan felírt program alapján annak dinamikus viselkedése, vagyis futás közbeni alakulása is áttekinthető.

A következő fejezetekben részletesen foglalkozunk még a strukturált programozás alapelemeivel, és ezeknek az elemeknek a folyamatábrán, struktogramon és a tárgyalt programnyelvekben megvalósítható formáival. A strukturált programozás alapmintáját mind folyamatábrán, mind pedig struktogramon be szeretnénk mutatni, mivel a folyamatábra előbb említett hátrányai ellenére a kevésbé gyakorlott olvasók számára érthetőbb, mint a struktogramon való ábrázolás, amit épp a strukturált programozás sajátosságainak kiemelésére kívánunk bemutatni.

A **struktogram** az algoritmusok felépítésének NASSI és SCHNEIDERMANN [4.16] által bevezetett grafikus ábrázolási módja. Legegyszerűbb eleme az ún. **elemi funkcionális blokk.** Elemi funkcionális blokkokból a következő szabályok szerint komplex blokkokat állíthatunk össze (l. [4.7, 4.10]):

- Egy funkcionális blokk csak egyetlen bemenettel és egyetlen kimenettel rendelkezhet.
- A funkcionális blokkok között vezérlésátadás csakis fentről lefelé történhet. Ez az elemi blokkokból összeépített stuktogramokra is vonatkozik.
- A blokk zárt funkcionális egység.
- A blokkok között semmiféle átfedés sem engedhető meg: A blokkoknak teljes egészében tartalmaznia kell az általa megvalósított logikai vagy algoritmikus funkciókat.
- Egy blokk kizárólag a közvetlen szomszédaival valósíthat meg adatforgalmat. A hierarchiában felette álló szomszédjától kapja a vezérlést, és az alatta állónak adja tovább.

Mindenekelőtt lássunk egy robotprogramozással kapcsolatos futtatásvezérlési problémát. Azoknál a programnyelveknél, amelyeknél a fordítóprogram állítja elő azt az egyszerű szimulációs modellt, amelynek segítségével azután a robot vezérlése szimulálható, a különféle programelágazások esetén nehézségek merülhetnek fel. (Ilyen pl. az amerikai AL-változat.) Emiatt az amerikai AL-ban bevezették a „Plantime-Assignment"-et (a tervezéskor végrehajtható kijelöléseket), aminek segítségével a programozó a fordítás közben jelölheti ki a szimulátor bizonyos adatait. A probléma azonban így sem



4.71. ábra. Strukturálatlan folyamatábra

oldható meg maradéktalanul, hiszen a program dinamikus működését általában nem lehet a statikus programszöveg alapján megfejteni.

Ez a probléma még tovább súlyosbodik, ha figyelembe vesszük a robot működése szempontjából oly fontos külső érzékelők által küldött jeleket és mérési adatokat is.

Az ilyen természetű nehézségek miatt a robotműködés előzetes szimulációjáról a karlsruhei egyetemen kifejlesztett AL-változatban eleve lemondtak [6].

A következőkben gyakran fogunk utasításokról elvontan is beszélni anélkül, hogy konkrétan meghatároznánk, milyen utasításról is van szó. (A szövegben ilyenkor csúcsos

zárójelek közé írva ⟨utasítás⟩ formában említjük.) Ezen tetszőleges utasítást értünk, elsősorban azonban futtatásvezérlő utasításokat. Ezek sorában különleges helyet foglal el az üres utasítás, amely megtalálható mind a PASCAL-ban, mind pedig az AL-ban. Az üres utasításoknak olyan programelágaztatásoknál van jelentősége, ahol bizonyos esetekben nincs szükség külön programág létesítésére (ilyen pl. a kiszámított GOTO-utasításnál is előfordulhat) (l. még a 4.7.1.2. pontot).

4.7.1. Programelágaztatások

A programelágaztatás legegyszerűbb módja a feltétel nélküli ugróutasítás használata. Mint már említettük, maga az ugróutasítás nem nyújt kellő felvilágosítást arra a kérdésre, hogy hogyan alakul a program szerkezete. Az ugróutasítással kapcsolatban használt címkék alkalmas megválasztásával a program áttekinthetőbbé tehető, ha pl. a címke kifejezi a vezérlésátadás célját vagy helyét stb. Azonnal szembetűnő a különbség, ha pl. azt írjuk, hogy

GOTO 100

vagy

GOTO cikluseleje

A HELP nyelvben címkeként szöveg használata is engedélyezett. Ez elegendő variációs lehetőséget nyújt. Ezzel szemben a PASCAL, a VAL és a SIGLA nyelvekben címkeként csak számjegyek használhatók. Nézzük az ugróutasítások alakját a különféle nyelvekben!

Általánosan minden nyelvben, ahol ugróutasítások mellett strukturált vezérlésátadó utasítások is léteznek (ilyenek a PASCAL és a HELP), fennáll, hogy ezekbe a strukturált utasításokba vagy be sem szabad ugrani, vagy ennek hatása már a compilertől függ, és emiatt mindenképpen elkerülendő.

PASCAL:

A *4.72. ábra* mutatja az ugróutasítás alakját. Itt a ⟨címke⟩ a program fejrészében deklarálandó, legfeljebb négy számjegyből álló címke. A *4.73. ábra* mutatja a program folytatását attól a ponttól, ahová ugrunk.

AL:

Az AL-nyelvben nincs ugróutasítás. Ehelyett a vezérlésátadás szervezésének különféle problémaorientált utasítástípusai léteznek.

VAL:

A *4.74. ábrán* bemutatott utasítástípusban a ⟨címke⟩ csak 32 767-nél kisebb szám lehet, és csakis számjegyekből állhat. A *4.75. ábra* mutatja a program folytatását a címkétől.

HELP:

A *4.76. ábrán* bemutatott utasítástípusban a ⟨címke⟩ tetszőleges számú alfabetikus jellel kezdődő alfanumerikus karakterből állhat. Csak az első 6 karakter szignifikáns. A *4.77. ábra* mutatja a program folytatását a címkétől.

A HELP nyelvben a bonyolult blokkszervezés miatt az ugróutasítás használata erősen korlátozott. HELP-ben megkülönböztetünk statikus és dinamikus blokkot. A statikus programblokk kódja állandóan a memóriában marad, a dinamikus blokké pedig a végrehajtás után törlődik (l. 4.6. szakasz). Emiatt külső blokkba történő beugrásnál csakis statikus blokk utasítására adható át a vezérlés és ezt az utasítást tartalmazó statikus blokknak, ill. címkének meg kell előznie az ugróutasítást, tehát a programban

csak hátrafelé lehet ugrani. Dinamikus blokkon belül ezzel szemben előre is, hátra is ugorhatunk. Dinamikus blokkba kívülről beleugrani nem szabad.

SIGLA:

A *4.78. ábrán* feltüntetett ⟨címke⟩ 1 és 255 közé eső egész szám, paraméter, számlálóváltozó vagy indirekt címzésű számláló lehet, ezek tartalma azonban szintén csak az előbb megadott értéktartományba eshet. Az ugrási cél a *4.79. ábrán* látható, ahol ezen utasításkód a tulajdonképpen végrehajtandó utasítás előtt áll.

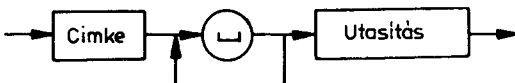
ROBEX:

A ⟨címke⟩ használatának a *4.80. ábrán* bemutatott módját [13] nem részletezi, azonban más utasításokból és a [14]-ben közölt példákból arra lehet következtetni, hogy a címke maximálisan hat – alfabetikus jellel kezdődő – alfanumerikus jelből állhat. A *4.81. ábrán* látható, hogy hogyan folytatódik a program a címkétől.

4.72. ábra. Feltétlen vezérlésátadás PASCAL-ban

4.73. ábra. A címkézett utasítás PASCAL-ban

4.74. ábra. Feltétlen vezérlésátadás VAL-ban



4.75. ábra. A címkézett utasítás VAL-ban

4.76. ábra. Feltétlen vezérlésátadás HELP-ben

4.77. ábra. A címkézett utasítás HELP-ben

4.78. ábra. Feltétlen vezérlésátadás SIGLA-ban

4.79. ábra. A vezérlés folytatása a címkétől SIGLA-ban

4.80. ábra. Feltétlen vezérlésátadás ROBEX-ben

4.81. ábra. A címkézett utasítás ROBEX-nél

Ezek az ugróutasítások a program futása közben befolyásolják a vezérlésátadást. A ROBEX-ben emellett létezik olyan – compilernek szóló – vezérlésátadási utasítás is, amely azt idézi elő, hogy maga a fordítás folytatódjon az adott címkétől. Ezek a nyelvi megoldások azonban nem tartoznak a szorosan vett nyelvhez (csupán a nyelvleírás kiegészítéseként említik a G függelékben), ezért ezzel a kérdéssel nem foglalkozunk részletesebben.

A feltétel nélküli ugróutasításon és a 4.5.1. pontban tárgyalt külső mérési adatoktól függő programelágazáson kívül a ROBEX-ben nincs más segédeszköz a programfutás (vezérlésátadás) befolyásolására. Mivel a ROBEX-ben a VAL-hoz és a SIGLA-hoz hasonlóan nincsenek meg a később tárgyalandó feltételes utasítások, elágazások és ciklusok közvetlen szervezésére alkalmas eszközök, ezért ezeknek a standard típusait a programban külön meg kell szervezni explicit programozással. A ROBEX-nél pl. a feltételes vezérlésátadásoknál csak igen korlátozott mértékben lehet figyelembe venni az érzékelőktől bejövő külső jeleket.

4.7.1.1. Feltételes vezérlésátadás

Tekintsük át először a VAL és a SIGLA feltételes vezérlésátadási utasításait, és azután a PASCAL, az AL és a HELP strukturált IF-utasításait.

VAL:

A *4.82. ábrán* az alapszavak jelentése: EQ (equal): egyenlő, NE (not equal): nem egyenlő, LT (less than): kisebb mint, GT (greater than): nagyobb mint, LE (less than or equal): kisebb vagy egyenlő és GE (greater than or equal): nagyobb vagy egyenlő. Ha az előírt feltétel teljesül, akkor a vezérlés a ⟨címke⟩-re adódik át, egyébként pedig a program a következő utasításnál folytatódik.

SIGLA:

SIGLA-ban háromféle feltételes ugróutasítás ismeretes *(4.83. ábra).* Az alapszavak jelentése a következő: BE (branch equal): egyenlő, BG (branch greater): nagyobb és BL (branch less): kisebb. Ha a megadott feltétel teljesül, akkor a ⟨címke⟩-re adódik át a vezérlés, ellenkező esetben a soron következő utasítással folytatódik.

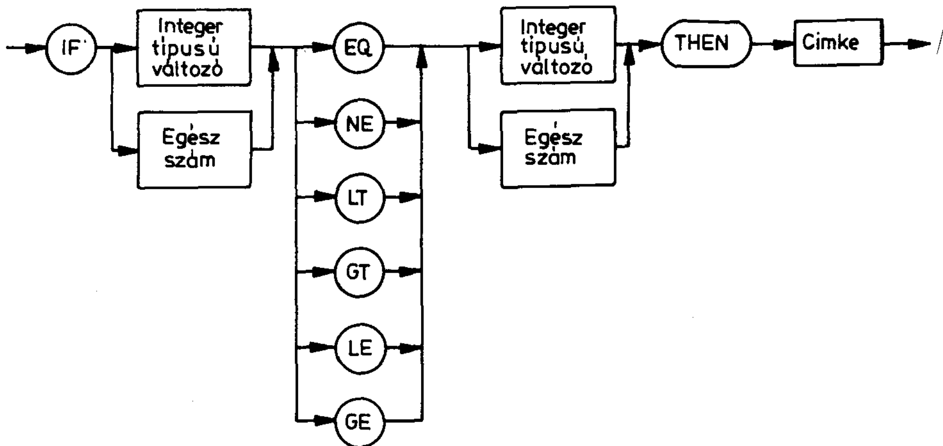
A strukturált IF-utasítás egy feltételes utasítás kiadását és két alternatíva közüli választás lehetőségét biztosítja. Az utasítás általános alakja a *4.84.* és a *4.85. ábrán* látható.

PASCAL és AL:

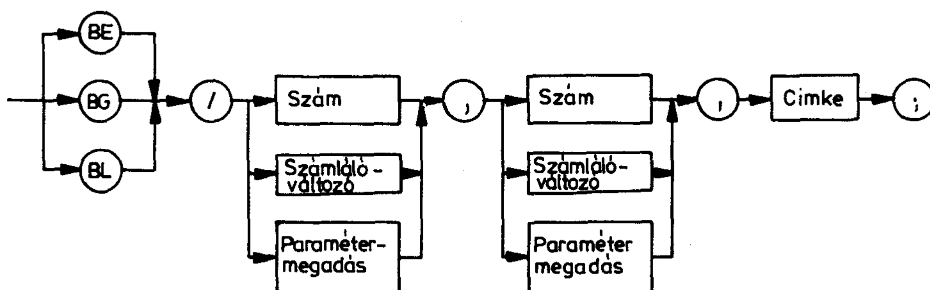
A PASCAL és az AL ide vonatkozó utasításai megegyeznek (l. *4.86. ábrát*). Itt a ⟨feltétel⟩ logikai kifejezést jelent.

A **HELP**-nél olyan utasítássorozatot kell alapul venni, amely egy utasítást mindenképpen tartalmaz, ezért az IF-utasítást mindig END zárja le (l. *4.87. ábra*).

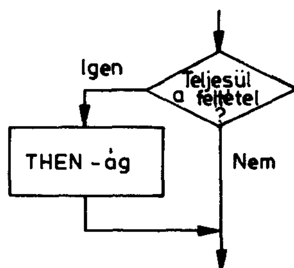
Egy utasítás, ill. strukturált programblokk ismét csak tetszőleges számú vezérlésátadási utasítást tartalmazhat, és így a programelágazások hierarchikus struktúrája alakítható ki. Ilyen összetett IF-utasítást mutat a *4.88. ábra.* A *4.13. programrészlet* AL-ban, ill. PASCAL-ban megfogalmazott összetett feltételes utasítást mutat be (PASCAL-ban a PRINT-utasítást WRITE-utasítás helyettesítené!).



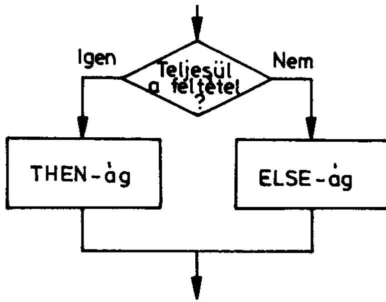
4.82. ábra. Feltételes vezérlésátadás VAL-ban



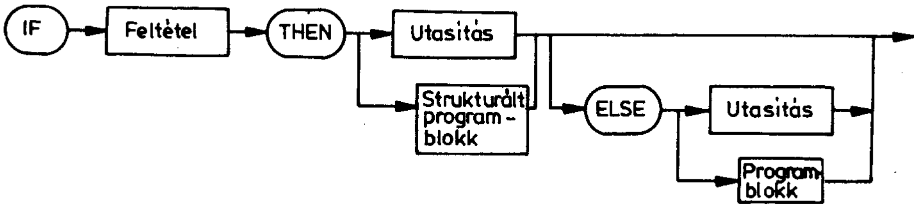
4.83. ábra. Feltételes vezérlésátadás SIGLA-ban



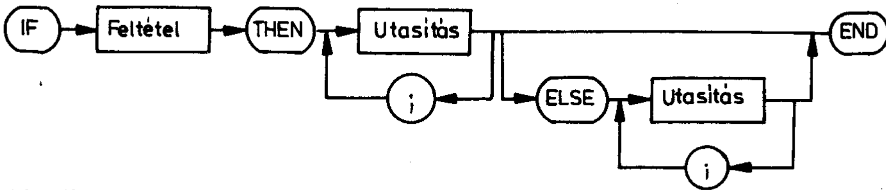
4.84. ábra. Feltételes utasítás



4.85. ábra. Választás két alternatíva közül



4.86. ábra. IF-utasítás szerkezete AL-ban és PASCAL-ban

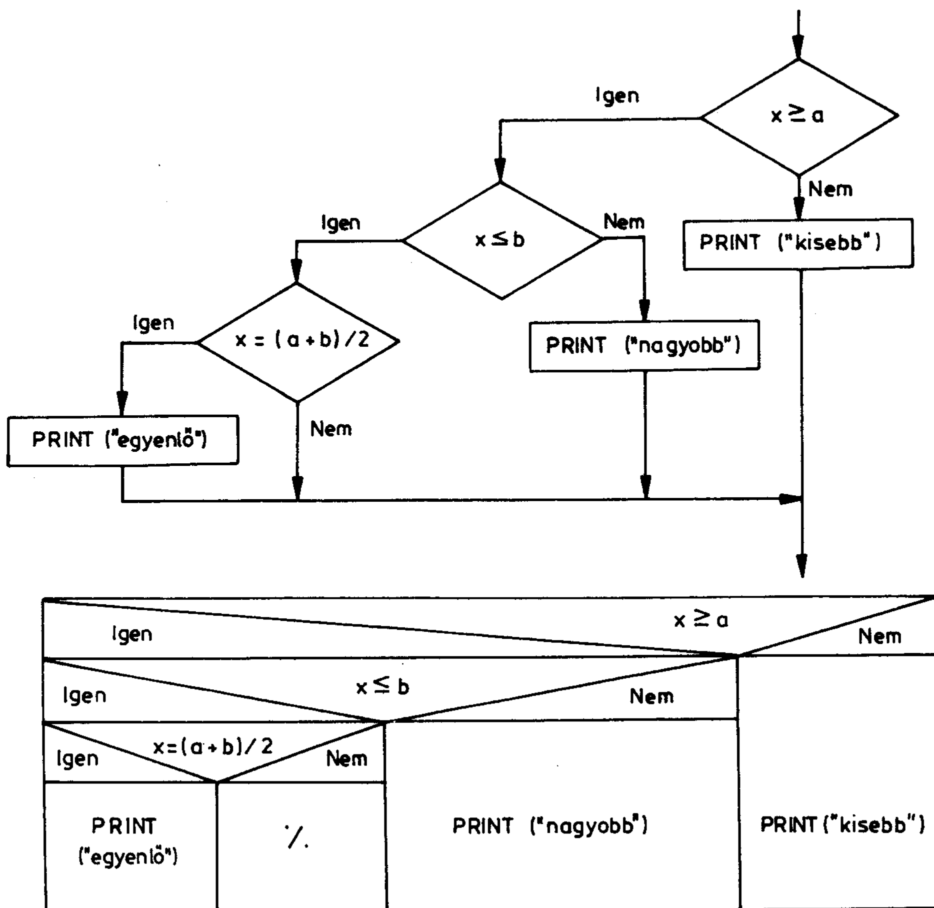


4.87. ábra. IF-utasítás szerkezete HELP-ben

A harmadik feltétel BEGIN-END zárójelek közé került. Ennek az az oka, hogy a harmadik IF-nek nincs ELSE-ága, egymásba ágyazott IF-utasításokra pedig fennáll az az általános szabály, hogy az ELSE-ág ahhoz a hozzá legközelebb álló IF-hez tartozik, amelynek még nincs ELSE-ága. A BEGIN-END zárójelezés következtében a harmadik IF már nem befolyásolja az ELSE PRINT („nagyobb") ágat, ezért az a második IF-hez tartozik.

A HELP-ben minden IF-utasítást END-del zárunk le, ezért HELP-ben egyértelmű, hogy melyik ELSE melyik IF-hez tartozik. Ami a program strukturális tagolását illeti, nincs jelentős eltérés az AL-hoz és a PASCAL-hoz képest (l. a *4.14. programrészletet*). Az IF után álló feltételeket úgy választottuk meg, hogy a *4.88. ábrának* megfelelő algoritmus változatlan maradjon.

A többi nyelvben már kevésbé áttekinthető a helyzet. Vegyük először a VAL-t, majd a SIGLA-t *(4.15.* és *4.16. programrészlet).*



4.88. ábra. Egy egyszerű példaprogram folyamatábrája és struktogramja

```
IF (x> a) or (x = a)
THEN IF (x < = b) OR (x = b)
        THEN IF x = (a + b)/2
                THEN PRINT („egyenlő")
                END
        ELSE PRINT („nagyobb")
        END
ELSE PRINT („túl kicsi")
END;
```

4.14. programrészlet. A *4.88. ábrának* megfelelő HELP programrészlet

```
        IF x GE a THEN 10
        TYPE túl kicsi
        GOTO 100
10      IF x LE b THEN 20
        TYPE túl nagy
        GOTO 100
20      SETI help = a + b
        SETI help = help/2
        IF x NE help GOTO 100
        TYPE egyenlő
100
            :
```

4.15. programrészlet. A *4.88. ábrának* megfelelő VAL nyelven írt programrészlet

```
BL/M1, M2, 1;
BG/M1, M3, 2;
SE/M4, M2;
IC/M4, M3;
SE/M5, 0;                   / \
NU/10;                      / |
IC/M5, 1;                   / |
IC/M4, -2;                  / > 2-vel való osztás
BG/M4, 0, 10;               / |
BE/M4, 0, 11;               / |
IC/M5, -1;                  / /
NU/11;
BE/M1, M5, 3;
JU/100;                     / fennáll, hogy:
NU/1;                       / M1 tartalmazza x-et
NT/túl kicsi, M1;           / M2 tartalmazza a-t
JU/100;                     / M3 tartalmazza b-t
NU/2;                       / M4 tartalmazza (a + b)-t
NT/túl nagy, M1;            / M5 tartalmazza az osztás eredményét
JU/100;                     / (Egészosztás)
NU/3;
NT/ egyenlő, M1;
NU/100;
```

4.16. programrészlet. A *4.88. ábrának* megfelelő programrészlet SIGLA-ban

A PASCAL, az AL és a HELP strukturált utasításszervezésével ellentétben a VAL és a SIGLA nyelvekben az egyes feltételek egymásba ágyazásának szerkezete a leírt programszöveg alapján csak igen alapos elemzés után tisztázható.

A SIGLA program sokkal hosszabb és bonyolultabb, mivel még az osztási műveletre is külön programot kell írni (l. *3.12. programrészlet*).

Az AL, a PASCAL és a HELP programrészletek jól olvashatók annak ellenére, hogy a programokat nem láttuk el kommentekkel. A VAL program olvashatósága már rosszabb, a SIGLA program kommentek nélkül pedig már áttekinthetetlenné válik, gyakorlatilag olvashatatlan és emiatt könnyen elhibázható.

4.7.1.2. Programelágaztatás

Ha kettőnél több alternatíva közül kell dönteni, akkor vagy egymásba ágyazott IF-utasításokat konstruálunk, vagy pedig programelágaztatáshoz folyamodunk. A *4.89. ábra* szemlélteti ennek folyamatábráját és struktogramját.

Ilyen utasítástípus csak a PASCAL-ban, az AL-ban és a HELP-ben található.

PASCAL:

A *4.90. ábrán* ⟨**const**⟩ helyébe olyan számértékeket kell írnunk, hogy a ⟨**kifejezés**⟩ ilyen számértékeket felvehessen. Példánkban a ⟨**kifejezés**⟩ lehet INTEGER vagy CHAR típusú.

AL:

A CASE-utasításnak kétféle típusa létezik: a címkézett és a címke nélküli (l. *4.91. ábra*).

A ⟨**skalárkifejezés**⟩-t előbb kiszámítjuk, ennek Integer része megadja, hogy melyik utasítást kell végrehajtani. Az utasításokat 0-tól kezdődően számozzuk. Mind az AL-ban, mind pedig a PASCAL-ban létezik üres utasítás is. A CASE-utasítás második típusánál a konstans nem lehet negatív.

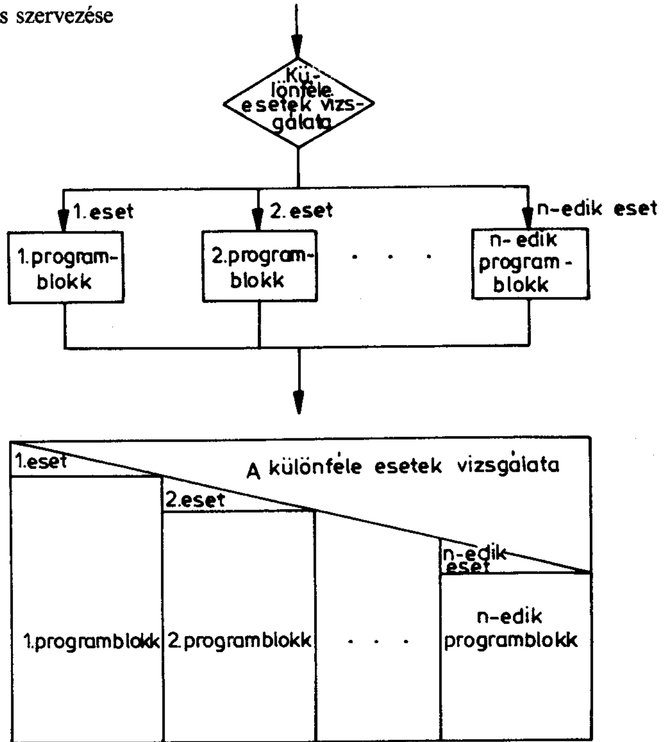
Ha a szelektor kifejezés eredményének megfelelő konstanst nem adjuk meg, ill. ilyen sorszámú utasítás nem található, akkor AL-ban még akkor is hibaüzenet jelenik meg, ha a második típusnál az ELSE-ág megvan. Erről az esetről a PASCAL sajnos nem rendelkezik, a program viselkedése tehát ilyenkor a fordítóprogramtól függ. PASCAL-nál is helyesebb lenne, ha ugyanúgy hibaüzenettel állna le a fordító, mint AL-ban az ELSE-ágnál.

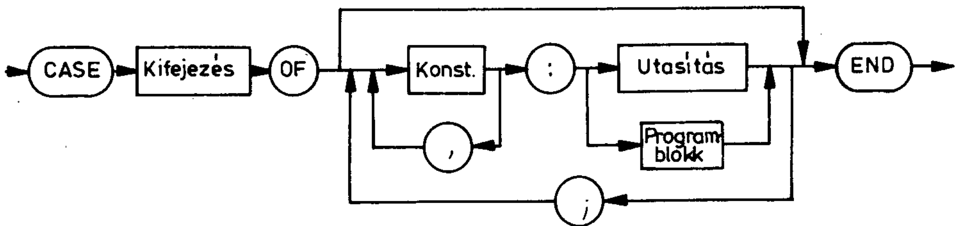
HELP:

A HELP-ben nincs CASE-utasítás, csak a FORTRAN-ból ismert kiszámított GOTO-utasítás. Ennek segítségével könnyen összeállítható a kiszámított programelágaztatás.

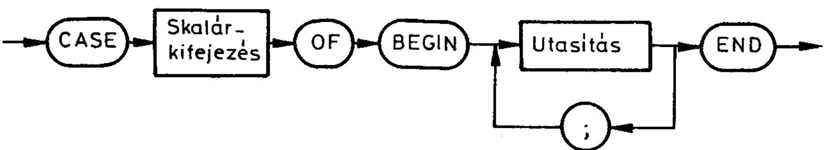
Az utasítás szintakszisát a *4.92. ábra* szemlélteti. Itt ⟨**kifejezés**⟩ skalár típusú kifejezést jelent, ami esetenként kerekítődik. Ennek értékétől függően választja ki a vezérlés az i-edik ⟨**címke**⟩-t és ide adódik át a vezérlés. Az ugrási címkék számozása 1-től

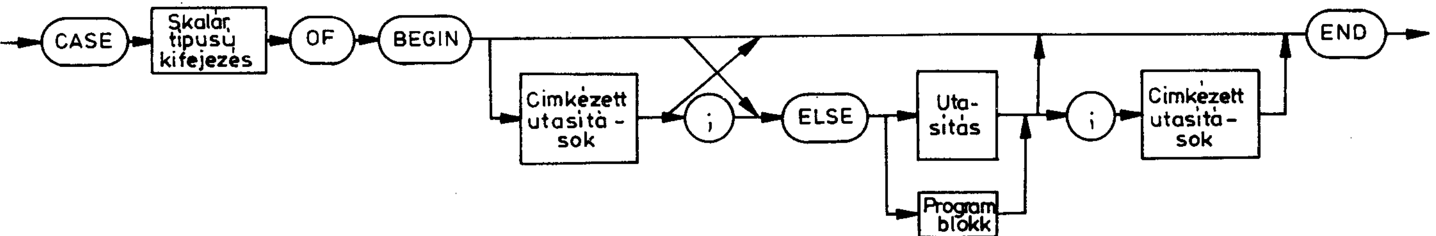
4.89. ábra. Programelágaztatás szervezése

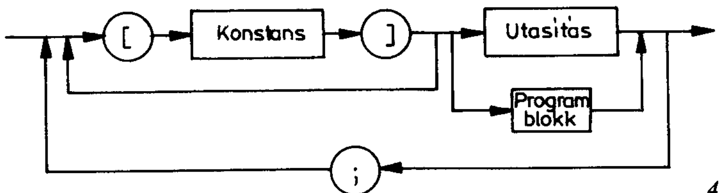




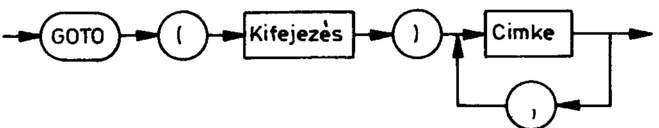
4.90. ábra. CASE-utasítás a PASCAL-ban







4.91. ábra. CASE-utasítás az AL-ban



4.92. ábra. Kiszámított GOTO-utasítás HELP-ben

kezdődik. Ha a kifejezés értéke nagyobbnak adódik, mint a címkék száma, akkor a program nem hajt végre ugrást, hanem a soron következő utasítással folytatódik a feldolgozás. Az ellenkező eset – amikor a kifejezés kisebb mint 0.5 – a HELP-ben nem létezik. A kiszámított GOTO esetében is érvényesek a 4.7.1. pontban a vezérlésátadási utasításokkal kapcsolatban felsorolt megszorítások. Ezek figyelembevételével HELP-ben az összetett programelágaztatás a *4.17. programrészletnek* megfelelően állítható össze.

```
           GOTO (esetek) esetl, eset2, ..., esetn;
esetl;     ⟨utasítások⟩
           GOTO esetvége;
eset2:     ⟨utasítások⟩
           GOTO esetvége;
           :
esetn:     ⟨utasítások⟩
esetvége:  :
```

4.17. programrészlet. Elágazás kiszámított GOTO-utasítással HELP-ben

A bevezetőben már említettük, hogy a programelágaztatás egymásba ágyazott IF-utasítások segítségével is létrehozható. Ezt a megoldást akkor kell előnyben részesíteni, ha

- az egyes esetek erősen eltérő valószínűségűek, és a program írásakor ez a gyakoriságeloszlás már előre látható;
- másrészt az egyes esetek megkülönböztetése nem redukálható valamilyen index kiszámítására.

A VAL-ban és SIGLA-ban a programelágaztatást GOTO-utasításokkal kell megoldani, a ROBEX-ben programelágaztatás csak külső érzékelők jeleivel kapcsolatban szervezhető.

