# tanulmányok

**166/1985**

MTA Számítástechnikai és Automatizálási Kutató Intézet Budapest

MAGYAR TUDOMÁNYOS AKADÉMIA
SZÁMITÁSTECHNIKAI ÉS AUTOMATIZÁLÁSI KUTATÓ INTÉZETE

# INFORMÁCIÓS RENDSZEREK SZÁMITÓGÉPES TERVEZÉSE

Irta

*Radó Péter*

Tanulmányok 166/1985

A kiadásért felelős:

*DR VÁMOS TIBOR*
igazgató

Főosztályvezető:

*DEMETROVICS JÁNOS*

ISBN 963 311 187 O
ISSN O324-2951

Hozott anyagról sokszorosítva

8515606 MTA Sokszorosító, Budapest. F. v.: dr. Héczey Lászlóné

Tartalomjegyzék

# 0. BEVEZETÉS

Nyilvánvaló, jól követhető tendencia ugy Magyarországon, mint az egész világon, a szoftver előállitására költött összegek dinamikus növekedése /ld. pl. [1]/. Ez a hatalmas tőkebefektetés nemcsak mennyiségi, hanem minőségi változásokhoz is vezet.

Nem csupán arról van szó ugyanis, hogy egyre több különböző számitógéphez, növekvő számu alkalmazáshoz kell szoftvert gyártani, Maguk a rendszerek is egyre bonyolultabbak lesznek, a velük szemben támasztott követelmények egyre magasabbak. Ennek a folyamatnak kedvez a hardver - még a szoftvernél is látványosabb - fejlődése /ld. pl. [2]/, mely lehetővé teszi nagy rendszerek létrehozását és üzemeltetését.

A szoftvergyártás technológiája korunkban a számitógéptudomány önálló ágának számit /software engineering/. B. Boehm [1]-ben közölt meghatározása szerint ez tudományos ismereteink gyakorlati alkalmazása számitógépprogramok tervezésére, előállitására és a programok fejlesztéséhez, müködtetéséhez, karbantartásához szükséges dokumentáció előállitására.

A jelen dolgozatban olyan számitógépes rendszerekkel kivánok foglalkozni, melyek feladata a fenti definicióban foglalt tevékenységek segitése. A szoftverkészitás igen széles területén belül elsősorban információs rendszerek készitésének problémáira, ill. az ezek megoldását segitő számitógépes rendszerekre helyezzük a fő hangsulyt.

Vizsgáljunk meg néhányat azok közül a problémák közül, melyek megoldása, vagy meg nem oldása döntő jelentőségü lehet egy-egy projekt szempontjából. Megjegyezzük, hogy példáink főként a rendszer életének

korai szakaszaiból, a felmérés, tervezés stádiumából valók. Ez nem véletlen: ezek a korai munkafázisok különösen fontosak. Ennek indoklásául [1] két megállapitására hivatkozunk:

- valami alapos elvégzését könnyü elodázni, vagy teljesen elkerülni;
- az ily módon elkövetett hibák kijavitása később igen nehéz és költséges.

/Később pontositani fogjuk a rendszer időbeni fejlődéséhez kapcsolódó fogalmakat, külön kiemelve a minket érdeklő periódusokat./

Kezdjük az alapprobléma egy frappáns megfogalmazásával /[3] nyomán/: a nagy rendszerek készitőinek figyelmét sokszor elkerüli az a nyilvánvaló tény, hogy a megfogalmazatlanul maradt kérdés nem tekinthető megválaszoltnak. Arról van szó ugyanis, hogy a hiányos felmérés, a következetlen tervezés, a pontatlan dokumentáció lényeges problémákat elsikkaszt, ahelyett, hogy felhivná rájuk a figyelmet, ill. megoldaná őket.

Lényegében az ilyen hibák okairól ir [4] is, az információs rendszerek tervezőinek szemszögéből vizsgálva a helyzetet. Szerinte a nehézségek a következőkből adódnak:

- Nehéz megismerni a /számitógépen modellezendő/ rendszert. A felhasználók jól ismerik ugyan, de más dolog jól ismerni valamit, és megint más elmagyarázni annak tevékenységét, vagy belső müködését, a lényeges részletek kiemelésével. /Ezért is gyakori a "nagyszerü rendszert készitettünk, de a felhasználónak nem kellett" jellegü panaszkodás./

- Az információáramlás forditott utját /a rendszer készitőjétől a felhasználó felé/ rögössé teszi a felhasználók elég általános számitástechnikai képzetlensége. /Feltehetően ez Magyarországon még sulyosabb gond, mint az USA-ban./
- A tul sok részlet nehezen áttekinthetővé teszi az összképet. Másfelől persze, a tisztázatlan részletek a rendszer egészét használhatatlanná tehetik.
- A rendszerterv általában többszáz oldalas, nehezen olvasható mü, nem alkalmas arra, hogy annak alapján a felhasználó elképzelje magának jövendő rendszerét.
- Ha a rendszerterv a felhasználó számára érthető, valószinüleg hiányosnak tünik majd az azt realizáló programozó számára. Általában még egy rendszertervet kell készitени, mely az implementáció fizikai részleteit tartalmazza. Ez dupla munka, és a két rendszerterv közötti eltérések további problémákat okozhatnak.

Sajnos ezzel nem végeztünk a nehézségek felsorolásával /[4] sem állitja, csupán ezeket emeli ki/. Mutatóba még néhány - tapasztalataink szerint az előzőekhez hasonlóan sulyos - probléma:

- A modellezendő rendszer maga is folyton változik. Ez nemcsak azt jelenti, hogy a számitógépes programoknak elég rugalmasnak és könnyen változtathatóknak kell lenniük, hanem azt is, hogy a rendszerszervezőnek fel kell

készülnie arra, hogy a rendszertervben magában kell változtatnia. /Nem speciálisan magyar probléma ez sem![5]is foglalkozik vele./

- Nagyobb rendszereket nem egy személy tervez - ez fizikai képtelenség volna -,hanem több tagból álló team. A tervező csoport tagjai felosztják valamilyen módon a feladatot maguk között, és ki-ki csak a saját részével foglalkozik. A részrendszerek összehangolása nehéz feladat, párhuzamos tevékenységekhez, sok hibához, pontatlansághoz vezethet. A több tagból álló csoportok esetén az összes többi probléma is sokkal élesebben vetődik fel.

- A számitógépes rendszer karbantarthatósága érdekében azt dokumentálni kell. Erre a célra a rendszerterv általában nem alkalmas, elsősorban azért, mert az esetek nagy részében a kész rendszer nem pontosan azt és ugy csinálja, ahogyan a rendszertervben le van irva. A dokumentálás meglehetősen népszerütlen munka, léteznek rendszerek, ahol a dokumentáció készitése a rendszer müködésének kezdetétől számitva tovább tartott, mint amennyi időt a müködő rendszer előállitására forditottak - vagy nem készült el soha.

Az értekezés ezeknek és hasonló problémáknak a megoldására javasol módszereket. A Bevezetésben a továbbiakban néhány alapvető és később sürün használt fogalmat /információs rendszer, szoftver életciklus, tervező rendszer, stb./ definiálunk, majd az értekezés felépitését, főbb eredményeit ismertetjük.

## 0.1. Információfeldolgozó rendszerek

Az "információfeldolgozó rendszer" fogalma csak az utóbbi néhány évtizedben terjedt el, noha valójában az ilyen jellegü rendszerek több évezredes multra tekinthetnek vissza. Ez az első pillantásra talán meghökkentőnek tünő állitás valójában nyilvánvaló trivialitás.

A munkamegosztás, a tevékenységek összehangolása azonnal szükségessé teszi a szervezetek létrehozását. Ezek valamilyen cél elérése érdekében keletkeznek, erőforrásokkal, személyzettel, rendelkeznek, standard eljárások szabályozzák a tevékenységét [6]. A szervezetek részszervezetekre oszthatók, melyek felfoghatók ugy, mint a rendszer részrendszerei. A szervezet egyik ilyen részszervezete a szervezethez tartozó információs rendszer.

Nyilvánvaló, hogy minden szervezetben a döntések meghozatalához, a szervezet irányitásához, információra van szükség. Az információs rendszer feladata ennek az információnak az összegyüjtése ugy a szervezet környezetéből, mint annak részegységeiből. Az összegyüjtött információt tárolja, rendszerezi, összegzi, azaz előkésziti a döntéshozatalra.

Az információfeldolgozó rendszer ilyen általános meghatározása mellett könnyen érthető pl. az a megállapitás, hogy mint az i.e. 2000 táján keletkezett Hammurabi Törvénykönyve igazolja, már a régi Babilonban is léteztek információs rendszerek [7]. A kezdeti időkben az adatok gyüjtése, feldolgozása kézzel folyt, és maga az információfeldolgozás nem különült el a szervezet többi tevékenységétől. Ahogy egyre nagyobb, egyre bonyolultabb szerkezetü és müködésü szervezetek

jöttek létre, úgy vált az információfeldolgozás folyamata is egyre összetettebbé. A nagy szervezetek létrehozták azokat az elkülönitett részszervezeteket, melyek feladata pontosan megegyezett a modern információs rendszerekével.

A forradalmi lépés az információfeldolgozás történetében természetesen a számitógép megjelenése volt. A gyors, megbizható, áttekinthető információszolgáltatást ez tette lehetővé, jelentősen emelve ezzel az információ értékét /bár [7] jónéhány olyan számitógépes korszak előtti példát emlit, ahol az információ döntően befolyásolta vállalkozások sikerét és vagyonok sorsát/. Nem véletlenül kapcsolódik tehát a köztudatban az információfeldolgozás fogalma a számitógéphez.

## 0.2. A szoftverfejlesztési technológiák fejlődése

Semmiképpen nem vállalkozunk teljes történeti áttekintésre - ez nem feladata a dolgozatnak - néhány, a további fejezetekben használt fogalom kialakulásának folyamatát vesszük szemügyre.

[1] a szoftverfejlesztési feladatokat két csoportba osztja:

- részletes rendszerszoftver tervezése és kódolása viszonylag gazdaság-független közegben;
- valamilyen gyakorlati alkalmazási lehetőség felmérése, az alkalmazási szoftver tervezése, tesztelése, karbantartása, gazdaság által irányitott közegben.

Megjegyzi, hogy a legégetőbb szoftverfejlesztési problémák az utóbbi csoportba tartozó feladatoknál jelentkeznek, mig a tudományos elvek inkább az első csoport feladatainál

találnak alkalmazásra. [1] felosztását követve először röviden, főleg az irodalomra hivatkozva az első csoporttal, - rendszerszoftvernek fogjuk nevezni - majd a másodikkal - alkalmazási szoftver - részletesebben foglalkozunk.

Az első periódus a számitástechnika hőskorának számitó 50-es évek. Erre az időszakra - a mi szempontunkból - a magasszintü nyelvek és az operációs rendszerek fejlődése a jellemző, azaz elsősorban a rendszerszoftver fejlődik. Ami az alkalmazási szoftvert illeti, [8] megállapitása szerint a korszakra a lyukkártya orientált, viszonylag egyszerü alkalmazások jellemzőek.

A 60-as években a fejlesztők egyre bonyolultabb rendszerek készitésével kisérleteztek. Olyan alkalmazási területek kerültek előtérbe, mint a repülőgép és szálloda helyfoglalási rendszerek, kórházi nyilvántartások, adatbáziskezelés, termelésirányitás [9].

Az elkészült rendszerek jelentős része az alkalmazás szempontjából sikertelennek bizonyult, sőt néhány projektet befejezni sem sikerült. Ez a jelenség idézte elő a "szoftver krizist", majd a válság megoldásaként a szoftverkészitési technológia gyorsütemü fejlődését.

A szoftverkészitők felismerték a rendszerek szisztematikus módszer szerint történő tervezésének és programozásának szükségességét. A 70-es évek elején született meg a felülről lefelé történő tervezés és a lépésenkénti finomitás elve [10], a modularitás fogalma [11], és a strukturált programozás [12].

A funkcionális absztrakció fenti fogalmai mellett, sőt időben valamivel korábban, a 60-as évek végén jelentek meg az absztrakt adatokat /felhasználó definiálta adattipusokat/ támogató nyelvek: SIMULA 67 [13], Algol 68 [14], stb. A nyelvek fejlődésének egy következő

lépcsőfokát jelentette a 70-es évek vége felé mindinkább tért nyerő felismerés a funkcionális és adatabsztrakció szoros kapcsolatáról, és az egységes absztrakciós mechanizmust biztositó nyelvek előnyeiről /pl. CLU [15], ADA [16], stb./.

Az emlitett programozástechnikai elvek természetesen ugy rendszerszoftver, mint alkalmazási szoftver feladatok megoldásánál sikerrel alkalmazhatóak. Mégis szükséges megemliteni, hogy mig az előbbi feladatok megoldásánál ezeknek az eszközöknek következetes használatával a siker elérhető közelségbe kerül, az utóbbiaknál ez önmagában kevés. Ez könnyen érthető, hiszen mig a rendszerszoftver a számitógép "belső felhasználására" készül, a felhasználói szoftver készitésénél szembe kell nézni a komplex gazdasági közeg által okozott problémákkal is /ezek közül ismertettünk néhányat a Bevezetés elején/.

A szoftvertermékkel kapcsolatos tevékenységek összességét a szoftver életciklusának /life-cycle/ nevezik. Több életciklus modell létezik - a tevékenységek sokfajta csoportositása, és időbeli sorrendjük jónéhány variációja képzelhető el - egy lehetséges közülük a következő tevékenységeket irja elő:

- Rendszerfelmérés: ennek eredményeként kell megszületnie az un. logikai rendszertervnek. Ez teljes, konzisztens, egyértelmü specifikációja annak, hogy mit fog csinálni a készitendő rendszer.
- Tervezés: miután a logikai rendszerterv meghatározta a feladatot, a következő lépés a megoldás módjának meghatározása, a "mit" után a "hogyan" meghatározása. Ezt végzi el a fizikai rendszerterv, mely már az implementáció részleteit /az egyes nagyobb modulok feladatai, az adatok szervezése, stb./ is tartalmazza.

- Programozás: ideális esetben ez csupán kódolást jelent. A felülről lefelé történő tervezés elve szerint az előző lépésben specifikált /környezettől független/ modulok fokozatos részmodulokra bontásával alakul ki a programkód.
- Karbantartás: a kész rendszer üzemeltetését, az üzemszerü müködés közben jelentkező hibák kiküszöbölését jelenti. Ebbe a kategóriába szokás sorolni a rendszer hozzáigazitását a változó környezethez is /módositás/.

Az életciklus modellel kapcsolatban szükséges néhány dolgot megemliteni. A legfontosabb, hogy egyetlen életciklus modell sem tekinthető pontosan követendő "receptnek" melynek lépésről-lépésre történő pontos végrehajtása a siker garanciája. /Egyes vélemények - pl. [17] - egyenesen megkérdőjelezik az ilyen modellek hasznosságát a projekt résztvevői számára, inkább mint a vezetésnek szóló határidőgyüjteményként tekintik./

Megjegyezzük, hogy a felsorolt lépések közé nem huzható merev elválasztó vonal, nehéz megmondani pontosan, hogy hol végződik az egyik, és kezdődik a másik. Nehéz pl. meghatározni a "nagyobb" modulok feladatait specifikáló tervezés végét, és az algoritmusokat egészen a kódolásig finomitó programozás elejét. A felmérés és a tervezés szoros összefüggése még nyilvánvalóbb, a "mit" megállapitásánál mindig gondolni kell a "hogyan"-ra is, a megvalósithatatlan célok elkerülése érdekében.

Nemcsak az egymást követő lépések "egymásbafolyása" bonyolitja az egyszerünek tünő modellt. Nyilvánvalóan előfordulhat pl. az is, hogy csak programozás közben derül fény a fizikai, vagy akár a logikai rendszertervben

elrejtett következetlenségre, és emiatt ujra kell tervezni /esetleg felmérni/ egyes részeket. A példából is látható tehát, hogy az amugy is csak bizonytalanul definiálható lépések sorrendje sem kötött, az egyes tevékenységek ciklikusan ismétlődhetnek.

Az egyes lépések tovább finomithatóak. A tervezést pl. két fázisra - nagyvonalu és részletes tervezés - osztja [1], vagy a próbaüzemeltetés is lehet külön lépés, stb. Számunkra a továbbiakban a feljebb felsorolt lépések elegendőek lesznek a továbbiakban az egyes tevékenységkörökre való hivatkozásra.

Magának az életciklus modellnek a megjelenése - minden hiányossága ellenére - már önmagában az alkalmazási szoftvert előállitó technológia lényeges eredményének számitott. Felhivta ugyanis a figyelmet a programozáshoz képest eléggé elhanyagolt, az eredmény szempontjából azonban döntő jelentőségü területekre: a felmérésre, a tervezésre és a karbantartásra.