4.7.2. Ciklusszervezés

A programon belüli vezérlésátadás alapvető eljárásai közül utoljára hagytuk a ciklusszervezés tárgyalását. A cikluson belül egy utasítás, vagy programblokk vétehajtása ismétlődik mindaddig, amíg egy meghatározott feltétel teljesül.

A ciklusszervezés különféle típusai egymástól abban térhetnek el, hogy milyen a megadott feltétel, vagy hogy a cikluson belül hol helyezkedik el a ciklus folytatásával kapcsolatos vizsgálat. A ciklus szervezésére megfelelő ciklusutasítások léteznek a PASCAL-ban, AL-ban és a HELP-ben. VAL-ban és a SIGLA-ban a ciklus szervezésére ugróutasítás felhasználásával külön kell elkészítenünk a megfelelő programrészletet.

ROBEX-ben csak a külső érzékelőkkel kapcsolatban lehet ciklust szervezni, ciklusszámlálóval nem.

Még egyszer szeretnénk utalni rá, hogy a ciklus belsejébe való beugrás általában előre nem látható következményekkel járhat. Ezt egyes compiler-programok eleve hibaként kezelik.

4.7.2.1. Ciklusszervezés ciklusváltozóval

A ciklusok egyik típusánál kijelölünk egy ún. ciklusváltozót (más néven számlálóváltozót), melynek értéke a cikluson történő minden egyes végighaladás előtt vagy után a megadott kezdőértéktől a végértékig inkrementálódik (növekszik) vagy dekrementálódik (csökken). A ciklus akkor fejeződik be, ha a ciklusváltozó a kijelölt végértéket túllépi. AL-ban, HELP-ben és PASCAL-ban ezt a fajta ciklust FOR-utasítással valósíthatjuk meg. AL-ban és PASCAL-ban a program a cikluson történő minden egyes végighaladás előtt ellenőrzi a ciklusváltozó értékét, tehát a legelső előtt is. A HELP nyelv ezt a kérdést nem szabályozza. A *4.93. ábra* olyan ciklus szervezésének általános folyamatábráját, ill. struktogramját mutatja, amelyben a ciklusváltozóval növekvő vagy csökkenő irányba számlálunk és a ciklusváltozó értékének vizsgálata a ciklus elején van.

Az ábrán bemutatott típusnál a ciklus magját képező utasításokat (az ún. blokkot) még akkor is végrehajtja a program, ha a ciklusváltozó értéke éppen egyenlő az előírt végértékkel. Ha ezt el akarjuk kerülni, akkor a vizsgálatot úgy kell felírni, hogy a feltételben „nagyobb egyenlő” helyett „nagyobb” álljon.

PASCAL:

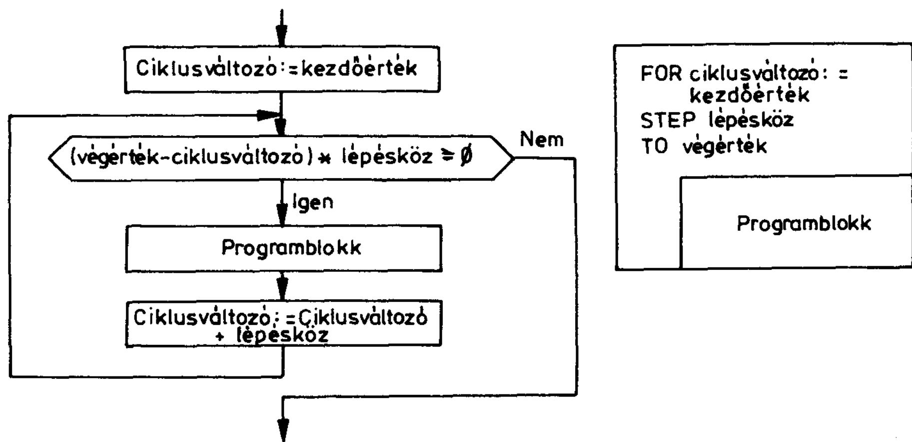
PASCAL-ban a számlálás mindig csak egyesével történhet *(4.94. ábra)*. A ciklusváltozó kezdő, ill. végértékét megadó kifejezéseket a program csak egyetlenegyszer értékeli ki, az utasítás elején. A ciklus belsejében a ciklusváltozó értékét semmilyen utasítással nem szabad megváltoztatni, ennek értéke a FOR-ciklus befejezésekor nem definiált. A ciklusváltozónak INTEGER vagy CHAR típusúnak, vagy a felhasználó által definiált más lineárisan rendezett adattípushoz tartozónak kell lennie.

AL:

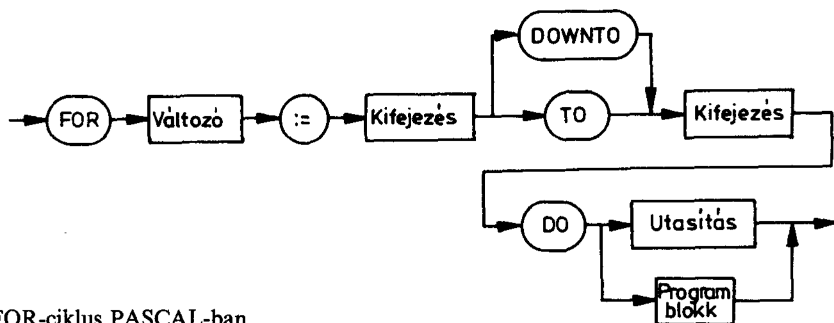
A FOR-ciklus AL-változata több variációs lehetőséget kínál, lásd a **4.95. ábrát!** Aszerint, hogy a számlálás inkrementumát megadó ⟨**skalárkifejezés**⟩ pozitív vagy negatív, a számlálás felfelé vagy visszafelé történik mindaddig, amíg a ciklusváltozó túl nem lépi az előírt végértéket. A kifejezéseket a program itt is csak egyszer, a ciklus megkezdésekor értékeli ki. A PASCAL-lal ellentétben a ciklusváltozó értéke az iteráció befejezését követően is egyértelműen definiált. A ciklusváltozónak a ciklus törzsén belüli megváltoztatásának hatása nincs rögzítve. A ciklusváltozót nem szükséges előre deklarálni, mivel az AL-ban létezik implicit deklaráció is.

HELP:

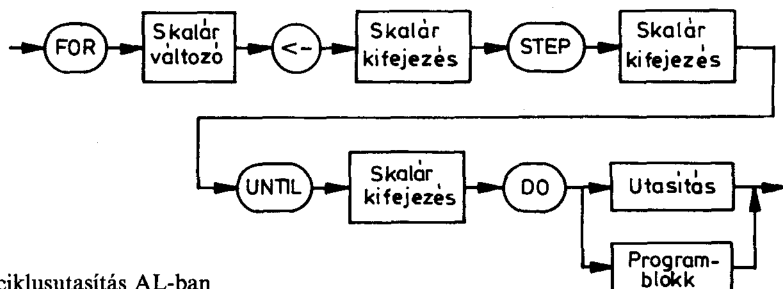
A HELP-ben a FOR-ciklusnak kétféle alakja is létezik. Az első megfelel az eddig bemutatott változatnak. Ez a *4.96. ábrán* látható. A számlálás csak növekvő irányban lehetséges. Az inkrementum értékének megadása el is hagyható: ebben az esetben a számlálás egyesével történik. A kifejezéseket a program itt is a ciklus végrehajtásának megkezdése előtt értékeli ki. Az eddig tárgyalt esetektől eltérően a HELP-ben kifejezetten meg van engedve, hogy a ciklus törzsén belül értéket adhassunk a ciklusváltozónak, amivel a ciklust lerövidíthetjük, esetleg idő előtt befejezhetjük.



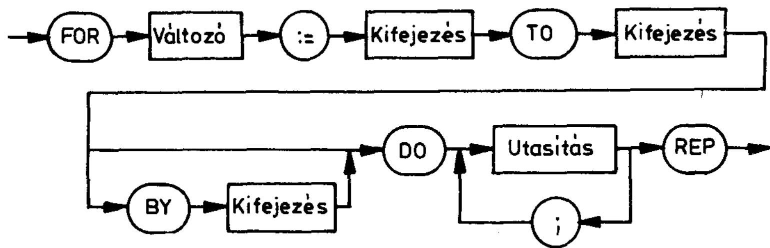
4.93. ábra. Ciklusszervezés ciklusváltozóval



4.94. ábra. FOR-ciklus PASCAL-ban



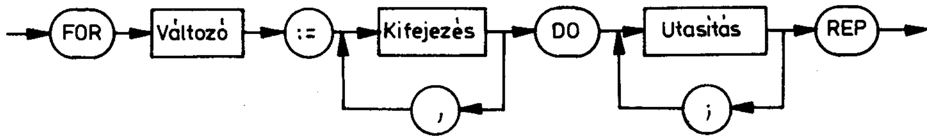
4.95. ábra. FOR-ciklusutasítás AL-ban



4.96. ábra. FOR-ciklusutasítás HELP-ben

A ciklusváltozó megőrzi utoljára felvett értékét, így az a ciklus befejezését követően is hozzáférhető.

A második változatot a *4.97. ábra* szemlélteti. Ennél a ciklusváltozó egymás után felveszi a megadott n számú kifejezés értékét, és ezekkel végrehajtódik a ciklus magját alkotó utasítássorozat.



4.97. ábra. FOR-ciklusutasítás HELP-ben

4.7.2.2. Feltételtől függő ciklus

Két alaptípusa van: az egyiknél a program a ciklus törzsének megismétlését megelőzően megvizsgálja a ciklusfeltételt, a másiknál a ciklustörzs végén a leállási feltételt vizsgálja. A két típus folyamatábráját és struktogramját a *4.98. ábra* mutatja.

PASCAL:

A PASCAL nyelvben használt WHILE és REPEAT ciklusok definíciói a *4.99.* és a *4.100. ábrákon* láthatók. Ha a ciklusmag több utasítást tartalmaz, akkor ezeket a WHILE-ciklusnál blokkban kell egyesíteni, míg a REPEAT-ciklus esetében ez nem szükséges.

AL:

Az AL-nyelvben használt WHILE-ciklus pontosan megegyezik a PASCAL-ban használttal, a REPEAT pedig csak szintaktikailag különbözik PASCAL megfelelőjétől *(4.101. ábra).*

HELP:

A HELP-ben csak a *4.102. ábrán* bemutatott WHILE-ciklus használható.

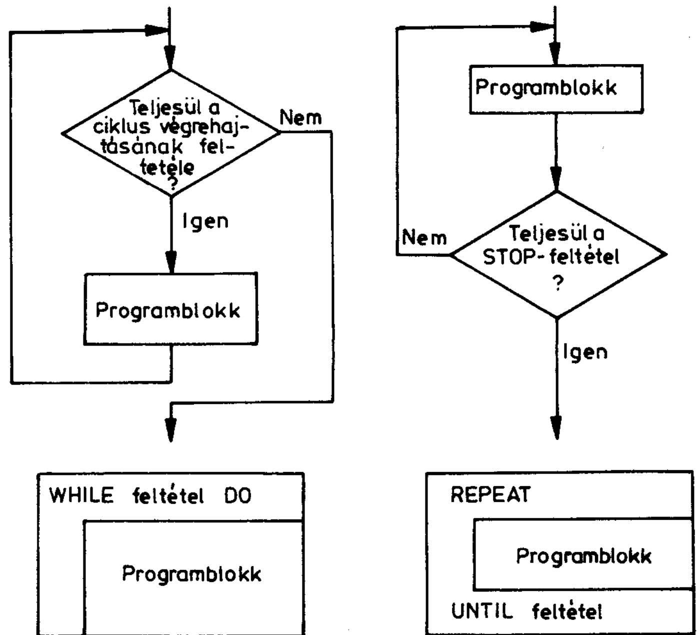
4.7.3. Szinkronizációs utasítások

A 2.2.6. pontban megismert párhuzamos feldolgozási folyamatok szinkronizálásához egy speciális változó, az ún. szemaforváltozó használható. Ennek – a logikai típusú változókhoz hasonlóan – logikai igen/nem értéket adhatunk (más szóval a szemafor beállítható, ill. visszaállítható). A szemafor ugyanakkor a ciklusváltozókhoz hasonlóan számlálásra is használható. Az első esetben a szemafor segítségével letiltható egy feldolgozási folyamat, ill. újra engedélyezhető. A második esetben a tiltások és engedélyezések számával lehet operálni (l. a 2.2.7. pontot). A párhuzamos feldolgozási folyamatok részleteire és szervezésükre a 7. fejezetben még visszatérünk.

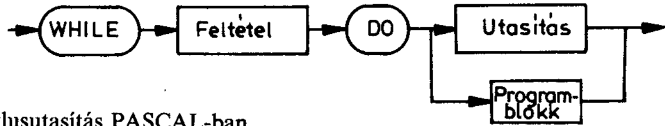
AL:

AL-ban a szinkronizálást EVENT (esemény) típusú változók segítségével szervezhetjük meg. A

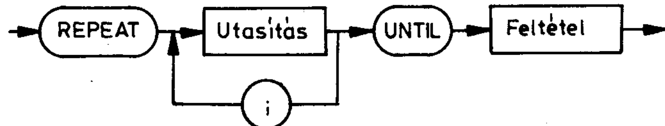
SIGNAL⟨eseményváltozó⟩



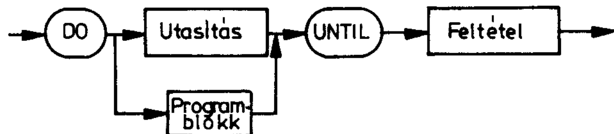
4.98. ábra. Feltételtől függő ciklus szervezése



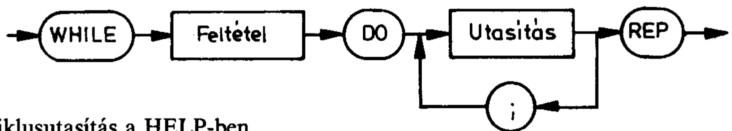
4.99. ábra. WHILE-ciklusutasítás PASCAL-ban



4.100. ábra. REPEAT-ciklusutasítás PASCAL-ban



4.101. ábra. DO-ciklusutasítás AL-ban



4.102. ábra. WHILE-ciklusutasítás a HELP-ben

megnöveli a szemafor ⟨**eseményváltozó**⟩ értékét. Ha az eredmény kisebb/egyenlő nullával, akkor a várakozó feldolgozási folyamat ismét felszabadul. A

WAIT ⟨eseményváltozó⟩

csökkenti az eseményváltozó (szemafor) értékét, és mihelyt az eredmény negatívvá válik, akkor annak a folyamatnak kell várakoznia, amelyben ez a parancs szerepel. Ha pedig a szemafor eléri a nulla vagy a pozitív értéket, akkor a feldolgozási folyamat ismét elindul. Ezzel a módszerrel két vagy több folyamat is szinkronizálható.

HELP:

HELP-ben csak logikai típusú szemafor használható. Az utasítások alakja hasonló az AL-ban megismert utasításokhoz. Ezek a

SIGNAL (⟨n⟩),
WAIT (⟨n⟩),

ahol ⟨n⟩ a szemafor sorszáma. Az, hogy maximálisan hány szemafor használható, az implementációtól függ. A WAIT addig vár, amíg a megadott szemaforváltozót egy SIGNAL-utasítás ismét beállítja. A WAIT ezután ismét visszaállítja a szemaforváltozót. Nincs rögzítve, hogy hogyan viselkedik a WAIT-utasítás, ha a szemafor már eleve beállított állapotban van. (Feltehető, hogy nem jön létre várakozás, csak a szemafort állítja vissza.)

A

TEST (⟨n⟩)

eljárás segítségével meghatározható egy szemafor állapota. Beállított szemaforváltozónál a TEST eredménye TRUE.

SIGLA:

A SIGLA nyelvben az

ES/⟨változó⟩;

utasítással beállítható egy számláló vagy paraméter, vagy egy indirekt címzésű számláló. Amikor a program elérkezik az

EW/⟨változó⟩, ..., ⟨változó⟩;

utasításhoz, akkor addig várakozik a program, amíg valamennyi hozzá tartozó változó be nem állítódik.

---

1. feldolgozási folyamat

```
:

WAIT megfogta;
:

SIGNAL tovább;
:

:

:
END (* hibakezelés vége *)
```

2. feldolgozási folyamat

```
ok ← false;
MOVE kar TO megfogpont
CLOSE kéz TO 5 * cm;
MOVE kar TO megfogpont + 5 * zhat * cm
  ON FORCE (zhat) > = önsúly
  DO BEGIN
SIGNAL megfogta;
ok←true;
END

WAIT tovább;
IF NOT ok THEN
BEGIN (* hibakezelés *)

  SIGNAL megfogta
```

4.18. programrészlet. Dead'lock keletkezése párhuzamos feldolgozási folyamatoknál AL nyelven írt programban

A VAL és a ROBEX nyelvekben nincs semmilyen nyelvi eszköz szemaforral végrehajtható szinkronizálás megoldására. Ha ennek ellenére szinkronizálási feladatot kell megoldani, akkor némi programozói ügyességgel külső kétállapotú jelek fogadására és kezelésére alkalmas utasításokhoz kell folyamodnunk. Ezeket az utasításokat eredetileg a megfelelő inputok, ill. outputok rövidre zárásával külső érzékelők és periferiális eszközök kezelésére hozták létre. Ilyenek a VAL-ban a WAIT- és SIGNAL-utasítások, míg ROBEX-ben a SWITCH- és a WAIT-utasítások.

A szinkronizált folyamatok dinamikai viselkedését igen gondosan kell elemezni – különösen kettőnél több feldolgozási folyamat vagy bonyolultabb programok esetén –, nehogy kölcsönös blokkolás állhasson elő (ún. dead-lock helyzet), l. a *4.18. programrészletet!* Ha itt a 2. feldolgozási folyamatban a robot nem talál alkatrészt, amit megfoghatna, akkor a rendszer leáll, mert a WAIT-utasítás – hibásan – nem a hibakezelés után, hanem előtte helyezkedik el.

4.7.4. Várakozási utasítások

Minden robotvezérlő nyelvben megvannak azok az utasítások, melyek hatására a program egy meghatározott esemény vagy szituáció bekövetkezéséig várakozási állapotba kerül. A várakozás okai pl. a következők lehetnek:

- időzítési feltétel;
- terminálról bekért válasz;
- meghatározott billentyű működtetése;
- vagy valamilyen külső jel beérkezése (l. még a 4.5. szakaszt).

A *4.2. táblázatban* áttekintjük a programvárakozást előidéző utasításokat. A HELP nyelv HIGH rutinjának kivételével a bemutatott utasítások, ill. standardeljárások nem

4.2. táblázat. Várakozási utasítások

Várakozási feltétel	AL	VAL	HELP	SIGLA	ROBEX
Időzítési feltétel	PAUSE ⟨s⟩ min	DELAY ⟨r⟩ min	DELAY (⟨s⟩) 20 ms-os egységben	WA/⟨a⟩ 20 ms-os egységben	DELAY/⟨n⟩ min
Terminálról várt válasz:	PROMPT (⟨printl⟩) P	PAUSE ⟨text⟩ PROCEED	./.	./.	./.
Meghatározott funkciós billentyűk működtetése	./.	./.	HOLD MAYRUN	HL	CSTOP
Külső jel	WAIT ⟨e⟩	WAIT ⟨kl⟩	HIGH (⟨k⟩, ⟨s⟩) ⟨s⟩ 20 ms-ben	./.	WAIT/ EVENT, ⟨k⟩

Paraméterek jelentése:

⟨k⟩ és ⟨kl⟩ csatornaszám, kl 1 és 8 közötti szám
⟨s⟩ SCALAR típusú
⟨r⟩ REAL típusú
⟨a⟩ számlálóváltozó, paraméter vagy indirekt címzésű számláló
⟨printl⟩ az AL ⟨printlist⟩-je
⟨e⟩ előre definiált eseményváltozó

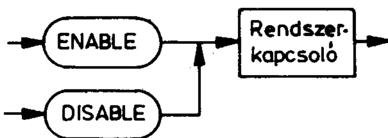
szorulnak különösebb magyarázatra. A HIGH rutin azt idézi elő, hogy a program **(s) * 20 ms** ideig várakozik a ⟨**k**⟩-adik csatornán beérkező 0/1-es jelátmenetre. Ha nem adunk meg időtartamot, akkor a várakozás 30 s-ig tart. Ha ezalatt az idő alatt jelváltás nem észlelhető, akkor üzenetet küld a terminálra (ha van terminál), majd HOLD-ba megy át, vagyis a program mindaddig várakozik, amíg a „RUN" billentyűt meg nem nyomják.

4.8. Rendszerkapcsolók

Egyes programozási nyelvekben lehetőség van a felhasználói programban elhelyezett utasítások segítségével bizonyos *rendszerkapcsolók* beállítására, ill. visszakapcsolására. Ezzel meghatározott rendszerfunkciókat lehet aktivizálni, ill. letiltani. Ennek a lehetőségnek biztonsági szempontból van jelentősége, de tárolóterület, vagy időmegtakarítás szempontjából is hasznos lehet. A robotberendezések programozásánál nagyon fontosak azok az utasítások, amelyek a robot *kalibrálására,* valamint a robottengelyek nullpozíciójának újradefiniálására valók.

A VAL nyelvben a rendszerkapcsolók beállítására, ill. visszakapcsolására az ENABLE-, ill. a DISABLE-utasítások szolgálnak *(4.103. ábra)*. Az egyes rendszerkapcsolók a következők:

CP	Pozicionálás közbenső frame-helyzetek érintésével (4.2.4. pont).
CRT	ROBOUT-billentyűvel rendelkező terminálra.
EHAND	A rendszerben van villamos hajtású, ill. pneumatikus megfogószerkezet is (4.3.1., ill. 4.3.2. pontok).
MESSAGES	A TYPE, ill. TYPEI utasításnál az üzenet terminálra küldése, ill. letiltása (l. 3.4. szakasz).
SRV.ERR	Speciális hardverhibák automatikus ellenőrzése, ill. ennek letiltása.
VISION	A vizuális rendszer inicializálása, ill. a kommunikációs kapcsolat kiépítése.



4.103. ábra. Rendszerkapcsolókat beállító utasítások VAL-ban

VAL:
Enable EHAND
OPEN 50
Ezzel előírjuk, hogy a rendszer a villamos hajtású megfogószerkezetnek adja ki a parancsokat. Ezután az OPEN-utasítással 50 mm-nyire nyitjuk a megfogószerkezetet.

4.9. Kivételes helyzetek kezelése

Kivételes helyzeteken olyan programmegszakítást értünk, amikor előre definiált kritériumok teljesülése idézi elő a normál programfutás megszakítását. Kivételes helyzet adódhat, ha pl. hibás adatok miatt nullával vagy túl kis számmal való osztásra kerül sor. Másrészt készüléktechnikai, vagy időzítési okok miatt is keletkezhet futási hiba: pl. hibásan beállított adatátviteli sebesség, vagy szinkronizációs hiba esetén. A vezérlőrendszer, ill. az operációs rendszer a kivételes helyzetekre általában *automatikusan* reagál: vagy megszünteti a hiba okát, vagy megállítja a programot. Arra azonban többnyire nincs mód, hogy a *rendszer reakcióit* a programból befolyásoljuk. Az újabb – elsősorban a folyamatirányítás terén használt – programozási nyelvekben megtalálhatók a különleges szituációk kezelésére alkalmas nyelvi szerkezetek. Ilyenek az Industrial-Real-Time-BASIC, a PL/1, az ADA és a PEARL. A robotvezérlő programozási nyelvek közül jelenleg egy sem ismeretes, amelyben megtalálhatók lennének a különleges helyzetek tradicionális kezelésére alkalmas megoldások, vagy akár csak távlatilag is szó lenne a *robotokra jellemző* speciális kivételes üzemállapotok programtechnikai kezeléséről. Emiatt egy, a PEARL nyelvből vett példán mutatjuk be a kivételes üzemállapotok kezelésének különféle módszereit, majd javaslatot teszünk arra, hogy ezeket a módszereket hogyan terjeszthetnénk ki a robotprogramozás speciális kivételes üzemállapotaira.

Épp a gyártási feladatok ciklikus feldolgozása az a terület, ahol nem megengedhető, hogy egy adatátviteli hiba a felhasználói program azonnali leállítását okozza. Célszerűbb előzetesen kialakítani egy olyan megoldást, hogy a program egyáltalán normálisan folytatódhasson, vagy legalább definiált programvége utasításon lépjen ki.

A *kivételes üzemállapotoknak* két típusát különböztetjük meg:

1. A rendszer adottságai miatt meglevő, előre definiálható kritériumokat. Ezek előre deklarált állapotjellemzővel, vagy azonosítónévvel jelölhetők.
2. A programozó által deklarált kivételes üzemállapotokat. Ezek logikai kifejezésekkel határozhatók meg: pl. DURATION > 5.

A PEARL-ben a rendszer kivételes helyzeteinek első típusát a felhasználó közvetlenül is megnevezheti, tehát kezelheti. Példák: OVERFLOW, ENDOFILE. Ha ezek a kivételes helyzetek fennállnak, akkor a normál programmenet megszakad, és valamilyen megadott programrészlet kerül végrehajtásra. A robotpozicionálás műveleténél maradva az AL és a HELP nyelveknél programon belül a 2. esetnek megfelelően deklarálhatunk kivételes üzemállapotot (l. 4.2.5.1. és 4.2.7. pontok). A kivételes üzemállapot fellépésekor aktivizált programrészlet – más néven *handler* – állhat akár egyetlen utasításból, vagy blokkból is, vagy lehet maga is egy speciális alprogram. A handlereket vagy a programon belül dinamikusan rendeljük hozzá a kivételes üzemállapothoz, vagy statikusan a handler deklarálásakor. PEARL-ben az utóbbi eljárást követik.

PEARL:

```
ON ENDOFTAPE: BEGIN;
:                    A kivételes állapot kezelésére alkalmas
                     utasítások
END;
```

Az ON-utasítás segítségével előre lerögzítjük, hogy mágnesszalag olvasásakor a szalagvége jel elérésekor a program a megadott blokkot aktiválja a kivételes üzemállapot kezelésére.

A PEARL nyelv ON-utasításának szerkezete sokban hasonlít az AL nyelv pozicionáló utasításánál az eseményfelügyelet, ill. az időzítés- és az érzékelőfelügyelet logikai szerkezetére (l. *4.104. ábra,* ill. 4.2.5.–4.2.7. pontok). A kivételes helyzetek eddig megismert kezelési módszerei tulajdonképpen a pozicionálás felügyeletének általánosabb eljárásába illeszthetők.

PEARL-ben szabadon lehet választani a rendszer által biztosított kezelési eljárás (jele: SYS) és a felhasználó által írt handler közt.

4.104. ábra. Pozicionálás-ellenőrző utasítás általános struktúrája az AL-ban

---

PEARL:
ON ENDOFTAPE: SYS;
A rendszer ekkor az ENDOFTAPE nevű kivételes állapotra saját hibakezelő rutinjával reagál.

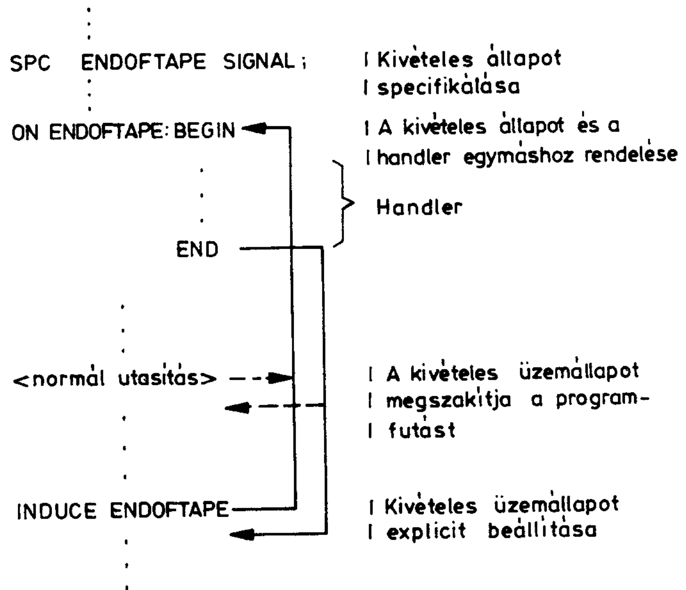
A kivételes üzemállapot bekövetkezését a rendszer általában akkor észleli, amikor a kérdéses helyzet előáll. A bemutatott példában egy olvasási utasítás végrehajtása közben a szalagvége jel olvasás hatására beállítódik az ENDOFTAPE feltétel. Teszteléskor azonban általában nem tudunk kivételes üzemállapotot mesterségesen előidézni, vagy csak igen költséges módszerekkel, és így az állapotfeltétel teljesülését és a handler behívását sem szimulálhatjuk. Az állapotfeltétel közvetlen beállítására ezért az INDUCE-utasítást használjuk.

PEARL:
INDUCE ENDOFTAPE:
Ugyanazt a reakciót váltja ki, mint a szalagvége jel tényleges olvasása.

Esetenként hasznos lenne, ha egyes programszakaszok végrehajtása közben valamely kivételes üzemállapot kezelését letilthatnánk. Ilyenkor az állapotfeltételt figyelmen kívül szeretnénk hagyni. Erre a PEARL-ban nincs megfelelő utasítás. Az Ipari Real-Time-BASIC (IRTB) nyelvben ezzel szemben a handler ENABLE-lel aktivizálható, DISABLE-lel pedig letiltható.

IRTB:
230 ENABLE HANDLER overflow
240 LET exprl = LOG (x/y+a)
250 DISABLE HANDLER
Az aritmetikai túlcsordulás kezelésére alkalmas HANDLER-t csak a 240-es utasítás időtartamára aktivizáljuk.

A *4.105. ábrán* összefoglalva bemutatjuk a kivételes állapotok kezelésének módjait PEARL-ben.



4.105. ábra. Kivételes állapot lekezelésének lehetőségei PEARL-ban

Robotvezérlések programozásánál az eddig megismert kivételes üzemállapot-típusokon kívül újabb speciális kivételes esetek fordulhatnak elő, ezek külön deklarálhatók. Példa erre a VAL nyelv GRASP-utasítása (l. 4.3.3. pont). Itt a különleges helyzet az, hogy „nem sikerült alkatrészt megfogni". Ilyenkor a programban a vezérlés egy megadott címkére adódik át. Ebben az esetben a kivételes állapotot helyzetérzékelők segítségével tudjuk észlelni. Ezek a megfogószerkezet összezáródása után a szerkezet záródásának mértékét érzékelik. Az ellenőrzésére vizuális rendszert is használhatnánk, amely az alkatrészek, ill. a megfogószerkezet pozícióit ellenőrizhetné.

Robotprogramozáskor a következő kivételes állapotok fordulhatnak elő:

- a robot ki van kapcsolva;
- a munkadarab hiányzik, vagy rossz helyre került;
- a program szerinti robotmozgás ütközéshez vezetne, vagy a pozicionálás célpontja kívül esik a robot munkaterén;
- túl nehéz tárgyat kell felemelni, vagy a tárgy rögzítve van;
- túl hosszú ideig kell várni a pozicionálás vége vagy más egyéb jelre;
- alkatrész beillesztésekor vagy becsavarásakor a robot nem találja a furatot;
- sorozatban kezelt alkatrészek közül az egyik nehezebb/könnyebb, nagyobb/kisebb, mint a többi stb. (selejtes alkatrész);
- a rakodóhely megtelt/kiürült;
- hirtelen erőhatás éri a robotkart, ill. a végrehajtó szervet.