A rendszerek készitői felismerték a könnyen áttekinthető dokumentáció - tehát a jól átgondolt tervezés - fontosságát. Egyes szervezetek pl. előirják a rendszer fejlesztése során elvégzendő tevékenységeket /életciklus/ és azt, hogy az egyes tevékenységek elvégzését milyen dokumentum igazolja. A dokumentum - természetesen a következő lépésben felhasználásra kerül.

Az életciklus modell rámutatott arra, hogy nem elegendő programozási módszereket megadni. A szoftverkészités komplex folyamat, egészében kell azt vizsgálni, és olyan módszerekre, eszközökre van szükség, melyek a folyamat egymással szoros kapcsolatban álló fázisait egymástól el nem szakitva képesek segiteni a fejlesztőt a teljes életcikluson keresztül. Ez persze

inkább törekvés, mint elért cél, a módszerek nagy többsége az életciklus egyes szakaszira koncentrál inkább. /A történeti fejlődés menetéből érthető módon jelenleg a felmérés és a tervezés fázisai a fő érdeklődési terület. Ezt jelentőségük, és a megoldatlan problémák sulya indokolja. Az olyan módszereket, melyek a teljes életciklus során segitik /segiteni igyekeznek/ a szoftvergyártót integrált tervezési módszernek, ill. rövidebben tervezési módszernek /[9] "integrated methodology" fogalma nyomán/, az ilyen módszereket támogató számitógépes rendszereket pedig integrált tervező rendszereknek - tervező rendszereknek - /kb. az angol "life-cycle support system" terminológiának felel meg/ fogjuk nevezni. A tervezési módszer tulajdonképpen valamilyen tervezési filozófia és az ezt támogató eszközrendszer /technológia/ együttese.

## 0.3. Az értekezés felépitése, fontosabb eredmények

Az értekezés általában a szoftver - főként az információs rendszerek - életciklusát nyomon követő tervező eszközökkel és módszerekkel foglalkozik. Részletesen tárgyalja a rugalmas számitógépes tervező rendszerek kialakitásának egyik lehetséges módszerét, a generálást. Müködő, széles körben használt rendszer bemutatásával illusztrálja a problémákat és ezek lehetséges megoldásait.

Az értekezés felépitése ezt a gondolatmenetet követi. Egyre szükebb megoldásosztályok egyre alaposabb elemzése vezet először a számitógépes tervező rendszerek, majd ezek generálásának, végül az általunk kidolgozott generálási módszer gondolatához.

Az 1. fejezet a tervező rendszereket mutatja be. A fejezet első részében /1.1/ különféle manuális eszközöket és technológiákat, majd egy számitógépes tervező rendszert ismertetünk. Ez egyrészt lehetőséget ad a módszerek egymás közötti összehasonlitására, másrészt érzékelteti az értekezés későbbi fejezteiben központi szerepet játszó irányzat létrejöttét megelőző fejlődési folyamatot.

1.2. a tervező rendszerek általános felépitésére közöl egy lehetséges sémát. A három alkotórész - a nyelvi processzor az adatkezelés és a lekérdező rendszer - sulyát, és a rendszeren belüli szerepét tisztázza.

A felhasználás tapasztalatait általánositja és rendszerezi 1.3. Értékeli a tervező rendszerek biztositotta előnyöket és rámutat a gyenge pontjaikra is. Az értekezés egyik kiindulópontja az az észrevétel, hogy a leiró nyelv változtathatatlansága a tervező rendszert rugalmatlanná, a konkrét alkalmazásokhoz nehezen adaptálhatóvá teszi.

A fejezetet záró 1.4. a tervező rendszert mint szoftver terméket a gyártó szemszögéből vizsgálja. Mint minden általános célu szoftvernek, ennek is egyszerre kellene általánosnak - sok feladatra alkalmazhatónak - és célorientáltnak - éppen az adott alkalmazásnak megfelelőnek - lennie. 1.4. a szoftver életciklusának egyes fázisaiban járja körbe ezt az ellentmondást. Itt is ugyanarra a következtetésre jutunk, mint amikor a felhasználó oldaláról közelitettük meg a tervező rendszereket: a leiró nyelv rögzitettsége nem csak használatukat, hanem a készitésüket, minőségüket is hátrányosan befolvásolja.

A 2. fejezet az előző gondolatmenetét folytatva az ott felvetett probléma megoldását keresi. A leiró nyelv változtathatóságának szükségességéből és formális leirhatóságának lehetőségéből közvetlenül adódik a tervező rendszer generátor gondolata. /Mint sok más esetben, itt is a programozási nyelvekkel való analógia adhatja az alapötletet./

2.1. a generálás módszerének a termék - a tervező rendszer - előállitására gyakorolt hatását igyekszik felmérni. A tapasztalat szerint mivel a tervező rendszert igénylő felhasználók, mindig a meghatározott feladatból adódó speciális igények kielégitésére képes rendszerrel kivánnak dolgozni, maguk készitenek ilyet, vagy egy létezőt módositanak, ami még több munkát jelent. Sorra véve a tervező rendszer életciklusának egyes szakaszait 2.1. megmutatja, hogy a generálás módszere valamennyi fázisban jelentős segitséget ad, a munka gépies részét /programozás, dokumentálás/ pedig szinte teljesen feleslegessé teszi, igy az adott cél - a speciális igényeket is kielégitő tervező rendszer előállitása - nagyságrenddel rövidebb idő alatt, sokkal kisebb ráforditással érhető el.

A generátor és a generált tervező rendszer felhasználójának tevékenységét veti össze 2.2. egy hagyományos tervező rendszer felhasználójáéval. Meg kell állapitani, hogy a leiró nyelv definiálhatósága a definició elkészitésének komoly számitógépes képzettséget és a leirandó rendszer alapos ismeretét igénylő feladatát is a felhasználóra rója, szemben a hagyományos tervező rendszerekkel, melyekhez kevesebb szaktudás is elegendő, hiszen csak használni kell tudni őket. Szerencsére a kétfajta felhasználó - a definiáló és a definiált nyelvet leirás készitéséhez használó -

feladatai jól elkülönithetőek, igy ez nem okoz feloldhatatlan feszültséget egy szervezeten belül. Megadjuk a definiciós szint - a generátor - felhasználójának feladatait /a leirás készitőjéé változatlanul az 1.3-ban specifikáltak/, és javaslatot teszünk egy definiciós módszerre is.

A generálás módszere a leiró nyelvek formális definiálhatóságán alapul. Olyan fogalomrendszert kell találni, mely alkalmas a tervező rendszerek általános leirására. Az egyes tervező rendszerekben használt fogalmak ezeknek az előfordulásai lesznek. Ez ugyanaz a mechanizmus, mint ahogy a leiró nyelvek tipusszerkezete modellezi az információs rendszert, csak az absztrakció egy fokkal magasabb szintjén.

Abból az észrevételből kiindulva, hogy egy tervező rendszer speciális célu információs rendszernek tekinthető, 2.3. a fogalomrendszer alapjául az ANSI/SPARC bizottság általános adatbáziskezelő rendszer modelljét ajánlja. A leiró nyelv és a lekérdező rendszer a külső, az adatkezelő interface a belső sémának feleltethető meg, és a kettő közötti leképezést megvalósitó konceptuális séma a tervező rendszerek esetében nem más, mint a valós világnak az a modellje, melynek terminusaiban a tervező rendszer felhasználója gondolkozhat. Három különböző konceptuális sémát mutatunk be. Közülük az egyik rögzitett leiró nyelvhez vezet, a másik kettő esetében a leiró nyelvek változtathatóak, de a különböző konceptuális sémák különböző módon jelölik ki a változtathatóság határait. A modell külső és belső sémájának felépitésével kapcsolatosan általános javaslatokat teszünk.

A generátor és a generált rendszer szoftvertechnikai megoldásaival foglalkozik 2.4.

A generátor a tervező rendszer formális leirását felhasználva /megmutatjuk, hogy az adatkezelő részt nem kell megadni, az standard lehet/ a generált rendszert vezérlő táblázatokat készit. Fontos kiemelni, hogy nem program, hanem csak táblázatok generálásáról van szó.

A 3. fejezet tárgya az SDLA rendszer. Nem felhasználói kézikönyv részletességü ismertetés volt a cél, inkább a tervező rendszer generátorokra vonatkozó konkrét javaslatok rendezett felsorakoztatása. A rendszer maga a szerzők közötti viták, kompromisszumok eredménye. A 3. fejezetben leirtak több ponton - főképpen a rendszer szemantikája esetében - egyoldalu /pl. tipusszerkezet/ vagy később kialakult /pl. szemantika definiálása tervező rendszerekben/ álláspontot képviselnek.

A tervező rendszer definiálásának eszközeit tárgyalja 3.1. A konceptuális és a külső séma elemei /fogalmak és formák/ megadásának módját irja le 3.1.1. és 3.1.2. A szemantika kérdéseivel foglalkozó részben /3.1.3./ felhasználva a 2.3-ban alkotott modellt, a leiró nyelvek szemantikájára általános definiciót adunk, majd egyenként vizsgáljuk az SDLA-ban megadható szemantikus összefüggések tulajdonságait. Ezek a konkrét konceptuális modellen alapulnak, igy más rendszerekre való általánosithatóságuk minden konkrét esetben külön vizsgálatot igényel. Figyelembe kell azonban venni azt, hogy - legalábbis az információs rendszerek tervezéséhez használható rendszerek esetében - nincsenek nagy eltérések az egyes konceptuális sémák között, igy az egyes összefüggések kisebb változtatásokkal könnyen leképezhetőek egy másik rendszer fogalmaira.

3.2. a definiálható leiró nyelvek közös tulajdonságait irja le. Ilyen az általános objektumazonositási

mechanizmus, az implicit definició megengedésének elve, az automatikus tipusfinomitás /3.2.1./. Valamennyi leiró nyelv szerkezete azonos: egymásba ágyazott szekciókból áll. A szekciók egymáshoz való viszonyát, felépitésük szabályait 3.2.2. tárgyalja.

A felülről lefelé tervezés módszerét, a feladat részfeladatokra bontását támogatja az alrendszerek léte. Ez koncepciójában a programozási nyelvek blokkszerkezetéhez hasonló. 3.2. utolsó témája a törlési és módositási mechanizmus. Több más rendszertől eltérően az SDLA-ban ez a leiró nyelv része, nem külön parancsrendszer.

A 3.3-ban tárgyalt lekérdező rendszer fő jellegzetessége, hogy használatához nem kell uj nyelvet megtanulni, a felhasználó a leiró nyelv segitségével definiálja, hogy a tárolt információhalmazból milyen objektumok milyen kapcsolatait kivánja kiiratni. A módszer nyilvánvaló előnye egyszerüsége, eleganciája. Könnyen általánositható más leiró nyelvekre is.

A disszertáció utolsó fejezetét záró 3.4. adatkezelési kérdésekkel foglalkozik. A külön tárgyalást a rendszer hatékonyságát döntően befolyásoló problémák sulya indokolja. A tárolt adatok elérése, módositása - a konceptuális séma szerkezetét figyelembe véve - relációs interface-en keresztül történik. Az interface funkcióinak megvalósitása egy CODASYL tipusu adatbáziskezelő rendszerrel történik. 3.4. a relációs referencia adatmodell CODASYL-ra való leképezésének és a hatékony CODASYL implementációnak megoldását ismerteti.

# 1. TERVEZÉSI MÓDSZEREK, TERVEZŐ RENDSZEREK

A szoftverkészités folyamatát, a szoftver életciklusát egy ház épitéséhez /tervezéstől a karbantartások elvégzéséig/ szokás hasonlitani /[4], [9] stb./. A hasonlat valóban találó: mindkét folyamatban megtalálhatók a jellegzetes fázisok: az igények felmérésének, a tervezésnek, a megvalósitásnak, majd a karbantartásnak megfelelő szakaszok. A hasonlóságok mellett a különbözőség is lényeges dologra mutat rá: a ház épitésénél a megrendelő - a dolog természetéből adódóan - sokkal pontosabban, jobban átgondoltabban képes kivánalmait megfogalmazni, mint egy nagy szoftver-rendszer megrendelője. Ez nem csupán számára, hanem a tervező, az épitész szempontjából is kellemesebb [4].

A hasonlatot továbbfejlesztve, ebben a részben az épités "klasszikus" eszközeinek megfelelő szoftverkészitési módszerekről lesz szó. A szabadkézi rajz nyilván pontatlan, - noha az épitész eszköztárából egy-egy megoldás első felvázolásához nem hiányozhat - a vonalzó használata nélkülözhetetlen. Az állitott rajztábla, rajta tolható csuklós rajzgéppel megkönnyiti a szerkesztést. A másoló berendezések szükségtelenné teszik a rajz sok példányban való elkészitését.

A nagyobb épületek vagy épületegyüttesek tervezésére és kivitelezésére csoportok, és nem egyének vállalkoznak. Ilyenkor már szükség van valamilyen módszerre, tervezési filozófiára, és ezt támogató eszközrendszerre is. A munkát fel kell osztani a csoport tagjai között, el kell kerülni az átfedéseket, valamilyen tevékenység többszöri elvégzését, vigyázni kell arra, hogy az egyes részmegoldások összehangolhatók legyenek, meg kell szervezni a munka időbeli ütemezését, stb. Mindehhez valamilyen

tervezési /kivetelezésnél technológiai/ módszerre van szükség. A terveknek egységeseknek kell lenniük /szabvány/, az egyes részleteknek illeszkedniük kell egymáshoz, a tervezés egyszerüsitése érdekében jól bevált korábbi megoldásokból át lehet venni elemeket, stb. A módszernek olyannak kell lennie, hogy a fenti követelmények minél természetesebben - lehetőleg automatikusan - teljesüljenek.

A tevékenységek jelentős része mechanikus vagy azzá tehető /adatok összegyüjtése, tárolása, kikeresése, egyszerü ellenőrzések, standard megoldások illesztése, számolások, stb./. Ezek elvégzése gépesithető, létrehozható a számitógépes tervező rendszer. Ennek természetesen a már kialakitott módszert és az ahhoz kapcsolódó technológiát kell támogatnia /pl. a rajzgéppel - plotter - készitett rajzoknak meg kell felelniük a kialakult szabványnak/.

Ebben a fejezetben először a szoftverfejlesztés kézi eszközeiről lesz szó. Ezeknek jelentős része a számitógépes rendszerek bevezetésével nem változott, a gépi támogatás használatukat csak egyszerübbé tette, módositotta. A rögzitett leiró nyelvvel rendelkező számitógépes rendszereket a PSL/PSA /Problem Statement Language/ Problem Statement Analyser/ példáján keresztül mutatjuk be. /[39] szerint ez a legelterjedtebb tervező rendszer/. Az ilyen rendszerek általános felépitésével, használatukkal általánosan is foglalkozom, elemzésük elvezet a tervező rendszerek generálásának gondolatához.

## 1.1. A szoftverkészités segédeszközei

A technológia eszközökön, vagyis a módszert realizáló eljárásokon, annak véghezviteléhez segitséget adó termékeken alapul. Az előző paragrafusban emlitett technológiák közül pl. a programozást - ugy rendszer, mint alkalmazási szoftver esetén - támogatják az egyes elvek - felülről lefelé tervezés, absztrakt adattipusok, stb. - gyakorlati alkalmazására kifejlesztett magasszintü programozási nyelvek.

Most az eszközökről, és néhány, ezeken alapuló technológiáról adunk rövid áttekintést, különös figyelmet forditva az életciklus első két fázisában, a felmérésben és a tervezésben alkalmazhatóakra. Először az egy-egy fázisban felhasználható de önálló technológiának aligha nevezhető szoftverkészitési eszközökről lesz szó. A segédeszközök második csoportjába soroltuk a számitógéptől többékevésbé független, alapvetően manuális technológiákat, és ezek eszközrendszereit. A kifejezetten számitógép-orientált technológiák alkotják a harmadik csoportot.

### 1.1.1. Eszközök

Eszközökben leggazdagabb - mint erre O.2-ben B.W. Boehm nyomán utaltunk is - programozási fázis. Néhány cimszó az időbeli fejlődés érzékeltetésére: assembler, könyvtár, operációs rendszer, file-kezelés, magasszintü nyelvek, programcsomagok.

Ezeknek és a hasonló, a számitógéphez szorosan kapcsolódó eszközöknek alapvetően a programozáshoz van csak közük, bár pl. a magasszintü nyelvek használata nagymértékben könnyiti a karbantartást, sőt a tervezéshez is adhat ötleteket. A továbbiakban néhány olyan eszközről

lesz szó, melyek jelentős szerepet játszottak ugy a manuális, mint a számitógépes technológiák kifejlődésében.

### 1.1.1.1. Blokkdiagram

A programozás megjelenésével szinte egy időben kezdték használni a blokkdiagramokat /flowchart/ [18]. Sokáig jó programtervező eszközként tartották számon, - tulajdonképpen a tervezés és a programozás szakaszai közötti láncszemként szolgált, sőt mint dokumentálásban nélkülözhetetlen eszköz, a karbantartáshoz is segitséget nyujtott - de az utóbbi néhány évben népszerüsége csökkent. Néhány érv a használata ellen [9].