5. A betanítási eljárás a programírásban

Az ipari gyakorlatban jelenleg a legtöbb robotot nem szövegszerűen, hanem betanítási eljárással (teach-in-eljárás) programozzák (l. 2.1. szakasz). Ennek az az oka, hogy egyrészt maguk a feladatok sem igazán bonyolultak, másrészt nem alkalmazzák az érzékelővezérlést, nem alkalmaznak környezetleíró adatbázist, és hogy a pozicionáláshoz szükséges frame-adatokat nem lehet absztrakt módon derékszögű vagy hengerkoordinátákban programozni. Ezek a problémák többnyire egy alkalmas robotvezérlő programnyelv használatával oldhatók meg, az utolsóként említett problémán azonban már egy jó szimulációs rendszer is igen sokat segíthet. Ismeretes, hogy a *szövegszerű programozás* egyik gyenge pontja a pozicionálás frame-adatainak pontos definiálása. Ehhez a koordinátaadatokat programozáskor ténylegesen bemért adatokkal kell megadnunk, hiszen ezt sem becsléssel, sem a térbeli viszonyok egzakt, programozott leírásával nem lehet helyettesíteni. Emiatt a legtöbb szöveges programnyelv a frame-adatok definiálására a betanítási eljárást használja.

A betanítási eljárásnál a *pályapontokat,* ill. a pálya jellegzetes frame-helyzeteit azzal határozzuk meg, hogy a robotot, ill. a robotra szerelt végrehajtó szerv TCP pontját (*T*ool *C*enter *P*oint, azaz a szerszám középső pontja, 2.4.2. pont) a kérdéses pozícióba állítjuk, miközben a végrehajtó szerv meghatározott térbeli pozícióját, ill. orientációját is rögzítjük. A programozó ilyenkor különböző funkcióbillentyűk működtetésével vezérelheti a robotberendezést, és amíg a billentyűt lenyomva tartja, a robot a billentyűnek megfelelő funkciót hajtja végre. Eközben vizuálisan is ellenőrzni kell a robotot, és el kell dönteni, hogy a robot beállt-e már a kívánt pozícióba, és egyben megfelel-e a beállított orientáció. Ezzel kapcsolatban megemlítjük, hogy a funkcióbillentyűket a robot térbeli mozgását szemléltető vonalas ábrákkal szokták jelölni (pl. a nyíllal ellátott félkörív ↷ a megfelelő irányú elfordítást jelöli). A rajzos szimbólumok kifejezőbbek, a felhasználó az így jelölt funkciókat térbelileg jobban el tudja képzelni, és így gyorsabban meg tudja oldani az információbevitelt. Az információbevitel nehézségei egyben arra is rávilágítanak, hogy a kezelő által vezérelt explicit pozicionálásnak is megvannak a maga korlátai. A pozicionálás explicit programozásakor a betanítási eljárás alkalmazásának az a nagy előnye, hogy folyamatosan ellenőrizhetők a frame-adatok és az esetleges hibát azonnal lehet javítani. Robotberendezés használata nélkül ilyen visszacsatolásra nincs lehetőség, szimulációval is csak korlátozott mértékben. A betanítási eljárásnak ugyanakkor megvan az a hátránya, hogy a programozási munka közben a robot kiesik a termelésből. Ezért a betanítási rendszer on-line részét csak azokra a pozicionálási frame-adatokra szokták korlátozni, amelyek szövegszerű programozása túlságosan időigényes lenne, a programozás többi részét pedig robot nélkül off-line módszerrel fejezik be. A teljes programozási folyamat tehát két részre bontható:

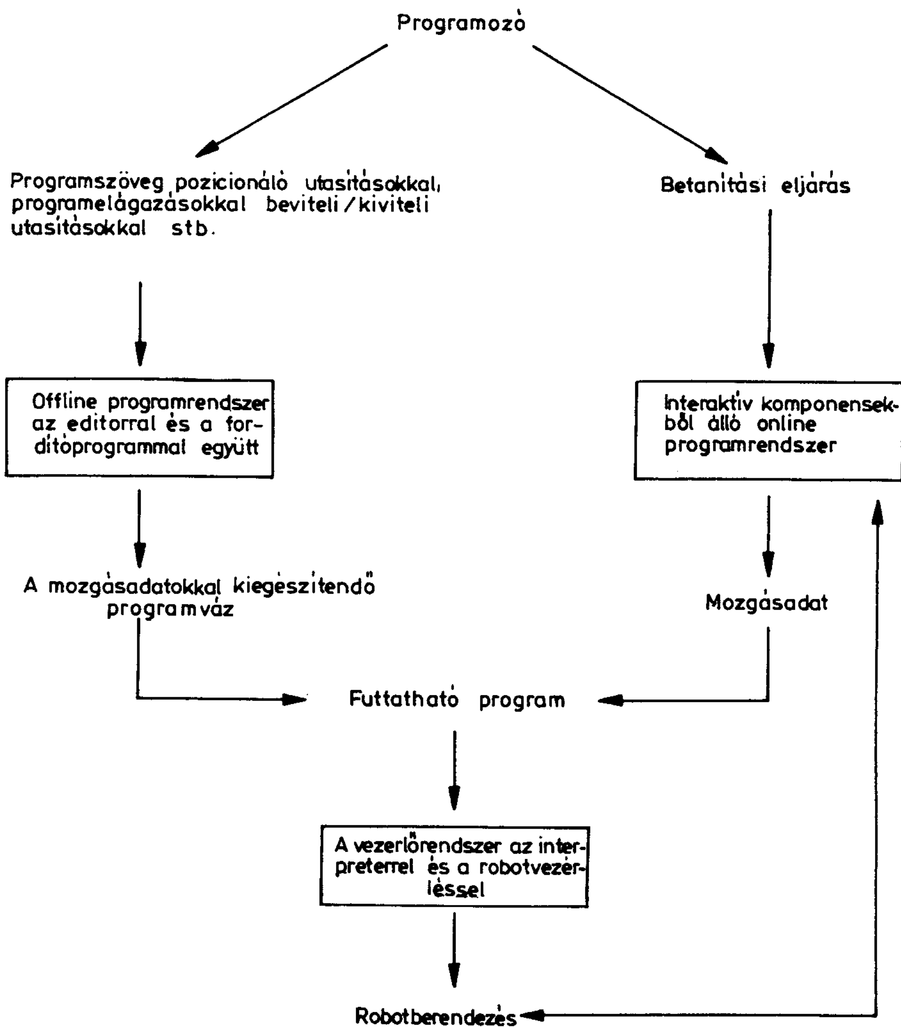
1. A programrendszer off-line részének segítségével előállítjuk a program egy még kiegészítendő változatát, amely már minden utasítást tartalmaz: a pozicionálásokat is és az explicit konstansokat is. Ilyenkor a pozicionáláshoz használatos frame-rend-

szereknek már csak azok a koordinátaadatai hiányozhatnak, amelyek programbeli szöveges megadása lehetetlen.
2. A programrendszer on-line (interaktív) komponensének felhasználásával betanítjuk a robotot a hiányzó pályaadatokra, ill. frame-adatokra.

A futtatásvezérlő rendszer a frame-adatokból és a vázlatos programkódból előállítja a futtatható programot (l. *5.1. ábra*). Ugyanez a futtatásvezérlő rendszer vezérli a kész program végrehajtását is.

A pozicionálásokhoz és a pályameghatározásokhoz tartozó frame-adatok koordinátaadatai a következő két alapelv szerint rendelődhetnek hozzá a megfelelő pozicionáló utasításokhoz:

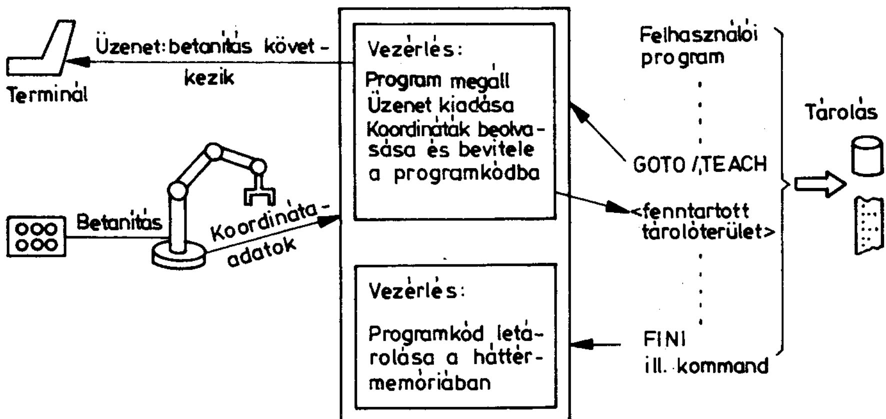
1. A programkódban címkét kap minden pontos adatával még meg nem határozott frame-rendszer, ill. a hozzá tartozó pozicionáló utasítás, valamint ezek koordinátái-



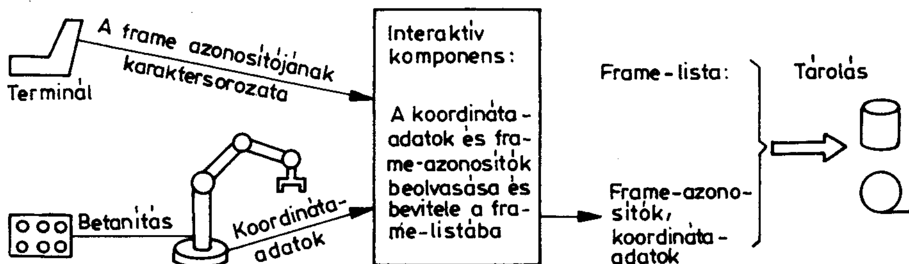
5.1. ábra. Ipari robotok kombinált on-line és off-line programozása

ra a tárolóterületet is lefoglaljuk. A program első futásakor a futtatásvezérlő rendszer a felcímkézett pozicionáló utasításoknál megállítja a felhasználói programot és kiad egy üzenetet. Ekkor a programozónak a hiányzó frame-adatokat kell definiálnia.
Programozói üzenet hatására a koordinátaadatok bekerülnek a programba, és a program futása folytatódik (l. *5.2. ábra*). Ennek a módszernek megvan az a hátránya, hogy az első programfutás alkalmával az összes utasítást betanítási címkével kell ellátni. Ez nehezen teljesíthető különféle programelágazások és érzékelővezérelt szubrutinhívások esetén. Ha pl. az első futás alkalmával egy logikai IF...THEN utasításnál a logikai feltétel nem teljesül és a THEN ágban frame-definíció következik, akkor a hiányzó adatok betanítási eljárással nem adhatók meg. Ha ezután a THEN-ág, mondjuk a 40. programciklusban végrehajtódik, amikor a logikai feltétel a megváltozott érzékelőadatok, ill. változók tartalma miatt már teljesül, akkor a futtatásvezérlő rendszer, ill. a robotvezérlés a termelési folyamat kellős közepén megállítja a programot a kérdéses pozicionáló utasítás előtt. Ezenkívül a teljes programkódot újból le kell tárolni a háttérmemóriában, vagy pedig a program minden egyes betöltése előtt meg kell ismételni a betanítást. Mégis e módszer mellett szól az, hogy normál programfutás közben nem merül fel semmilyen programszervezési feladat.

2. A pozicionálások frame-adatai a futtatásvezérlő rendszertől teljesen függetlenül is definiálhatók. A betanítási eljárással beállított frame-eket rendszerparanccsal beolvassuk, és a megadott frame-azonosító neve alatt egy ún. frame-listába jegyeztetjük be. Ilyenkor a program végrehajtása közben a futtatásvezérlő rendszer, ill. az interpreter valamennyi pozicionáló utasításnál az adatmegadás nélküli frame(ek) koordinátaadatait ebből a frame-listából veszi. Az egyes frame-eket azonosítójuk alapján ismeri fel *(5.3. ábra)*. Ennek a módszernek az az előnye, hogy teljesen független a program generálásától, a betanítás előtte is, utána is elvégezhető, és a definiált frame-adatok utólag mégis egyszerűen megváltoztathatók. Valamelyes hátrányt jelent, hogy a program futása közben jelentősebb programszervezés szükséges, mivel a program minden egyes indításakor be kell olvasni a frame-listából az adatokat és ezekkel az adatokkal fel kell tölteni a frame-változókat.



5.2. ábra. A betanítási eljárás kapcsolata a programmal címkézett utasítások használatával ROBEX-ben



5.3. ábra. A betanítási eljárás kialakítása frame-lista segítségével az AL és a VAL nyelvek esetén

Az AL és a VAL nyelvekben a betanítási eljárás frame-listával történik. AL-ban csak a derékszögű koordinátákban kifejezett adatok tárolhatók, VAL-ban azonban lehetőség van mind derékszögű koordinátákkal definiált frame-adatok, mind pedig robotkoordinátákban kifejezett ún. precíz frame-ek megadására. Az AL nyelvben ezzel szemben a programon belül explicit számadatokkal is megadható a frame, és így sok esetben elkerülhető a betanítási lépés. A VAL nyelvben a frame-lista nem egy elkülönített tárolóterületen jön létre, hanem láncolt listaként a programkódterületen (abban elszórtan) helyezkedik el.

VAL-ban ezenkívül van egy nagyon kényelmes módszer, amivel a pozicionálások programozása betanításos módszerrel történhet, és ugyanakkor az egyes pozicionáló utasítások *jól olvasható* és *javítható* programszöveget alkotnak. Ezzel kapcsolatban az *editor* egy különleges eljárást hajt végre, ha az operátor megadja a T vagy a TS opciókat. Emellett a T-t, ill. TS-t követően egy frame-azonosítót (pl. tárolópolc) is meg kell adni. Ezután nem szöveget kell megadni, hanem a robottal betanításos módszerrel felvett frame-helyzetek sorozata következik. Amikor a kívánt frame-helyzetet sikerül beállítani, megnyomjuk a betanító terminál RECORD billentyűjét, amire a VAL programrendszer bejegyzi az 1. tárolópolc, 2. tárolópolc stb. frame-helyzetekre irányuló pozicionáló utasítást a megfogóeszköz pillanatnyi nyitási adataival együtt. A terminál CR (carriage return) kocsi vissza billentyűjének leütésével a rendszer kilép a betanítási üzemmódból, és az editálás folytatódhat. A T-opció kiválasztásának hatására a pályavezérlésekre a program a tengelyinterpolációs eljárást (MOVET) írja elő, míg a TS-opcióval a lineáris derékszögű koordinátás interpolációt (MOVEST) valósíthatjuk meg (l. még a 4.2.4. pontot is).

VAL:

T magazin	Editáláskor a betanítási fázis kezdete
	Az 1. frame betanítása RECORD-billentyű lenyomásával
MOVET magazin 1, 0.0	Szövegbevitel. A megfogószerkezet zárva volt
.	A 2. frame betanítása RECORD-billentyű lenyomásával
MOVET magazin 2, 40.7	Szövegbevitel. A megfogószerkezet nyílása 40.7 mm

A bemutatott eljárással egész mozgássorozatok programozhatók anélkül, hogy erre külön szövegszerű programot kellene írnunk.

A HELP nyelvben a betanítás megvalósításához egy ettől kissé eltérő módszert alkalmaznak, amelyben címkézett utasítást, ill. különleges utasításokat kell igénybe venni. Programozáskor először meg kell írni az ún. „Self-teach" (önbetanító) programokat, és az ezekben elhelyezett MANUAL-utasítás hatására megkezdődik a robot betanítása. A koordinátaadatokat a program segítségével olvassuk be, majd ugyanezeket az adatokat a program az értékadó utasításokban használja. Ezután azonban nem az egyes változókat tároljuk le a periferiális tárolóegységen, hanem magát a változót tartalmazó utasítást. Erre a célra a 3.1.5.3. pontban megismert fájlba történő írási utasítást kell használni.

HELP:

MANUAL(1);	A betanítási üzemmód behívása; az 1. kar betanítása
CREATE('teachprog');	A **teachprog** fájl létrehozása
PRINT('teach frame kezd:')	Terminálra küldött üzenet; (tanítsd be a kezd nevű frame-et!)
COORD(1);	Várakozik a Joy-Stick-en egy billentyű lenyomására, ill. az aktuális pozíció koordinátaadatainak beolvasására
RECORD (#0, kezd :=', AX(1),';');	Az utasítások felírása a **teachprog** nevű fájlba
RECORD(#0, 'kezdy := ', AY(1),';');	
:	
RECORD(#0, 'MOVE(1, #1, kezdx, #2, kezdy, ..., #6, kezdyw)');	további utasítások ill. a betanítás folytatása
:	
RECORD(#0, 'STOP');	Programvége felírás a fájlba
CLOSE	A **teachprog** nevű fájl lezárása
STOP	Programvége

Az 1. robotkarra vonatkozó betanítási üzemmód behívását követően a program létrehoz egy fájlt, ahova az utasításokat írja, majd kiküld egy üzenetet a kezelőnek, és ezután lehetővé válik a **kezd** nevű frame-re történő pozicionálás betanítási üzemmódban. A robotberendezést ilyenkor a botkormánnyal – az ún. joy-stick-kel kormányozzuk. A COORD-utasítás hatására a program addig várakozik, amíg a beállítás megtörténik. Ezt a joy-stick-en egy külön gomb megnyomásával nyugtázzuk, mire a koordinátaadatokat a rendszer beolvassa. Az ezt követő utasításokba ezek az adatok kerülnek be, majd az így kitöltött utasítások a **teachprog** fájlba íródnak. Ugyanez az eljárás a többi frame-re is megismételhető.

A SIGLA nyelvben nincs betanítási üzemmód. A ROBEX nyelvben a betanítást a pozicionáló utasításban feltüntetett TEACH alapszóval írjuk elő (l. az *5.2. ábrát*). A program első futását követően a teljes programkódot ismét ki kell tenni a háttérmemóriába.

6. Alprogramok, eljárások és függvényeljárások

A 2.2.4. és 2.2.5. pontokban már szóltunk az alprogram, az eljárás, a függvényeljárás, valamint a rekurzív eljáráshívás általános fogalmáról. Most áttekintjük az ezekkel kapcsolatos különféle programozási nyelvbeli lehetőségeket, és kitérünk a gyakorlati alkalmazás során felmerülő általános problémákra.

A jól tagolt – ún. strukturált – programban nem pusztán az ugróutasítások használatát kell elkerülni, hanem az egyes programbeli részfeladatok elemi funkciókig való precíz és finom felbontásával az egész programot alárendelt feladatokat megvalósító egységekre célszerű felbontani. Ezekre az elemi feladatokra, vagy legalábbis a nagyobb részfeladatokra célszerű olyan programrészleteket írni, hogy ha ezek belsejében valamit változtatni kell, annak ne legyen kihatása a programbeli környezetre. Erre a célra valók az illesztési pontok, ahol azt definiáljuk, hogy a kérdéses részfeladatnak melyik adatállománnyal van kapcsolata. Amennyiben egy programnyelvben lehetőség van eljárások vagy függvényeljárások szerkesztésére, úgy ilyenkor ezeknek az eljárásoknak a paraméterátadási mechanizmusa a legalkalmasabb nyelvi eszköz az illesztés megvalósítására. Magát a feladatot az eljárás törzsében megvalósított programrészlettel hajtjuk végre. Az eljárást természetesen tovább tagolhatjuk további lokális feladatokat kezelő eljárások deklarálásával. Ezek az alprogramok azonban külső blokkból nem hozzáférhetők, és ezért más részfeladat kezelésekor már nem használhatók fel. Az ilyen több részfeladatban is felmerülő elemi feladatokat kezelő programrészleteket a programban célszerű globálisan definiálni.

Nagyon fontos, hogy az egész programban az egyes részfeladatok illeszkedési felületeit tisztán és áttekinthetően szerkesszük meg. Ezzel a programbiztonságot nagymértékben fokozhatjuk. Elvárható, hogy minden magasabb szintű programnyelvben megtalálhatók legyenek azok a nyelvi elemek, amelyek ezt elősegítik. Ezek az elvárható megoldások a következők:

- a paraméterek specifikációja az eljárás fejrészében;
- a bemeneti paraméterek átadása érték szerinti paraméterátadással *(Call-by-value);*
- azoknál a paramétereknél, amelyek értékét az eljárás megváltoztathatja, hivatkozás szerinti paraméterátadás *(Call-by-reference)* biztosítása a mellékhatások kiszűrésére:
- az eljárásoknak csak egyetlen kilépési pontja legyen.

A tárgyalt nyelvek közül csak a PASCAL és az AL teljesítik maradéktalanul ezeket a követelményeket.

Ha biztosítani tudjuk, hogy a rutin valamennyi bemeneti és kimeneti adatot az előbbi követelményeknek megfelelően kezeljen, akkor az illeszkedési pont megbízhatóan és jól alakítható ki. Nem áll ugyanez az olyan illesztési pontra, ahol éppen az ún. mellékhatások nyújtotta lehetőségeket kívánjuk kihasználni. *Mellékhatáson* azt értjük, amikor az eljárás törzsében a globális adatokat közvetlenül – tehát az alprogram paramétereinek közbeiktatása nélkül – változtatjuk meg.

Mellékhatásokat tartalmazó program vagy helytelen programozói stílusra vall, vagy

a feladatot nem sikerült tisztán elkülönülő részekre tagolni, ami igen gyakran logikai hibákhoz vezet. Mellékhatásokat csak kivételes esetben indokolt kihasználni, pl. olyan alprogramban, amelynek egyetlen feladata az adatállományok megváltoztatása. Ilyenek az ún. tárfunkciók.

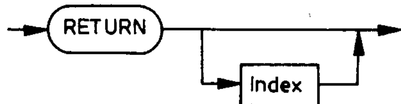
Minthogy a robotprogramozás során nemcsak belsőleg ábrázolt adatokon végzünk műveleteket, hanem elsősorban valós eszközöket működtetünk, ezért egy külön eljárásba beillesztett pozicionáló utasítással is előidézhetünk mellékhatásokat. Az ilyen eljárás után ugyanis a robotkar nem ugyanabban az állapotban marad, mint az eljárás aktivizálása előtt volt. Ezt az állapotot tehát, amely legalábbis a megváltozott pozíció- és orientációadatokkal, esetleg a megfogott munkadarabbal jellemezhető, közölni kell a programbeli környezettel is, éspedig az illesztési ponton keresztül.

6.1. Alprogramok

Mint a *3.3. táblázatból* kitűnik, a ROBEX nyelv kivételével a többi nyelvben megvan a lehetőség alprogramok szerkesztésére. Lássuk először a VAL és a HELP megoldásait:

VAL:
A *6.1. ábrán* látható utasítás hatására a vezérlés átadódik a megnevezett program első utasítására, majd ez a program hajtódik végre a RETURN-utasítás megjelenéséig (l. *6.2. ábra*).
Ha az ⟨index⟩-változó nincs megadva, vagy tartalma kisebb egyenlő nullával, akkor a program annál az utasításnál folytatódik, amelyik a hívóprogramban a GOSUB után következik. Ha ⟨**index**⟩ tartalma pozitív, akkor ez hozzáadódik a következő utasítás címéhez. Így ha ⟨**index**⟩ tartalma 2, akkor a program a GOSUB-utasítást követő harmadik utasítással folytatódik. A főprogramon belül elhelyezett RETURN-utasítás a STOP-utasításhoz hasonlóan működik, vagyis leállítja a programot. Mint a 4.2.6. pontban láttuk, az alprogramhívást a REACT- és REACTI-utasítások segítségével is

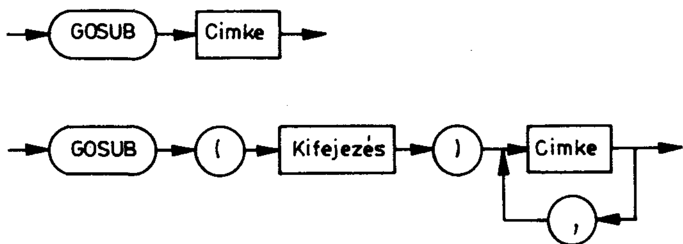
6.1. ábra. Alprogram hívása VAL-ban



6.2. ábra. RETURN-utasítás VAL-ban

meg lehet valósítani. Ilyenkor a RETURN REACTI esetén a megszakított, REACT-nál pedig a soron következő utasításra vonatkozik. VAL-ban az alprogramok egymásba ágyazása tízes mélységig van megengedve.

HELP:
HELP-ben az alprogramhívás szerkezete hasonló a VAL-hoz, azzal a különbséggel, hogy a RETURN-utasítás minden esetben a főprogram soron következő utasítására adja vissza a vezérlést, és hogy az alprogramot magában a főprogramban címke segítségével deklaráljuk. Az alprogram hívásának lehetséges két változatát a *6.3. ábra* szemlélteti. Itt a második változat egy olyan programelágazás, amely az alprog-



6.3. ábra. Az alprogram hívási módjai HELP-ben

ramhívásnak egy, a kiszámított GOTO-utasításhoz hasonló lehetőségét képviseli (l. a 4.7.1.2. pontot). Az alprogram szelekciós hívásánál is érvényesek a 4.7.1. pontban a szelekciós ugróutasítás elugrási pontjaira tett megjegyzések.
A nyelvleírásból sem az nem tűnik ki, hogy milyen mélységben egymásba ágyazott hívási struktúra engedhető meg, sem az, hogy van-e lehetőség rekurzióra. Ezek feltehetően implementációtól függő kérdések. Nyitott kérdés az is, hogy hogyan viselkedik a főprogram, ha ráfut egy RETURN-utasításra. Ez olyankor következhet be, ha pl. az alprogramban ugróutasítással programelágazás jön létre.

HELP nyelven írt programszövegben az alprogramok első pillantásra alig különíthetők el, különösen, ha a megszokott ugróutasítások célutasításai vegyesen fordulnak elő az alprogramok címkéivel. Ezen a bajon természetesen lehet segíteni, ha az alprogramokat részletes megjegyzésekkel (kommentekkel) egészítjük ki. Szerencsésebb lenne azonban, ha a megkülönböztetést a nyelv is elősegítené, pl. úgy, hogy az alprogram a csupasz címke helyett mondjuk a következő deklarációval kezdődne:

SUBROUTINE⟨címke⟩;

A probléma lényegét a következőkben bemutatott *6.1.*, *6.2.* és *6.3. programrészletek* segítségével szeretnénk érzékeltetni.

Kérdés, hány alprogramot deklaráltunk a *6.1. programrészletben,* kettőt vagy hármat? Az L30-as címkétől az utolsó RETURN-ig terjedő rész tulajdonképpen önálló alprogramnak tekinthető. Ha viszont csak két alprogram írása állt szándékunkban, akkor azt inkább a 6.2. programrészlet szerint kellene megírni. Ha mégis három alprogramot szeretnénk elkülöníteni, akkor a 6.3. programrészlet szerinti szerkezet lenne a helyesebb.

Ezzel az is biztosítható, hogy egy alprogramból csak egyetlen ponton lehessen kilépni. Ezt egyébként sem a HELP, sem a VAL nem írják elő kötelező érvénnyel. További probléma adódik abból is, hogy a HELP koncepciója szerint egy szubrutinba több különböző helyen is be lehet lépni. Ezt példázza a *6.4. programrészlet.*

Nézzük, hány belépési pontja van ennek az alprogramnak! Ez a kérdés csak úgy dönthető el, hogy a program fennmaradó részét elemezzük, és megvizsgáljuk, hogy lehet-e GOSUB-utasítással hivatkozni az L11 és L12 címkékre. Célszerűbb lenne azonban, ha a szubrutin programszövegéből ez már eleve kitűnne.

Összefoglalva elmondhatjuk, hogy a HELP alprogramszerkesztési és hívási rendszere nem elégíti ki minden szempontból a strukturált programozással szemben támasztott követelményeket, és ezáltal nehezen áttekinthető. Ezzel egy olyan programszerkesztési stílust honosít meg, amelyben a tévesztéseknek nagyobb a veszélye. A VAL nyelv koncepciója sem strukturált. Megkülönbözteti azonban az ugróutasítás célpontját a szubrutin nevétől, és nem engedi meg a több belépési pontú alprogramok használatát sem, miáltal stílusa is áttekinthetőbb, mint a HELP-nek.

```
L10:
        :
     RETURN;
L20:
        :
     IF a<b THEN GOTO L30 END;
     RETURN;
L30:
        :
     RETURN;
```

6.1. programrészlet. HELP nyelven írt alprogramok

```
L10:
        :
     RETURN;
L20:
        :
     IF a<b THEN RETURN END;
        :
     RETURN;
```

6.2. programrészlet. HELP nyelven írt két alprogram

```
L10:
        :
     RETURN;
L20:
        :
     IF a<b THEN GOSUB L30 END;
     RETURN;
L30:
        :
     RETURN;
```

6.3. programrészlet. HELP nyelven írt három alprogram

```
L10:
        :
     IF a<b THEN GOTO L11 END;
L12:
        :
L11:
        :
     IF b>5 THEN GOTO L12 END;
        :
     RETURN;
```

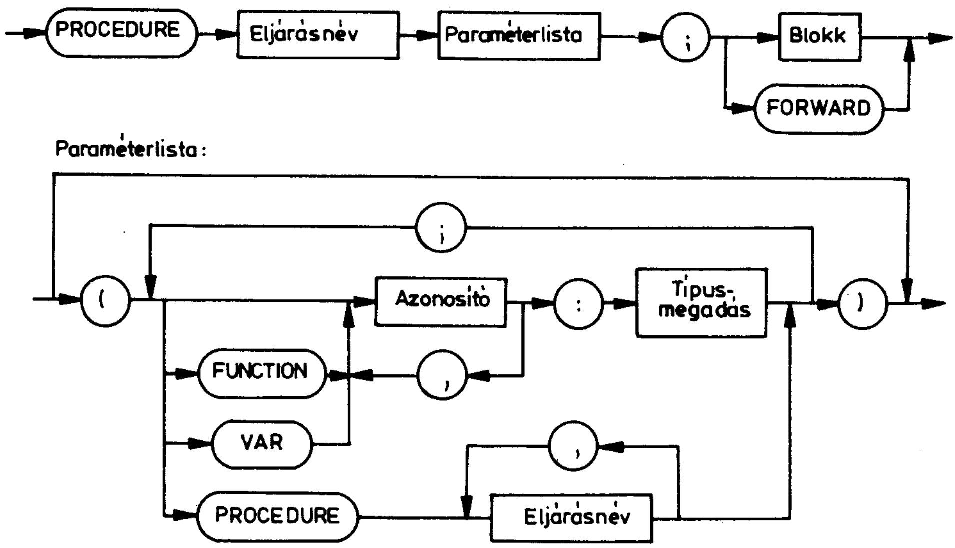
6.4. programrészlet. HELP-ben írt alprogram több belépési ponttal

6.2. Eljárások

A következőkben bemutatjuk az eljárások deklarálásának és hívásának formáit a PASCAL, az AL és a SIGLA nyelvek esetében.

PASCAL:

Az eljárás deklarációját PASCAL-ban a *6.4. ábra* szemlélteti. Az ábrán szereplő **(blokk)** a PASCAL-programhoz hasonlóan szintén tartalmaz deklarációs részt, amelyben az eljárásban lokális érvényű címkéket, változótípusokat, változókat, eljárásokat és függvényeljárásokat definiálhatunk, valamint ez tartalmazza a BEGIN–END-del lehatárolt utasításrészt, más néven az eljárástörzset (l. a 4.6. szakaszt). A VAR kulcsszóval megjelölt eljárásparamétereknél hivatkozás szerinti paraméterátadás történik, egyébként

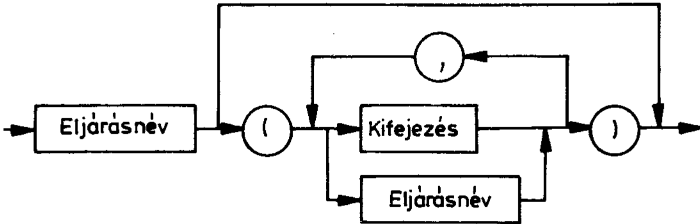


6.4. ábra. Eljárásdeklaráció PASCAL-ban

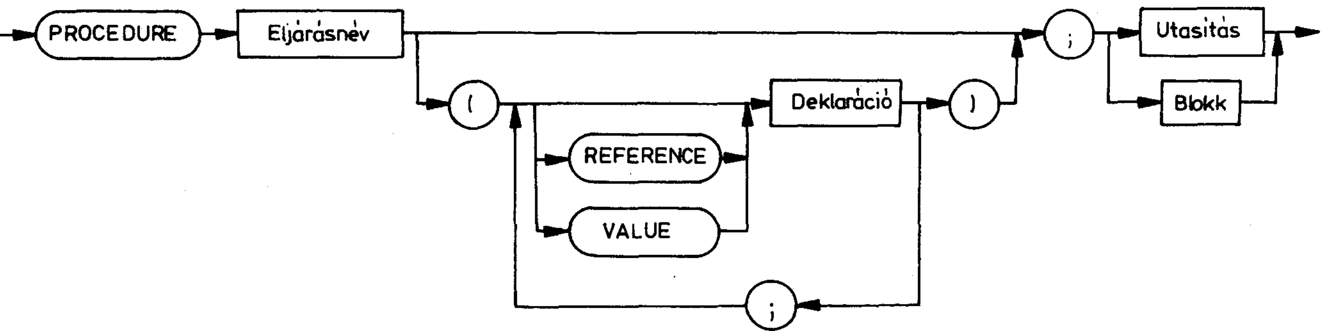
pedig érték szerinti. A hívott alprogramnak a szokásos paramétereken kívül eljárások, ill. függvényeljárások is paraméterként adhatók át.

Minthogy PASCAL-ban az eljárásokat használatuk előtt először deklarálnunk kell, ezért bevezették a FORWARD-deklarációt, amely az előrehivatkozást biztosítja. Ennek segítségével kereszthivatkozások is szerkeszthetők. Amikor pl. a P eljárás hívja Q-t és a Q hívja P-t, akkor a deklarációban az eljárás törzsét a FORWARD alapszó helyettesíti, és csak később következhet az eljárás törzsének deklarálása, de ekkor már a paraméterlista megismétlése nélkül.

Az eljárás ott fejeződik be, ahol az eljárástörzsnek vége van. Így tehát a PASCAL nyelv eljáráshívási technikája megfelel a bevezetőben említett követelményeknek. Az eljáráshívást a *6.5. ábra* szemlélteti. Az eljárás aktivizálásakor a rendszer először ellenőrzi, hogy az adatok típusa és az aktuális paraméterek száma megegyezik-e a deklarációban rögzített típusokkal, ill. paraméterekkel, valamint, hogy a hivatkozás szerinti értékátadással deklarált paraméterek esetében tényleg változó áll-e az illető paraméter helyén, nem pedig esetleg konstans számadat.



6.5. ábra. Eljáráshívás PASCAL-ban



6.6. ábra. Eljárásdeklaráció AL-ban

AL:

Az AL nyelvben az eljárásokat hasonlóképp deklaráljuk, mint PASCAL-ban *(6.6. ábra)*. Ezen az ábrán ⟨**deklaráció**⟩-n azt a fajta deklarációt kell érteni, ahogyan AL-ban a változókat, ill. tömböket általában deklaráljuk, az eljárás törzse pedig egy blokk, vagy akár egyetlen utasítás is lehet. A PASCAL-lal ellentétben a paraméterátadás alapvető típusa AL-ban a hivatkozás szerinti paraméterátadás. Ezt az alapértelmezést a VALUE kulcsszóval meg lehet változtatni, a REFERENCE kulcsszóval pedig külön is lehet hangsúlyozni! Az eljárás hívása megegyezik a PASCAL-nál megismert hívási móddal azzal a különbséggel, hogy eljárást nem lehet paraméterként átadni. FORWARD típusú deklarációra AL-ban nincs szükség. Az eljárás AL-ban is az eljárástörzs utolsó utasításával fejeződik be, így a bevezetőben felsorolt követelmények az AL nyelvnél is teljesülnek.

SIGLA:

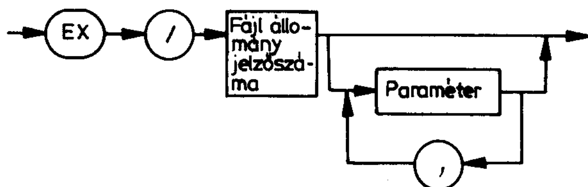
Az AL és PASCAL nyelvekkel ellentétben a SIGLA-ban nem lehet eljárást deklarálni. Egyik programból azonban fel lehet hívni egy másikat és annak paramétereit is lehet átadni. A hívott program – ugyanúgy, mint a VAL esetében – önmagában is futtatható, ebben különbözik az AL-nál és a PASCAL-nál használt eljárásoktól. Paraméterátadásokra 16 memóriarekesz áll rendelkezésre. Ezeket P betűvel és az utána írt számmal jelöljük, ill. különböztetjük meg. Az egyes programokban, ill. eljárásokban csakis ezek a paraméterek lokális érvényűek. Az M1-től M1023-ig jelölhető memóriaszámlálók globális érvényűek. Ha a programot eljárásként hívjuk, akkor a paraméterátadás az egyes változók tartalmának – tehát pillanatnyi értékének – átmásolásával megy végbe, azaz érték szerinti hívás történik. Az indirekt címzés ügyes kihasználásával azonban hivatkozás szerinti paraméterátadás is megvalósítható (l. a *6.5. programrészletet*). A SIGLA nyelv tehát az eljárások kezelése tekintetében szinte assemblerszerű figyelemre méltó lehetőségeket nyújt.

Az eljárást a *6.7. ábrán* bemutatott módon lehet hívni. A sémán a ⟨fájl jelzőszáma⟩ a hívott alprogram sorszámát jelenti. Mind a ⟨fájl jelzőszámá⟩-ra, mind a ⟨paraméter⟩-ekre meg van engedve konstansok, paraméterek, direkt vagy indirekt címzésű számlálók használata.

Végezetül bemutatunk egy példát a hivatkozás szerinti paraméterátadásra SIGLA-ban.

```
Program 2:
                    SE/M1, P1
                    IC/M1, P2
                    SE/I3, M1
Főprogram:          SE/M10, 5
                    SE/M20, 17
                    EX/2, M10, M20, 100
```

6.5. programrészlet. Hivatkozás szerinti paraméterátadás SIGLA-ban



6.7. ábra. Eljáráshívás a SIGLA nyelvben

A főprogram a következő paraméterekkel hívja a 2. programot

P1 = **M10** tartalma, vagyis **5**,

P2 = **M20** tartalma, vagyis **17**,

P3 = **100**

A 2. programban **P1**-et és **P2**-t **M1**-be összegezzük, majd az eredményt az **I3**-mal jelölt számlálóban helyezzük el. Ennek tartalma a híváskor **P3**-ban található szám. Végül az **M100**-as számlálóba kerül az eredmény. Ennek értéke **23**. Az **M100** az alprogramnak hivatkozási változóként adódik át.

6.3. Függvényeljárások

Függvényeljárás csak PASCAL-ban és AL-ban lehetséges. A függvényeljárások eredménye egyetlen adat (számérték vagy logikai érték stb.), ezért a függvényeljárás közvetlenül képletben is szerepeltethető. Az eljárásokkal összehasonlítva ez a leglényegesebb különbség, hiszen az eljárásoknál is megvan a hivatkozás szerinti paraméterátadás, ill. a mellékhatások révén való paraméterátadás lehetősége (de nemcsak egyetlen kimeneti paraméter adható át, mint a függvényeljárásnál).

Ahhoz, hogy egy eljárást függvényeljárássá alakítsunk át, mind PASCAL-ban, mind pedig az AL nyelvben két fontos változtatást kell végrehajtanunk: egyrészt a deklaráció során az eredmény típusát is deklarálnunk kell, másrészt az eredmény értékét egy értékadó utasításban át kell adnunk magának a függvényeljárás azonosítójának, a függvényeljárás nevének tehát szerepelnie kell egy értékadó utasítás bal oldalán.

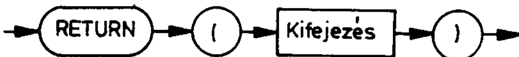
Ezek a módosítások PASCAL-ban a következőképp végezhetők el: PROCEDURE helyett a deklarációban FUNCTION-t kell írnunk, és a paraméterlistát követően deklarálnunk kell az eredmény típusát (l. *6.8. ábra*). AL-ban az eredmény típusát a PROCEDURE alapszó előtt kell kiírni.

PASCAL-ban értékadó utasítással át kell adni az eredményt az eljárásnévnek, míg AL-ban az eredmény a *6.9. ábra* szerinti RETURN-utasítás hatására adódik át.

AL-ban az eredmény bármilyen típusú lehet, kivéve az EVENT (esemény) típust. PASCAL-ban az eredmény INTEGER, CHAR, BOOLEAN, REAL vagy pointer típusú lehet, de megengedett a felhasználó által definiált bármilyen lineárisan rendezett adattípus deklarálása is.

Ezekután a 2.2.4.2. pontban hipotetikus nyelven vektorok elemeinek összeadására írt függvényeljárásunkat most már AL-ban vagy PASCAL-ban is meg tudjuk írni. PASCAL-nál a *3.9. programrészletben* látott **vektordefiníciót** használtuk (l. a *6.6.*, ill. a *6.7. programrészleteket*). Mint látható, AL-ban erre a feladatra nem feltétlenül szükséges

6.8. ábra. A függvényeljárás típusának deklarálása PASCAL-ban



6.9. ábra. RETURN utasítás AL-ban

```
SCALAR PROCEDURE vektorszumma (VALUE VECTOR v);
  RETURN (v. VECTOR (1, 1, 1));
```

6.6. programrészlet. A vektorszumma függvényeljárás AL nyelvben

```
FUNCTION vektorszumma (v: vektortyp); REAL;
BEGIN (* vektorszumma *)
  vektorszumma := v[1]+v[2]+v[3];
END (* vektorszumma *);
```

6.7. programrészlet. A vektorszumma függvényeljárás PASCAL nyelvben

külön eljárást készíteni, hiszen ebben a nyelvben a vektorok skalárszorzatának kiszámítására alkalmas művelet is van.

A következő példában láthatjuk, hogy hogyan szerepeltethető a függvényeljárás közvetlenül kifejezésben, ill. képletben.

Legyen

kezdőváltozó ←5 + vektorszumma (célvektor);

Itt a **kezdőváltozó** skaláris mennyiség (PASCAL-ban, REAL és az értékadás jele is másképp írandó).

PASCAL-ban tilos a függvényeljárást külön utasításban meghívni, mint az eljárásokat. Nem így az AL-ban. AL-ban írhatjuk, hogy

vektorszumma (xyzvektor);

Ilyenkor az utasítás végrehajtása után az eredmény természetesen elvész.

Ezen utasítás használatának ezért ilyen formában nincs értelme. Vannak azonban olyan esetek, amikor szükségünk lehet a függvényeljárás végrehajtására anélkül, hogy annak eredménye érdekelne bennünket. Ilyen eset pl. a verem szervezésére alkalmas standard függvényeljárás is (l. még a 2.2.2. pontot). A *6.8.* és a *6.9. programrészletben* egy olyan megoldást látunk, amikor a program a **darabsz** nevű változóban tárolt számú adatot tölt be egy verembe, majd az **index** változó tartalmának megfelelő számú adatot töröl (kihagy a feldolgozásból), és végül eredményként kiadja a verem legfelül maradt elemének tartalmát.

Figyeljük meg, mennyire hasonlít egymáshoz a két program! Vegyük észre, hogy a PASCAL-ban az adatok szervezésére előnyösebb strukturált szervezési lehetőség kínálkozik.

AL-ban az **üres** és **veremkorlát** változókat nem lehetett konstansként deklarálni, csak változónak. Ez implicit változódeklarációval történt. AL-ban azok a változók sem foglalhatók össze semmilyen strukturált adattípusba se, amelyeket a program a verembe tölt be.

PASCAL-ban a WITH-utasítás használatával a rekordok elemei mindkét függvényeljárásban közvetlenül hozzáférhetővé válnak, és így nincs szükség pl. a **k. verem** használatára. A PASCAL-változatban alkalmazott input/output utasítások abban különböznek a 3.3.2.1. pontban megismert utasítástól, hogy az itt látott utasítások egy sorban azonnal ki is adják a megadott paraméterek tartalmát.

A FOR-ciklus annyiszor üríti a vermet, amennyi az **index** tartalma. A FOR-ciklusban jól megfigyelhető, hogy a két nyelvnél a függvényeljárás hívásának módja miben tér el egymástól. PASCAL-ban a függvényeljárás eredményét valamilyen módon fogadni kell, ezért adtuk át az eredményt a **dummy** változónak. AL-ban erre nincs szükség. Az AL változatban egyébként lehetett volna dinamikus tömböket is használni (l. a 3.1.5.1. pontot).

```
PROGRAM stack (input, output);
  CONST
    üres              = - maxint;
    veremkorlát       = 10;
TYPE
  veremtípus =
    RECORD
      verempointer:    integer;
      verem            : ARRAY [1.. veremkorlát] OF real
    END;
VAR
  verem              : veremtípus;
  darabsz            ,             (* az input adatok száma                    *)
  index              ,             (* a kimaradó adatok száma                  *)
  i                  : integer;    (* segédváltozó                             *)
  adat               ,             (* egyetlen számérték                       *)
  dummy              : real;       (* segédváltozó pop művelethez              *)
FUNCTION pop (VAR k: veremtípus): real;
( * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * )
(*                                                                                             *)
(*        Ez a függvényeljárás a verem tartalmának legfelső                                    *)
(*        elemét adja eredményül, miközben kitörli az ehhez az                                 *)
(*        elemhez tartozó bejegyzés tartalmát. Ha a verem üres,                                *)
(*        akkor a függvényeljárás eredménye is üres.                                           *)
(*                                                                                             *)
( * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * )
BEGIN (* pop *)
  WITH k DO
    IF verempointer = 0 THEN
      pop := üres
    ELSE
      BEGIN
      pop := verem[verempointer];
      verempointer := verempointer -1;
      END;
END (* pop vége *);
FUNCTION push (VAR k: veremtípus;
                 adat: real): boolean;
( * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * )
(*                                                                                             *)
(*        Ha a verem még nem telt meg, akkor a függvény-                                       *)
(*        eljárás az adatot letárolja és a true logikai értéket                                *)
(*        adja eredményül. Ha megtelt, akkor az eredmény a                                     *)
(*        false logikai érték.                                                                 *)
(*                                                                                             *)
( * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * )
BEGIN (* push *)
  WITH k DO
    IF verempointer = veremkorlát THEN
      push := false
    ELSE
    BEGIN
    verempointer := verempointer + 1;
    verem[verempointer]: = adat;
    push := true;
    END;
END (* push vége *);BEGIN (* veremszervező program; PASCAL-változat *)
  verem.verempointer := 0;
  (* A verem inicializálása *)
  writeln ('Adja meg az input adatok számát.,veremkorlát,'-nál nem lehet nagyobb');
  readln (darabsz);
  FOR i := 1 TO darabsz DO
    BEGIN
    writeln (i, '. adat: ');
    readln (adat);
```

```
  IF NOT push (verem, adat) THEN
    writeln (i – verem. verempointer, '. Veremtúlcsordulás');
  END;
writeln ('Hány adatot kell kihagyni a beolvasáskor?');
readln (index);
FOR i := 1 TO index DO
  dummy := pop (verem);
adat := pop(verem);
IF adat = üres THEN
  writeln ('A verem üres!')
ELSE
  writeln ('adat = ', adat);
END * A veremszervező program vége *).
```

6.8. programrészlet. Verem szervezése PASCAL-ban

```
BEGIN „Veremszervező program; AL-változat”
  üres           ← –32767;
  veremkorlát    ← 10;
  SCALAR ARRAY verem[1 : veremkorlát];
  SCALAR adat    ,                          (* Egyetlen adat              *)
         i       ;                          (* Segédváltozó               *)
SCALAR PROCEDURE pop (REFERENCE SCALAR ARRAY k[1: veremkorlát];
                      REFERENCE SCALAR kpointer);
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
(*                                                                                       *)
(*        Ez a függvényeljárás a verem tartalmának legfelső                              *)
(*        elemét adja eredményül, miközben kitörli az ehhez az                           *)
(*        elemhez tartozó bejegyzés tartalmát. Ha a verem üres,                          *)
(*        akkor a függvényeljárás eredménye is üres.                                     *)
(*                                                                                       *)
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
IF kpointer = 0 THEN
  RETURN (üres)
ELSE
  BEGIN
  RETURN (k[kpointer]);
  kpointer ← kpointer –1;
  END;
SCALAR PROCEDURE push (REFERENCE SCALAR ARRAY k[1: veremkorlát];
                       REFERENCE SCALAR kpointer;
                          VALUE SCALAR adat);
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
(*                                                                                       *)
(*        Ha a verem még nem telt meg, akkor a függvényeljárás                           *)
(*        az adatot letárolja, és a true logikai értéket adja                            *)
(*        eredményül. Ha megtelt, akkor az eredmény a false                              *)
(*        logikai érték.                                                                 *)
(*                                                                                       *)
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
IF kpointer = veremkorlát THEN
  RETURN (false)
ELSE
  BEGIN
  kpointer ← kpointer + 1;
  k[kpointer] ← adat;
  RETURN (true);
  END;
(* A veremszervező program főprogramja; AL-változat *)
verempointer ← 0; (* a verem inicializálása *)
PRINT („Adja meg az input adatok számát.”, veremkorlát, „-nál nem lehet nagyobb.”);
```

```
FOR i ← 1 STEP 1 UNTIL INSCALAR DO
  BEGIN
  PRINT (i, „. adat:");
  IF NOT push(verem, verempointer, INSCALAR) THEN
    PRINT (i - verempointer, |. Veremtúlcsordulás");
  END;
PRINT („Hány adatot kell kihagyni a beolvasáskor?");
FOR i ← 1 STEP 1 UNTIL INSCALAR DO
  pop (verem, verempointer);
  adat ← pop (verem, verempointer);
  IF adat = üres THEN
    PRINT („verem üres!")
ELSE
  PRINT („adat = ", adat);
END „Veremszervező program AL-változatának vége"
```

6.9. programrészlet. Verem szervezése AL-ban

6.4. Rekurzív eljárások és függvényeljárások

A *rekurzív eljáráshívás* lehetőségét a 2.2.5. pontban már részletesen tárgyaltuk egy robotvezérlési példa kapcsán. A tárgyalt robotvezérlő nyelvek közül csak az AL-ban szerkeszthető *rekurzív eljárás,* ill. *függvényeljárás,* de ezt a programban nem kell külön megjelölni.

Annak illusztrálására, hogy a rekurzív eljáráshívás robotvezérlő programok írásakor is előnyösen alkalmazható, bemutatjuk a környezeti modell affix hozzárendeléseit kezelő AL példaprogramot (vö. a 4.1. szakasszal). Ez a példa megmutatja, hogy rekurzív eljáráshívások segítségével milyen hatékonyan és áttekinthetően kézben tartható a környezetleíró modell hierarchikus adatstruktúrája.

A *6.10. programrészletben* az összes szereplő **frame-rendszer** letárolása céljából felvettünk egy **frame-tömb**-öt, éspedig a **framenr** tartalmának, vagyis a frame-rendszerek számának megfelelő méretűt. Ugyanakkor felvettük az „affixtömb" nevű integer-, ill. skalár tömböt is az affix relációk ábrázolására. Ez a **framenr** számú egységből áll. Egy egység a következő három komponenst tartalmazza:

1. egy értéket, ami a **frame-tömb** frame-indexe,
2. egy pointert, amely az első hozzárendelt frame-re mutat,
3. egy másik pointert, amely a következő olyan frame-re mutat, amely szintén ugyanahhoz a frame-hez lett hozzárendelve, mint a szóban forgó egység által reprezentált frame.

A *6.11. ábra* mutatja az egységek pointerezett listáját. Ez a *6.10. ábra* szerint az affix hozzárendelések fa-struktúrájának felel meg. Mindenesetre figyelembe kell venni, hogy az affix hozzárendelési reláció itt fordított irányú, tehát a hivatkozás a fölérendelt frame-től a hozzárendelt frame felé mutat.

A bemutatott rekurzív eljárás neve **kereső.** Az eljárás paramétere az **affixtömb**-lista egyes elemeire mutató listapointer. Feladata az, hogy ebben az **„affixtömb"**-ben megtalálja a keresett **keresframe** azonosítójú frame-et. Ha megtalálta, akkor a **keresframe** pointerét az ún. **keresframepointer**-t erre a frame-re állítja be. Annak érdekében, hogy minden részletében kézben tartsuk a pointerezéseket, magán a **kereső** eljáráson belül szükség van a **kereső** eljárás rekurzív hívására. Ellenkező esetben a visszaállítási pontok

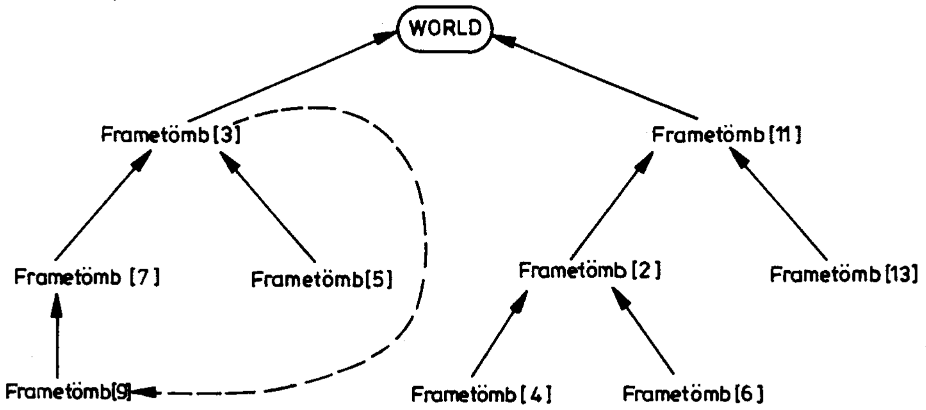
```
BEGIN „Keretprogram a * Példa * nevű programhoz"
  SCALAR framenr;
  :
  BEGIN „Példa"
    FRAME       ARRAY frametömb[1:framenr];
    SCALAR      ARRAY affixtömb[1:framenr, 1 : 3];
    SCALAR      keresframe, affixframepointer, célframepointer, keresframepointer, kapcso-
                                                   lat, hibajelzés;
    ( * * * * * * * * * * * * * * * * * * *   kereső   * * * * * * * * * * * * * * * * * * * )
    PROCEDURE kereső (VALUE SCALAR pointer);
    BEGIN „kereső"
      IF affixtömb[pointer, 1] = keresframe THEN              (* Megvan-e a frame? *)
      BEGIN „megtalálta"
        IF keresframepointer> 0 THEN          (* Kétszer van meg a frame              *)
                                              (* a fa-struktúrában?                   *)
          hibajelzés ← 1;                     (* Hibajelzés beállítása,               *)
        keresframepointer ← pointer;          (* a pointert a megtalált               *)
                                              (* frame-re állítja                     *)
      END „megtalálta";
      IF affixtömb[pointer, 2]> 0 THEN        (* Vége van-e a lefelé                  *)
                                              (* terjedő pointerezésnek?              *)
        kereső(affixtömb[pointer,2]);         (* Keresés lefelé                       *)
      IF affixtömb[pointer, 3]> THEN          (* Oldalirányú pointerezés              *)
                                              (* véget ért?                           *)
        kereső(affixtömb[pointer,3]);         (* Keresek oldalirányban                *)
END „kereső"
( * * * * * * * * * * * * * * * * * * * hozzárendelés * * * * * * * * * * * * * * * * * * * )
PROCEDURE affixrendel (VALUE SCALAR affixframe, célframe);
BEGIN „affixrendel"
  SCALAR kerespointer;
  kerespointer ← kapcsolat;                   (* a kapcsolat a lista elejére          *)
                                              (* mutat                                *)
  affixframepointer ← 0;
  DO                                 (* Egy keresési ciklus, amelyben megálla-        *)
                                     (* pítjuk, hogy az affixframe a WORLD-höz        *)
                                     (* van-e kapcsolva                               *)
   IF affixtömb[kerespointer,1] = affixframe THEN
     affixframepointer ← kerespointer;             (* Megtalálta a frame-et           *)
   IF affixtömb[kerespointer, 3] <> 0 THEN         (* Van-e még                       *)
                                                   (* következő frame?                *)
     kerespointer ← affixtömb[kerespointer,3]
                                     (* Értékadás, a következő olyan frame,           *)
                                     (* amely WORLD-höz kapcsolódik                   *)
   ELSE
     kerespointer ← 0;               (* keresést leállítja                            *)
   UNTIL kerespointer = 0 OR affixframepointer <>;
   keresframe ← célframe;
   keresframepointer ← 0;
   kereső(kapcsolat);                         (* Megkeresi azt a frame-et,            *)
                                              (* amelyhez az affix hozzárende-        *)
                                              (* lést végre kell hajtani              *)
   célframepointer ← keresframepointer        (* Fölérendelt frame                    *)
                                              (* megjelölése                          *)
   IF affixframepointer = 0 THEN
   BEGIN „hozzárendelendő frame nem kapcsolódik a WORLD-höz"
     keresframe ← affixframe;
     keresframepointer ← 0;
     keres(kapcsolat);                        (* Leellenőrzi, hogy a hozzáren-        *)
                                              (* delendő frame már affix hoz-         *)
                                              (* zárendelési kapcsolatban van-e       *)
     IF keresframepointer > 0 THEN
       hibajelzés ← 2                         (* Hiba, ha a frame már affix           *)
                                              (* kapcsolatban állt                    *)
     ELSE
       affixkapcsolás;                        (* Ez egy közelebbről nem defini-       *)
                                              (* ált eljárás, amely a frame-et        *)
                                              (* felveszi a listára                   *)
```

```
      END „hozzárendelendő frame nem kapcsolódik a WORLD-höz vége"
      ELSE
      BEGIN „hozzárendelő frame kapcsolódik a WORLD-höz"
        keresframe ← célframe;
        keresframepointer ← 0;
        kereső(affixframepointer);         (* Leellenőrzi, hogy a                *)
                                           (* fölérendelt frame alárendelt kapcsolat *)
                                           (* ban áll-e a hozzákapcsolandóval, vagyis *)
                                           (* hogy körkapcsolat keletkezik-e      *)
        IF keresframepointer > 0 THEN
          hibajelzés ← 3                   (* Hiba! körkapcsolat van             *)
        ELSE
          affixkappcsolás;                 (* A korábban már használt eljárás,   *)
                                           (* amelyik a frame-et felveszi a listára *)
      END „hozzárendelendő frame kapcsolódik a WORLD-höz vége"
  END „affixrendel vége"
* * * * * * * * * * * * * * * * * * * * * * Példa * * * * * * * * * * * * * * * * * * * * * *
   kapcsolat ← 1                           (* Listakezdet beállítása *)
   FOR i := 1 STEP 1 UNTIL framenr DO
   BEGIN „affixtömb inicializálása"
     affixtömb[i,1] ← 0;
     affixtömb[i,2] ← 0;
     affixtömb[i,3] ← 0;
   END „affixtömb inicializálás vége";
       :
   affixrendel(14,6);                      (* a 14 frame-et a 6-oshoz kapcsolja   *)
   IF hibajelzés > 0 THEN
   BEGIN „hibakezelő eljárás"
       :
   END „hibakezelő eljárás vége";
       :
   END „Példa vége";
END „Keretprogram vége"
```

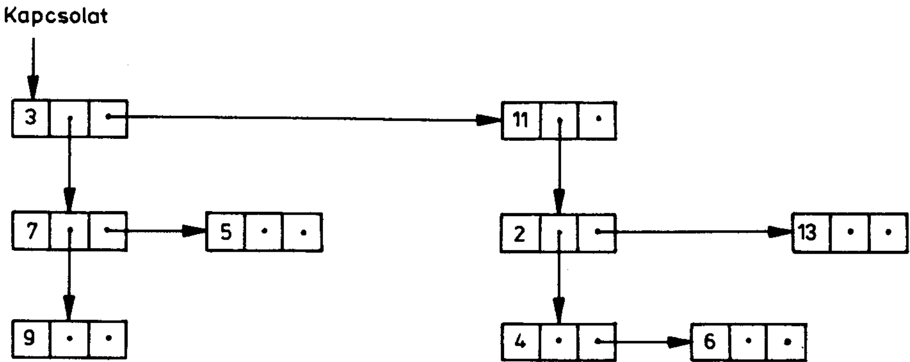
6.10. programrészlet. A „kereső" eljárás rekurzív hívása az affix-hozzárendeléseket kezelő AL nyelvű programban

tárolásának kézben tartásához nagyon bonyolult algoritmusra lenne szükség, hogy amikor egy ágat vízszintes irányban már feldolgoztunk, lefelé is megtalálhassuk a következő folytatási pontot. A **kereső** eljárás legelőször lekérdezi, hogy a keresett frame azonos-e azzal a listaelemmel, amelyre a pointer mutat. Ha azonos, akkor ellenőrzi, hogy ugyanez az elem előfordult-e már a listában, és hogy ennek megfelelően a **hibajelzés** tartalma be van állítva 1-esre. Ezután a **kereső** eljárás további rekurzív hívásai következnek a lefelé irányuló pointerezésnek megfelelően. Amikor elérjük a pointerláncolat legalsó végét, akkor a legutolsó elemnél a keresés a hivatkozási lánc oldalirányú feldolgozásával folytatódik. Ha itt ismét előfordul lefelé mutató pointer, akkor először ezt dolgozzuk fel, és így tovább. Amikor pl. az **affixrendel** eljáráson belül a lista kezdetére mutató **kapcsolat** pointerrel és a 6-os **keresframe**-mel aktivizáljuk a kereső eljárást, akkor a *6.11. ábrának* megfelelően a *6.12. ábra* szerinti feldolgozási sorrend adódik. A zárójelek a rekurzív egymásba ágyazódásra utalnak.