- A blokkdiagram jelölésmódja ugy a felmérés, mint az implementáció /programozás/ jelöléseivel inkonzisztens.
- A blokkdiagram nehezen olvasható, bonyolultsága a program komplexitásával nő.
- A blokkdiagramnak nincs számitógépes támogatása.
- A blokkdiagram nem választja jól szét a lényeges és lénylegtelen részleteket. Egyes dobozok magas szintü müveleteket tartalmaznak, de ezekkel egyenrangu az "i:=i+1" doboz.

[18] kisérletekkel igazolta, hogy nincs lényeges előnye a blokkdiagram használatának sem a program tervezése, sem pedig megértése, belövése, vagy módositása szempontjából. Arra a következtetésre jut, hogy a blokkdiagram nem más, mint a programnyelv utasitásai által tartalmazott információ /a kódhoz képest/ redundáns ábrázolása.

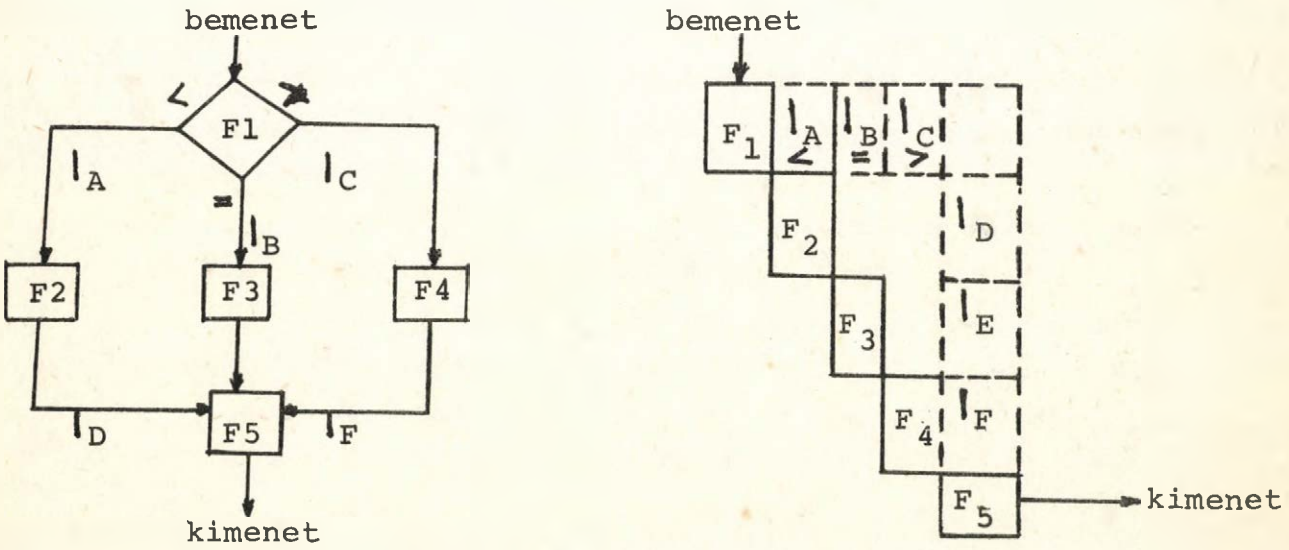
### 1.1.1.2. $N^2$-diagram

A blokkdiagram eredeti alapgondolata - a grafikus ábrázolás áttekinthetőbb és tömörebb a szövegnél - több más formális leiró technikánál megjelenik. Ezek közül is említünk még néhányat.

Az $N^2$-diagram /$N^2$ Chart/ [19] egy rendszert alkotó részek, és az ezek közötti kapcsolódások /interface-ek/ szemléletes ábrázolására szolgál. Onnan kapta a nevét, hogy N négyzettel ábrázolt N részrendszer /funkció/ között $N^2$-N interface létezhet. Az interface-ek maguk is négyzetek lesznek.

Az 1. ábra [19] egy blokkdiagramot és vele ekvivalens $N^2$-diagramot mutat. Az ábrán jól követhetők az $N^2$-diagram felépitésének szabályai [19]:

- az összes funkció az átlón helyezkedik el;
- a rendszer kimeneteit vizszintes nyilak jelzik;
- a rendszer bemeneteit függőleges nyilak jelzik;
- a nem funkciót reprezentáló négyzetek a velük egy sorban és oszlopban lévő funkciók közötti egyirányu interface-t jelölnek.




1. ábra

Blokkdiagram és ekvivalens $N^2$-diagram

[19] szerint az $N^2$-diagram technikát már sok projektben sikerrel alkalmazták. Több formája létezik, tulajdonképpen csak a feljebb felsorolt szabályok betartása kötelező, az eredményül kapott ábra kiegészithető pl. a funkciók ciklikus ismétlődését jelző nyilakkal, vagy az interface-ek ábrázolhatók körökkel, a négyzetek helyett, stb.

1.1.1.3. Adatszerkezet diagram

A blokkdiagram feladata a programmüködés logikájának a szemléletessé tétele, vagyis a program funkcionális szerkezetének, a funkcióknak és sorrendjüknek a leirása. /Erre a célra természetesen más eszközök is léteznek./ Mint [20] kisérleti eredményekkel alátámasztott - és egyébként is nagyon hihetőnek tünő - hipotézise állitja azonban, egy program müködésének megértése szempontjából a használt adatok szerkezetének ismerete legalább olyan fontos, vagy inkább fontosabb, mint a funkcionális szerkezeté. Az adatszerkezet dokumentálására még nem alakult ki olyan egységes szerkezetü grafikus rendszer, mint a blokkdiagram /az adatszerkezet-diagramra több konvenció létezik pl. [5] is közöl egyet/. [20] kisérleteinek tanulsága szerint nem okoz lényeges eltérést a dokumentáció érthetősége szempontjából, hogy grafikus, vagy szöveges formában irjuk le az adatok szerkezetét - de ilyen leirásra mindenképpen szükség van.

Érdemes megemliteni azonban, hogy a gazdag adatdefiniciós lehetőséggel rendelkező nyelvek legalábbis csökkenthetik az adatszerkezet leirásának jelentőségét. Arról van szó ugyanis, hogy mig egy nyelvben csak egyszerü adatdefiniciós lehetőségek léteznek, addig a bonyolultabb adatszerkezetek realizálása ezeknek az esz-

közöknek a segitségével és /esetleg elég bonyolult/ algoritmussal történik. /Képzeljük el például, hogy fastrukturát kivánunk FORTRAN-ban ábrázolni!/ Nyilvánvaló, hogy ilyen esetekben szükség van az adatszerkezet leirására, mert egyébként az algoritmusból kell azt kibogarászni. A fejlettebb adatdefiniciós lehetőségekkel - elsősorban absztrakt adattipusokra gondolunk - rendelkező nyelveknél maga a definició tartalmazza az adatszerkezetet, és az önmagában is elég érthető. /Lásd pl. a fastruktura PASCAL ábrázolását [21]./ Ez a tendencia hasonló ahhoz, ahogyan a magasszintü programozási nyelvek terjedése fokozatosan szoritja ki a blokkdiagramot a programtervezés és dokumentálás gyakorlatából.

### 1.1.1.4. Adatáramlás diagram

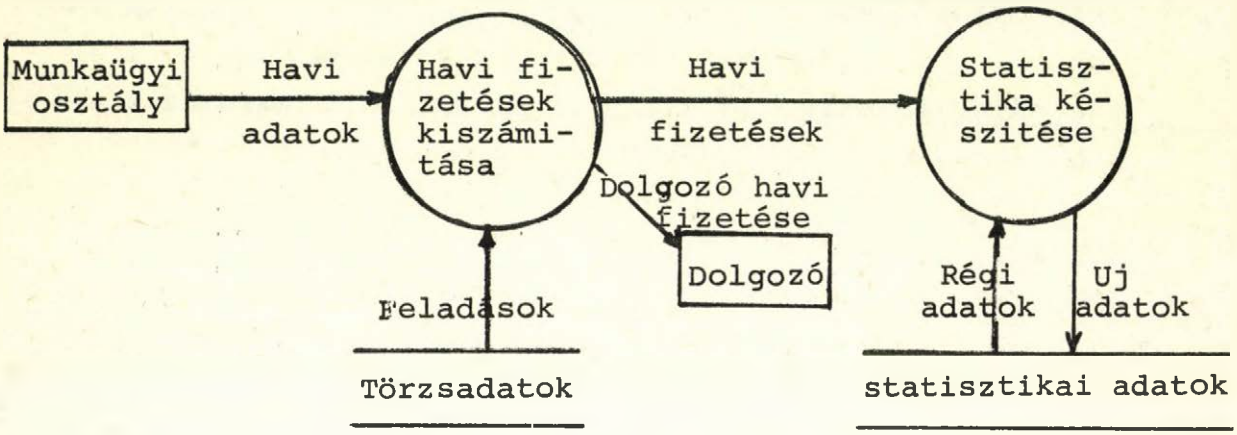
Az egyre bonyolultabbá váló adatszerkezetnek a funkcionális szerkezettel való fokozatos "egyenjogusodását" jelzi az adatközéppontu technikák fejlődése. Noha - mint emlitettük - a statikus adatszerkezet leirására nincs egységes formális apparátus /éppen bonyolultsága miatt/, az adatok rendszeren belüli mozgásának szemléletes, grafikus ábrázolására létezik ilyen: ez adatáramlás /data flow/ diagram.

Az adatáramlás diagram a következő objektumtipusok ábrázolását teszi lehetővé.

- Adatáramlás - a rendszer adatainak mozgása. Jelölése nyillal, és a mellé irt adatnévvel történik.
- Folyamat - az adatokat átalakitó algoritmus. Jelölése /pl. [5]-nél/ kör, benne a folyamat nevével.
- Fájl - adathalmaz tárolása. Pl. két párhuzamos rövid szakasszal és a közéjük irt névvel jelölhető.

· Forrás - a rendszer határain kivül eső személy, vagy szervezet, mely adatokat szolgáltat, vagy fogad. Téglalappal jelöljük.

A 2. ábrán egy bérelszámolás folyamatának vázlatos adatáramlás diagramja látható.

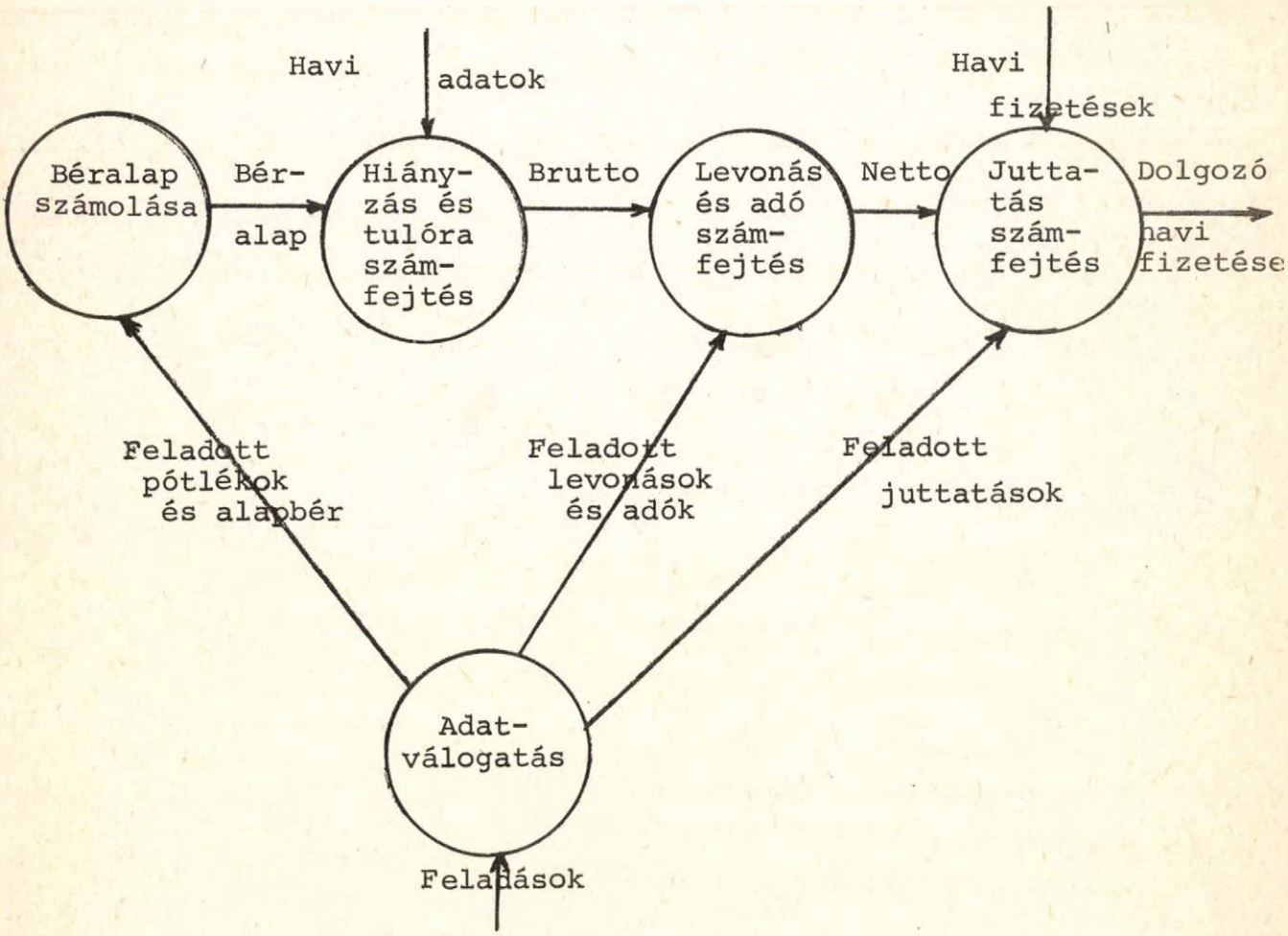



2. ábra
Bérelszámolás adatáramlás diagramja

Az ábrán - amellett, hogy szemléletessé teszi a rendszer felépitését és könnyebbé teszi egyes hibák /felhasználatlan adat, vagyis semmibe mutató nyil, csak létrehozott, de sehol fel nem használt fájl, vagyis olyan amiből nem vezet ki nyil, stb./ kiszürését - érzékelhetővé teszi az adatáramlás diagram /és a többi grafikus eszköz/ használatának legfőbb akadályát is: a papir végességét. Ha tovább bonyolitanánk a rajzot, alaposabban részletezve

a feldolgozás folyamatát bonyolult, és - tetszőleges részletességig lemenve - tetszőleges méretü ábrát kapnánk.

Erre a problémára megoldásként a felülről-lefelé tervezési technika grafikus változatát szokás ajánlani megoldásképpen. Esetünkben ez azt jelenti, hogy a részletesebb specifikálást nem akárhogyan, a már elkészült ábra figyelmen kivül hagyásával, hanem a 2. ábra folyamatainak részletesebb megadásával végezzük, minden részletezéshez ujabb ábrát készitve. A "Havi fizetések kiszámitása" folyamat pl. a 3. ábrán látható módon bontható részegységekre.




3. ábra

"Havi fizetések kiszámitása" adatdiagram

Ujabb és ujabb ábrákkal egyre tovább lehet finomitani az egyes részleteket.

Ezzel a problémát - elvileg - megoldottuk, bár fizetnünk kell érte, ugyanis most már az egyes leirási szintek konzisztenciáját /be- és kimenő nyilak egyezősége/ is ellenőrizni kell. /A gyakorlatban további problémát jelent a felülről lefelé történő tervezési technika következetes véghezvitele./

1.1.1.5. Struktura diagram

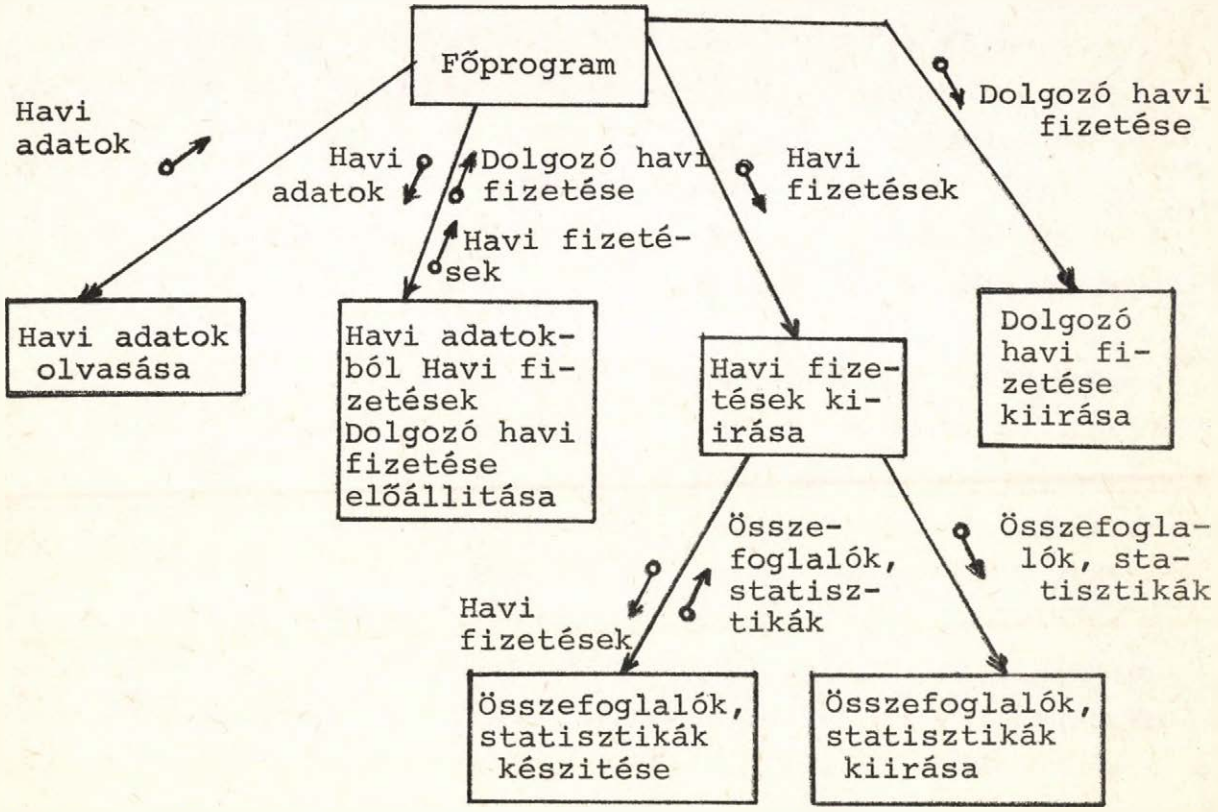
Az adatáramlás diagram használatával rögzithetőek a rendszerrel szemben támasztott igények: mit kell csinálnia, milyen adatokból milyeneket kell levezetnie. Az Az adatáramlás diagramhoz kapcsolódó másik grafikus technika, a struktura diagram /Structure Chart [5]/ a "hogyan" kérdésre keresi a-"mit" megválaszolásával többé-kevésbé meghatározott - megoldást.

A struktura diagram modulok /programok, rutinok/ egymás közötti kapcsolatát ábrázoló eszköz. Alkotó elemei:

- A téglalappal és a benne lévő névvel ábrázolt modul.
- Két modul közötti kapcsolat, melyet egyenes nyil jelöl. Ez azt jelenti, hogy az egyik modul hivja a másikat.
- Paraméterátadás. Jelölje az adat neve, mellette az átadás irányát jelző nyil.

Az adatáramlás diagram /"mit"/ meghatározza a struktura diagramot /"hogyan"/. [5] közli azokat a lépéseket, melyekkel az adatáramlás diagram struktura diagrammá alakitható. Mi itt a 4. ábrán a 2. ábra bérelszámolási

rendszerének megfelelő struktura diagramot közöljük.




4. ábra

A bérelszámolás struktura diagramja

### 1.1.1.6. Adatszótár és magasszintü specifikációs nyelv

A fentiekben tárgyalt eszközök alapvetően grafikus jellegüek, és - éppen ezért - kevés számitógépes támogatást élvezők voltak. Most két olyan eszközt emlitünk, melyek ellenkezőleg, inkább szöveges jellegüek, és számitógép-orientáltak: az adatszótárról /data dictionary/ és a magasszintü specifikációs nyelvekről /requirements and specifications languages/ van szó. Megjegyezzük, hogy az óvatos fogalmazás itt nem felesleges: az adatáramlás-diagram pl. számitógéppel /grafikus display/ is készithető, elemeztethető [22], másfelől viszont pl. [5] a kézi adatszótár használatát ajánlja, ha nincsen megfelelő számitógépes programcsomag, ill. a hibrid automatikus-kézi adatszótárat, vagyis pl. szövegszerkesztő megfelelő konvenciókat betartó használatát adatleirásra. A magasszintü specifikációs nyelvek egy része merevebb szintaxissal rendelkezik, tehát számitógépes feldolgozásra alkalmas, másoknál csupán a leirás szerkezete kötött, igy ezeknél a gépi támogatás jelentéktelen. Az adatszótár a rendszer adatainak a leirását tartalmazza. Megjelenési formája igen változatos: létezhet számitógépes adathordozón megfelelő programcsomag által kezelhető módon, lehet jegyzetfüzet valamilyen módon szervezett bejegyzésekkel, állhat ábrákból /vagy tartalmazhat ábrákat is/, stb.
Néhány alapvető követelményt ki kell elégitenie [5].

- Az adat neve alapján a leirásának könnyen kikereshetőnek kell lennie.
- Az adatszótár nem lehet redundáns.
- Az adatszótár és a rendszerterv többi komponense között nem lehet tul nagy redundancia.

- Az adatszótárnak könnyen módosithatónak kell lennie.
- A konvenciók alkalmazásával következetesnek kell lenni.

A kézi technikák elég nyilvánvalóak, és kisebb rendszereknél elégségesek is. A probléma akkor válik sulyosabbá, amikor a rendszerben szereplő adatnevek mennyisége tulmegy bizonyos határon, és az egész áttekinthetetlenné válik. Ilyenkor - illetve a nagyobb rendszerek tervezésekor, ezt a helyzetet megelőzendő - célszerü automatikus /számitógépes/ adatszótárt használni.

A számitógépes adatszótár összetevői [23]:

- az adatleirásokat /metaadatokat/ tartalmazó adatbázis;
- lekérdező, elemző lehetőségek;
- az adatszótár titkosságát, integritását, visszaállithatóságát, osztott használatát biztositó software eszközök;
- a software interface-ek, melyek segitségével programok adatokat kérhetnek az adatszótárból, ill. módosithatják annak tartalmát.

Az első három komponens lényegében egy speciális célokra használt számitógépes adatbáziskezelő rendszert határoz meg. A negyedik összetevő az adatbáziskezelő rendszerek adatszótárainál jelentős igazán, ezek teszik lehetővé pl. a forditóprogramoknak vagy adatleirás processzoroknak az adatszótárban lévő információ elérését ill. felujitását.

A számitógépes és a kézi adatszótár viszonya tehát

hasonló a számitógépes és kézi adatfeldolgozáséhoz, a kézi nyilvántartás gépi adatbázisba szervezése a számitógépes adatszótár. Ugyanazokat az adatokat képesek tárolni és kezelni, csak a gép természetesen hatékonyabb, rendszeresebb, mint az ember, nem veszit el adatokat, olyan listákat képes produkálni, melyeket kézi nyilvántartással sokkal nehezebb, vagy gyakorlatilag lehetetlen lenne.

Az adatszótár a rendszer adatainak többé kevésbé formális leirását tartalmazza. Minél "gép-közelibb" az adatszótár, természetesen annál szigorubb a szintaktika, annál formálisabb a leirás. A lehetőségeknek igen széles skálája képzelhető el, a kötetlen, ábrákkal teletüzdelt szöveges leirástól a forditóprogram által ellenőrzött adatleiró nyelvig.

A magasszintü nyelv a programszerkezet többé kevésbé formális leirására szolgáló eszköz, ilyen értelemben az adatszótár kiegészitője, ellenpárja. Nyilvánvaló a kettő közötti átfedés, hiszen adat és programszerkezet nem képzelhető el egymás nélkül. A specifikációs nyelvekre is jellemző a megvalósitás sokfélesége, a nyelvek széles skálája egyszerütől bonyolultig, gép-orientáltól kötetlenig.

Mint feljebb utaltunk rá, a "magasszintü specifikációs nyelvek" az angol "requirements languages" ill. "specifications languages" az irodalomban /ld. pl. [24]/ sokszor megkülönböztetett fogalmait vonja össze. Az elsővel lényegében az életciklus első fázisában, a felmérésben, az utóbbival a második fázisban, a tervezésben használt nyelvet jelölik.

Mint ezt korábban megjegyeztük a két fázis meglehetősen összemosódik, igy eszközeik is hasonlóak. A magasszintü specifikációs nyelvek közül valamennyi a

természetes nyelvből indul ki, erre alkalmaz szerkezeti, szintaktikus és szemantikus megszoritásokat. A nyelvek között az igazi különbség a megszoritások szigoruságának a mértékében, vagy.s az elérhető számitógépes támogatás fokában van. [24] mind a két kategóriában emlit teljesen formális, forditóprogrammal, szókészlettel rendelkező számitógépes rendszereket és kötetlen - pl. csak a használható mondatok szerkezetére vonatkozó megkötéseket tartalmazó - nyelveket /számitógépes támogatással/, [5] "Structured English" nyelve pedig csupán néhány egyszerü szabály betartásával készitett szöveges leirás.

A magasszintü specifikációs nyelv tervezésénél a két szélsőség - teljesen kötetlen leirás /[5] a "viktóriánus regény" jelzővel jellemzi/, ill. programozási nyelvi szintü konstrukciókig menő részletességü, és ehhez méltóan szigoru szintaktikáju nyelv - között szokás kompromisszumos megoldást találni. A kötetlen leirás előnye az érthetőség a laikus felhasználó számára - nem utolsó szempont, a rendszer készitője és felhasználója közötti szakadék áthidalása - a kötött nyelv viszont tömörséget biztosit, és megadja a számitógépes támogatás lehetőségét. A kompromisszumos kiskapu általában a nyelvek által biztositott "megjegyzés" lehetősége. Ezzel ugy lehet élni, /és visszaélni/ ahogy a felhasználó akar, igy végső soron a legkötöttebb nyelv is használható tetszőlegesen kötetlen leirás készitésére. A specifikációs nyelvek - és általában a fenti áttekintésben szereplő eszközök - csupán a felhasználás lehetőségét bocsáthatják a felhasználó rendelkezésére, nem kényszerithetik az alkalmazási filozófia helyes használatára.

### 1.1.2. Manuális technológiák

Az integrált tervezési módszer technológiája és az eszközök valamilyen csoportja abban különbözik egymástól, hogy az előbbi alkalmazásával az életciklus egyes szakaszai közötti szerves kapcsolat az egyes fázisokban használt eszközökön keresztül magától értetődő természetességgel valósul meg. Ideális technológiánál az eszközök átfedik egymást, egyik használata feltételezi és támogatja a másikat, mint ahogy az életciklus egyes szakaszai elkülönithetetlenek egymástól.

#### 1.1.2.1. Structured System Analysis

A Structured System Analysis /SSA/ [4],[5] tervezési módszernek tekintendő, hiszen az életciklus több szakaszát - felmérés, tervezés, karbantartás - lefedni kivánó eszközrendszerről, és egységes filozófiáról - strukturálás, felülről lefelé tervezés - van szó. Technológiája a következő eszközökön alapul:

- adatáramlás diagram;
- adatszótár;
- relációs szerkezetü logikai adatok;
- magasszintü specifikációs nyelv.

Ezek közül csak a harmadikat nem tárgyaltuk O.1.1.-ben. Az SSA a logikai adatait a legegyszerübb adatmodell - a relációs - alapján irja le, felhasználva ehhez a relációs adatbázisok elméletének több elemét /funkcionális függőségek, relációs müveletek, stb. [25]/, mintegy feltételezve, hogy az adatkezelés egy ideális relációs adatbáziskezelő rendszerrel történik majd.

A technológia első lépésként a "fizikai" adatáramlás diagram elkészitését javasolja. Ez azért fizikai, mert a megvalósitás részleteit - osztályok nevei, emberek nevei, eljárások, eszközök, stb. - is tartalmazhatja.

Második lépés a "logikai" adatáramlás diagram és ezzel párhuzamosan az adatszótár készitése. Az adatáramlás diagram esetén a "fizikai"-ból "logikai"-vá transzformálás elég nyilvánvaló. Az adatszótárba kerülő adatoknál ez messze nem triviális, a jövendő rendszer adatszerkezete nem feltétlenül egyezik meg a régiével, hiszen ki kell szürni a redundanciákat. A segédeszköz ehhez a lépéshez a relációs adatmodell, a funkcionális függőségek elmélete [26]. /Ez a feladat jól automatizálható [27]/.

A harmadik lépés az eljárások algoritmusainak megadása. A specifikáció eszköze a magasszintü specifikációs nyelv /bár más eszközöket, pl. döntéstáblákat is megenged az SSA/.

Érdemes megjegyezni, hogy minden lépésnél a felülről lefelé történő tervezés elve érvényesül. Ezt már az első lépés meghatározza, hiszen az adatáramlás diagram - mint ezt O.1.1.-ben láttuk - többszintes, a szintek, elvileg a részletek finomitásával jönnek létre. Az adatszerkezet kialakitásánál először az entitásokat majd azok attributumait kell definiálni, és formális lépések sorozataként jönnek létre a normálformáju relációk. Az algoritmusok szerkezetét eleve meghatározzák az adatáramlás diagram szintjei, ezek felépitésével automatikusan megtörténik az algoritmus strukturálása, szintekre bontása is. Itt az SSA a struktura diagramot javasolja eszközként [5] ezzel mintegy hidat épitve a strukturált tervezés /Structured System Design [28]/

felé. Mi a két tervezési módszert - pl. [5] vagy [9] véleményeivel egyetértve - egynek, pontosabban egymás folytatásának tekintjük, és a strukturált tervezést, az SSA következő lépésként vizsgáljuk.

A negyedik lépés tehát eszközként a struktura diagramot hasznája. Az adatáramlás diagramból mechanikusan levezetett struktura diagram által meghatározott modul szerkezet persze nem feltétlenül optimális. Ahhoz, hogy ilyenhez jussunk, először is az "optimális" fogalmát kell megközeliteni.

Az SSA szerint ideális rendszer egymástól független, egyenként pontosan definiált, egy-két mondattal elmagyarázható funkciókat ellátó modulokból áll. A függetlenséget a kapcsolódás /coupling/ mértékével, egy modul helyes definicióját, az általa végzett tevékenységek összetartozását a modul belső kohéziójával /cohesion/ jellemzik. A kapcsolódásnak kicsinek, a kohéziónak nagynak kell lenni.

A kapcsolódás mértékének meghatározására [29] algoritmust és mérőszámot közöl. A kiemelkedő jelentőséggel biró tényezők közül néhány.

- A modulok közötti kapcsolat tipusa. Ha pl. lehetőség van egyik modulból GO TO segitségével átkerülni a másikra az igen veszélyes. Patologikus jel ugyancsak a nem explicit adatcsere /pl. FORTRAN COMMON/ is.
- A modulok közötti kapcsolatot fenntartó adatok tipusa, mennyisége és azok iránya. Nyilvánvaló, hogy nem segiti elő a modulok függetlenségét, ha az egyik a másiknak olyan adatot kénytelen átadni, melynek alapján az, a müködésére vonatkozó döntést hoz. Ha mégis ilyen adatokra van szükség, akkor célszerü, ha a döntést

implikáló változóérték alul születik, és a döntés végrehajtása felül történik, ugyanis egy lefelé irányuló döntéshozó változó jelentését az alacsony szintü modul nem láthatja át teljesen, és igy integritása csorbát szenved.

A kohézió mértéke tulajdonképpen a meghatározásból adódik. Nagy kohéziós erővel biró, vagyis önmagában zárt és teljes, jól definiált funkciót teljesitő modul könnyen elnevezhető, és a neve az általa ellátott tevékenységet jellemzi. Ha nehéz megfelelő nevet találni egy modul számára, vagy a név semmitmondó, az hibás tervezésre, a nem megfelelő definicióra utal.

A kohézió és a kölcsönhatás nem elkülönithetőek. Nyilvánvaló, hogy ha a modulok között nagy a kölcsönhatás /pl. ha egy alacsony szintü modul a magasabb szintütől kap vezérlő paramétert/, az egyes modulok kohéziója is kicsi /esetünkben az alacsony szintü modul a vezérlő paramétertől függően lát el több, különböző funkciót/.

Az SSA filozófiája szerint tehát, az adatáramlás diagramból előállitható struktura diagramok közül a legnagyobb kohézióval biró és a minimális kölcsönhatásban álló modulokat tartalmazót kell kiválasztani. Ennek érdekében - ha szükséges - az adatáramlás diagram módositható /az életciklus első két fázisa, a felmérés és a tervezés iterálódhat/.

Az SSA az életciklus több szakaszát - a felmérést, a tervezést, a menet közben készitett dokumentumok jóvoltából a karbantartást - lefedő gazdag eszközkészlettel rendelkező tervezési módszer. Az eszköztár tulzott gazdagsága bizonyos mértékig nehezíti is használatát, ugyanis nem homogén, hanem egymást többé kevésbé kiegészitő, különböző jellegü eszközökből áll. Több

jelölésrendszerrel kell egyszerre dolgozni:

- adatáramlás diagram;
- adatszótár;
- relációs adatmodell;
- magasszintü specifikációs nyelv;
- struktura diagram.