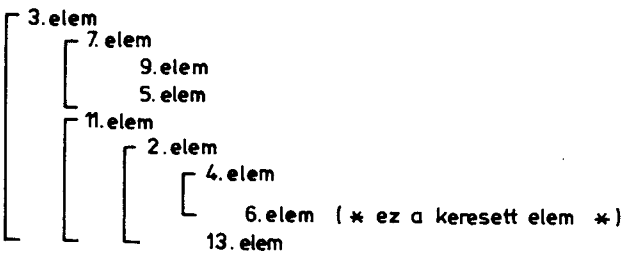
A **kereső** eljárásra az **affixrendel** eljárásban van szükség, amely az affix hozzárendelést bejegyzi az **affixtömb** nevű listába. Az **affixrendel** nevű eljárás kezdetén a program ellenőrzi, hogy a hozzárendelő frame-hez mint fölérendelt frame-hez nincsenek-e már esetleg további frame-rendszerek hozzákapcsolva. Ilyen tulajdonságú frame csak a környezetleíró modell WORLD-jelölésében (l. *6.10. ábra*), ill. a **kapcsolat**-tal kezdődő



6.10. ábra. Az affix hozzárendelés fa-struktúrája



6.11. ábra. Az affix hozzárendelési struktúra ábrázolása pointerezett lista segítségével



6.12. ábra. A „kereső" eljárás rekurzív hívásai

vízszintes irányú hivatkozási láncban szerepelhet. Ha létezik ilyen frame, akkor azt a hozzárendelési fa-struktúra egész hosszú csatlakozó alsó részével együtt hozzárendeljük a megadott fölérendelt frame-hez. Ennek kapcsán hiba léphet fel, ha az új fölérendelt frame annak a frame-nek a hozzárendelési fáján is szerepel, amelyet éppen affix-relációba akarunk hozni a fölérendelt frame-mel. Ilyenkor körkapcsolat jönne létre, és később,

amikor a körkapcsolat valamely tagjának helyváltoztatásakor módosítani kell az affix-kapcsolatba hozott frame-et, az interpreter végtelen ciklusba esne. A *6.10. ábrán* a szaggatott vonal azt jelzi, hogy a **frametömb** [3] a **frametömb** [9]-cel affix hozzárendelési relációban van, tehát körkapcsolat áll fenn. A *6.9. programrészletben,* miután a program megkeresi a listában a fölérendelt **célframe**-et, megvizsgálja azt is, hogy a hozzárendelendő frame a hozzárendelési fa egész alsó részével szerepel-e valahol a listában. Ha igen, vagyis ha az **affixframepointer** a WHILE-ciklusban be lett állítva, akkor a **célframe**-et teszi meg **keresframe**-nek, majd az eljárás megvizsgálja a hozzárendelési fa alsó részét, hogy a fölérendelt **célframe** szerepel-e abban. Ezt a keresést a

kereső(affixframepointer);

eljárás hajtja végre. Ha a **célframe** nevű frame nincs bejegyezve a listába az említett hozzárendelési fával, akkor nem jöhet létre körkapcsolat. Még azt is ellenőrzi, hogy a hozzárendelő frame szerepel-e egyáltalán az **affixtömb** nevű listában. Ha igen, akkor a **hibajelzés** tartalmát 2-re állítja, hiszen a frame-et nem lehet kétszer hozzárendelni valamihez. A frame-nek az affix-relációk rendszerébe való tényleges beépítését az **affixkapcsolás** nevű eljárás végzi, amelyet az áttekinthetőség kedvéért már nem részleteztünk. Az affix hozzárendelési reláció létrehozását a főprogramban az

affixrendel(14,6);

eljárás végzi. Ha az **affixtömb** nevű listába való bejegyzés közben hiba lép fel, akkor a **hibajelzés** nevű változó lekérdezésével a hiba természete megállapítható.

7. Eljárások időbeli koordinálása

Ipari alkalmazásokban a robotberendezéseknek legtöbbször más gépekkel, ill. további robotokkal kell együttműködniük. Ilyenkor felmerül az egyes feldolgozási folyamatok koordinálásának problémája, éspedig attól függetlenül, hogy a résztvevő gépegységek mindegyike programvezérelt-e, vagy sem. Ha programvezéreltek, akkor programjaikat szinkronizálni kell. Ellenkező esetben, meghatározott készültségi állapotokban (pl. egy-egy alkatrész elkészültekor stb.) a robotnak üzenetet kell küldenie vagy az őt kiszolgáló robot programja felé, vagy egy felsőbb szintű felügyelőprogram felé. Ezzel a probléma a paralel folyamatok programozástechnikai kezelésére egyszerűsíthető le – eltekintve most az arra alkalmas érzékelők és egyéb jelzőberendezések műszaki problémáitól.

A 4.7.3. pontban megismert szinkronizációs eljárások alkalmazásakor alapvetően az a veszély fenyeget, hogy a konkurens folyamatok kölcsönösen blokkolhatják egymást. Ilyesmi többnyire éppen olyankor fordul elő, amikor kivételes helyzetek, ill. hibaüzenetek lekezelésekor a programnak intelligensen kellene reagálnia.

7.1. Párhuzamos blokkok

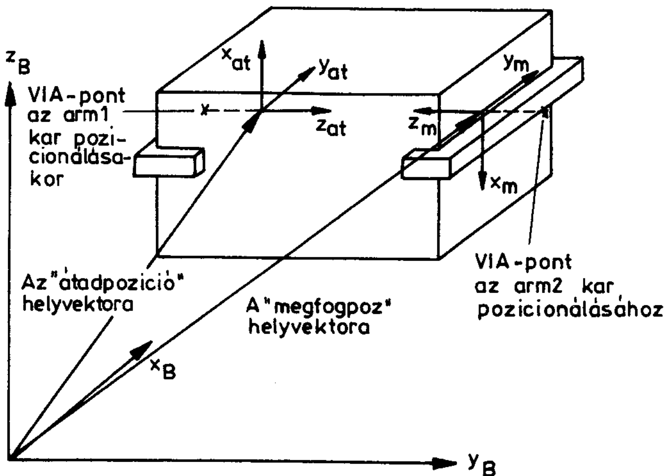
A párhuzamos blokkok használata a párhuzamos műveletsorozatok programbeli leírásának igen kézenfekvő segédeszköze. Az itt tárgyalt nyelvek közül csak az AL nyelvben használatos, így ennek lényegét egy AL nyelvben írt programpéldán mutatjuk be. A paralel blokkokat AL-ban a COBEGIN–COEND alapszavakkal jelöljük (innen ered a *CO-Blokk* elnevezés is). A szokásos blokkokkal ellentétben a program a paralel blokk összes utasítását látszólag szimultán hajtja végre. Azért csak látszólag, mert n számú folyamat tényleges paralel végrehajtásához valójában n számú processzorra lenne szükség. Általában azonban csak egyetlen processzor interpretálja a programot, bár ez esetleg bizonyos részfeladatokat, így pl. a geometriai számításokat alárendelt processzoron is futtathat, de maga a fő processzor több különböző folyamat kisebb egységekre bontott részeit egységenként igen kis időszakokra szabdalva szekvenciálisan, tehát valójában egymás után dolgozza fel. Így tehát állandóan váltogatja a feldolgozás alatt álló feladatokat, és ezért kívülről úgy látszik, mintha valódi paralel feldolgozás folyna a gépen. A feladatok közti átkapcsolások problémáival, valamint a processzoridőnek a feladatok közötti megosztásának kérdéseivel e helyen nem kívánunk foglalkozni, mert ez inkább az operációs rendszerek témakörébe tartozik, és ugyanakkor a kérdést különbözőképpen közelítik meg a nagyszámítógépes és a folyamatirányító számítógépes rendszereknél.

AL-ban a paralel futó blokkok szinkronizálására a 4.7.3. pontban megismert SIGNAL és WAIT szinkronizáló utasítások használhatók. A *7.1. porogramrészlet* a paralel feldolgozást egy példa kapcsán mutatja be, amelyben a robot **arm1** karja átad egy szekrényt az **arm2** karnak. A két paralel feldolgozási folyamatot összesen öt részletre bontottuk (1, 2, A, B, C). Ezek közül ténylegesen csak az első részletek (az 1-es és az A) futnak kváziparalel módon. Az ezt követő programszakaszokat már szemaforok szink-

```
BEGIN
  EVENT átadás, megfogta, elengedte;
  FRAME szekrény, átadpozíció, megfogpozíció, cél;
  COBEGIN
   BEGIN „arm1"
(        MOVE arm1 TO szekrény;             (* arm1 kar                       *)
(        CENTER arm1;                       (* megfogja a szekrényt           *)
(        AFFIX szekrény TO arm1;            (* beállítja az átadási           *)
                                            (* pozícióba                      *)
(        MOVE szekrény TO átadpozíció
(          WITH APPROACH = -6*cm;
(        SIGNAL átadás;                     (* arm1 kész                      *)
         WAIT megfogta;                     (* arm1 várakozik arm2-re         *)
(        OPEN hand1 TO 6*cm                 (* arm1 elengedi a                *)
2        UNFIX szekrény FROM arm1;          (* szekrényt                      *)
(        SIGNAL elengedte;                  (* és ezt jelzi is                *)
      END arm1";
    BEGIN „arm2"
(        OPEN hand2 TO 6*cm;                * A kinyitott hand2               *)
A        MOVE arm2 TO megfogpoz             (* a robotkezet beállítja a       *)
                                            (* megfogási pozícióba            *)
(          WITH APPROACH = -6*cm;
         WAIT átadás;                       (* és várakozik arm1-re           *)
(        CENTER arm2;                       (* arm2 megfogja a                *)
B        AFFIX szekrény TO arm2;            (* szekrényt                      *)
(        SIGNAL megfogta;                   (* és ezt jelzi is                *)
         WAIT elengedte;                    (* arm2 várakozik arm1-re         *)
(        MOVE arm2 TO cél                   (* arm2 viszi a szekrényt         *)
C          WITH DEPARTURE = -6*cm;
         END „arm2";
      COEND;
    END;
```

7.1. programrészlet. Párhuzamos blokkok használata AL-ban

ronizálják, és így feldolgozásuk a következő sorrendben egymás után történik: B, 2, C. Az átadás pillanatában a robotkarok mozgását úgy kell vezérelni, hogy a két kar ne ütközhessen össze. Ezt a *7.1. ábra* szemlélteti. A pozicionáló utasításokat emiatt a 4.2.4. pontban említett megközelítési pont alkalmazásával úgy kell módosítani, hogy a két



7.1. ábra. A megfogószerkezetek pozíciója alkatrész átadásakor

robotkar a két végrehajtó szerv tengelyeit összekötő képzeletbeli egyenes mentén közelítsen egymás felé. A példában a paralel futó ún. CO-blokk két egymással kváziparalel módon végrehajtott összetett utasításból áll, éspedig az **„arm1"** és az **„arm2"** blokkokból. Mint a bevezetőben már említettük, akár kettőnél több utasítás (ill. blokk) szinkronizálása is megvalósítható, ha pl. további robotkarok vagy gépi berendezések együttes vezérlésére van szükség.

A CO-blokk összes kváziparalel futó utasításának végrehajtása után a program a COEND alapszót követő utasítással folytatódik. Minthogy a CO-blokkhoz képest minden adat globális érvényű, ezért a párhuzamos feldolgozási folyamatok számára minden adat hozzáférhető és így meg is változtatható, kivéve a CO-blokkon belül deklarált blokk esetleges lokális érvényű adatait. Ez az utóbbi megoldás erősen elbonyolíthatja a programot és nem csekély veszélyt rejt magában a szinkronizálás biztonságos lebonyolítását illetően.

A példaprogramban nem szerepel semmiféle ellenőrzés arra vonatkozóan, hogy az egyik robotkar elengedte-e a szekrényt és a másik valóban biztonságosan megfogta-e A gyakorlatban ilyen jellegű ellenőrzésre ténylegesen szükség van. Külön fel szeretnénk hívni az Olvasó figyelmét arra, hogy a példaprogram erősen leegyszerűsítve ábrázolja az alkatrész átadás-átvétel programját, ezért a tényleges problémákat nem szabad alábecsülni.

7.2. Task-kezelés

A taskoknak a CO-blokkokkal megvan az a közös tulajdonságuk, hogy lehetővé teszik a paralel vagy legalábbis kváziparalel feldolgozást. A task legfontosabb jellemzői a következők:

- általában lokális tárolóterülettel rendelkezik;
- rendszerparancs, vagy egy másik task segítségével kívülről indítható, leállítható, folytatható, ill. abortálható, a task ugyanakkor képes saját magát is megállítani, abortálni vagy ciklikusan – ill. mint új megvalósulás – újra indulni.
- különböző rendszereknél a task egyidejűleg több felhasználó által is hívható; vagyis az eljárásokhoz hasonlóan a tasknak is különböző reprezentációi futhatnak;
- az operációs rendszertől függően különböző szolgáltatásokat nyújtó formában a *taskok közötti kommunikáció* is megoldható.

A task-kezelést legtöbbször úgy oldják meg, hogy a taskok önálló programokat alkotnak. Így működik a karlsruhei egyetemen kifejlesztett bővített AL nyelvi implementáció is.

A HELP-nél más megoldást alkalmaztak. Itt a task-kezelést egyazon programon belül kezelt alprogramokkal valósították meg, ami sok tekintetben hasonlít az eredeti task-fogalomhoz.

AL:

A PSTART, ill. PEND parancsokkal a programok task-módjára indíthatók, ill. abortálhatók, PHALT és PCONT segítségével pedig megállíthatók, ill. folytathatók. A szóban forgó program nevét a kommand alapszava után ≪ ≫-jelek közé téve adjuk meg, pl.:

```
PSTART ≪PROG1≫;
```

Mivel a taskot programként szerveztük, ezért a task lokális tárolóterülettel dolgozik. Egy tasknak különféle reprezentációi lehetnek, ha az AL interpreter olyan, hogy ezt lehetővé teszi. (Ilyen pl. a karlsruhei változat.)

Az adatforgalommal kapcsolatban a helyzet bonyolultabb, mivel az AL nyelvben erre vonatkozóan nincsenek meg a megfelelő nyelvi elemek. A robotkar és -kéz változói mindenképpen globális érvényűeknek tekintendők. Ahhoz, hogy további előre deklarált változókat (így pl. esemény típusúakat) globális érvényűvé tegyünk, nem kell sem a nyelvet, sem a compiler-programot se bővíteni, se módosítani, ha az interpretert ennek megfelelően módosítjuk. Ezzel a módszerrel a kívánt mennyiségű globális változó létrehozható, ezek terjedelme azonban már sajnos nem változtatható a különféle alkalmazási eseteknek megfelelően.

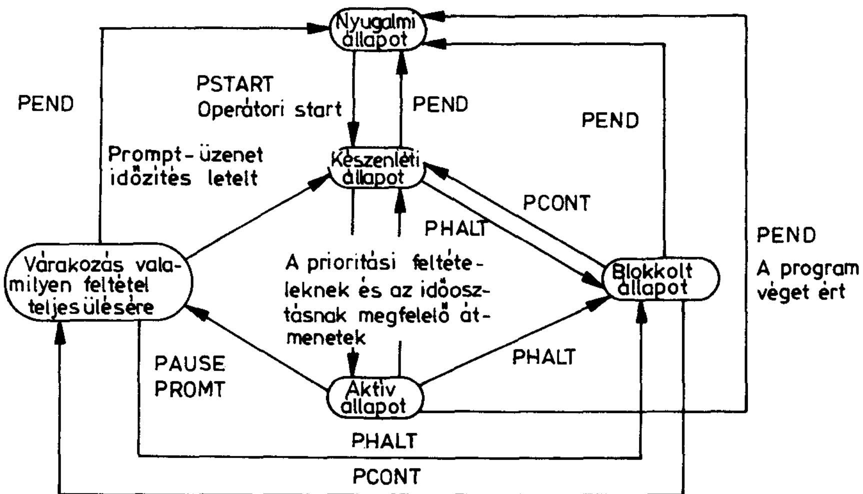
A *7.2. ábra* a task állapotdiagramját mutatja AL nyelv esetében. Az ábrán folytonos vonallal húztuk alá azokat a parancsokat, amelyek mindenképpen külső folyamatból érkeznek. A szaggatott aláhúzás azt jelenti, hogy az illető parancs magából a kérdéses taskból is kiadható.

Kurzív kisbetűvel vannak nyomtatva azok az események, amelyek a folyamatot tekintve kívülről, de nem másik feldolgozási folyamatból érkeznek.

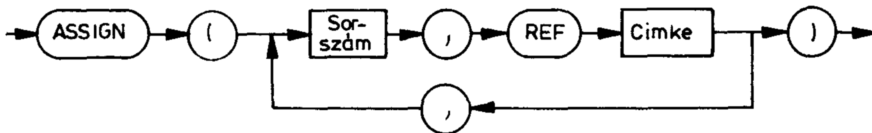
HELP:

A *7.3. ábra* szerinti utasítás segítségével a ⟨**cimke**⟩-vel jelölt alprogram (esetleg több alprogram) task-ként deklarálható, amely task azután az ⟨**nr**⟩ sorszámhoz lesz hozzárendelve. Ez a sorszám meghatározza a task *prioritását* is. A legmagasabb prioritáshoz az 1. sorszám tartozik, ami a program abortálása végett az ún. „End Process" számára van fenntartva.

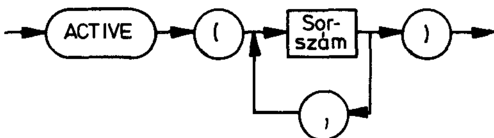
A *7.4. ábrán* szereplő ACTIVE-utasítás segítségével taskokat lehet aktivizálni. A *7.5. ábra* szerinti ERASE-utasítás leállítja a megadott taskot, ill. ha a ⟨**nr**⟩ sorszámot elhagyjuk, akkor az összes futó taskot. A SIGNAL- és a WAIT-utasítások a 4.7.3. pont értelmében a taskok szinkronizálására valók. Az ERASE-zel leállított task egy ACTIVE-



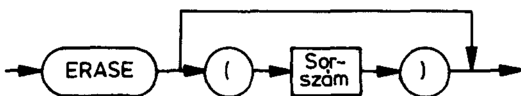
7.2. ábra. Taskok lehetséges állapotainak diagramja az AL nyelvnek a karlsruhei egyetemen kifejlesztett változatánál



7.3. ábra. Task deklarációja HELP-ben



7.4. ábra. Task aktivizálása HELP-ben



7.5. ábra. Task leállítása HELP-ben

utasítással újraindítható. Mivel a HELP-ben csak globális változó használható, ezért a task-ként deklarált alprogramokhoz sem tartozik önálló tárolóterület, és így ezeknek a taskoknak feltehetően nem is futtathatók további reprezentációi sem. Ennek ellenére a HELP alprogram kezelését is a task-kezelések közé soroltuk, mivel a HELP alprogram-hívási rendszere még a task-kezeléshez áll a legközelebb. Pontos leírás hiányában e helyen nem áll módunkban részletesen tárgyalni a HELP task-kezelési állapotdiagramját. Ha HELP-ben a taskot megtestesítő alprogramot olyan programciklusba építjük be, amely ezenkívül a robotberendezés műveleti ciklusát is tartalmazza, akkor a HELP-ben „Cycle-program"-ként ismert feldolgozási programhoz jutunk.

7.3. Társrutinok

A kváziparalel futtatható CO-blokkokkal és a taskokkal ellentétben a társrutinok (együttfutó rutinok) szervezése olyan, hogy vagy teljes egészében, vagy csak részben szekvenciálisan kerülnek feldolgozásra, ugyanakkor azonban az egyes rutinok önálló adatterületei az inaktív fázisban is megmaradnak. Az együttfutó rutinoknak lehet több felélesztésük is. A társrutin első hívásának hatására kijelölődik az adatterület. Ezután a rutin törzsének utasításai kerülnek végrehajtásra egészen addig, amíg valamilyen megszakítást előidéző utasítás meg nem állítja a futást (pl. SIMULA 67-ben a RESUME vagy a DETACH). A DETACH visszaadja a vezérlést a főprogramnak, míg a RESUME hatására a már korábban aktivizált társrutinok ismét aktívvá válnak. A főprogram egy további utasítással ismét elindíthatja a korábban felfüggesztett társrutint.

Az egész eljárásban az a különleges, hogy a társrutin mindig azon a ponton folytatódik, ahol korábban meg lett szakítva. A robotberendezés műveleti ciklusa elvileg egyetlen ciklikusan futtatott társrutin segítségével is programozható. Ilyenkor a külső társrutinok

felől érkező megszakításkérések arra valók, hogy segítségükkel le lehessen kezelni a hibás szituációkat, vagy különleges, több robotkart is érintő feladatokat tudjunk megoldani.

Az általunk tárgyalt robotvezérlő nyelvekben nincs megoldva a társrutinok kezelése, sőt jelenleg nem is ismeretes olyan robotvezérlő nyelv, amely a társrutinokat kezelni tudná. Ennek ellenére megemlítjük a társrutinok kezelésének lehetőségét, hiszen párhuzamos folyamatok feldolgozására alkalmas eszközről van szó, és egyben a lokális érvényű adatkezelés, valamint az igen jól áttekinthető szinkronizálás lehetősége nagyobb programozási biztonságot nyújt, mint az AL nyelvből ismert CO-blokkok használata. Sajnos a társrutinoknál nem megoldható az egymástól független műveletsorozatok kváziparalel feldolgozása, ami jelentős hátrányként könyvelhető el.

A társrutinokat kezelő nyelvek közül meg kell említeni a SIMULA 67-et. Ez a nyelv az AL-hoz és a PASCAL-hoz hasonlóan az ALGOL nyelvjáráshoz tartozik, és mint ilyen blokkorientált, strukturált programozásra alkalmas, és az eljárások, ill. függvényeljárások rekurzív hívása is megengedett.

7.4. Paralel feldolgozás a VAL, a SIGLA és a ROBEX nyelvekben

A paralel feldolgozás szinkronizálására csak a SIGLA-ban van kidolgozott megoldás, éspedig a 4.7.3. pontban tárgyalt szemaforkezelő utasítások felhasználásával. A feldolgozások SIGLA nyelven írt programból alprogramok formájában indíthatók. Mivel ez külső jelektől függően is történhet (l. pl. REACT-utasítás), így ennek segítségével a szokásos érzékelőkre vonatkozó utasítások szinkronizálását is meg kell oldani.

ROBEX-ben is megvan a lehetőség, hogy az érzékelőkre vonatkozó utasításokkal több programot szinkronizáljunk. ROBEX-programmal azonban nem lehet másik programot indítani.

8. A programfejlesztő és a futtatásvezérlő rendszerek

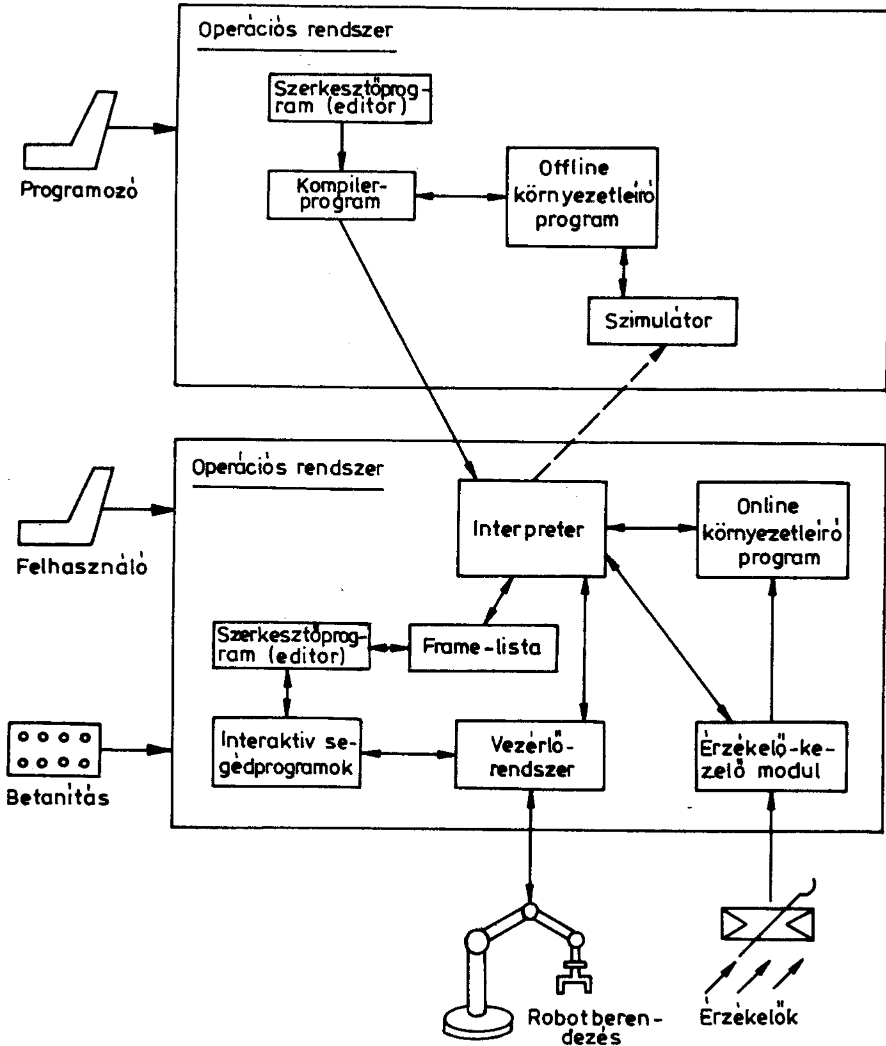
A programok írásához, karbantartásához és kezeléséhez a számítógépes adatfeldolgozás minden szintjén szükség van szoftver-eszközök egész rendszerére. Ilyen szoftver segédeszközök az editor (szövegszerkesztő), a fájlkezelő rendszer, a fordítóprogram, az interpreter (értelmező), a hibakereső stb. Ezek közül némelyek – így a fájlkezelő rendszer vagy a szövegszerkesztő – inkább az általánosabb szoftver segédeszközök közé sorolhatók. Ezek már nemcsak egyik vagy másik programozási nyelv támogatására, hanem általánosabb felhasználói célok kielégítésére is alkalmasak. A többi szoftver már általában robotorientált feladatokat lát el. Az ilyen szorosan egymáshoz kapcsolódó szoftver segédeszközök összességét programrendszernek nevezzük (BLUME [8.1], SANDEWALL [8.2]). A rendszer szerkezetét, a szükséges elemeket és ezek szolgáltatásait az határozza meg, hogy milyen a megvalósítani kívánt programozási technika, milyen a nyelv, az alapul vett hardver-konfiguráció, valamint az, hogy milyen szintre kívánjuk emelni a rendszer kezelhetőségének színvonalát. A programrendszer kezelhetőségének színvonalával nem a felhasználók valamiféle kényelmességét vagy többé-kevésbé fölösleges luxusigényét kívánjuk szolgálni, hanem elsősorban olyan kulturált viszonyok kialakítására gondolunk, amely a programbiztonság, a karbantarthatóság és az egyszerű módosíthatóság tényezőit hivatott támogatni és tökéletesíteni. Ezek a rendszernek igen fontos tényezői, melyek a hosszú távon felmerülő szoftver költségeket jelentős mértékben befolyásolják. Ezen a téren az idejekorán megtett beruházások jelentőségét gyakran sokszor alábecsülik.

Egy robotvezérlő nyelv futtatásvezérlő rendszeréhez tartozik a nyelv utasításértelmező programja és az interaktív rendszer, de ez nincs meg minden esetben. A futtatásvezérlő rendszer az operációs rendszer funkciói közül is többet igénybe vesz, elsősorban a paralel folyamatok szervezésekor. Ha az operációs rendszer ezeket a szolgáltatásokat nem tudná nyújtani (pl. egyetlen felhasználói rendszer esetén), akkor a futtatásvezérlő rendszerben kell ezeket megvalósítani.

A *8.1. ábra* nagyon jó áttekintést nyújt a különféle szoftver-komponensek együttműködéséről. Nem említettük még az ábrán feltüntetett szimulátorprogram szerepét. A szimulátor a program tesztelésekor nyújt hasznos segítséget azzal, hogy a manipulátorok működését szimulálja, és ellenőrzi (l. a 8.5. szakaszt). Ezzel a programtesztelési munka döntő részben helyettesíteni képes a manipulátort.

A rendszer teljesítményétől és kialakításától függően a compiler-program olyan részleteket is tartalmazhat, amelyek egy többé-kevésbé bonyolult környezetleíró modell előállítására és kezelésére is alkalmasak.

A következőkben bevezető jelleggel áttekintjük a robotrendszerek programozásához nélkülözhetetlen fejlesztő és futtatásvezérlő rendszer lényeges elemeit.

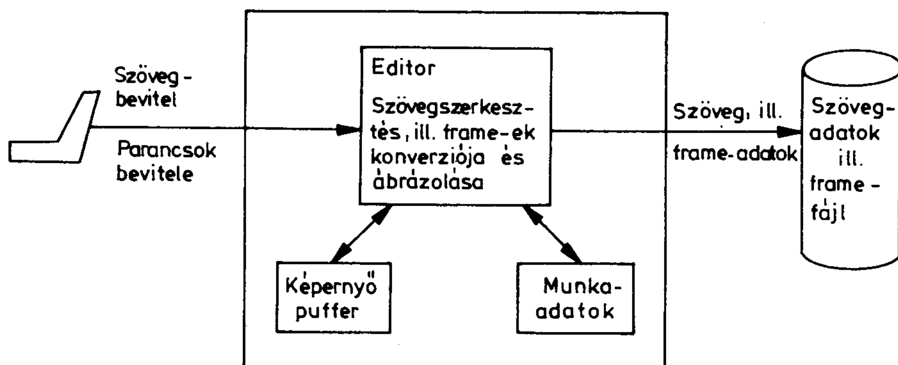


8.1. ábra. A programfejlesztő és a futtatásvezérlő rendszerek elemeinek kapcsolatai a magasabb szintű programnyelveknél

8.1. Az editor

Az editor (vagy szövegszerkesztő) nagyobb adattömeg előállítására és módosítására alkalmas kiszolgáló program, így pl. szövegek vagy frame-ek, valamint ezek adatainak szerkesztésére alkalmas (l. *8.2. ábra*).

A szövegszerkesztő programok általában a rendszer standard kiszolgáló programjai közé tartoznak, ezért ezeket csak röviden érintjük. A legegyszerűbb editorok csak soronként képesek feldolgozni a szöveget, ez pedig főleg programjavításoknál igen nehézkes megoldás. Ezért újabban a legtöbb szövegszerkesztő már olyan utasításkészlet-



8.2. ábra. A szöveg- ill. a frame-szerkesztő programok rendszerkapcsolatai

tel rendelkezik, amelynek segítségével a szöveg karakterenkénti feldolgozása is megoldható. Ezeknél a kurzorral a szövegben tetszőleges helyre ugorhatunk, és ott a szükséges karakter beszúrható, javítható vagy törölhető. A fejlettebb szövegszerkesztőknek van kereső funkciójuk, ezenkívül több szövegfájl egyesíthető, sőt a szövegfájlok részekre is bonthatók.

A különféle frame-szerkesztő programok ezzel szemben tipikusan robotorientált szoftver segédeszközök. A frame-szerkesztő különféle szolgáltatásainak segítségével a felhasználó könnyen előkészítheti a szükséges frame-listákat. Ezek a szolgáltatások tipikusan az alábbiak:

- a szükséges frame-ek a kezelő számára érthető formában listázhatók;
- az új frame-rendszerek bevitele az 5. fejezetben tárgyalt interaktív betanítási eljárással vagy szöveges adatbevitellel történhet;
- az egyes frame-rendszerek különféle adatai közvetlenül javíthatók, ha pl. a frameszerkesztővel új pozíció- vagy orientációadatokat adunk meg;
- segítségükkel a megszerkesztett frame-listák külső tárolóegységen is tárolhatók.

8.2. A compiler-program (fordítóprogram) és a processzor

A compiler olyan program, amely a *forrásnyelven* írt programot vagy programrészletet egy másik nyelvre, az ún. *célnyelvre* fordítja le. A forrásnyelv általában magasabb szintű algoritmikus nyelv. Ilyenek pl. a könyvünkben bemutatott PASCAL, AL és HELP nyelvek, valamint bizonyos megszorításokkal a VAL nyelv is. A célnyelv lehet gépi kód, amely a gép által közvetlenül végrehajtható, vagy lehet egy közbenső ún. *pszeudokód,* amelyet egy másik program, az *interpreter* (értelmező) hajt végre. Az NC-nyelvek fordítóprogramja az ún. processzor, amelyre később még visszatérünk. Az NC-nyelvek processzora nem tévesztendő össze a számítógépes hardver központi egységének nevével (angolul Central Processor), amit röviden szintén processzornak hívunk.

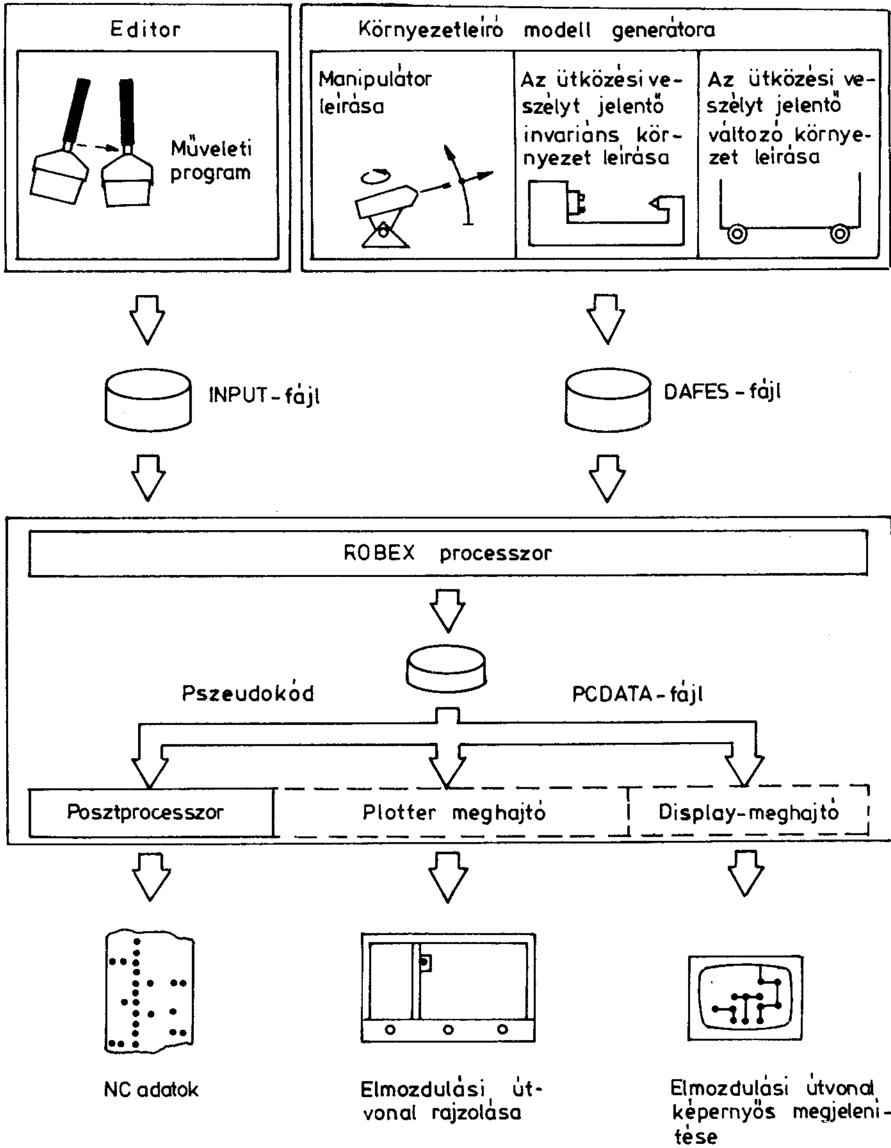
A compiler-program feladatainak zöme semmiben sem tér el a számítástechnikában megszokott feladatoktól, ezért ezekre nem térünk ki, csak utalunk az irodalomra [8.3, 8.4, 8.5, 8.6]).

A bevitt programot – amely véges hosszúságú karaktersorozatnak tekinthető – lexikai és szintaktikai elemzésnek kell alávetni. A lexikai elemzés a bevitt jelsorozatot felbontja az ún. alapszimbólumokra, így alapszavakra (pl. BEGIN, IF stb.), számokra, azonosítókra és különleges jelekre, ill. jelcsoportokra (ilyenek pl. =, := stb.). A szintaktikai elemzés fázisában azt kell megvizsgálni, hogy a fenti alapszimbólumok sorozata a nyelv grammatikai szabályainak megfelelően (vö. 2.3. szakasz) érvényes mondatot alkot-e? A compiler (fordítóprogram) egyik legfontosabb jellemzője az a mód, ahogyan a szintaktikai hibákat kezeli. A legegyszerűbb – és legkevésbé szerencsés – esetben a compiler egyszerűen csak jelzi a hibát, azonban egy szintaktikai programhiba miatt általában olyan sok további hiba keletkezik, hogy ilyenkor a compiler további futásának már aligha van értelme. A jó compiler (fordító)-programok a hibát egyetlen utasításra határolják be, kiadnak egy hibaüzenetet, amely találóan jellemzi a hiba okát, majd folytatják a szintaktikai elemzést, jóllehet a korábbi hibából adódó további szintaktikai hibák már nem szűrhetők ki (jellegzetes példa egy változó hibás deklarálása). Vannak olyan compilerek is, amelyek a legegyszerűbb szintaktikai hibákat – pl. hiányzó vesszőt, pontosvesszőt stb. – képesek korrigálni, ill. pótolni. A szoftver kifejlesztésére fordított idő a compiler szolgáltatásainak színvonalától is függ, ezért ez a kérdés gazdasági szempontból is jelentős. A szintaktikai elemzés után a szemantikai ellenőrzések következnek, ilyen pl. annak ellenőrzése, hogy az operandusok típusa összeegyeztethető-e a kijelölt művelettel, vagy hogy megfelelő-e az aktuális paraméter típusa egy adott eljárás hívásakor. Ebben a fordítási fázisban blokkorientált nyelveknél a compiler megszervezheti a változók számára fenntartott tárolóterület kiosztását is (l. 2.2.3. pont).

Az ezután következő fordítási fázisban a fordítóprogram meghatározza a vezérlésátadó utasítások ugrási címeit és a kódot a célnyelven állítja elő. Ilyenkor a kód még sokféle szempontból optimálható. A programkód megváltoztatható pl. úgy, hogy a gyakran használt adatokat a program, ill. eljárás kezdetén egyszer betöltjük a kijelölt regiszterekbe, de megoldható az is, hogy relatív ugróutasításokat alkalmazunk, amely kisebb ugrásoknál kevesebb ráfordítással jár. Az egész eljárás úgy is megszervezhető, hogy a compiler a teljes programot egy belső ábrázolási formában letárolja, majd elejétől a végéig többször átdolgozza. Attól függően, hogy a betöltött programon hányszor fut végig a compiler, egymenetes vagy többmenetes fordításról beszélhetünk. Az egymenetes fordítóprogramok általában gyorsabbak, ha a program szintaktikailag rendben van. Ha azonban hiba volt a programban, akkor az egymenetes compiler feleslegesen állítja elő a kódot, ami többmenetes compilereknél már elkerülhető, és ezért ilyenkor az utóbbi típus a gyorsabb.

Az NC-vezérlő nyelveknél az ún. processzor feladatai hasonlóak a compilerek feladataihoz, azzal a különbséggel, hogy az NC-nyelveknél általában egy közbülső ún. pszeudokód előállítására van szükség. Ezt a pszeudokódot egy postprocesszor fogadja, amely azt az alkalmazott NC-gép, esetleg robotberendezés vezérlési rendszeréhez illesztve a célgéptől függően dolgozza fel *(8.3. ábra)*.

További eltérés mutatkozik a compilerekhez képest abban is, hogy a pszeudokódban már megvannak a forrásprogramban előforduló számítások eredményei is. A pszeudokód tehát lényegében csak olyan utasításokból áll, amelyeket a gép vezérlőrendszere már közvetlenül működtetésekké tud átalakítani anélkül, hogy a program futása közben lehetősége lenne a cél kiszámítására (pl. az érzékelők adatai alapján). A processzorok használatának ez a legnagyobb hátránya, és ugyanígy a ROBEX nyelvé is. Időközben néhány olyan NC-nyelv is megjelent, amely már az információfeldolgozás tágabb lehetőségeivel is rendelkezik. Ilyen pl. a McDonell Douglas cég MCL nyelve.



8.3. ábra. A ROBEX programfejlesztő rendszer felépítése [15]

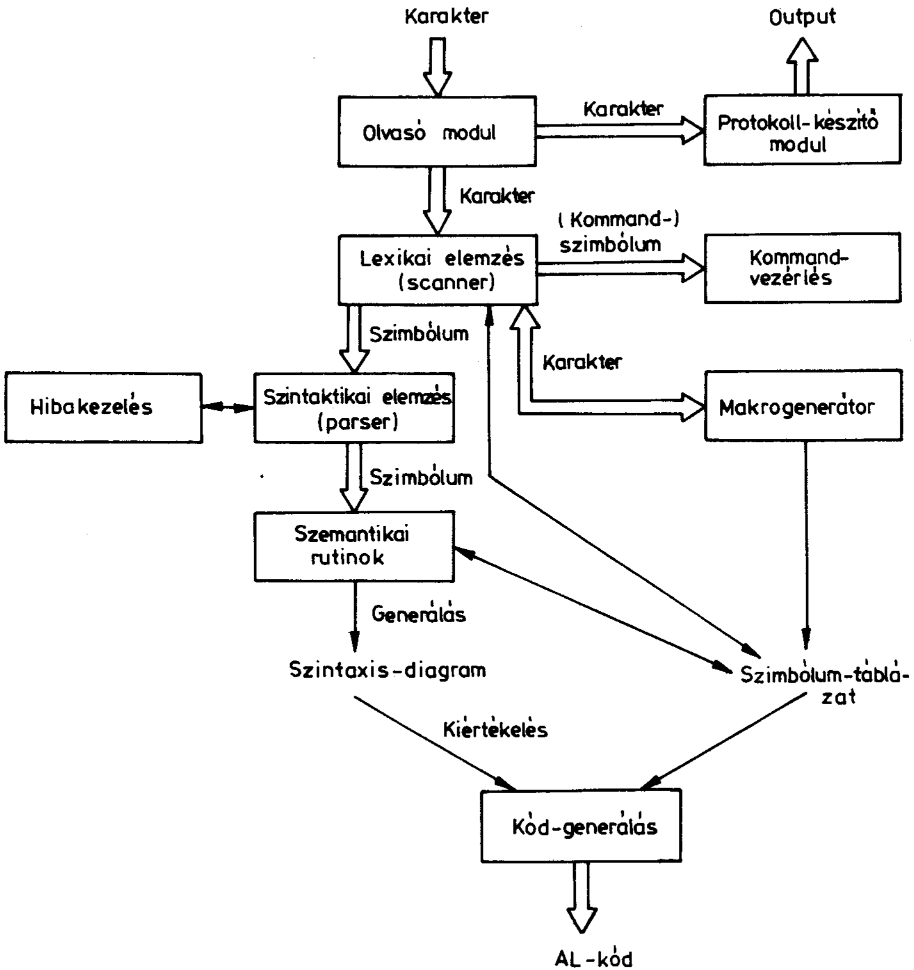
Amennyire a SIGLA nyelvleírás dokumentációiból [11, 12] megállapítható, az Olivetti cég a tradicionális NC-technikától eltérően olyan interpretert alkalmaz a forrásprogramok fordításához, amely a SIGMA-berendezések numerikus vezérlését már közvetlenül is működteti. SIGMA-berendezéseken olyan rendszerek értendők, amelyek „a szokásos NC-gépektől eltérően rendellenes üzemállapot fellépésekor döntéshozatalra is képesek" [12]. Az Olivetti cég ezek közé sorolja az ipari robotokat is.

A robotvezérlő nyelvek fordítására alkalmas compilerek egyik sajátságos jellemvonása az, hogy általában egy interaktív komponenssel is rendelkeznek. Ezt vagy eleve beszer-

kesztik a compilerbe, vagy interfész kialakításával oldják meg (l. 5. fejezet). A compilerek másik fontos jellemvonása, hogy környezetleíró modellt képesek generálni – már legalábbis azoknak a nyelveknek az esetében, amelyek az implicit programozás koncepcióját követik.

Az elsőnek említett tulajdonsággal csaknem mindegyik eddig kifejlesztett robotvezérlő nyelv compilere rendelkezik, az utóbbi pedig fokozatosan egyre nagyobb jelentőségre tesz szert.

A környezetleíró modell adatainak nagyobb részét a program futása közben a robot pozícióváltoztatásaival együtt állandóan aktualizálni kell. Mármost egy egyedi feladatot tekintve statikus és általános környezetleíró modellből a compiler (fordító) kiemeli az adott feladatra nézve lényeges adatokat, amelyek alapján előállít egy a konkrét problémához illő környezetleíró modellt, és ezeket az információkat felhasználja a program generálásakor. A robotprogramozásnak ez a modernebb irányzata egyelőre még csak kibontakozóban van, így a bemutatott nyelvekben ezeknek a gondolatoknak még a csírájuk sincs megvalósítva.



8.4. ábra. Az AL-compiler felépítése

A ROBEX nyelvnek a *8.3. ábrán* felvázolt környezetleíró modellgenerátora is inkább célkitűzésnek tekinthető, mint megvalósult állapotnak. Ebben a témakörben az IBM AUTOPASS rendszere (l. LIEBERMANN [2.10, 8.7]) és a Massachusetts Institute of Technology mesterséges intelligenciával foglalkozó laboratóriumának LAMA rendszere (l. LOZANO-PEREZ [8.8]) érdemelnek említést, mint létező fejlesztőrendszerek.

A robot-compilerek (fordítók) egy további sajátos tulajdonságaként meg szeretnénk említeni az amerikai AL változat compilerében megvalósított ún. Worldmodeller-t – azaz környezet modellező részt. Ez a program kezeli az összes frame-rendszert, és a velük kapcsolatos affix hozzárendelési struktúrákat, tehát azt, hogy melyik frame melyikkel van összekapcsolva. Ez a program már a fordítás során számításba veszi a pozicionálásokkal kapcsolatos lehetséges változatokat, tehát mintegy előre figyelembe veszi a pozicionáló utasítás várható végleges tartalmát. Ezeket a számításokat a tényleges futás közben természetesen még helyesbíteni, ill. pontosítani kell, mivel a fordítás időpontjában az egyes pozícióknak még nem minden adata ismert. Ugyanakkor az is igaz, hogy programozáskor minél inkább kihasználjuk az AL, és hasonló magasabb szintű programnyelvek tulajdonságait, annál kevésbé biztosítható, hogy a szóban forgó adatok ismertek legyenek a program tényleges futása előtt. A „Worldmodeller" szerinti eljárás egyik indítóoka éppen az a célkitűzés volt, hogy a robotvezérléseknél a bonyolult pályaszámításokat háttérgépre tegyük, tehermentesítve ezzel a robotvezérlő számítógépet. Időközben azonban a kisszámítógépek olcsóbbak és mindenképpen nagyobb teljesítőképességűek lettek, ez az érv tehát veszített jelentőségéből, és ma már ez az elgondolás általában háttérbe szorul.

A compilerek lehetséges szolgáltatásainak illusztrálására bemutatjuk a karlsruhei egyetemen kifejlesztett alapverzióra implementált AL-compiler néhány tulajdonságát (l. [4, 5, 6]). Az AL-compiler szerkezeti felépítését a *8.4. ábra* szemlélteti.

A forrásszöveg bevitele vagy közvetlenül terminálról, vagy a memóriában tárolt egy vagy több szövegfájlról történhet, de e két lehetőség tetszőlegesen kombinálható is. Így a program előre elkészített modulokból a képernyőn szerkeszthető össze. Az összeállított forrásprogram, vagy annak egyes részei ugyanakkor külön fájlokba el is menthetők későbbi felhasználásra. A compiler vezérlésére egy külön kommand (parancs)-nyelv áll a felhasználó rendelkezésére. Ezt az utasításkészletet az AL nyelv REQUIRE-utasításából fejlesztették ki. A fordítás kezdetén a compiler-program beolvassa az USERDEF.AL-fájlt, amelybe a felhasználó előzőleg beírja az általánosan használt változók, rutinok stb. definícióit. A compiler a következő output fájlokat állítja elő:

- a rekordfájlt a forrásprogrammal együtt (opcionális);
- a listafájlt a kilistázott programmal (ez is opcionális), (programhiba esetén minden esetben keletkezik listafájl, amelybe a hibás sorok és a hibaüzenetek kerülnek bele);
- a kódfájl, amibe a generált AL-pszeudokód kerül (ez csak a standard- és a tesztváltozatban jön létre).

A compiler (fordító) hibakezelő eljárása igen sokoldalú szolgáltatásokat nyújt, lehetővé teszi a kommand (parancs)-nyelv utasításaiban előfordult összes hiba interaktív javítását, és önállóan helyesbíti az egyszerűbb szintaktikai hibákat. A kétfokozatú szintaktikai hibakezelő rész az első menetben megkísérli, hogy a hibát az egyes alapszimbólumok kiegészítésével, helyettesítésével vagy törlésével hárítsa el. Ha ez nem sikerül, akkor a compiler keres egy olyan támpontot (ez minden esetben az utasításokat elválasztó pontosvessző), amelyhez egy vagy több szimbólum törlésével vagy beszúrásával jut el (l. DENKER [8.9, 6]). Ezzel a módszerrel sorban kijavítja a programhibákat vagy

pedig javaslatot ad a javításra, mégpedig úgy, hogy a programban a következő pontosvesszőig terjedő részen belül lokalizálja a hibát. Ezzel a módszerrel messzemenően csökkenthető a korábbi hibák következményeként adódó hibák száma. A hibakezelő eljárás természetesen pontosan dokumentálja a listában azokat a módosításokat, amelyek ezzel a módszerrel készültek. A kísérletek azt tanúsították, hogy az egyszerűbb szintaktikai hibákat az eljárás helyesen korrigálja. Egy példa: A hibás

PRINT („A értéke:" A;
IF A = B X←A ELSE X←B;

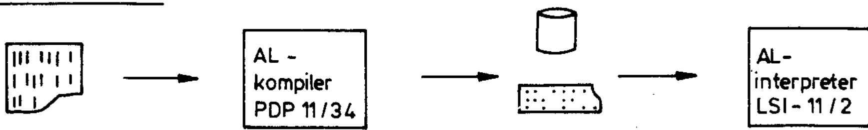
utasításokat az automatikus hibakezelő a

PRINT („A értéke:",A);
IF A = B THEN X←A ELSE X←B;

utasítássorozatra javítja.

A compiler (fordító) interaktív szolgáltatásait úgy vehetjük igénybe, hogy megadjuk egy frame-fájl nevét, amelybe az egyes frame-ek nevei és velük együtt a frame-eknek a betanítási eljárás segítségével nyert adatai vannak felsorolva. A futás során az interpreter minden olyan frame deklarálásakor, amely a frame-fájlban előfordul, az ott megadott adatokkal inicializálja a frame-et. A compiler (fordító) háromféle üzemmódban működhet (l. *8.5. ábra*).

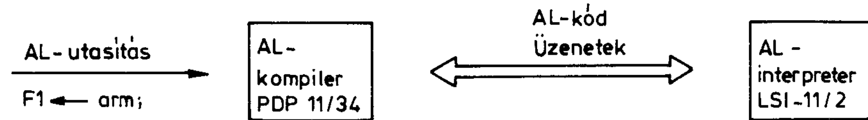
Standard üzemmód



AL nyelvű programszöveg (forrásprogram)

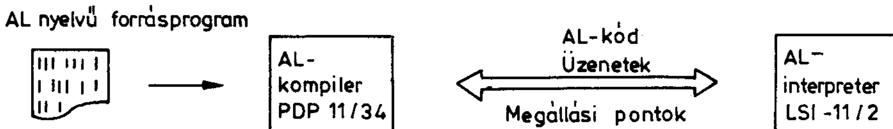
AL - kód

Interaktív üzemmód

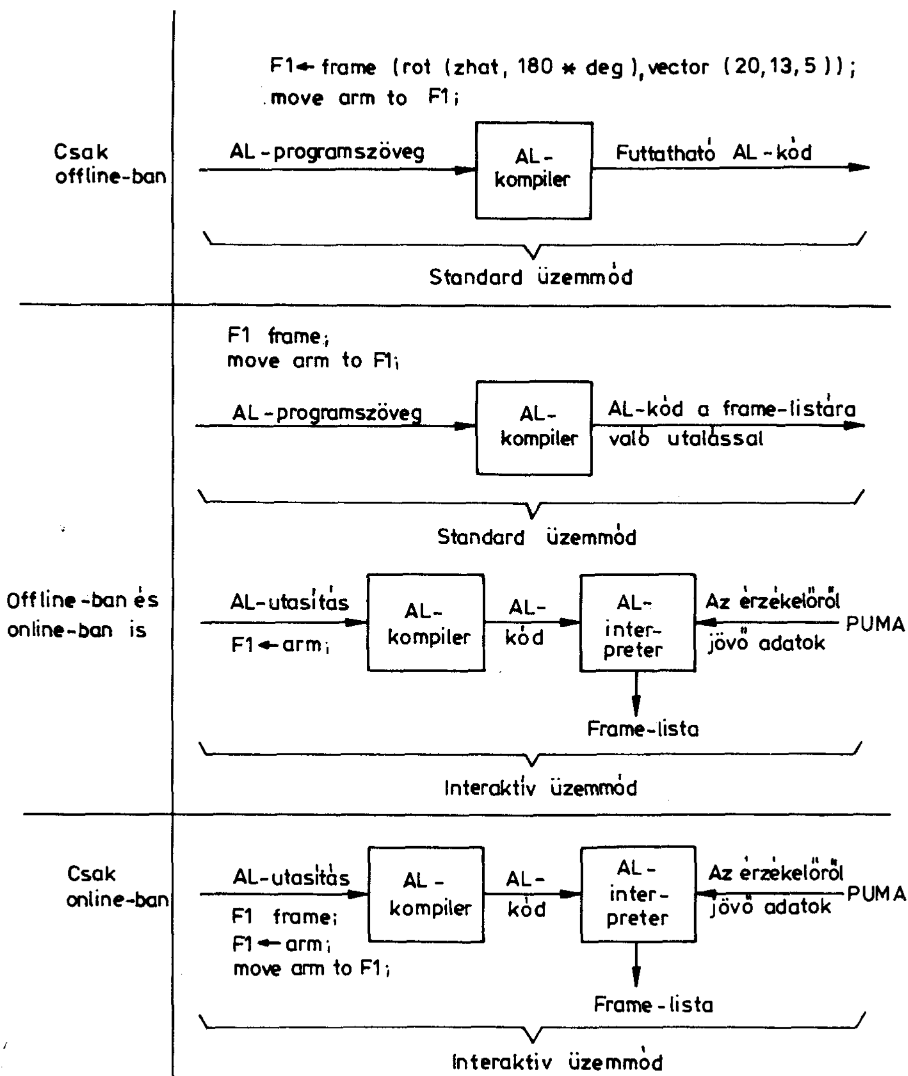


Ellenőrző üzemmód

AL nyelvű forrásprogram



8.5. ábra. Az AL-compiler üzemmódjai



8.6. ábra. Az AL-compiler használatakor választható programozási lehetőségek

Standard üzemmód
Mint már említettük, a compiler (fordító) előállít egy fájlt, amibe az AL pszeudokód kerül. Ezt az eljárást off-line módon hajtja végre, de az interpreterrel ilyenkor nincs kapcsolatban. Ezt a pszeudokódot később az interpreterrel akárhányszor végre lehet hajtani.

Ellenőrző üzemmód (Tesztmódus)
A compiler (fordító) ilyenkor az első menetben ugyanúgy fordítja a programot, mint a standard üzemmódban, a lefordított AL programkódot azonban átadja az interpreternek, amely azt végrehajtja. Eközben az interpreter minden utasításnál üzenetet küld a compilernek, amely ezt az üzenetet feldolgozza és a soron következő utasítás végrehajtá-

sára ad parancsot. A felhasználó a compilernek küldött kommandok segítségével szüneteltetheti az egyes utasítások végrehajtását, beszúrhat megállási pontokat, ill. törölheti azokat, ellenőrizheti és javíthatja a változók tartalmát, vagy újra is indíthatja a programot. Emellett vissza lehet állítani a legutolsó pozicionáló utasítás előtti állapotot.

Interaktív üzemmód

Ebben az üzemmódban a compiler az AL program minden egyes utasítását a bevitel után azonnal lefordítja, átadja az interpreternek (értelmezőnek), ez pedig végrehajtja. Emellett a felhasználónak arra is megvan a lehetősége, hogy betanítást hajtson végre, vagyis a robottal a pálya megfelelő pontjára állva felvegye a pályaponthoz tartozó frame pozíció- és orientációadatait. Ezeket a frame-adatokat a felhasználó tárolhatja a frame-fájlba, vagy ha onnan olvassa be, akkor módosíthatja is a kívánt adatokat.

Az ellenőrző üzemmódból a frame-fájlban tárolt frame-adatok javítása céljából át lehet váltani az interaktív üzemmódba. Ha a javítás után ismét az ellenőrző üzemmódot hívjuk meg, akkor a program elölről indul, tehát nem ott folytatódik, ahol a javítás miatt félbeszakítottuk, mivel a programtesztelés célja a megváltozott kiinduló adatokkal végrehajtott műveletek ellenőrzése.

Mint a *8.6. ábrán* látható, a rendszer háromféle programozási üzemmódban használható:

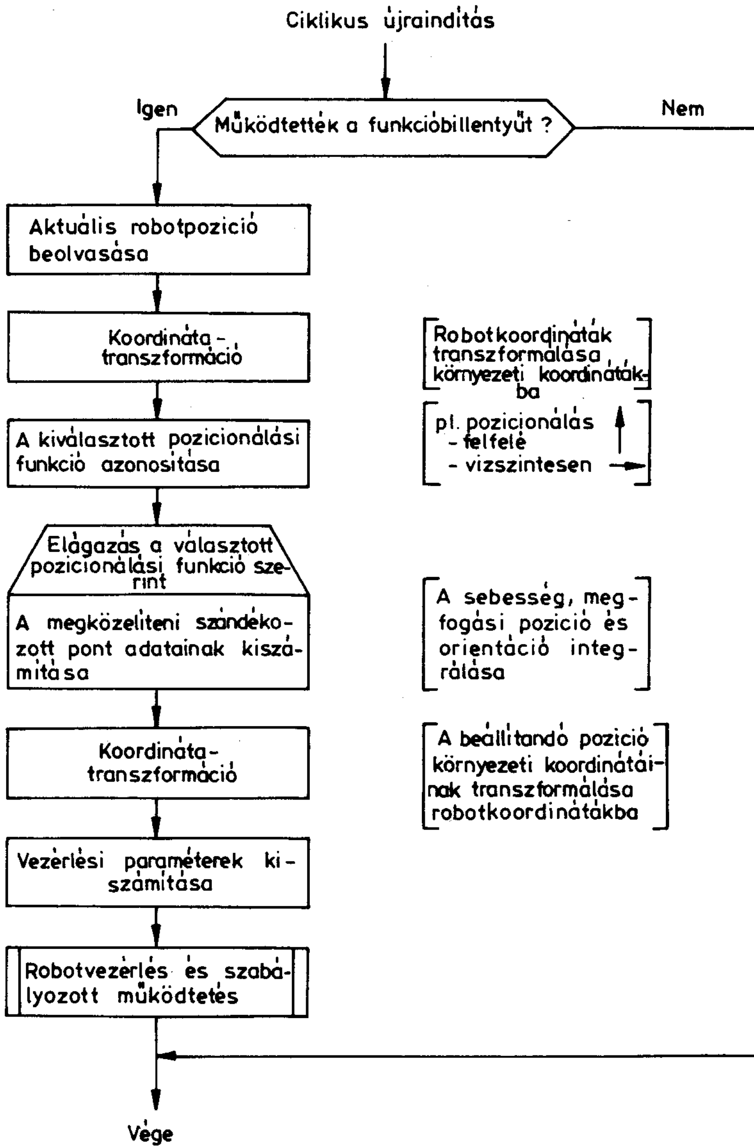
- tisztán off-line-programozásban;
- kevert off-line és on-line programozásban;
- és tisztán on-line programozásban.

Ezekkel a lehetőségekkel a legváltozatosabb felhasználói igényeket is ki lehet elégíteni.

8.3. Interaktív összetevő

Az 5. fejezetben megismert betanítási eljárás a rendszer interaktív programozási szolgáltatásainak segítségével valósítható meg. A betanítási eljáráshoz egyrészt az ún. teach-box ciklikus lekérdezésére is szükség van. A teach-box a betanításhoz használt billentyűzettel ellátott kezelői periféria. Ennek lekérdezésekor a program megállapítja, hogy a kezelő melyik funkcióbillentyűt működtette – pl. a robotkoordináta-rendszer valamelyik irányába történő pozicionálást vagy pl. a második robottengely elforgatását vezérlő billentyűt stb. Ennek alapján a robot pillanatnyi helyzetéből kiindulva a beállított sebességnek megfelelően a rendszer kiszámítja a kívánt pozíció és orientáció adatait, majd a robot vezérlőrendszere elvégzi a működtetéseket *(8.7. ábra)*. A programrendszer a betanítást végző elemek ciklikus aktivizálásán kívül állandóan ellenőrzi a billentyűzetet is, hogy nem történt-e adatbevitel. Amikor a kezelő az említett kommandok segítségével megad egy frame-nevet, akkor beolvassa az aktuális robotpozíció-, ill. orientációadatokat, majd ezeket a koordinátaadatokat a frame-listában a frame neve alatt tárolja (l. *8.8. ábra*).

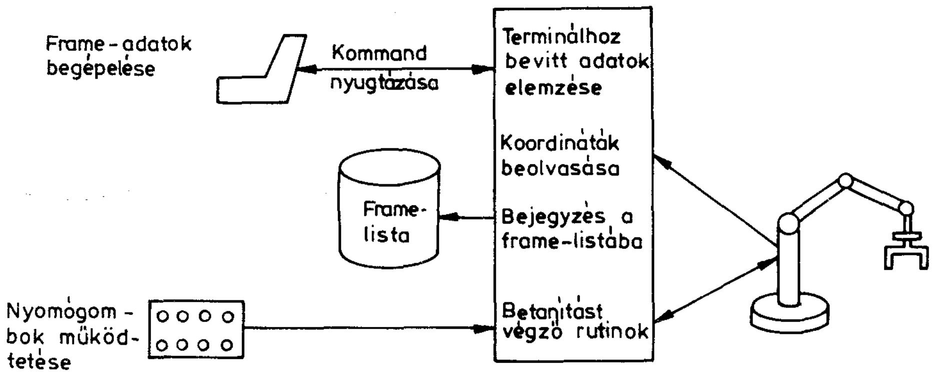
Az interaktív összetevő a futtatásvezérlő rendszerhez is tartozhat, ha az egyes framerendszereket a programfutás első menetében kell betanítani a rendszernek – ilyen pl. a ROBEX (vö. 5. fejezet). Ha az interaktív rendszer csak a frame-lista összeállítására szolgál, akkor annak ellenére nem tartozik a futtatásvezérlő rendszerhez, hogy a robottal csak on-line kapcsolatban futtatható.