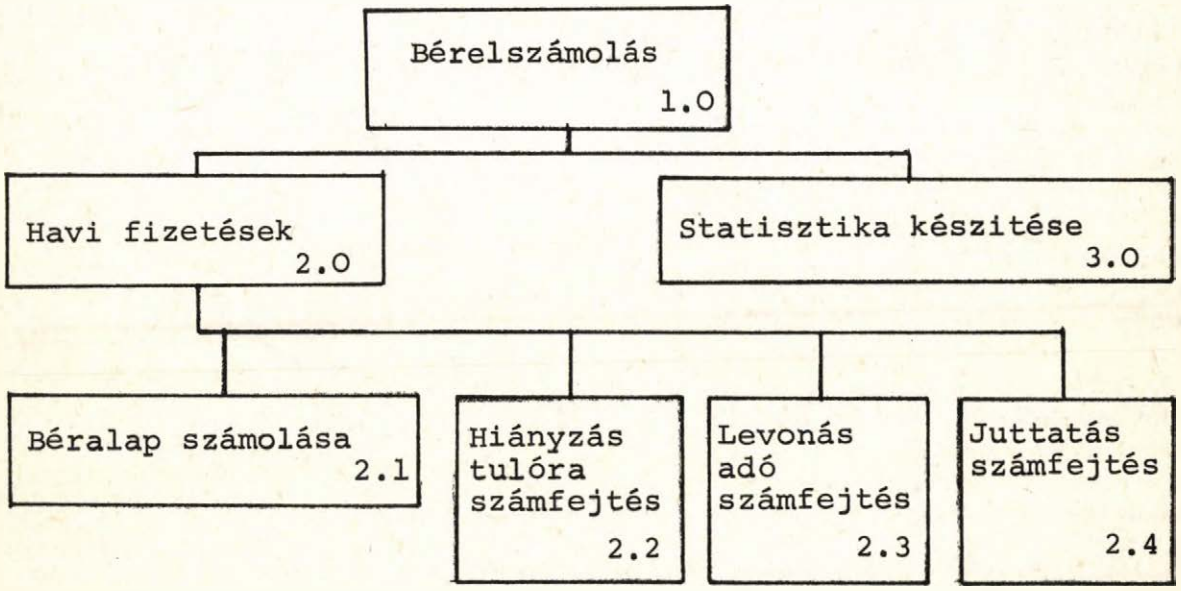
1.1.2.2. HIPO

A HIPO /Hierarchy plus Input-Process-Output/ technológia széles körben elterjedt /mint általában az IBM támogatta rendszerek/. Eredetileg dokumentációs eszközként fejlesztették ki /az OS/VS Program Logic Manualok HIPO technikával készültek, ld. pl. [30]/, de használható a rendszerfelmérés és tervezés fázisaiban is, igy módszernek tekinthető.

A HIPO módszertani alapja a felülről lefelé tervezés filozófiája: a feladatot megfogalmazásában részfeladatokra kell bontani, a részfeladatok mindegyikét kisebb részfeladatokra és igy tovább az elemi, további bontást nem igénylő egyszerüen megoldható feladatokig. A részfeladatokra bontás természetesen általában nem egyértelmü: az optimális felbontás érdekében érdemes a strukturált tervezésben használt kritériumokat, a kapcsolódást és a kohéziót használni [31].

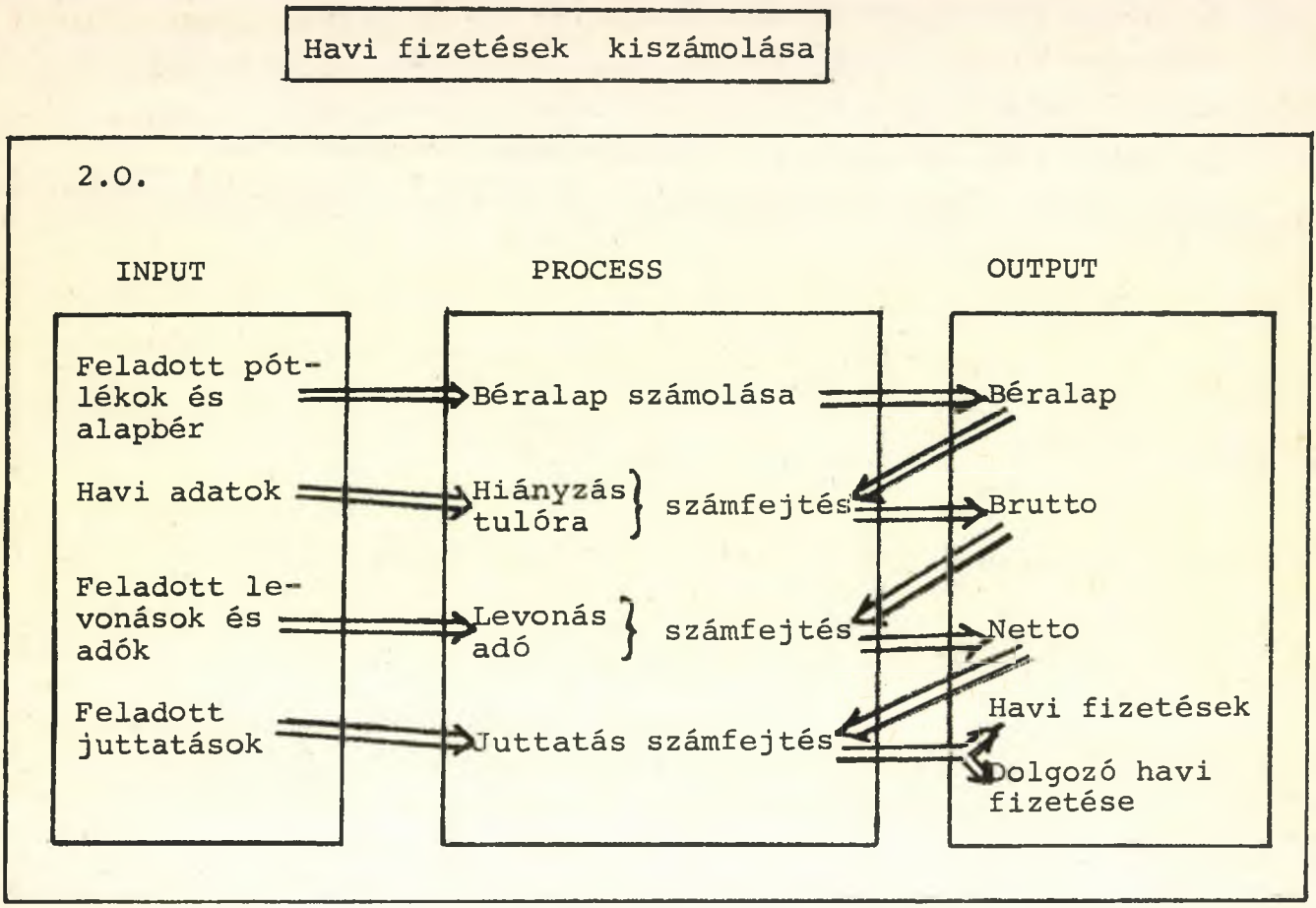
A technológiai rész eléggé egyszerü: a HIPO két ábratipust használ. Az egyik a hierarchikus diagram, mely az egyes részek helyét mutatja a hierarchián belül, a feladatok részletezése nélkül. Az egyes részek azonositása névvel és/vagy azonositó számmal történik. A másik ábra az immár hierarchikus rendszerbe illesztett részfeladat részletezése. Ez a feladat lépéseinek leirását,

valamint azok input és output adatait tartalmazza /input-process-output diagram/. Példaként az 1.1.1.4.-ben már vizsgált egyszerü példa hierarchikus diagramját /5. ábra/, és az egyik rész input-process-output diagramját /6. ábra/ mutatjuk be.




5. ábra

"Bérelszámolás" hierarchikus diagram




6. ábra
"Havi fizetések kiszámolása" input-process-output diagram

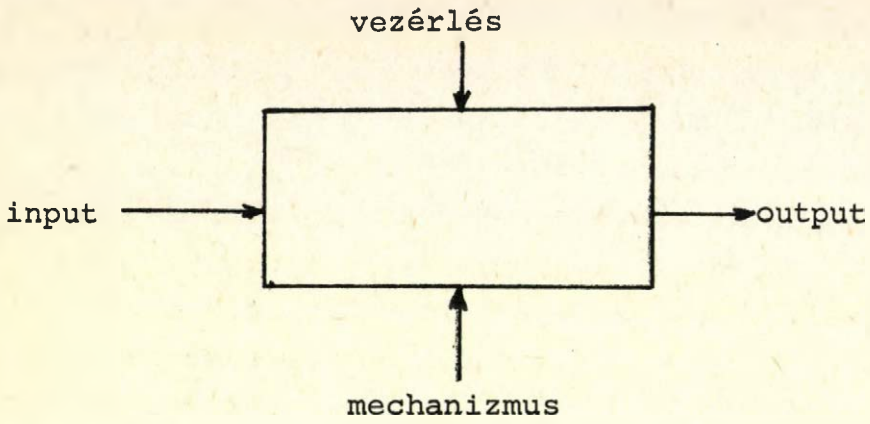
A HIPO diagramjai fokozatosan épithetők fel a felmérés és a tervezés fázisaiban. A folyamat iterativ jellegü, de a tendenciának - az eszköz helyes használata mellett - fokozatosan felülről lefelé kiépülő, egyre részletesebbé váló diagramokat kell eredményeznie.

A HIPO elsősorban dokumentációs eszközként jelentős. Nagy előnye egyszerüsége, könnyen elsajátitható. Alkalmazható a felmérés és a tervezés során is, hidat képezve a két fázis között. /Előfordulhat, hogy a részletesebb tervezés során előkerülnek olyan problémák, melyek miatt a rendszer egy részét ujra kell tervezni. Arról van szó ugyanis, hogy az optimális modulszerkezet nem feltétlenül esik egybe a feladatok felmérés során keletkezett felbontásával, igy uj felbontást kell késziteni a strukturált tervezés elveinek fokozottabb figyelembevételével./ A hierarchikus felbontás támogatja az absztrakciót, az input-process-output diagram elősegiti a rendszer modularitását. Hátránya a HIPO-nak viszonylagos kötetlensége, nincs formális lehetősége az interface-ek és a felbontás helyességének ellenőrzésére.

1.1.2.3. SADT

Az SADT-t /Structured Analysis Design Technique/ alkotói általában bonyolult rendszereket modellező eszköznek szánták, és valóban, használata nem korlátozódik adatfeldolgozási problémákra, általános módszerként alkalmazzák a müszaki élet különböző területein. Maga a technológia grafikus, és sokban hasonlit az adatáramlás ill. a struktura-diagramra, noha azoknál általánosabb és egységesebb.

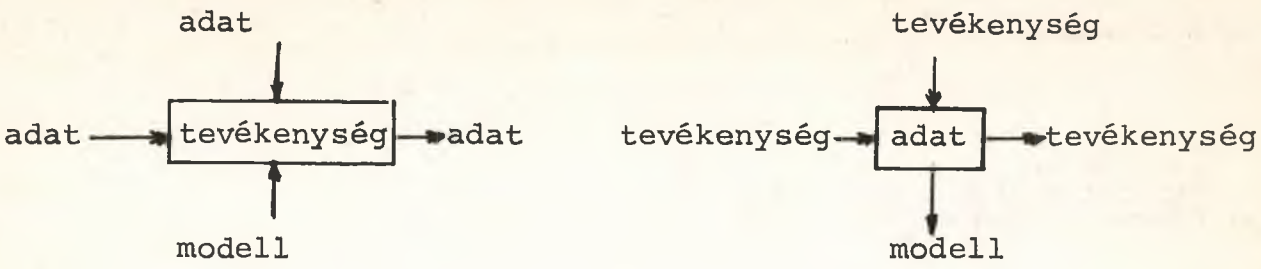
Az SADT dobozokból és nyilakból épitkezik. Mindegyik doboznak és nyilnak egyedi neve van. A dobozok és nyilak szerepét - általában - a 7. ábra [32] szemlélteti:




7. ábra
Dobozok és nyilak az SADT-ben.

A nyilak geometriai elrendezése határozza meg jelentésüket. A balról érkező nyil mindig a doboz inputja, a jobbra távozó output-ot jelent. A felülről érkező nyil az input-output átalakitás vezérlése, az alsó pedig az átalakitást biztositó mechanizmus.

A dobozoknak kettős jelentésük van, az SADT kettős diagramrendszerének megfelelően. A modellezendő rendszert ugyanis, az SADT két szemszögből, a tevékenységekéből és az adatokéból vizsgálja. A tevékenységek szemszögéből modellezett rendszerben a dobozok tevékenységeket, az adatokéból modellezettben pedig adatokat jelentenek. A 8. ábra tulajdonképpen a 7. ábra két speciális esete.



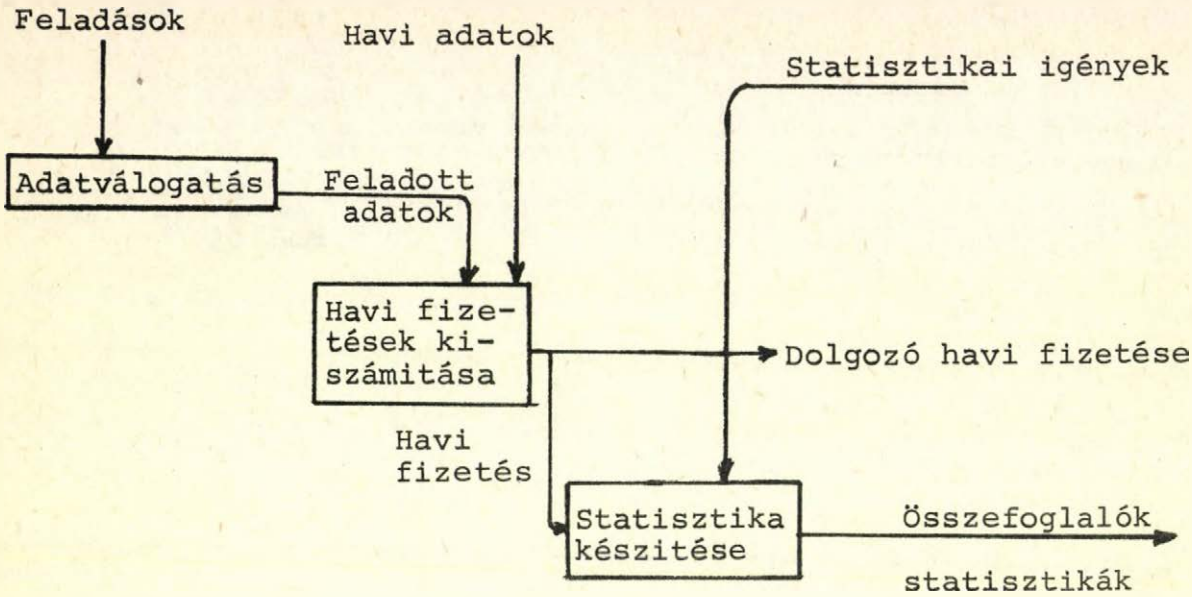

a/ b/

8. ábra

Nyilak és dobozok kettős jelentése.

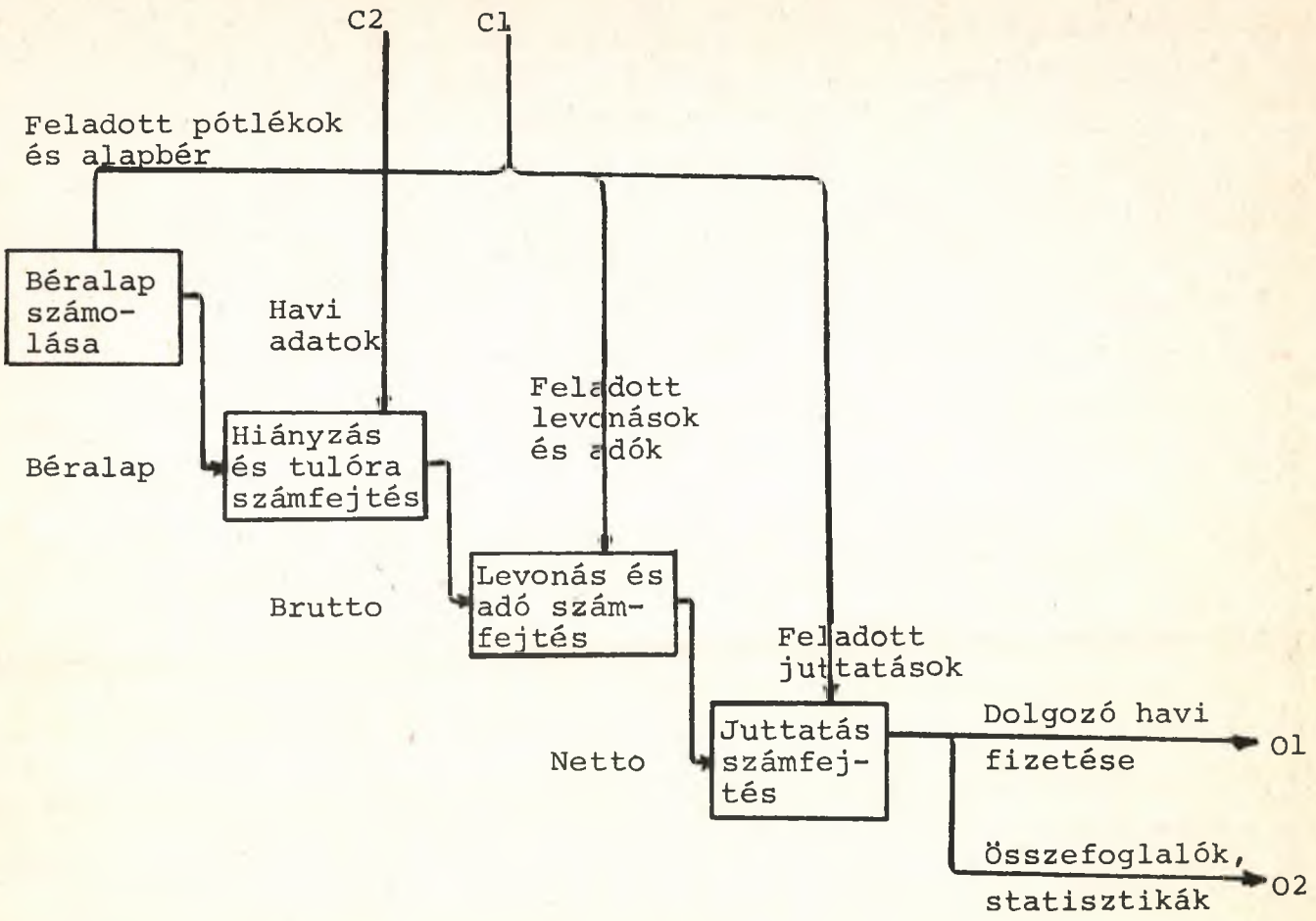
A 8.a/ ábrán látható rajzból, mint alapegységből felépitett, tevékenység-szemszögü SADT leirásokat tevékenység-diagramnak /actigram/ nevezzük. Az adat-szemszögü leirás neve adat-diagram /datagram/, alkotóelemeit a 8.b/ ábra mutatja.

A dobozok nyilakkal történő összakapcsolása elég nyilvánvaló: egy doboz outputja valamely más dobozba mint input , vagy mint vezérlés futhat be. A 9. ábra a bérelszámolás SADT tevékenység diagramja.




9. ábra
A bérelszámolás tevékenység diagramja

Az SADT egyik szabálya szerint egy ábrán 3-6 doboz szerepelhet. A rendszer tervezése természetesen itt is felülről lefelé történik, a részekre bontás uj szintek, uj ábrák bevezetésével történik, minden szint valamennyi ábrájánál megőrzi a 3-6 dobozos méretet. A "Havi fizetések kiszámitása" tevékenység lebontása a 10. ábrán látható:




10. ábra
"Havi fizetések kiszámitása" tevékenység diagram

Az SADT jellegzetessége, hogy a technológia és a tervezési filozófia /felülről-lefelé/ mellé szervezeti, irányitó mechanizmust is előir: pontosan megjelölt szerepek várnak a projekt résztvevőire /szervező, szakértő, könyvtáros, stb./ gondos előirások szabályozzák a munka menetét, a dokumentumok készitésének, ellenőrzésének,

jóváhagyásának módját. Jellemző a "szerző-olvasó" /author-reader/ ciklus, melynek során az elkészült diagramot egy vagy több, a modellezendő témát ill. az SADT-t ismerő szakember átnézi és ehhez megjegyzéseket füz. Feltétlenül irásban, ugyanis, ez is előirás.

Az SADT-t a manuális rendszerek között tartják számon, mivel igy terjedt el. Feltétlenül emlitésre méltó, azonban, hogy biztató kisérletek történtek számitógépes támogatásának biztositására [22].

### 1.1.3. Számitógépes technológia

A rendszertervezési technológiák "elszámitógépesedése", azaz a számitógép egyre növekvő mértékü felhasználása a rendszer életciklusának korai szakaszaiban tulajdonképpen magától értetődő jelenség. A felmérés során a modellezett rendszer müködését leiró adatokat kell gyüjteni, megjegyezni, csoportositani. Ezek elemzése alapján születik meg a jövendő számitógépes rendszer felépitését tartalmazó logikai rendszerterv. A tervezés fázisában ez tovább finomul, implementációs részletekkel bővül.

Adatok gyüjtésére, tárolására, adott szempontok szerinti csoportositására a számitógép jól alkalmas. A számitástechnika egyes ágainak eredményei készen alkalmazhatóak. A formális nyelvek, a forditóprogramok segitségével lehetővé válik az adatok számitógépre vitele, a felvitt adatok módositása, a konzisztencia ellenőrzése. Adatbáziskezelő rendszer gondoskodik a hatékony tárolásról, lekérdező nyelvével előre adott, vagy akár ad hoc lekérdezések végezhetők, automatikusan csoportositott, válogatott adatokról.

A gép pontossága, gyorsasága ezen a területen is

jól kihasználható. Nem felejti el, nem keveri össze az adatokat, kiszüri az ellentmondásokat, ellenőrzi a megadott feltételek teljesülését, stb. Képes rövid időn belül áttekinthető listákat adni a rendszert készitő egyént éppen érdeklő adatokról, azokat válogatva ki az adathalmazból melyek szükségesek.

A fejlődés a számitógép irányába mutat tehát. A számitástechnika belső fejlődése /interaktiv rendszerek, egyre olcsóbb mikrogépek, számitógéphálózatok, stb./ ezt a folyamatot támogatja. Egyes kézi eszközök, technológiák is egyre fokozottabb számitógépes támogatást kapnak /specifikációs nyelvek, SADT, stb./, a számitógépes tervező rendszerek pedig egyre fejlettebbek, egyre általánosabb és magasabb igények kielégitésére képesek. A következőkben a számitógépes technológiát a legelterjedtebb rendszer, a Michigan Egyetemen az ISDOS project keretében kifejlesztett PSL/PSA [36] példáján keresztül mutatjuk be.

Először a rendszernek a felhasználó számára legfontosabb alkotórészével, a leiró nyelvvel /PSL, Problem Statement Language/ foglalkozunk. Ezen a nyelven adhatja meg a felhasználó a modellezendő rendszer leirását, a felmérés során és az életciklus további fázisaiban összegyült adatokat.

Tételezzük fel, hogy egy vállalat számára bérügyi rendszert kviánunk szervezni. A PSL módszertana - természetesen a felülről-lefelé tervezés - szerint először elegendő a feladat hozzávetőleges meghatározására. Ez magyarul pl. a következőképpen fogalmazható meg:

A bérügyi rendszer felhasználva a munkaügyi osztályról érkező havi adatokat, kiszámitja a fizetéseket, és különböző listákat, statisztikákat készit, melyeket a

munkaügyi osztály és a vállalat dolgozói kapnak kézhez. A rendszer müködése során kialakitja a munka és bérügyi törzsállományt /v.ö. 1.1.1.4. 2. ábra/.

A rövid leirásban objektumok és a közöttük lévő kapcsolatok nevei szerepelnek. Az objektumoknak és kapcsolataiknak - implicite - tipusuk van. Az objektumok és kapcsolatok tipusai egymást definiálják, hasonlóan az absztrakt adattipusok axiomatikus definiálásához /ld. pl. [35] 3.5./. Esetünkben objektumok pl. a "havi fizetések kiszámitása", "munkaügyi osztály", "havi adatok", stb. és ezek kapcsolatait jelölik a "felhasználva", "érkező", stb. szavak.

A PSL nyelv az információs rendszerek leirásához jól átgondolt tipus és kapcsolatrendszert bocsájt a felhasználó rendelkezésére. A példában szereplő leirás pl. közvetlenül megadható PSL-ben:

```
PROCESS bérügyi rendszer;
   UTILIZES havi fizetések kiszámitása, statisztika
   készitése;
PROCESS statisztika készitése;
   USES havi fizetések;
   GENERATES összefoglalók, statisztikák;
   UPDATES statisztikai adatok;
INTERFACE munkaügyi osztály;
   GENERATES havi adatok;
   RECEIVES összefoglalók, statisztikák;
   INTERFACE dolgozó;
   RECEIVES dolgozó havi fizetése;
SET törzsadatok;
   USED BY havi fizetések kiszámitása.
```

11. ábra

"Havi fizetések kiszámitása" PSL nyelvü leirás

Az ábrán jól látható az objektumok és egymáshoz való viszonyuk /a kapcsolatok/ megadásának módja. A leirás szekciókra tagolódik, ahol minden szekció egy objektumot ir le. A szekció első állitása definiálja az objektumot ⟨tipusnév⟩⟨objektumnév⟩; szintaktikával /pl. SET törzsadatok;/. A szekción belül szereplő állitások /pl. RECEIVES dolgozó havi fizetése;/ a definiált objektum és más objektumok közötti kapcsolatokat irják le. Egy-egy ilyen állitás egyben az állitásban szereplő objektum tipusát is megadja, pl. a "dolgozó havi fizetése" OUTPUT tipusu objektumként definiálódik a "GENERATES" állitásban való szerepeltetésével.

A 11. ábra leirása minden további nélkül számitógépbe vihető és lista kérhető róla, pl. a FORMATTED PROBLEM STATEMENT PSA /Problem Statement Analyser/ parancs segitségével. Itt már némi hasznát vehetjük a számitógép használatának. A lista ugyanis ugyanolyan formátumu lesz, mint a 11. ábrán látható, de az ottani 14 sor helyett jóval hosszabb lesz, ugyanis a rendszer minden objektum minden kapcsolatát visszaadja. Tehát például mig felvitelnél a "statisztika készitése" PROCESS-nél adtuk csak meg, hogy "GENERATES" összefoglalók, most az "összefoglalók" nevü, OUTPUT tipusu objektumnál - természetesen külön szekciója lesz - is meg fogjuk találni a megfelelő inverz bejegyzést, vagyis, hogy "GENERATED BY statisztika készitése;". Egy objektum valamennyi kapcsolata tehát egy helyen megtalálható, nem kell keresgélni a leirásrészleteket. A lista természetesen redundáns lesz, de a gépi ellenőrzés miatt garantáltan ellentmondásmentes. Hibaellenőrzés is történik: ha pl. kihagytuk volna a 11. ábra leirásából a "GENERATES összefoglalók" állitást, a gép figyelmeztető üzenetet adott volna, hiszen az

"összefoglalók" objektumot egy másik /"munkaügyi osztály"/ használja /RECEIVES/, de nem szerepel előállitó /"GENERATES"/ állitásban, vagyis nincs rögzitve, hogy hogyan került be egyáltalán a rendszerbe. /Ez a probléma a 11. ábra "dolgozó havi fizetése" objektumnál valóban jelentkezik is, a hiányos leirás miatt./

A felvitt leirás természetesen folytatható és továbbfejleszthető. A PSL/PSA alkotói a felülről lefelé történő tervezést ajánlják /de nem teszik kötelezővé, a nyelvben nincs erre kényszeritő eszköz/ [34]. Részletezzük példánk leirását az 1.1.1.4. pont 3. ábrájának megfelelően.

```
PROCESS havi fizetések kiszámitása;
   UTILIZES béralap számolása, hiányzás és tulóra
   számfejtés;
   UTILIZES levonás és adó számfejtés, juttatás
   számfejtés;
   UTILIZES adatválogatás;
   DERIVES havi fizetések, dolgozó havi fizetése;
PROCESS adatválogatás;
   USES törzsadatok;
   DERIVES feladott pótlékok és alapbér, feladott
   levonások és adók, feladott juttatások;
PROCESS béralap számolása;
   USES feladott pótlékok és alapbér  TO DERIVE béralap;
```

12. ábra

"Havi fizetések kiszámitása" PSL PROCESS finomitása

Ez természetesen csak az ábra egy részét irja le, az ábrában szereplő összes objektum és kapcsolat felsorolása elég hosszadalmas volna. Az mindenképpen kiderül az ábráról, hogy a "havi fizetések kiszámitása" tevékenységet /"PROCESS"/ az "UTILIZES" állitással négy résztevékenységre bontottunk. A PSL nyelv természetesen az adatok hasonló felbontását is lehetővé teszi, mint az a 13. ábrán látható.

```
OUTPUT összefoglalók;
  SUBPART munkaügyi összefoglaló, bérügyi összefoglaló;
INPUT havi adatok;
  SUBPART munkaügyi adatok, bérügyi adatok;
SET munka és bérügyi törzsállomány;
  SUBSET statisztikai adatok, törzsadatok;
```

13. ábra
Adatok finomitása a PSL-ben

A PSL állitások aspektusok szerint csoportosithatók. A következőkre fogalmazhatók meg állitások:

- a rendszer folyamatának leirása /RECEIVES,GENERATES, stb./,
- a rendszer strukturájának leirására /SUBPART, UTILIZES, stb./,
- az adatstrukturák leirására /COBOL-szerü rekordleirás a GROUP,ELEMENT tipusu objektumokból álló ENTITY,INPUT,OUTPUT tipusu objektumokról/;
- az adatszármaztatás leirására /USES,DERIVES/,
- a rendszer idő és térbeli terjedelmének leirására,
- a rendszer dinamikájának leirására /EVENT, TRIGGERS,stb./,

- a rendszer tulajdonságainak leirására /ATTRIBUTE, KEYWORD/;
- a rendszerterv kezelésének leirására /PROBLEM-DEFINER állitással adhatók meg az egyes részekért felelős szervezők, ill. programozók nevei/.

A PSL állitásokat az interpreter mágneslemezen tárolja. A beolvasott leirás helyességét nemcsak önmagában, hanem a már tárolt régebbi adatokkal való összefüggésében is ellenőrzi.

A tárolt információ /mint már láttuk/ bővithető, ezen kivül természetesen módositható, lekérdezhető is. A lekérdezés során különböző tartalmu és formátumu listák nyerhetők. Emlitettük korábban a FORMATTED PEOBLEM STATEMENT parancsot, mely PSL nyelvü listát készit az adatokról, ill. azok kijelölt részhalmazáról /pl. az összes INPUT tipusu objektumról/. Látványos, grafikus lista a PICTURE, mely az adatáramlást rajzolja ki, némileg az adatáramlás diagramhoz hasonlóan. Az adatok szerkezetének listázására több parancs is szolgál, különféle grafikus és szöveges adatszerkezet diagramok nyerhetők. A DICTIONARY parancs adatszótár-szerü formában irja ki a rendszerben /ill. meghatározott részhalmazában/ szereplő adatok neveit /érdekes, hogy [5] a PSL/PSA adatszótár funkcióját emeli ki, és mint ilyet emliti a rendszert, noha ennél sokkal többet tud/.

A PSL/PSA rendszer - az eredeti elképzelések [36] szerint - az életciklus felmérés és tervezés, továbbá karbantartás /jól áttekinthető és könnyen módositható, konzisztens dokumentáció/ szakaszaiban alkalmazható. Több továbbfejlesztése /SODA[37], Software Development Facility [38], stb./ is megcélozta a programozás fázisának segitését /a SODA automatikus kódgenerálási lehetősége annak teljes gépesitését/, de ezek a rendszerek nem arattak olyan sikert, mint az eredeti.

## 1.2. A tervező rendszerek általános felépitése

A legegyszerübb tervező rendszer a szövegszerkesztő. Valóban a tervezendő rendszer leirása /ez grafikus, vagy táblázatos is lehet/ gépbe vihető, tárolható. A tárolt szöveg módositható, terminálra iratható, lista készithető róla, s a fejlettebb szövegszerkesztő rendszerekben a lista formátuma szabályozható is.

A leglényegesebb különbség a szövegszerkesztő és egy korszerü tervező rendszer között az a mód, ahogy az egyik ill. a másik a bevitt szöveget kezeli. Az előbbi számára a leirás szöveg, melynek csak formája létezik, a tartalma közömbös. A tervező rendszer a bevitt leirást nem csak tárolja és visszairja a terminálra vagy listázza, hanem értelmezi és elemzi is azt.

Ehhez először persze a leirandó rendszer modelljének megalkotása szükséges. A leirás maga a modellben definiált fogalmak segitségével történik. Ezek a fogalmak a rendszert alkotó objektumok és kapcsolataiknak osztályokba sorolásával egy absztrakciós folyamat eredményeként alakulnak ki. A létrejött osztályok lesznek majd a leirásban szereplő objektumok és kapcsolatok tipusai /a PSL-ben pl. INPUT, ELEMENT, USES, DERIVES, stb./. A tipusok és kapcsolataik rögzitésével és egy - lehetőleg egyszerü és könnyen olvasható - szintaktika megadásával már lehetőség van a szöveg gépi elemzésére, létrejön a tervező rendszer legfontosabb alkotórésze, a leiró nyelv.