8.7. ábra. A betanítási rendszer működése

8.4. A futtatásvezérlő rendszer

Futtatásvezérlő rendszer címszó alatt összefoglaló néven azokat a szoftver segédeszközöket értjük, amelyek az alkalmazói program normál futásához általában nélkülözhetetlenek. Itt csak a normál üzemmódra korlátozott szolgáltatásokat értjük, mivel néhány rendszernél a programteszteléshez szükség van a compilerre is (pl. az AL-nál, vö. 8.2. szakasz), ill. az editorra (VAL), vagy pl. a programfutás első menetében az interaktív

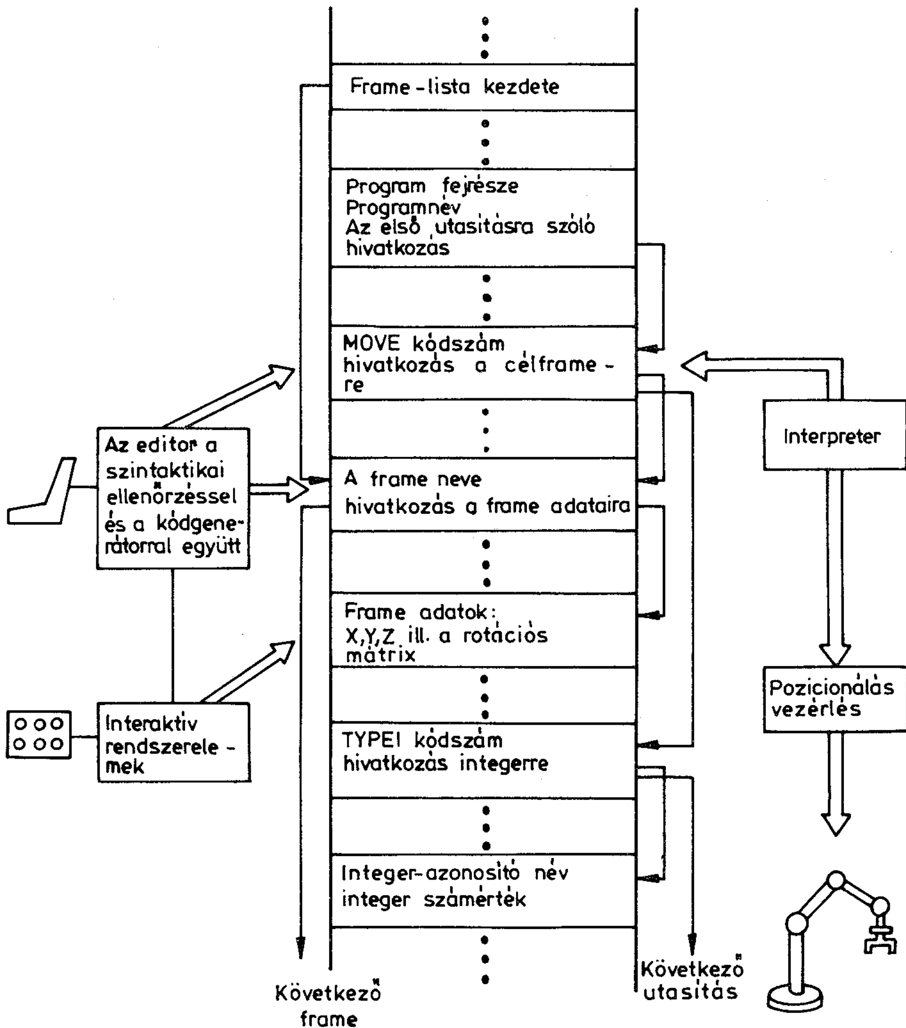


8.8. ábra. A frame-lista előállításához használható interaktív segédprogramok

Programozó
Interaktív, funkciót ellátó rutinok
Editor
Kompiler
Program-kód-listák
Interpreter
Környezetleíró modell
Pozicionálás vezérlése
Érzékelőket kezelő modul
Robotberendezés
Érzékelők

8.9. ábra. Az ipari robotok futtatásvezérlő programrendszereinek elemei

rendszerre is (pl. ROBEX, vö. 5. fejezet), emiatt pedig a programfejlesztő és a futtatásvezérlő rendszer már sok tekintetben fedné egymást. Az összefüggéseket a *8.9. ábra* szemlélteti. Az ábrán szaggatott vonallal jelöltük azokat az elemeket, amelyekre csak kivételes esetekben van szükség. Különleges szerepet játszik a környezetleíró modell, mivel az alkalmazói program futásakor ez még csak fő vonalaiban áll össze (l. a 4.1. szakaszt).



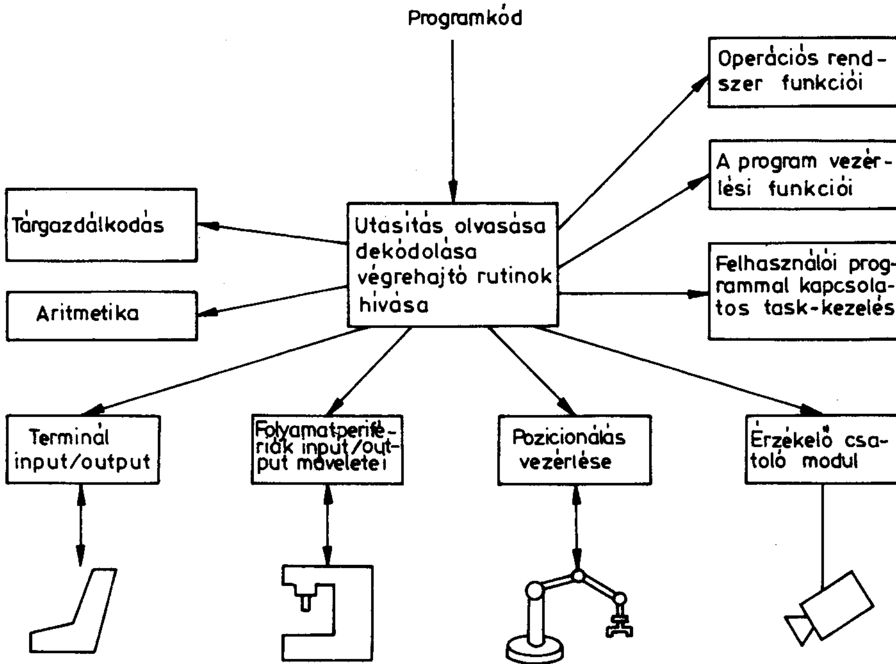
8.10. ábra. A programfejlesztő és a futtatásvezérlő rendszerek együttműködése a VAL nyelvnél

Az egyes szoftver-elemek egymásba fonódására mutat példát a *8.10. ábra.* A VAL programfejlesztő rendszer editora és az interpreter paralel futnak, az editorral bevitt programkódot az interpreter azonnal végrehajtja. Az editornak ezenkívül együtt kell működnie az interaktív rendszerrel is, mert a program futása közben korrigálni kell a frame-adatokat is.

A továbbiakban az interpreter és a pályavezérlő programelemeket tárgyaljuk. A szoftver-rendszernek az érzékelőket kezelő modulja – amely a különféle érzékelők önműködő vezérlését látja el – jelenleg még fejlesztési stádiumban van.

8.4.1. Az interpreter

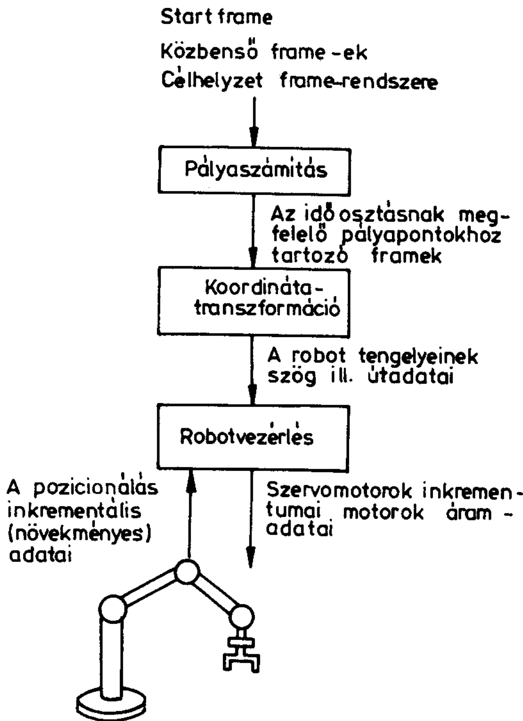
Mind a mai napig még nem helyeztek üzembe olyan számítógépes rendszert, amely az ipari robotok közvetlen vezérlését is ellátná, tehát a robot működtetésére alkalmas gépi utasításokat közvetlenül adná ki. Ezért az alkalmazói programok végrehajtásához interpreterre (értelmezőre) van szükség. Ez egy olyan program, amely elolvassa a compiler által generált programkódot, dekódolja az egyes programokat, majd hívja a megfelelő rutinokat (l. *8.11. ábra*). A változók kezelése, tehát tárolóterület foglalása vagy ismételt szabaddá tétele a 2.2.1. pontban bemutatott séma szerinti tárkezelő eljárás segítségével történik. A képletek kiszámítását az aritmetikai modul végzi, amely a 2.2.2. pontban tárgyalt operandusverem segítségével látja el feladatát. Az ábrán szereplő érzékelőcsatoló modul arra az alapesetre vonatkozik, amikor az interpreter az adott programfejlesztő rendszer érzékelőihez speciális meghajtóegységeken keresztül csatlakozik, azonban tetszőleges érzékelőre alkalmazható szenzormodullal nem rendelkezik. A robotvezérlő nyelvek többnyire stand-alone programok, tehát önállóan, operációs rendszer nélkül képesek futni a robotvezérlő számítógépen. Emiatt olyan modulokkal is rendelkeznek, amelyek tulajdonképpen egy operációs rendszer bizonyos funkcióit látják el. Ilyenek a fájlkezelés floppy-diszken (hajlékony lemezes tárolón), a rendszeróra, a megszakításkérések kezelése és a felhasználó kezelői parancsait feldolgozó modul.



8.11. ábra. Az interpreter programkomponensei

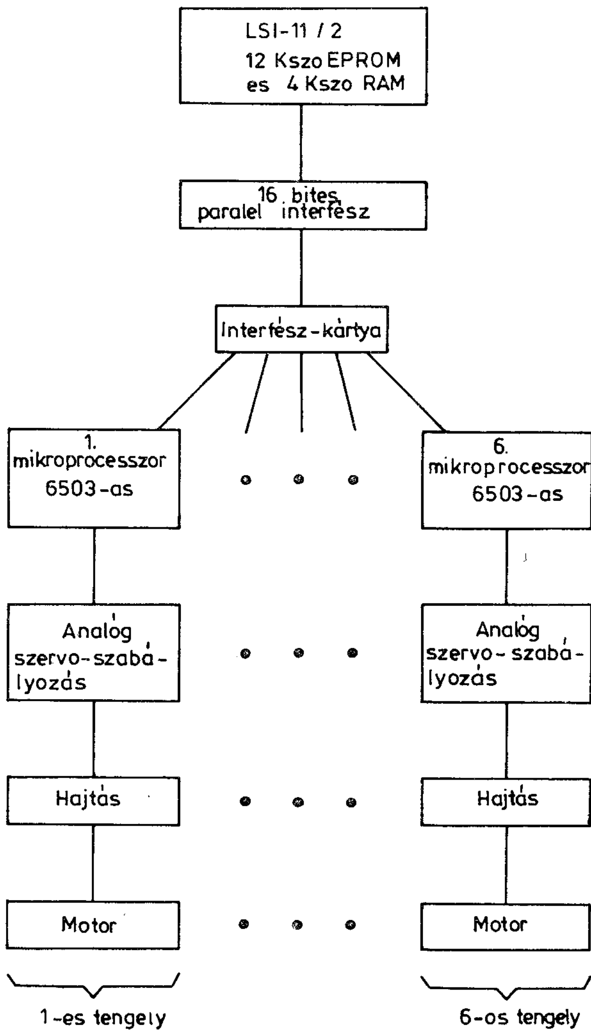
8.4.2. A pályavezérlés

A *pályavezérlés* feladata az, hogy a robotberendezés abba a frame-helyzetbe állítsa be, amely a programban megadott pozicionáló utasításban szereplő célpontnak felel meg. Ennek végrehajtásához a pálya kiszámítását végző modulnak át kell adni a kiindulási frame, a közbenső frame-ek és a célframe adatait. Ez kiszámítja azokat a frame-eket, amelyeket a robotnak a pálya bejárása közben meghatározott időintervallumonként érintenie kell. (A VAL nyelvben az időosztás pl. 28 ms.) A koordinátatranszformációt végző modul ennek alapján minden egyes frame-et a 2.4.2. pontban bemutatott eljárás szerint áttranszformál a robot ízületi koordináta-rendszereibe, és előállítja az egyes tengelyek szögelfordulási, ill. elmozdulási adatait (l. *8.12. ábra*). A szó szűkebb értelmében vett robotvezérlő (tulajdonképpen működtető) rendszer a tengelyelfordulási és eltolási adatokból kiszámítja azokat a növekményadatokat, áram-, ill. feszültségértékeket, amelyek a szervomotorok működtetéséhez szükségesek. Ezeket a vezérlőadatokat az időosztásnak megfelelő időpillanatokban kiadja, ehhez felhasználja a rendszer saját belső óráját, vagy valamilyen más időjelet. Az egyes robottengelyekre a vezérlőrendszer által kiadott pozíciókra történő tényleges beállást úgy ellenőrzi, hogy összehasonlítja az alapjelet az ellenőrző jellel. Ennek érdekében a robot minden egyes tengelyén egy-egy elmozdulásérzékelő található, amelyek segítségével az egyes tengelyek által megtett elmozdulások mérhetők és visszajelezhetők a vezérlésnek. A szabályozás ezután úgy történik, hogy az alapjel és az ellenőrző jel eltérésétől függően működtetik a megfelelő hajtást. Az esetek többségében minden tengelynek külön szabályozása van, az analóg



8.12. ábra. A robot pozicionáló-mozgásvezérlő programcsomag elemei

szervoszabályozást mikroprocesszor ellenőrzi. Ekkor minden egyes tengelyhez tartozik egy ún. ízületi vezérlő és szabályozó processzor, ezeknek a vezérlő számítógép központilag adja meg a szükséges paramétereket. Példaként a *8.13. ábrán* bemutatjuk az UNIMATION cég PUMA 600-as robotberendezés vezérlésének hardverét. Jelenleg a fejlesztések arra irányulnak, hogy a pályavezérlési és koordinátatranszformációs műveleteket egy esetleg több különálló aritmetikai processzorra tegyék, tehermentesítve ezzel a vezérlő számítógépet, amelyen az interpreter (értelmező) fut.



8.13. ábra. Az UNIMATION cég PUMA 600-as robotjának vezérlő hardver-rendszere

8.5. A szimulátor és a program ellenőrzése

A gyakorlati felhasználást megelőzően a robotvezérlő programot alapos és sokoldalú ellenőrzésnek kell alávetni minimálisra csökkentve ezáltal a robotberendezésnek vagy más berendezésnek az esetleges programhibák miatt bekövetkező sérüléseit. Még élesebben vetődik fel a probléma, ha a robotberendezés közelében – akár csak időszakosan is –, de emberek is dolgoznak. A balesetek megelőzésére általában további különleges intézkedéseket kell tenni.

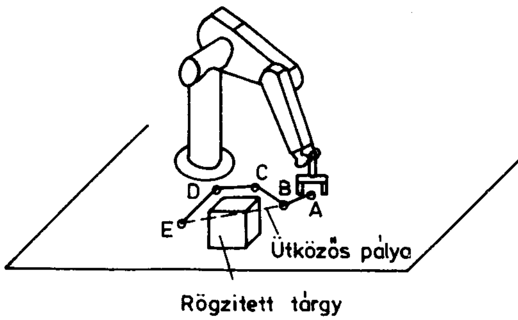
A robotok alkalmazásával kapcsolatos balesetvédelmi intézkedések komplex témaköre mindenképpen túlnő könyvünk keretein, ezért – bármilyen fontos is ez a terület – ennek a kérdéskörnek a tárgyalását kénytelenek voltunk mellőzni.

A programrendszerek helyes működésének bizonyítására sajnálatos módon mind a mai napig nem létezik gyakorlatilag elfogadható és kezelhető módszer; annak ellenére, hogy évek óta folynak ilyen irányú kutatások (l. ELSPAS [8.10]).

Maradnak a megszokott programtesztelési módszerek, amelyeknél a programot véletlenszerűen vagy szisztematikusan kiválasztott adatok tömegének felhasználásával futtatjuk. Ezzel a módszerrel természetesen sohasem bizonyítható be, hogy egyáltalán semmilyen programhiba nem léphet fel, csupán a kipróbált esetekről és az esetleg talált hibákról alkothatunk véleményt. A program tesztelésének a következőkben felsorolt segédeszközei ismertek (ezek közül a fontosabbakról már szó volt a 8.2. szakaszban az AL-compilerrel kapcsolatos példában):

- a programlefutás egyes lépéseinek kinyomtatása (angolul Trace);
- a központi tár tartalmának kinyomtatása (angolul Dump);
- a program futása közben szimbolikus azonosítójuk segítségével használt változók tartalmának olvasása és esetleges megváltoztatása;
- a program megállítása tetszőlegesen kijelölhető, ill. törölhető pontokon;
- a program újraindítása változatlan, ill. módosított kiinduló adatokkal;
- visszaugrás a program egy korábbi utasítására, esetleg a vezérlésátadás megváltoztatása.

A tárkiírás és a trace viszonylag gyakran használt programtesztelési segédeszközök. Ezek segítségével a program leállítása után a program működése részletesen elemezhető. Az ezután következő három alpontban az ún. szimbolikus hibakeresők alapvető funkciói vannak felsorolva. Ezek az ellenőrzési módszerek a program futása közben alkalmazhatók. Kézenfekvő, hogy a program gyorsabban és alaposabban tesztelhető hibakereső segítségével. Még a legjobb hibakeresők használata esetén is fennáll azonban az a veszély, hogy a program tesztelése közben hibás működés következtében megsérül a robot, vagy annak munkaterében elhelyezett más értékes berendezés, sőt éppen a tesztelés következtében további veszélyeket jelentő faktorok jöhetnek létre. Erre mutat be egy esetet a *8.14. ábra.* Itt a robot már beállt az A-val jelölt pozícióba, majd a program további utasításai szerint kikerüli az előtte levő tárgyat a B, C, D, E útvonal mentén. Tegyük fel, hogy az E pozíció elérésekor programhibát észlelünk, és arra gyanakszunk, hogy a hiba a B pozíció elérése előtti programszakaszon van. Ha ekkor a programban a vezérlést visszaállítjuk a B pont elérése előtti utasításra, akkor a következő végrehajtásra kerülő utasítás az lesz, hogy „Állj a B pozícióra". A robotkar ilyenkor az E helyzetből közvetlenül B-re pozicionál, és eközben összeütközik az előtte álló lerögzített munkadarabbal. Ez az oka annak, hogy a robotvezérlő programok tesztelésekor a programot mindig célszerűbb elölről újraindítani, mint egyszerűen csak visszaállni egy korábbi utasításra.



8.14. ábra. A robotműködés tesztelési problémái egy tárgy kikerülésénél

Másrészről azt is látni kell, hogy ha minden egyes tesztelési lépésnél újraindítjuk a programot, akkor minden egyes alkalommal végigfutunk a már előzetesen részletesen ellenőrzött programrészeken is, ami – figyelembe véve a pozicionálások és egyéb műveletek időigényét – igen hosszadalmassá teheti a program tesztelését. A tesztelés időigényének lecsökkentése és a berendezések sértetlenségének biztosítása két olyan egymásnak ellentmondó követelmény, amely a robotberendezés közvetlen használata helyett szimulátorok alkalmazása mellett szól. A szimulátor olyan program, amely lehetővé teszi az összes robotművelet tesztelését a legkülönfélébb sebességek mellett, mégis sokkal gyorsabb, mint ahogy a robot a valóságban végrehajtaná a műveleteket. A szimulátorban lejátszódó folyamatok láthatóvá tétele érdekében grafikus megjelenítés is megvalósítható, ekkor a kezelő a robot működése közben megfigyelheti a robot helyzetét a fontosabb tárgyakhoz képest. A *8.1. ábrán* látható kapcsolatrendszer szerint a szimulátor felhasználja a környezetleíró modellt is. Ennek adataiból saját célra egy másik modellt állít elő, amely a fontosabb objektumok grafikus ábrázolásához szükséges információkat tartalmazza. Hogy mely objektumokat tekintünk fontosnak, azt vagy a compiler, vagy a felhasználó dönti el. Mindenképpen az ábrázolandó objektumok közé tartoznak a következők:

- a robotberendezés vagy berendezések;
- mindazok az objektumok, amelyek részt vesznek az egyes mozgásokban.

Ezek körét a compiler is meghatározhatja. Ha azonban a kezelő további objektumokat is meg akar jeleníttetni – pl. bizonyos akadályokat, rakodóberendezéseket stb. –, akkor ezeket külön is megadhatja.

Miután a szimulátor az inicializálási fázisban előállította a modellt, akkor ezt a modellt minden egyes robotműködtetéskor és a tárgyak közötti logikai kapcsolatokra vonatkozó minden egyes utasítás végrehajtásakor még módosítani is kell (pl. AL-ban az affix hozzárendelések elvégzésekor). Eközben mód nyílik arra, hogy ütközésellenőrzéseket hajtsunk végre, ill. a biztonsági távolságok betartását tesztelhessük. A szimuláció gyorsítása érdekében egy-egy kiválasztott programszakaszra le lehet tiltani a kép megjelenítését is. A szimulációt mind ez ideig főleg új robottípusok fejlesztésénél és a robotokkal kapcsolatos kutatásoknál alkalmazták (l. BROOKS [8.12]). Különféle szimulátorokat tartalmaznak az AUTOPASS (l. LIEBERMAN [8.7]), az AL karlsruhei egyetemen kifejlesztett változata (l. ERENZEL [8.13]), az ALFA (l. ZAMBUTO [8.14]) és a TL nevű programrendszerek (l. KUNO [8.15]).

A szimulátorok alkalmazása gazdasági megfontolásokkal is indokolható, hiszen egy robot ára ma körülbelül kétszerese az ilyenkor nélkülözhetetlen grafikus hardvernek, ami elfogadható minőségben kb. 50 ezer DM-ért már kapható. Ha szimulátort használunk, akkor egyrészt tehermentesíthető maga a robot, másrészt elkerülhető a robot megrongálódása, harmadsorban pedig a programtesztelés kevesebb időt igényel.

Ezzel a módszerrel szemben bizonyos mértékig új alternatívát kínálnak az új mikrorobotok. Ezek ára 10 ezer DM alatt van. A mikrorobotok ugyanakkor sokkal kisebbek és lényegesen kisebb tömeget képesek mozgatni, ezért a robot környezetét és a mozgatott objektumokat méretarányosan megfelelően le kell kicsinyítani. Szükség van ezenkívül egy ún. illesztő szoftverre, amely a működtető utasításait a mikrorobot működtetésére teszi alkalmassá. Mikrorobot alkalmazásával a tesztelés időigénye nem csökkenthető.

Általános felhasznált és ajánlott irodalom

[1] JENSEN, K.–WIRTH, N.: Pascal–User Manual and Report. New York, Springer-Verlag, 1975.

[2] MUJTABA, S.–GOLDMAN, R.: Al User's Manual. Stanford University, 1979.

[3] BLUME, C.: Al – ein textuelles Programiersystem für Industrieroboter. PDV-Berichte, KfK-PDV 187, Karlsruhe, 1980.

[4] BLUME, C.: Vorstellung der Sprache Al und die Besonderheiten der Implementierung an der Universität Karlsruhe. In: Verbesserte Programmierung von Robotern in der Montage. PFT-Entwicklungsnotiz, KfK–PFT–E2, 17–53. p. 1981.

[5] JAKOB, W.: Der Al-Compiler aus der Sicht des Benutzers. In: Verbesserte Programmierung von Robotern in der Montage. PFT-Entwicklungsnotiz, KfK–PFT – E2, 54–76. p. 1981.

[6] JAKOB, W.: Entwurf und Implementierung eines Compilers für eine höhere Manipulatorsprache zur Programmierung eines Roboters. Diplomarbeit. Universität Karlsruhe, 1980.

[7] BLUME, C.: A Structured Way of Implementing the High Level Programming Language Al on a Mini- and Microcomputer Configuration. Proc. of 11 th ISIR*, Tokyo, 1981.

[8] BLUME, C.: VAL – ein Roboterkontrollsystem von Unimation. Bearbeitete Übersetzung der VAL-Beschreibung, Universität Karlsruhe, 1980.

[9] User's Guide to VAL – A Robot Programming and Control System Version 12 Unimation Robotics Inc., Danbury, Conn., 1980.

[10] Guide to HELP-Language. Digital Electronic Automation (DEA), Turin.

[11] PRAGMA A 3000 – Operating and Programming Manual. Digital Electronic Automation (DEA) Turin.

[12] SALMON, R.: SIGLA – The Olivetti Sigma Robot Programming Language. 8th ISIR, Stuttgart, 1978.

[13] SIGMA/MTG – Handbuch und Programmierung. Release. A 10, Olivetti NC-Systeme GmbH.

[14] WECK, M.–ZÜHLKE, D.: Short Reference Manual of the ROBEX-Language. Technische Hochschule Aachen, 1980. In: Verbesserte Programmierung von Robotern in der Montage, PFT-Entwicklungsnotiz, KfK–PFT–E2, 95–107. p. 1981.

[15] WECK, M.–EVERSHEIM, W.–ZÜHLKE, D.: ROBEX – ein Programmiersystem für numerisch gesteuerte Handhabungsgeräte. In: Verbesserte Programmierung von Robotern in der Montage, PFT-Entwicklungsnotiz, KfK–PFT–E2, 77–94. p. 1981.

[16] RAIL Reference Manual. AUTOVISION. Burlington (USA). Automatix Inc., 1981.

[17] RAIL Software Reference Manual ROBOVISION and CYBERVISION. Burlington (USA), Automatix Inc. 1982.

[18] KAPPATSCH, A.–MITTENDORF, H.–RIEDER, P.: PEARL. München, Wien, Oldenbourg-Verlag, 1979.

* ISIR International Symposium on Industrial Robots

Speciális irodalom az egyes fejezetekhez

[2.1] COLE, A.: Macro Processors, Cambridge, 1976.

[2.2] ROHLFING, H.: SIMULA. Mannheim, Wien, Zürich. Bibliographisches Institut, 1973.

[2.3] Wettstein, P.: Aufbau und Struktur von Betriebssystemen, München, 1978.

[2.4] DENAVIT, J.–HARTENBERG, R.: A Kinematic Notation for Lower Pair Mechanisms Based on Matrices. Journal of Applied Mechanics Vol. 22, Trans. ASME Vol. 77, 1955.

[2.5] BLUME, C.: Sprachen und Programmiersysteme für Industrieroboter. Vorlesungsskript, Universität Karlsruhe, 1982.

[2.6] PAUL, R.: Robot Manipulators, Cambridge, The MIT Press, 1981.

[2.7] PAUL, R.: Modelling Trajectory Calcula-

tion and Servoing of a Computer Controlled Arm. Stanford Artificial Intelligence Lab., Stanford University, Stanford, Al Memo 177, 1972.

[2.8] BINDER, D.: Interpolation in numerischen Bahnsteuerungen. Berlin, Heidelberg, New York, Springer-Verlag, 1979.

[2.9] WECK, M.: Werkzeugmaschinen, Band 3, Düsseldorf, VDI-Verlag, 1979.

[2.10] LIEBERMAN, L.–WESLEY, M.: AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly. IBM J. Res. Dev. 21. 4, 1977.

[2.11] BERNORIO, M., et al.: Programming an Industrial Robot in Italian. 7th ISIR, Tokyo, 1977.

[2.12] MAROY, J.–BERTHOD, M.: Natural Language Understanding by a Robot: A Pattern Recognition Problem. Pattern Recognition, Vol. 10 (1978), 63–71. p.

[2.13] TAYLOR, R.: The synthesis of manipulator control programs from task-level specifications. Dissertation, Stanford University, Standford, 1976.

[2.14] BAUER, F.–GOODS, G.: Informatik. Teil 1 und Teil 2. Berlin, Springer-Verlag, 1971.

[2.15] HAHN, R.: Höhere Programmiersprachen im Vergleich. Wiesbaden, Akademische Verlagsgesellschaft, 1981.

[2.16] SCHNEIDER, H.: Problemorientierte Programmiersprachen. Stuttgart, Teubner-Verlag, 1981.

[2.17] HERSCHEL, R.: Einführung in die Theorie der Automaten, Sprachen und Algorithmen. München, Oldenbourg-Verlag, 1974.

[3.1] STOER, J.: Einführung in die numerische Mathematik, Berlin, Heidelberg, New York, Springer-Verlag, 1972.

[3.2] WILKINSON, J. H.: Rundungsfehler, Berlin, Heidelberg, New York, Springer-Verlag, 1969.

[4.1] PAUL, R.–LUH, J., et al.: Advanced Industrial Robot Control Systems. 4th Report, Purdue University, 1980.

[4.2] AMBLER, A.–BEATTIE, R.–CORNER, D.: RAPTI Users' Manual. University of Edinburgh, 1982.

[4.3] LUNDSTRÖM, G.–GLEMME, B.–ROOKS, B.: Industrial Robots-Gripper Review, International Fluidics Services Ltd., Oxford (England), 1977.

[4.4] AUER, H.: Beitrag zur Steigerung der Flexibilität von Handhabungseinrichtungen im Bereich der Einzel- und Kleinserienfertigung. Dissertation, TU Berlin, Berlin, 1977.

[4.5] LIEBERMAN, L., et al.: Three dimensional Modelling for Automated Mechanical Assembly. Research Report des IBM Watson Research Center, New York, 1979.

[4.6] DIJKSTRA, E. W.: Structured Programming. In: Software Engineering Techniques. Rept. on a Conference, Rome (27th–31th Oct. 1969). Brüssel, NATO-Science Committee, 1969.

[4.7] DAHL, O. J.–DIJKSTRA, E. W.–HOARE, C. A. R.: Structured Programming, New York; Academie Press. 1972.

[4.8] MILLER, E. F.–LINDAMOOD, G. E.: Structured Programming: Top Down Approach. In: Datamation 19 (1973), 55. p.

[4.9] SCHNUPP, P.: Systemprogrammierung. Berlin, New York, De Gruyter, 1975.

[4.10] SCHNUPP, P.–FLOYD, C.: Software, Programmentwicklung und Projektorganisation. Berlin, New York; De Gruyter, 1976.

[4.11] DENNING, P. J.: Is it not time to define „Structured Programming"? ACM Operating System Review 8, 1. 1974. 6. p.

[4.12] ZELKOWITZ, M. V.: It is not time to define „Structured Programming", ACM Operating System Review 8, 2. 1974, 7. p.

[4.13] FAULK, C. R.: Yet another attempt to define „Structured Programming", ACM Operating System Review, 8, 3. 1974. 7. p.

[4.14] DENNING, P. J.: Is „Structured Programming" any longer the right term? ACM Operating System Review 8, 4. 1974. 7. p.

[4.15] DIJKSTRA, E. W.: GOTO-Statement Considered Harmful. In communications, of the ACM 11 (1968). 147. p.

[4.16] NASSI, I.–SHNEIDERMANN, B.: Flow Chart Techniques for Structured Programming. In: Sigplan Notices 8. (1973). 12. p.

[8.1] SANDEWALL, E.: Programming in an Interactive Environment: The „LISP" Experience, Computing Surveys. Vol. 10. No. 1. 1978.

[8.2] BLUME, C.–DILLMANN, R.: Frei programmierbare Manipulatoren – Aufbau und Programmierung von Industrierobotern. Würzburg, Vogel-Buchverlag, 1981.

[8.3] AHO, A.–ULLMANN, J.: Principles of Compiler Design. Reading, Mass.: Addison-Wesley, 1977.

[8.4] AHO, A.–ULLMANN, J.: The Theorie of Parsing, Translation and Compiling. Englewood Cliffs, N. J.: Prentice Hall, Volume 1, Parsing, 1972 und Volume 2, Compiling, 1973.

[8.5] GRIES, D.: Compiler Construction for Digital Computers, New York, John Wiley and Sons, 1971.

[8.6] BACHMANN, P.: Grundlagen der Compilertechnik. München, 1975.

[8.7] LIEBERMAN, L.: Model Driven Vision for Industrial Automation. Research Report des IBM Watson Research Center, New York, 1978.

[8.8] LOZANO-PEREZ, T.–WINSTON, P.: LAMA: Language for Automatic Mechanical Assembly. Proc. 5th International Joint Conference on Artifical Intelligence, Boston, 1977.

[8.9] DENKER, P.: Ein neues LALR-System. Diplomarbeit, Universität Karlsruhe, 1977.

[8.10] ELSPAS, B.–LEVITT, D.–WALDINGER, R.–WAKSMAN, A.: An Assessment of Techniques for Proving Program Correctness. In ACM Computing Surveys 4. 1972. 87. p.

[8.11] Unimation: Unimate 600 PUMA. Danburry, Connecticut, 1980.

[8.12] BROOCKS, R.–GREINER, R.–BINFORD, T.: The Acronym Model-Based Vision System. Proc. of the 6th International Joint Conference on Artifical Intelligence, Tokyo, 1979, Vol. 1., 105–113. p.

[8.13] FRENZEL, A.: Entwurf unf Implementierung eines Robotersimulationssystems. Diplomarbeit, Universität Karlsruhe, 1981.

[8.14] ZAMBUTO, D.–CHANEY, J.: An Industrial Robot with Minicomputer Control. 6th ISIR. Nottingham, 1976.

[8.15] KUNO, T., et al.: Robot Performance Simulator, 9th ISIR, Washington, 1979.

Függelék: Az AL nyelv alapdefiníciói

Az eredeti német mű egy kilenc alfejezetből álló terjedelmes függeléket tartalmaz. Ez az anyag nagyrészt szerepel a könyv előző nyolc – érdemi – fejezetében is, ill. az egyes ismertetett speciális nyelvek szintaxisának részletkérdéseivel foglalkozik.

Minthogy a könyv semmiképpen sem tekinthető egy programozási nyelvkönyvnek, az sokkal inkább elvi bevezetést nyújt a robotprogramozás speciális kérdéseihez, ezért mutatkozott célszerűnek, hogy a magyar kiadásban ezt a függeléket elhagyjuk. Kivételt kellett azonban tennünk egyetlen függelék-alfejezettel, mégpedig azzal, amelyik az AL nyelv alapdefinícióit tartalmazza, mert ebben olyan fogalmak, jelölések jelentését, ill. értelmezését találja az Olvasó, melyeknek magyarázatát a szövegből egyébként nem kaphatja meg annak ellenére, hogy ott előfordulnak. Ezért ezt a függelékrészt a következőkben teljes terjedelemben közöljük.

Az AL jelkészlete

Az itt bemutatott jelkészlet a karlsruhei egyetemen implementált AL-változatban használt jelkészlet. Az amerikai AL-változattal szemben kismértékű eltérés tapasztalható. Az &, az @ és a # jelek kivételével a többi karakter megvan az amerikai AL-változatban is.
Betűk: A...Z, a...z.
Számjegyek: 0...9
Különleges jelek: (,), [,], {, }, |, *, /, +, −, ∧, <, >, =, :, „ ;, &, @, _, #, ”.

A lefoglalt AL alapszavak a következők:

ABORT	CENTER	DIV	FOR
ABOUT	CLOSE	DO	FORCE
AFFIX	CLOSE_FILES	DRIVE	FORCE_FRAME
ALL	COBEGIN	DURATION	FRAME
ALONG	COEND		FROM
ALWAYS	COMMENT	EC	
AND	COPY	EDIT	HAND
APPROACH		ELSE	
ARRAY	DEBUG	END	IF
AT	DEFINE	EQV	IN
	DELETE	EVENT	INTO
BEGIN	DEPARTURE	EXIT	INV
BY	DEPROACH		
	DIMENSION	FCONSTRUCT	JOINT
CASE	DISPLAY	FF	

LF

MAX
MIN
MOD
MOVE

NILDEPROACH
NO_NULLING
NONRIGIDLY
NOT
NOTE
NULLING

OF
ON
OPEN
OR

PARK
PARKALL
PAUSE
PCONT
PEND
PF
PHALT
PRINT
PROCEDURE
PROMPT
PSTART

QDELETE
QREAD
QWRITE

READ
REFERENCE
REMOVE
REQ
RESET
RETURN
RF
RIGIDLY
ROT
SCALAR
SENSOR
SF
SIGNAL
STEP
STOP
SUBTREE

TEACH
THEN
TO
TORQUE
TRANS

UNFIX
UNTIL
VALUE
VECTOR
VELOCITY
VIA

WAIT
WHEN
WHERE
WHILE
WITH
WOBBLE
WORLD
WRITE

WRT

XOR

Különleges jelentésű karakterek és szimbólumok

& a Condition (feltétel) Monitornak szóló önmagában álló utasítást zár le
@ a kar frame-rendszere, kizárólag MOVE-utasításban használható
; utasítások elválasztására szolgál
, listaszeparátor jel
: tömbök elhatárolására szolgáló szeparátor
( ) eljárásutasításokban, függvényeljárásoknál és képletekben használható zárójelek
[ ] tömbindexek írására használt zárójelek
(**) a commentek (megjegyzések) megjelölésére szolgáló zárójelezés
= összehasonlítási műveletben egyenlő
< összehasonlítási műveletben kisebb
> összehasonlítási műveletben nagyobb
< = összehasonlítási műveletben kisebb egyenlő
> = összehasonlítási műveletben nagyobb egyenlő
< > összehasonlítási műveletben nem egyenlő
* aritmetikai műveleti jel, szorzás
/ aritmetikai műveleti jel, osztás
+ aritmetikai műveleti jel, összeadás
− aritmetikai műveleti jel, kivonás
· aritmetikai műveleti jel, vektorok skaláris szorzása
∧ aritmetikai műveleti jel, hatványozás
| abszolútérték-képzés
→ aritmetikai műveleti jel, relatív transzformáció

– > aritmetikai műveleti jel, relatív transzformáció (karlsruhei változat)
← értékadás
< – értékadás (karlsruhei változat)
? kérdés közlése a REQ-utasításban
makrodefiníció elhatároló jele
” szöveg határoló jele

Előre definiált dimenziók:

ANGLE
ANGULAR VELOCITY
DIMENSIONLESS
DISTANCE
FORCE
TIME
TORQUE
VELOCITY

Előre definiált konstansok:

CM
CRLF
DEG
FALSE
GM
NILROT
NILTRANS
NILVECT
PARKPOS
PARKPOS1
PARKPOS2
PARKPOS3
PARKPOS4
PARKPOS5
PARKPOS6
PARKPOS7
PI
SEC
STATION
TRUE
XHAT
YHAT
ZHAT

Előre definiált változók:

ARM
ARM1
ARM2
ARM3
ARM4
ARM5
ARM6
ARM7
HAND
HAND1
HAND2
HAND3
HAND4
HAND5
HAND6
HAND7

Tárgymutató

A, Á

ábrázolási hossz bájtokban, standard alaptípusé 93
– tartomány, standard alaptípusé 93
abszolút cím 21
abszolútérték 116
abszolút szög- és távolságadatok esetén DRIVE-utasítás 138
adatbeviteli eljárások 121
adatbevitel párbeszédes technikával (interaktív módszer) 120
adategyüttesek 98
adat nevű változó 20
adatobjektum 83
adatstruktúra 17
adattömbök 97
adott gyorsulású mozgás végrehajtása hipotextikus nyelven írt programmal 37
a értéknevű változó 20
affixedon változó 128
affixframe pointer 128, 234
affix hozzárendelés fa-struktúrája 233
– – struktúra ábrázolása pointerezett lista segítségével 233
– -kapcsolat 106
– -relációk ábrázolása 107
affixrel transzformáció típusú változó 128
affixrendel eljárás 232
affixtömb 230, 234
AFFIX-utasítások 131
aktív állapot, folyamaté 41
alakfelismerés 183
alapszavak, AL nyelvé 265
alapszimbólum ábrázolása 49
algoritmus 18
AL adattípusainak deklarálása PASCAL nyelvben 99
– -compiler felépítése 246
— üzemmódjai 248
ALGOL programnyelv 84
AL jelkészlete 265
állandó gyorsulás, robotkaré 39
AL nyelvben definiálható vibrációs mozgás 152
AL nyelv CENTER-utasítása 175
– – DRIVE-utasítása 138
AL nyelv geometriai típusú változókra értelmezett standard függvényei 116
– – mozgásvezérlő utasításai 144
– – utasítása relatív elmozdulás végrehajtására 142
– programnyelv 5, 77, 83
alapprogram 30, 220
– definíciója 87
– HELP nyelven írva 222
– hívása VAL nyelvben 220
– hívási módjai HELP nyelvben 221
– (szubrutinok) 90
– több belépési ponttal, HELP nyelven írva 222
alternatívák ábrázolása 49
approach pointer 128
APT NC-gépek vezérlésének nyelve 78
aritmetikai műveletek 108
– standard függvények 115
automatikus típusdeklaráció 85
AUTOPASS programnyelv 77
azonosítóként használható karaktersorozatok hossza 84

B

Backus–Naur forma 47
beállítási eljárás 16
BEAM (téglatest) 97
beolvasás, skalár és logikai típusú értéké 116
beolvasott pozíció és orientációadatok kezelése a HELP nyelvben 123
betanítási eljárás kialakítása frame-lista segítségével AL és VAL nyelvek esetén 216
– (teach-in) eljárás 16
betanított rendszer működése 251
betolás (push) 22
beviteli utasítások 120
blokkok és utasítássorozatok PASCAL nyelvben 189
blokkokkal definiált vektor és rotációs mátrix 95
blokkszervezés 26
– AL nyelvben 188
– HELP nyelvben 189
BNF 47
– ábrázolása szintoxisdiagramokkal 49
BODY (elemekből összetett test) 97
BOOLEAN 92

box nevű frame koordináta-rendszer geometriai értelmezése 54

C

calibkamera frame 185
calibrobot frame 184
call by name 33
-- reference 33
-- value 33
CASE-utasítás AL nyelvben 200
— PASCAL nyelvben 199
célframe 234
CHARACTER 92
- adattípus 85
ciklusszervezés 201
- ciklusváltozóval 203
ciklusváltozó 202
címdeklaráció 87
címkék 90
címkézett utasítás HELP nyelvben 193
-- PASCAL nyelvben 193
-- ROBEX nyelvben 193
-- VAL nyelvben 193
CIRCLE (kör) 97
CLOSE 102
CO-blokk 235
compiler 20, 243
- által előállított output fájlok 247
- -program 243
Continuous Path (CP) vezérlés 71
Controlled Path vezérlés 71
CONE (körkúp) 97
CP vezérlés 71
CREATE 102
CYLINDER (henger) 97

CS

csukló elfordulási szögértéke, Θ 58
–, roboté 14

D

dead-lock helyzet 206
deklaráció 19
Denavit-Hartenberg-mátrix 55
departure pointer 128
Descartes koordináta-rendszerben végrehajtott lineáris interpoláció AL nyelven írt eljárásának ismertetése 110
DH-mátrix 55
dhm-mátrix 97
dinamikus blokk 188
- modell 14
- tárkezelés 27
- változók 102
direkt rekurzív hívás 35
DO ciklusutasítás AL nyelvben 205
DONAU rendszer 81

E, É

editor 242
effektor nyitása és zárása ROBEX nyelvben 171
effektorvezérlő utasítások 169
-- AL nyelvben 172
-- eseményfelügyelettel 174
egyenlőség 115
egyetlen „formaszekrénytyp" típusú elemből álló szekrény listája 103
egy frame rendszer transzformációja 111
egyszerű effektorvezérlő utasítások 169
- mozgásvezérlő utasítások 136
--- SIGLA-ban 138
együttfutó rutinok 239
ekvivalencia (egyenlőség) 115
elemi funkcionális blokk 190
élettartam, változóké 26
elhagyási frame 150
- ill. megközelítési frame definíciója 152
eljárás definiálása AL nyelvben 32
-- hipotetikus nyelvben 32
- definíciója 87, 90
eljárásdeklaráció AL nyelvben 224
- PASCAL nyelvben 223
eljáráshívás PASCAL nyelvben 223
- SIGLA nyelvben 225
ellenőrző üzemmód, AL-compileré 248
elmozdulás 14
előre definiált dimenziók 267
erő- és nyomatékparaméterek megadása AL nyelvben 159
erőmérő cella frame-rendszerének megadása AL nyelvben 154
-- jelének határérték-beállítása HELP nyelvben 157
értékadás 119
értékadási utasítások írásmódjai 119
értékadás pointer típusú változónak 124
érték szerinti paraméterátadás 33
értelmező (interpreter) 243
érvényességi tartomány, változóé 26
érzékelő 14
- adatának beolvasása SIGLA nyelvben 181
- jelének fogadása SIGLA nyelvben 158
- jelétől függő feltételes AL-utasítások 182
---- utasítás alakja AL nyelvben 177
----- HELP nyelvben 179
érzékelőkkel ellenőrzött mozgásvezérlő utasítás értelmezése AL nyelvben 155
érzékelő specifikációja AL nyelvben 179
érzékelővezérelt keresőmozgás felírása SIGLA-ban 158
eseményfelügyelettel szervezett mozgatás ROBEX-ben 164

ÉS, logikai 115
EVENT 92
explicit mozgásvezérlő utasítások 135
- - - ROBEX-ben 140
- pályavezérlés támogatására bevezetett geometriai adattípusok 94
exponenciális függvény 116
exp változó 28
ezegyblokk nevű eljárás 188

F

fájlkezelő rendszer funkciói 101
fájlok (file) 100
fa-struktúra 23
fékberendezés szerelési utasítás AUTOPASS nyelven 80
feldolgozási folyamat 40
- - (task) 18
fejlesztés, eljárásé 40
feltételes utasítás 195
- vezérlésátadás 194
- - SIGLA nyelvben 195
- - VAL nyelvben 195
feltételtől függő ciklus 204
- - - szervezése 205
feltétlen vezérlésátadás HELP nyelvben 193
- - PASCAL nyelvben 193
- - ROBEX nyelvben 193
- - SIGLA nyelvben 193
felvevőkamera derékszögű koordináta-rendszer 184
- koordináta-rendszerének meghatározása a robot saját koordináta-rendszerében 185
file (fájl) 100
folyamatállapotok 41
FOR-ciklus AL nyelvben 203
— PASCAL nyelvben 203
- -ciklusutasítás AL és HELP nyelvben 203
fordítóprogram (compiler) 20
fordított lengyel jelölés 23
formaszekrények elrendezése munkaasztalon 106
formaszekrény megfogószerkezettel 99
formaszekrénytípus deklarálása PASCAL-ban 99
„formaszekrénytyp" típusú pointerezett lista 103
FORTRAN programnyelv 84
főblokk főprogram 188
FRAME 94
frame-eknek alaphelyzet szerinti értelmezése egy hattengelyű robot esetén 66
frame-fájl deklarálása PASCAL nyelvben 101
- (kísérő) koordináta-rendszer 52
- -lista előállításához használható interaktív segédprogramok 252
- -rendszer lineáris eltolására alkalmas VAL-utasítások 113
frame-fájl-szerkesztő programok rendszerkapcsolatai 243
- -távolság 31
- típusú koordináta 22
frametömb 234
frame-változó segítségével felírt mozgásvezérlő utasítás AL-ban 141
fúrási művelet előírt nyomóerővel AL nyelvben 161
futtatásvezérlő rendszer 214, 241, 251
futtató rendszer alprogram meghívásakor 31
függvényeljárás deklarációja 87, 90
- típusának deklarása PASCAL nyelvben 226

G, Gy

geometriai adatok NC-programozáshoz 82
- adattípusok 94
- - deklarálása AL nyelvben 95
- - - PASCAL nyelvben 95
- műveletek 110
- - AL nyelvben 111
globális változó 26
gyorsulás, robotkaré 39

H

határértékfigyelés 182
HELP fájlkezelő utasításai 102
- nyelven írt alprogramok 222
- nyelv mozgásvezérlő utasításai 144
- programnyelv 5, 84
helyzet beállítása explicit koordináták megadásakor 140
hibajelzés nevű változó 234
hivatkozás szerinti paraméterátadás 33
- - - SIGLA nyelvben 225
homogén transzformáció 55
humanoid 15

I

IBM AUTOPASS rendszer 79
időben egymást követő sorrend 18
időraszteres interpoláció 70
időzítési feltétel figyelése AL nyelvben 165
IF-utasítás szerkezete 196
implicit mozgásvezérlő utasítások 134
- típusdeklaráció 86
INDA programnyelv 77
indexelt fájl 100
indirekt rekurzív hívás 35
init blokk 188
INTEGE adattípus 85
INTEGER 92
interaktív összetevő 250
- üzemmód, AL-compileré 248
interpreter (értelmező) 243, 254
- programkomponensei 254

interrupt 18
invcalibkamera frame 188
ipari robot 13
ismétlés ábrázolása 49
–, mint opcinális lehetőség ábrázolása 49
ízületi koordináták 52
– koordináta-rendszer 52

J

japán kisméretű robot 71
jel fogalma 17
jelkészlet 17
jelsorozat ábrázolása 49
jelzők (szemaforok) 42

K

kamat nevű változó 28
kamerarendszer nevű frame 186
kapcsolt adatmezők 98
karakterek 266
karakterisztikus alakfelismerő vektorok 183
kassza nevű változó 28
kép felvételét és feldolgozását vezérlő RAIL-utasítások 186
keresframepointer 230
kereső 230
– eljárás 230
– – rekurzív hívása 233
– – – – az affix-hozzárendeléseket kezelő AL nyelvű programban 231
keret (frame) 31
készenléti állapot, folyamaté 41
kétállapotú jelek lekérdezésének technikája a VAL nyelvben programelágaztatással 179
– jeltől függő programoztatás ROBEX nyelvben 181
két folyamat szinkronizálása 43
– rotáció láncolása 111
– transzformáció láncolása 111
kezdő szimbólum 45
kézi programozás 16
kéz koordináták 52
kifejtési szabályok ábrázolása 49
kihúzás (pop) 22
kinematika 14
kisérő (frame) koordináta-rendszer 52
kiszámított GOTO utasítás HELP nyelvben 200
kivételes állapot kezelésének lehetőségei PEARL nyelvben 211
– helyzet 209
– üzemállapotok 209
kizáró VAGY, logikai 115
kombinált on-line és off-line program, roboté 214
komplex kifejezések 117
konstans értékadás 119
konstansnevek deklarálása 91
konstansok deklarálása 87
koordinátatranszformáció 57
koordinátatengely körüli rotáció értelmezése 56
koszinusz függvény jellegű mozgás (lassulás, gyorsulás), robotkaré 39
környezeti modell 81, 127
– – definiálása és a végzett műveletek PASCAL nyelven írt program segítségével 130
– modellre vonatkozó utasítások AL-ban 132
környezetleíró modell 102
követőprogramozás 16
közbenső frame-helyzetekkel megadott pálya az AL nyelvben 149
– — – – egybesimítása VAL nyelvben 149
közvetlen rekurzív eljáráshívás 35
közvetett rekurzív hívás 35
kulcsos hozzáférésű fájl 100
különféle frame koordináta-rendszerek egymáshoz viszonyított helyzete 54
különleges jelentésű karakterek és szimbólumok 266
külső megszakításkérelemre végrehajtott rutinhívás deklarációja VAL nyelvben 162

L, LY

lassulás, robotkaré 39
látszólagos időbeli párhuzamosság 18
lebegő tizedesvesszős ábrázolás 92
lefoglalt AL alapszavak 265
lineárisan változó mozgás, robotkaré 39
lineáris interpoláció *Descartes* koordinátákban 112
– – pályavezérlésnél 74
– tengelyinterpoláció 74
LINE (egyenes) 97
logaritmus 116
logikai műveletek 114
lokális változó 26
lyuk nevű frame-rendszer 128

M

makró 30
makrodefiníció 34
makrodeklaráció 87, 90
makrogenerátor 34
makró hívása 34
– név 34
makrotörzs 34
MAL programnyelv 77
Master-slave-programozás 16
megengedett típusdeklarációk 87
megfogási pont 52
megfogószerkezet 168
– előzetes beállítása ROBEX-ben 171

megfogószerkezet érzékelővezérelt működtetése VAL nyelvben 173
– geometriai viszonyai 57
– jellemző adatai 59
– nyitása és zárása VAL nyelvben 169
– pozíciója 54
– – alkatrész átadásakor 236
– rendszerének *z* irányába eső erőkomponens ellenőrzése AL nyelvben 156
megközelítési frame 150
–, ill. elhagyási frame definíciója 152
megszakításkezelő rutinok 87, 90
megszakítások kezelése 18
mértékegységek deklarálása 87, 88
minimális biztonsági távolság mozgatáskor 100
MOVE alapszó 133
– -utasítás 134
mozgásprogramozás eljárásai 16
mozgásvezérlő utasítás 133, 143
– – AL nyelvben eseményfelügyelet mellett 161
– – – – frame-rendszer explicit megadásakor 140
– – eseményfelügyelettel VAL nyelvben 164
– – HELP nyelvben 140
munkatér, roboté 14
működési tartomány 15
műveletek 108
– prioritási sorrendje 118

N

nagyszámítógépes programozási nyelvek 76
NC-rendszerek 78
négyzetgyökvonás 116
négyzetreemelés 116
NEM, logikai 115
nemterminális szimbólum 45
– szimbólum ábrázolása 49
név szerinti paraméterátadás 33
névtelen (Standford Research Institute) programnyelv 77
NONRIGIDLY alapszó 133
Numerical Control (NC) 78

NY

nyelvben használt jelek 45
nyelvtan, nyelvé 44
nyomatékparaméterek megadása AL nyelvben 159
nyugalmi állapot, folyamaté 41
– helyzet beállítása AL utasítással 167
– – – VAL-utasítással 167

O, Ö

off-line program, roboté 214
on-line program, roboté 214
OPEN 102
operációs rendszer 18
operandusverem 23
opcionális lehetőségek ábrázolása 49
orientáció megadása VAL nyelvben 136
– meghatározása a forgatás (csavarás), billentés és elfordítás szögeivel 136
összehasonlítási műveletek 114

P

pálya simítása HELP nyelvben 149
pályatervezés 57
pályavezérlés 71, 255
pályavezérlésre bevezetett adattípusok 94
paralel blokk 235
paraméterátadási módszerek 32
paraméteres effektorvezérlő utasítások 171
párhuzamos blokk használata AL nyelvben 236
PART (testekből összeállított alkatrész) 97
PASCAL nyelvből vett példa deklaráció leírására 89
– programnyelv 11, 83
PATTERN (ponthalmaz) 97
példaprogram NC-gépek vezérlésére alkalmas APT nyelven 78
perifériás készülékek ki/be kapcsolása SIGLA nyelvben 171
P frame-rendszer orientációjának kiszámítása 62
PLANE (sík felület) 97
POINT (pont) 97
– to Point vezérlés 71
pointer-típus 102
pontvezérlés 71
pop művelet 22
pozicionálás-ellenőrző utasítás általános struktúrája AL nyelvben 210
– időzítési feltételének explicit ellenőrzése HELP nyelvben 166
processzor 243
progmatika, nyelvé 16
programelágaztatás 192, 198
– mikrokapcsoló lekérdezésekor SIGLA nyelvben 181
– szervezése 199
programfejlesztő és a futtatásvezérlő rendszerek együttműködése a VAL nyelvnél 253
– – – – elemeinek kapcsolatai a magasabb szintű programnyelveknél 242
programmegszakítás 18
programozási nyelvek előnyei 11
program pointer típusú változók alkalmazására PASCAL-ban 104
– tesztelésének segédeszközei 257
PTP-vezérlés 71

PUMA 600-as robot vezérlő hardver-rendszere 256
- nevű robot 146
push művelet 22

R

REAL 92
- INTEGER típusátalakítás 116
- típusú számokkal való ábrázolás 92
RECORD 102
rekordok 98
rekurzív eljáráshívás 35, 230
- tulajdonságú szabály 46
relatív cím 20
- frame-rendszer geometriai viszonyai 184
- szög- és távolságadatok esetén DRIVE-utasítás 138
rendszerkapcsolatok 208
rendszerkapcsolókat beállító utasítások VAL nyelvben 208
REPEAT-ciklusutasítás PASCAL nyelvben 205
RETURN utasítás AL nyelvben 226
— VAL nyelvben 220
RIGIDLY alapszó 133
ROBEX néhány geometriai adattípusa 96
- nyelv 5, 84
- - DRIVE utasítása 138
- - mozgásvezérlő utasításai 144
- - utasítása relatív elmozdulás végrehajtására 143
- programfejlesztő rendszer felépítése 245
robotberendezés frame-adatainak beolvasása HELP-ben 123
robot frame-transzformációjának paraméterértékei 67
- futtatásvezérlő programrendszereinek elemei 252
robotfüggő programozási nyelvek 77
robotkar alaphelyzetének definiálása 59
- mozgását leíró diagramok 39
robot kombinált on-line és off-line programozása 214
robotkoordináták 52, 184
robot mozgása közben végrehajtott eseményfigyelés HELP nyelvben 164
robotmozgás megállapítása AL nyelvben 175
- - HELP nyelvben 176
robotműködés tesztelési problémái egy tárgy kikerülésénél 258
robot nyugalmi helyzetének definiálása ROBEX nyelvben 167
- pillanatnyi pozíció- és orientáció-adatainak beolvasása 122
- pozicionáló-mozgásvezérlő programcsomag elemei 255
- - tengelyei 72
robotprogramozás kivételes állapotai 211
robot vezérlési módjai 75
robotvezérlő programozási nyelvek 76
ROT 94
rotációs mozgás 15
rotmatrix rotációs mátrix 94

S

segédszimbólum 45
SIGLA programnyelv 5, 84
- - mozgásvezérlő utasításai 144
skaláris szorzat 111
skalárral szorzás 111
soros fájl 100
sorozatos rekurzív eljáráshívások feldolgozása 38
SPHERE (gömb) 97
standard alaptípusok 92
- - ábrázolási hosszai és ábrázolási tartományai 93
- - és alapszavaik 93
- függvények 115
- üzemmód, AL-compileré 248
statikus blokk 188
struktogram 190
strukturálatlan folyamatábra 191
strukturált programozás 189

SZ

szabadságfok 14
szakasztípus deklarálása PASCAL nyelvben 89
számlálótípusú változók 94
szegmentálási mélység 21
szekvenciális hozzáférésű fájl 100
szemaforok (jelzők) 42
szemaforral vezérelt szinkronizálás 43
szemantika, nyelvé 16
szerszám előzetes beállítása ROBEX-ben 171
- szorítóerejének megadása és szabályozása AL nyelvben 172
szimbólum 17, 266
szimbólumtáblázat 20
szinkronizációs utasítások 204
szinkronizálás 41
szinkronizáló változó 41
- - val vezérelt szinkronizálás 43
szinkronizált feldolgozás 41
- hozzáférés alkatrészraktárhoz 43
szintaktikai elemzés 244
szintaxisdiagram 47
szintaxisfa 47
szintaxis, nyelvé 16
szövegfájl 101
szöveghelyettesítés 34
szövegszerkesztő 242
- programok rendszerkapcsolatai 243
szövegszerű programozás 213
szubrutin 30, 90

T

tárolótartalom 20
tárolóterület 20
társrutin 41, 239
taskok 18
- aktivizálása HELP nyelvben 239
- deklarációja 87, 90
- - HELP nyelvben 239
- kezelése 237
- leállítása HELP nyelvben 239
- legfontosabb jellemzői 237
- lehetséges állapotainak diagramja 238
TCP 52
teach-in eljárás 16
tengelyinterpoláció 74
természetes alapú logaritmus 116
terminális szimbólum 45
tesztmódus 249
Θ_1 szög számítása 59
Θ_2 és Θ_3 szögek meghatározása 60
típusdeklarációk 85, 87
típusillesztés 108
top-down tervezés 189
TRANS 94
transzlációs mozgás 15
transzformáció szerkesztése 111
tűzhely nevű változó 28

U, Ü

ugrási címek 90
ugróutasítás 90
UNFIX-utasítások 131
útinkrementumok 70
útraszteres interpoláció 70
üres kifejtési lehetőség ábrázolása 49

V

VAGY, logikai 115
VAL programnyelv 5, 83
- - mozgásvezérlő utasításai 144
- - utasítása relatív elmozdulás végrehajtására 142
VAL programnyelvben definiálható vibrációs mozgás 152
valós számokkal való számolás 92
választás két alternatíva közül 196
változódeklaráció 87
változók 19
- élettartama 26
- érvényességi tartománya 26
- helyfoglalása a tárban 28
- típusdeklarációjának módjai 87
várakozási állapot, folyamaté 41
- sor 42
- utasítások 207
vasúti szerelvényt összeállító nyelv szintaxis-diagramjai 50
VECTOR 94
végrehajtó szervek 14
vektoriális szorzat 111
vektorok hosszának előállítása 111
- összeadása, kivonása 111
- rotációja 111
- zsugorítása 111
vektorszumma függvényeljárás AL és PASCAL nyelvben 227
vektortípus deklarálása PASCAL nyelvben 89
vektortranszformáció 111
verem elv szerinti tárkezelés 30
veremfeldolgozás 30
verem szervezése AL nyelvben 229
- - PASCAL nyelvben 228
veremtárolás 22
vezérlés folytatása a címkétől SIGLA nyelvben 193
vezérlési módok pályagörbéinek összehasonlítása 75
VIAMOVE utasítások 35
- - sebesség – út diagramja 38
vizuális berendezés kezelése IIV változatban 183
- rendszer 183
V-művelet 42

W

WHILE-ciklusutasítás HELP nyelvben 205
— PASCAL nyelvben 205
WAIT-utasítás 42