A leiró nyelv definiciója döntő jelentőséggel bir a rendszer használata szempontjából. Itt dől el az, hogy a leirás természetesen fogalmazható-e meg az adott nyelven, vagy a valóságos objektumok és összefüggéseik a bevezetett fogalmakkal csak nehézkesen, "belemagyarázásokkal" irhatók-e le.

A leiró nyelv részét alkothatják egyes szemantikai ellenőrzések is. A PSL pl. ellenőrzi, hogy a használt /USED BY/ objektum kivülről került-e be a rendszerbe /GENERATED BY/, illetve ott jött-e létre /DERIVED BY/ és ha egyik állitásban sem szerepel, hibát jelez. Ugyancsak ellenőriz egyes 1:N kapcsolatokat, továbbá más bonyolultabb ellenőrzéseket is végez.

A PSL nyelv 22 objektum és 57 kapcsolattipust tartalmaz. A nyelv univerzális, - alkotói legalábbis annak szánták [36]- elvileg tetszőleges információs rendszer felméréséhez, tervezéséhez és karbantartásához ad segitséget. Az objektum és kapcsolattipusok rögzitettek, akárcsak a szemantikai jellegü megkötések, tehát azokon nem változtathat, nem vezethet be uj objektumtipusokat, vagy nem módosithat egy kapcsolat jellegén. A leirásra felhasználható eszközök - a PSL esetében nagyon alaposan átgondolt, rendszerszervezői szemszögből elég általános alkalmazásokat biztositó - megadása determinálja a rendszer használhatóságát. Egyrészt könnyit a felhasználó helyzetén: nem neki kell a nyelv definiálásának komoly gyakorlatot igénylő, rutinmunkának nem nevezhető feladatát elvégezni. Másrészt viszont mereven meghatározza azt a sémát, melyben gondolkozni kell.

A leirásban csak a nyelvben definiált tipusu objektumok és kapcsolatok szerepelhetnek. A PSL/PSA rendszer használata meggyőzött róla, hogy igen nehéz kialakitani olyan fogalomrendszert, mely nem tul általános ahhoz, hogy elég mély elemzéseket produkáljon, sem pedig nem tul speciális, egy-egy célfeladatra orientált fogalmak gyüjteménye. A probléma némileg a programozási nyelvek körében már korábban, a 60-as évek végefelé felmerült univerzális nyelv, célorientált nyelv választásra emlékeztet. Az univerzális nyelvekről kiderült, hogy tulzottan

komplikáltak, áttekinthetetlenül nagy eszközkészletet bocsátanak a felhasználó rendelkezésére /PL/1/, a célorientáltak alkalmazási lehetősége pedig tul szük, ill. nehézkesen oldhatók meg velük viszonylag egyszerü feladatok is /számolás COBOL-ban/.

A tipusok és az ellenőrzések szemantikai jellegü definiciója - alapvetően ezek határozzák és különböztetik meg az egyes nyelveket - mellett a leiró nyelveknek az elemezhetőség érdekében a szintaktikát is pontosan meg kell határozniuk. Itt is eltérő irányzatokkal találkozhatunk. A PSL vagy pl. a PROTEE [40] szöveges leirások gépre vitelét teszi lehetővé. /Ez a legelterjedtebb megoldás./ A számitógépes SADT [22] grafikus display-ről olvas be, elemez és tárol tevékenység és adat-diagramokat. Az ARIUS [41] táblázatok formájában várja az adatokat. Nyilvánvaló, hogy a szintaktika is lényegesen befolyásolja a rendszer használhatóságát, célszerü minél kényelmesebb lehetőséget biztositani a - sokszor nagyméretü - leirás gépre vitelére.

A felhasználó leirását annak interpretálása, ellenőrzése után tárolni kell a rendszer adatbázisában. A tárolás módja döntő jelentőséggel bir a hatékonyság, tehát a használhatóság egyik nagyon fontos tényezőjének szempontjából [33]. Az adatkezelés tehát a tervező rendszer második, jelentős összetevője.

Nyilvánvalónak tünik, hogy az adatkezelési funkciókat célszerü elválasztani a rendszer egészétől. A tárolt adatok módositása, lekérdezése több, különböző feladatokat ellátó, egymástól elkülönitett modulból történhet. Ezeknek közös része az adatkezelés, igy ezt nem érdemes mindegyik modulhoz külön megirni.

Az adatkezelés tervezésénél figyelembe veendő szempontok nem különböznek lényegesen az adatkezelő rendszerek

esetében általában szokásosoktól. A két fő szempont, a hatékonyság és a minél könnyebben változtatható, minél több lehetőséget biztositó felhasználói interface között kell megfelelő kompromisszumos megoldást találni. /A "felhasználói" jelző itt természetesen nem a tervező rendszer felhasználójára, hanem az adatokat használó, a tervező rendszer részeit alkotó programokra vonatkozik./ A szempontok mérlegelésénél egy felől azt kell figyelembe venni, hogy a rendszert sürün használják, korszerü rendszereknél terminálról, egyidőben többen, tehát a lassusága használhatatlanná tenné, másfelől viszont várhatóan uj listázási, lekérdezési igények merülnek fel a felhasználók részéről. A hatékonyság tehát a mindennapi gyakorlat szempontjából, a rugalmas adatkezelő interface pedig a rendszer áttekinthetősége és továbbfejleszthetősége szempontjából jelentős.

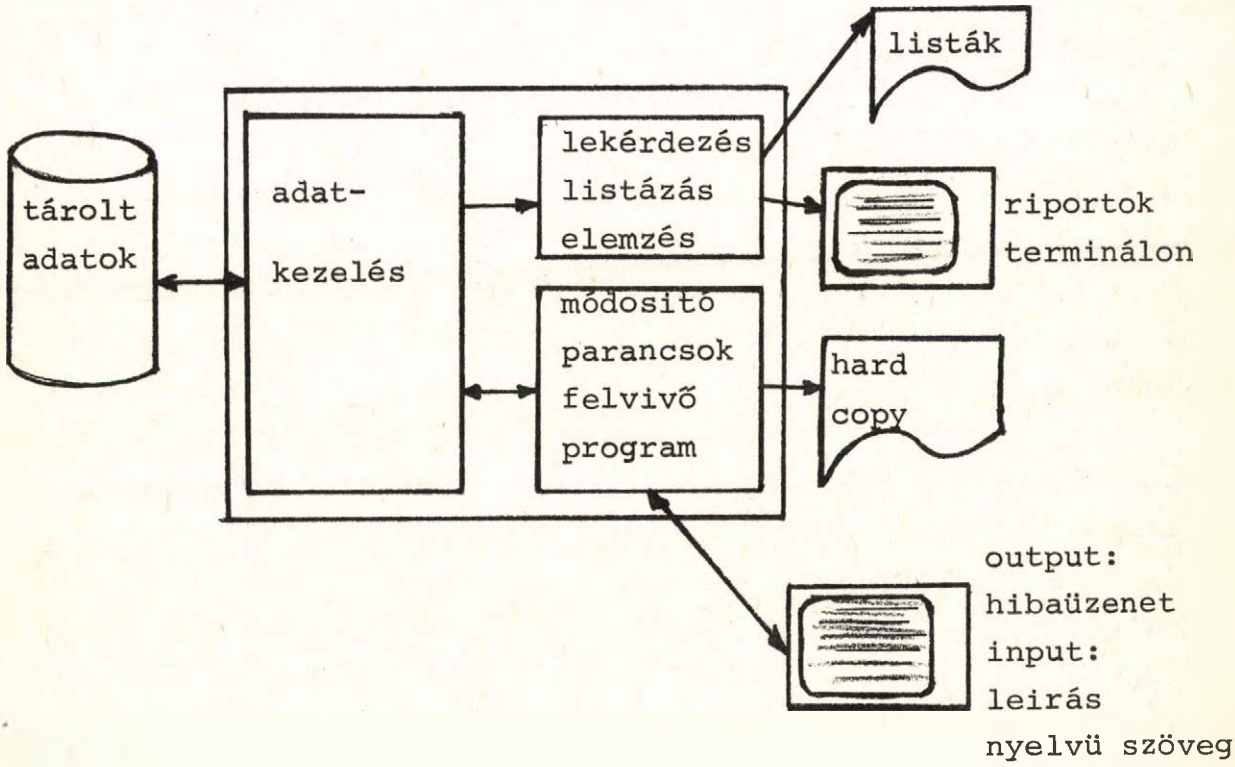
A tárolt adatokhoz való hozzáférés az azok módositását és lekérdezését biztositó parancsrendszeren keresztül történik. A PSL/PSA esetében ez a PSA, mely több, mint 20 parancsot tartalmaz.

A módositó parancsoknak lehetővé kell tennie a tárolt objektumok, kapcsolatok törlését, javitását, nevek megváltoztatását, stb. A felhasználó szempontjából az a kényelmes megoldás, ha nem kell külön módositó nyelvet megtanulni, hanem a leiró nyelv használatával végezheti a módositásokat.

A lekérdező és listázó parancsok sokféle feladatot látnak el. Más tipusu választ igényel a leirást a terminál mellett fejlesztő szervező, aki nem emlékszik valamilyen részletre, mást a projekt vezetője, aki a munka állásáról szeretne képet kapni, megint más feladat a rendszer dokumentációjának előállitása, stb. Ugy tünik,

hogy a tervező rendszernek ez a felhasználóhoz közeli része állandó továbbfejlesztést, módositást igényel. Célszerü igy is szervezni, általános listadefiniálási lehetőséggel, vagy megfelelő segédeszközöket biztositva uj parancsok viszonylag egyszerü programozásához /pl. könnyen használható, rugalmas adatkezelő interface-t./

A tervező rendszer alkotórészeinek kapcsolata a 14.ábrán látható




14. ábra. A tervező rendszer alkotórészeinek kapcsolata

## 1.3. A tervező rendszerek használata

Manuális technológiát, eszközöket használva a felmérés és a tervezés szakaszában az adatok, ill. a rendszerterv papiron, pl. az 1.1.1.-ben ill. 1.1.2-ben leirt eszközök meghatározta módon készül. A tervező rendszer használata a szoftver életciklusának különösen ezekre a szakaszaira hat, jelentősen megváltoztatja a munka jellegét, előnyökkel és hátrányokkal jár. A Bevezetésben egy sor problémát vázoltunk, mellyel a szervező szemben találja magát a rendszerterv készitése során. Most megkisérelünk a tervező rendszerek kinálta lehetőségekkel a problémákra megoldást, illetve megoldási módszert javasolni.

A tervező rendszer használata már a munka kezdetén a felmérés legelején kezdődhet /ld. 1.1.3./. A tapasztalatok szerint itt rögtön két hátránnyal kell számolni.

A dolog természetéből adódóan a rendszerszervezőnek intenziven használnia kell a számitógépet, hiszen minden uj adatot, összefüggést azonnal gépre kell vinnie. Ez a leghatékonyabb rendszer használata mellett is komoly gépidő, tehát a kézi technológiákhoz képest többletköltség. Megjegyzendő még, hogy tapasztalataink szerint tervező rendszert felmérési szakaszban csak interaktiv módon lehet használni. Gyakorlatilag nehezen képzelhető el ugyanis, hogy a szervező, aki intenziv felmérést végez, tehát nap mint nap nagytömegü uj információt szerez a modellezendő rendszerről, képes legyen ezzel párhuzamosan a kötegelt feldolgozásban megszokott legalább egy napos késéssel /lyukasztási idővel, gépleállással, stb. együtt ez akár egy hét is lehet/ gépre vinni az adatait, és főképpen az adatfelvitelről kapott lista /hard copy/ alapján javitgatni a hibákat

/miközben már ujabb adatokat kéne bevinni/.

A másik hátrány - lehet - a leiró nyelv. Könnyebb kötetlen formában megfogalmazni valamit, mint szigoru szabályok szerint, korlátozott szókinccsel, pontos szintaktikával, programszerüen leirni.[9] a PSL/PSA hátrányaként emliti nehezen tanulhatóságát /mellesleg ezt a problémát a nem számitógépes SADT-nél is emliti/. A fő nehézség tulajdonképpen nem a kulcsszavak és a szintaktika megjegyzése, hanem a stilus és gondolkodásmód elsajátitása. Az egész hasonlit egy programozási nyelv megtanulásához,ott sem a kódolás, hanem a gépi gondolkodásmód megértése, az algoritmusalkotási készség a fontos.

Ugy gondolom azonban, hogy az igazi nehézség nem ez. A leiró nyelv használatának problémáját az teszi - teheti - valóban sulyossá, hogy a nyelv általában nem pontosan a modellezendő rendszer leirására készült, nem eléggé "testreszabott". A kötött leiró nyelvvel rendelkező tervező rendszerek esetén ezt a problémát igen nehéz - általánosságban lehetetlen - megoldani. /Egy uj PSL állitás rendszerbe illesztése több emberhetes munkát jelent [33]. Ez a tény erősen motiválta a tervező rendszerek általános generálási lehetőségei irányába folytatott kutatásainkat./

A leiró nyelv használata azért alapvetően nem hátrány, hanem előny, még akkor is, ha nem a számitógép használatával járó szükséges rosszként tekintjük. Arról van szó ugyanis, hogy a szigoru szabályok szerint épülő leirás nem csak az ember-gép, hanem az emberek közötti információcserét is egyértelmübbé, világosabbá teszi. A rendszerszervező leiró nyelven készitett specifikációja félreérthetelenebb /és általában rövidebb/, mint a strukturálatlan szöveg, és a rendszer megrendelője könnyebben

tájékozódhat arról, hogyan látja a szervező a modellezendő rendszert, és milyen szolgáltatásokat ajánl a készitendő gépi rendszer. /[5] ez az előnyt általában a magasszintü specifikációs nyelvekkel kapcsolatban emliti./ Másfelől a leiró nyelv pontosabb programspecifikációt tesz lehetővé, igy a rendszerszervező kevésbé félreérthetően közölheti elgondolásait az azt realizáló programozóval. /Ez is igaz általában a magasszintü specifikációs nyelvekre./

A tervező rendszer leiró nyelvének használata természetesen nem önálló specifikációs nyelvként, hanem számitógépet használva előnyös igazán. A felmérés szakaszában gyorsan felgyülemlő adathalmaz áttekinthetőségét nagymértékben javitják azok a kigyüjtések, melyek manuálisan nem végezhetők el. A gép figyelmeztet a hiányzó adatokra, szigoruan ellenőrzi az adatdefiniciók konzisztenciáját. A felmérést nagyobb rendszerek esetében nem egyes ember, hanem egy csoport végzi. Ha a felmérés teljes anyaga a rendszerben tárolódik, a szervező team tagjainak kapcsolata egyszerüsödik, könnyebb elkerülni az átfedéseket, félreérthetetlenebbé válnak az interface-ek definiciói, az egymással való kapcsolat tartására szükséges idő csökken.

A tervezési fázisban - noha a tervezés kreativ folyamat, és igy nem gépesithető - is értékes segitséget tud adni a számitógép. A jó tervező rendszer képes bizonyos logikai ellentmondások kiszürésére. Ilyen ellentmondások főképpen olyankor keletkeznek, amikor a már kész vagy annak vélt rendszerterven - általában a megrendelő kivánságára - változtatni kell. A változások, és azok következményeinek végigvitele a rendszeren nyilvánvalóan egyszerübb a tervező rendszer biztositotta lehetőségek kihasználásával - melyek közül az egyik legfontosabb az

ellenőrzés.

A tervező rendszer egyik nagyon nagy előnye, hogy használatának során automatikusan létrejön a modellezett rendszer dokumentációja. A dokumentálás fárasztó és hosszadalmas folyamata a megfelelő listázások elvégzésére egyszerüsödik /rossz esetben a listázó programok megirására, hiszen nem tartalmazhat minden dokumentálási szabványra listázó programot egy rendszer/. Az igy készült dokumentáció - a gépi ellenőrzés miatt - pontosabb, ellentmondásmentesebb és teljesebb, mint a kézi. A karbantartás során komoly előnyt jelent, hogy amikor változtatni kell a rendszeren, a változás a tervező rendszer adatbázisában végezhető el /ennek előnyeiről már volt szó/, és az uj dokumentáció automatikusan generálható, nem a régit kell átirni.

## 1.4. Tervező rendszerek fejlesztése

A tervező rendszer maga is szoftver - méghozzá elég komplex - fejlesztése során ugyanazok a problémák merülnek fel, mint általában a szoftvergyártás során. Az életciklus modell a tervező rendszerek esetében is használható.

A felmérés fázisának meg kell határoznia a megoldandó feladatot. A tervező rendszer lehet célfeladatra orientált, mint pl. a CADES [42], melyet az ICL System VME/B operációs rendszer projektjének támogatására készitettek. Általánosabb feladat megoldása is kitüzhető célként. A PSL/PSA általában információs rendszerek tervezéséhez kiván segitséget nyujtani. Folyamatirányitási rendszerek specifikációjára szolgál az Espreso [17]. Az EPOS rendszer célja eszközöket biztositani valósidejü rendszerek leirásához, továbbá mikrogépes hardver-

fejlesztés támogatása [43]. A kitüzött célok tehát igen sokfélék lehetnek, de mint minden szoftverfejlesztésnél, a feladat pontos megfogalmazása, az igények gondos felmérése döntően befolyásolja a projekt sorsát. Modellt kell készitени, méghozzá annál bonyolultabbat, minél általánosabb a cél. A PSL/PSA-nak például olyan általános eszközöket kell biztositania, mellyel tetszőleges információs rendszer leirható, méghozzá az általánosságokon tulmenő pontossággal. Érdemes megjegyezni, hogy a feladat jellegéből adódóan itt a nem formális eszközökkel, pusztán természetes nyelven alkotott modell használata fokozottan magában rejti a pontatlan és hiányos definició lehetőségét.

A tervezés ad választ a "hogyan müködjék a rendszer?" kérdésre. Itt három fő szempontot kell kiemelni:

- a tervező rendszereket általában nem hivatásos programozók, hanem - alkalmazástól függően - esetleg viszonylag alacsony szintü számitógépes képzettséggel rendelkező emberek használják;
- a tervező rendszert sürün használják;
- a tervező rendszernek rugalmasnak kell lennie, hiszen a listázási lehetőségek bővithetősége mellett /ezeket gyakorlatilag reménytelen minden létező igényt még használható formában kielégitő módon megcsinálni/ felmerülhet a leiró nyelv kisebb-nagyobb módositásának igénye is.

Mindebből következik, hogy a rendszernek könnyen használhatónak, hatékonynak és kellőképpen rugalmasnak kell lennie.

A programozás fázisával kapcsolatosan emlékeztetünk arra a megjegyzésre, hogy a számitástechnika általános fejlődése kedvez a számitógépes tervező rendszerek létre-

hozásának, a konkrét eredmények jól alkalmazhatóak. A PSL/PSA teljes nyelvi elemzőjét az XPL forditóprogram generátor [44] felhasználásával készitették. Az adatok tárolására a CODASYL javaslatnak megfelelő adatbáziskezelő rendszer szolgál [33].

A tervező rendszerek karbantartása a felhasználók igényeitől és a rendszer minőségétől függ. Itt ismét találkozhatunk a már sokszor emlitett problémával, melyet itt élesen igy fogalmazhatunk meg: a tervező rendszer nem tud elég általános és ugyanakkor elég speciális lenni. Ahhoz, hogy a konkrét feladatra alkalmazható legyen, eléggé általánosnak kell lennie. Ahhoz viszont, hogy a megoldás hathatós segitséget adjon, rendelkeznie kell olyan eszközökkel /leiró nyelvi állitások, listák/ melyek egy adott fajtáju rendszer tervezését segitik csak.

Ilyen eszközöket utólag illeszteni müködő rendszerbe nehéz, munkaigényes feladat [33]. A probléma a tervező rendszerek számitógépes tervezésével és megvalósitásával, a rendszerek generálásával oldható meg teljes általánosságban.

## 2. TERVEZŐ RENDSZEREK GENERÁLÁSA

A számitógépes tervező rendszerek használatánál a legnagyobb problémát a modellezendő rendszert alkotó elemek és a tervező rendszer által kinált objektumok tipusainak a megfeleltetése jelenti. A tervező rendszer csupán olyan fogalmak fogadására, tárolására és elemzésére készül fel, melyet a rendszer készitői elképzelnek, és a rendszer létrehozatalakor rögzitenek. Az eléggé általános célu tervező rendszer alkotói azonban - természetesen - nem láthatják előre az összes lehatséges alkalmazásokat, nem számolhatnak valamennyi felhasználói igénnyel.

Vizsgáljuk most meg ugyanezt a problémát a másik oldalról, a tervező rendszer készitője szemszögéből. A tervező rendszer tipusszerkezete a valóság egy darabjának modellezési folyamata során, absztrakcióval keletkezik. A modell, majd ezt követően a tervező rendszer szoftverjének kialakitása során számolni kell a kézi szoftverkészitést általában jellemző nehézségekkel a szoftver életciklusának szakaszaiban.

A legproblematikusabb természetesen ebben az esetben is az életciklus első két fázisa, a felmérés és a tervezés szakasza. A második fázis eredményeként nyert rendszerterv pontatlan, félreérthető lehet, nem feltétlenül tükrözi készitői elképzeléseit, sőt ellentmondásokat is tartalmazhat.

A rendszer specifikációja pontosabbá tehető legfontosabb alkotóeleme, a leiró nyelv formális definiciójával. A programozási nyelvek formális definiciójára sok módszert dolgoztak ki, ezek átvihetők a leiró nyelvekre is.

A leiró nyelv változtathatóságának szükségességéből

és a formális leirás lehetőségéből közvetlenül adódik a generálás gondolata. Ha összehasonlitjuk a helyzetet a programozási nyelvek fejlődésével a 60-as évek vége felé, szembeötlő a hasonlóság: itt is az univerzális - célorientált ellentmondás vezetett a nyelvek gyorsütemü szaporodásához. Figyelemre méltó azonban, hogy a programozási nyelvek esetén a fejlődés másként alakult. Noha megjelentek a különféle forditóprogram generátorok /XPL [44], CDL[45], stb./, a döntő fordulatot az absztrakt adattipusok, azaz a felhasználó által a programban magában definiálható adattipusok és a rajtuk megadható müveletek, vagyis a bővithető nyelvek megjelenése jelentette. A leiró nyelvek esetén véleményem szerint ez kevésbé járható ut. Két érvet emlitek meg.

A leiró nyelv felhasználói gyakran nem programozási, számitógépes szakemberek, szakképzettségük inkább modellezendő feladatnak felel meg. Ők egyszerü, könnyen kezelhető, jól definiált eszközt akarnak használni. A bővithető nyelvek lényege ezzel szemben éppen az, hogy az eszközt a feladathoz alkalmazkodva a felhasználó maga késziti el az adattipus és müveletdefiniciókkal. A leiró nyelvek esetében célszerünek tünik a nem triviális, a megoldandó feladat ismeretén kivül komoly absztrakciós készséget, a definiciós nyelv kinálta lehetőségek pontos megértését igénylő tipusdefiniciótól megszabaditani az átlag felhasználót. Erre a célra megfelelőbb a generálási módszer, mellyel a konkrét feladat megoldására legalkalmasabb leiró nyelv definiálható, és utána az a tervező rendszer rögzitett leiró nyelveként használható.

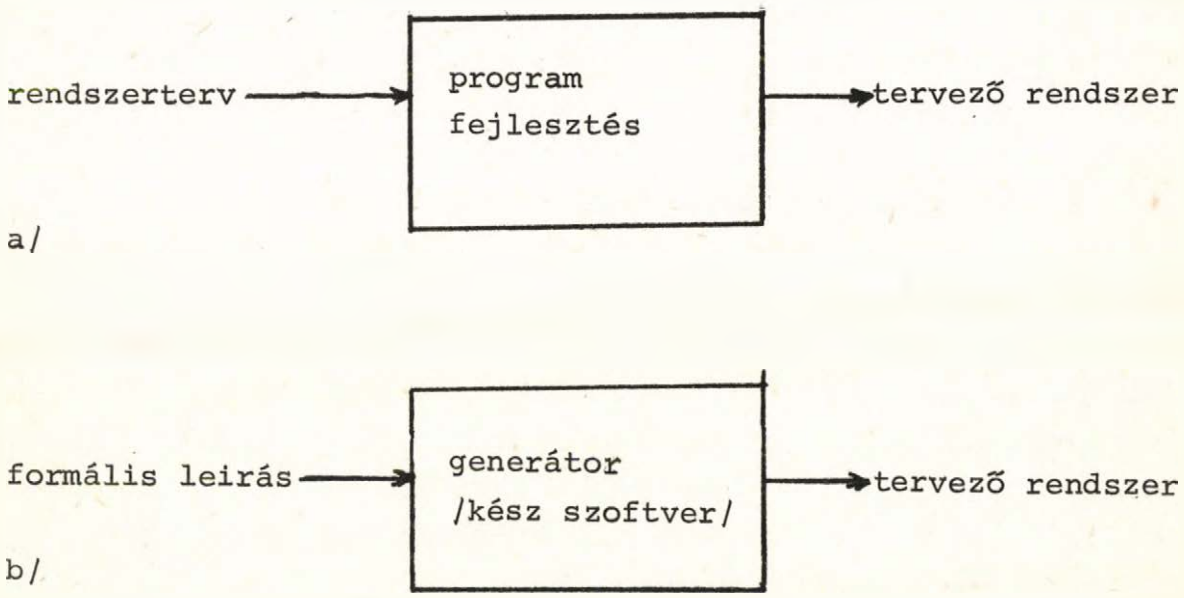
A másik oldalról szemlélve, a projekt egésze szempontjából sem lenne egyértelmüen jó megoldás felruházni valamennyi résztvevőt a tipusdefiniálás jogával. Az olvashatóság és az áttekinthetőség érdekében - ez itt még

fontosabb szempont, mint a programozási nyelveknél - valamilyen módon össze kellene egyeztetni a definiciókat, különben a leirás a többé vagy kevésbé célszerüen bevezetett, de valószinüleg feleslegesen nagyszámu, kevés előfordulással rendelkező tipus miatt nagyon nehezen követhetővé válhat. A definiciók automatikus összeegyeztetése egyelőre nagyon nehéz feladatnak tünik, a nem automatizált megoldás pedig tulzottan nagy terhet jelentene a projektvezetés számára.

Ez a fejezet a tervező rendszerek generálásával kapcsolatos általános jellegü meggondolásokat és értékeléseket tartalmaz. A generálás sémájának és a rendszer használati módjának ismertetése után a tervező rendszerek általános formális leirásának lehetőségeivel foglalkozunk, végül bemutatjuk a leirásból adódó architekturát és a generált tervező rendszerek szoftverjének felépitését.

## 2.1. A generálás módszere

A tervező rendszer hagyományos, kézi fejlesztése és a generálás közötti különbség nagyon egyszerüsitve a 15. ábra a/ és b/ részének összehasonlitásával látható.




15. ábra

Tervező rendszer készitése hagyományos módon illetve generátorral

A hagyományos módszer használata esetén a tervező rendszer készitése normális szoftvergyártó projektnek tekinthető. A generátor ezt feleslegessé teszi. Megjegyzendő, hogy ennek analógja a foditóprogramok irásánál pl. az XPL[44] /bár ez csak a forditóprogram szintaktikusan ellenőrző részét képes generálni, bonyolultabb szemantikai vizsgálatokat nem végez/, egyszerübb információs rendszerek készitésénél pedig pl. az ARIUS [41].

A hagyományos tervező rendszer fejlesztés fő motivációja az adott feladat vagy feladatok, melynek megoldásához segédeszközként elkészül a tervező rendszer /pl. [42]/. Általában a rendszer készitője és felhasználója ugyanaz a szervezet, gyakran ugyanazok a személyek.

A generálás módszerével készült tervező rendszereknél a fejlesztői és a felhasználói szerepkör elválik egymástól. A jövendő felhasználóknak nem kell fejlesztői munkát végezniük, hiszen ez a tervező rendszer előállitásához nem szükséges. Ez célszerü is, hiszen - ezt egy sor hazai és külföldi példa bizonyitja - meglehetősen nagy tehertétel egy projekten, ha a müködéséhez szükséges segédeszközöket is magának kell előállitania.

A tervező rendszereknek megvan a maguk életciklusa, éppen ugy, mint bármilyen szoftvernek. Vizsgáljuk meg, hogyan hat az egyes szakaszokra a generálás módszere.

A felmérés menetét a generátor létezése viszonylag kevéssé befolyásolja, a hatása inkább csak áttételesen, szemléletmódként jelentkezik. Már ebben a szakaszban figyelembe kell venni ugyanis, hogy a cél egy formális leirás elkészitése. A leirás elemei a készitendő tervező rendszer tipusai, azok összefüggései, előirt szemantikus jellegü ellenőrzések lesznek. A generátor a leirás elkészitésére definiciós nyelvet bocsát a felhasználó

rendelkezésére, és ennek a nyelvnek a szerkezete és lehetőségei - pl. milyen jellegü ellenőrzéseket biztosit - bizonyos mértékig meghatározza a készitendő modellt.

A tervezés fázisa tulajdonképpen nem más, mint a jövendő tervező rendszer leiró nyelvének végleges formális definicióját előállitó folyamat. Ennek során érdemes a javasolt nyelvi változatokat, egyes lehetőségeket rövidebb részletekkel esetleg egy részrendszer teljes leirásával kipróbálni. Ugyancsak ebben a fázisban kell meggondolni esetleges speciális listázó, lekérdező programok elkészitésének, rendszerhez illesztésének tervét. Összevetve a hagyományos módszerrel készitett tervező rendszereknél szükséges teendőket - a teljes szoftver megtervezése, a programozás előkészitése - a feljebb emlitettekkel, látható, hogy a generátor létezése teljesen megváltoztatja a tervezés menetét, a leiró nyelv formális definiciójának előállitására és esetleg egyes listák tervezésére korlátozódik /ez utóbbihoz is meghatározó keretet, jó esetben automatikus segédeszközöket biztosit/.

A programozás szakasza szinte teljes egészében kimarad az életciklusból. Amire esetleg szükség lehet, az a már emlitett, speciális listákat előállitó programok elkészitése, a tervező rendszer egésze automatikusan készül a formális definició alapján.

A tervező rendszer karbantartását a generálás módszere sokkal egyszerübbé teszi. Lehetőség van a dokumentáció zömének automatikus generálására. A formális definició /a definiciós nyelv részeként használható kötetlen formátumu magyarázó megjegyzésekkel együtt/ és a valamennyi generált tervező rendszert általánosan jellemző tulajdonságok együttes, jól összeillesztett kiiratásai

felhasználói kézikönyvekként szolgálhatnak. A dokumentálási munka megtakaritásánál is jelentősebb előny azonban, hogy az elkészült tervező rendszer módositása, átalakitása kevés munkával végezhető. A formális definició megfelelő változtatása után ujra kell generálni a rendszert, és az máris használható. /Tulajdonképpen ilyenkor uj rendszert állitunk elő, és nem a régit módositjuk - de ez csupán filozófiai jellegü különbség./

Láttuk tehát, hogy a generálás módszerével előállitott tervező rendszer életciklusa különbözik a kézzel, hagyományos módszerrel készitettétől. A generátor az életciklus valamennyi szakaszában segiti a tervező rendszer készitőjét. A formális definiciós nyelv tervezési módszert ad, szemléletmódot határoz meg. A generátort alkotó szoftver, mely a leirás alapján előállitja a tervező rendszert, ezt a módszert támogató számitógépes technológia. Mindebből az következik, hogy a generátor maga is speciális, tervező rendszerek készitésére szolgáló tervező rendszer.

Ha egyetlen, vagy néhány meghatározott tipusu tervező rendszerrel meg lehetne oldani általánosan a rendszertervezés - vagy legalább speciálisan az információs rendszerek tervezésének - problémáit, nem lenne szükség tervező rendszerek előállitását segitő eszközre. A helyzet hasonló a tervező rendszer és az információs rendszerek tervezésének viszonyára /éppen azért, mert a generátor nem más, mint speciális tervező rendszer /. Ha egy vagy több tipus információs rendszer általános megoldást tudna adni, ezek bármilyen probléma megoldására megfelelnének, nem lenne szükség tervező rendszerre.

A tapasztalatok szerint azonban általános megoldást jelentő tervezési rendszert készitení mindeddig nem sikerült

/a PSL/PSA-nak egy sor hiányosságát tapasztaltuk [33]/. Ennek oka a már emlitett ellentmondás: különböző jellegü leirásokat kell készitenі, tehát az általános megoldást jelentő leiró nyelvnek nagyon általánosnak kell lenni, ugyanakkor azonban olyannak, hogy a speciális tulajdonságok - melyek a rendszereket különbözővé teszik - is leirhatók legyenek benne.

A generálás módszere ezt az ellentmondást a valamennyi generált leiró nyelvet illetve rendszert jellemző általános tulajdonságok, és a formális definicióban megadható speciális jellemzők összeillesztésével kisérеli meg feloldani. A módszer főbb előnyei:

- nincs /vagy alig van/ szükség programozásra a tervező rendszer előállitásához;
- a leiró nyelv formális definiciójának elkészitése, mint feladat szemléletet ad a nyelv tervezéséhez, és biztositja a definició pontosságát;
- a tervező rendszer gyorsan előállitható /és változtatható/;
- a tervező rendszer felhasználója számára készülő dokumentációt a számitógép automatikusan, ráforditások nélkül előállitja .

## 2.2. A generátor használata

A hagyományos manuális technológiával készitett tervező rendszereknél a fejlesztő a felhasználó rendelkezésére bocsát egy leiró nyelvet, és az ezt támogató szoftvert. A nyelv, és a rendszer adva van, használatukról 1.3-ban volt szó.

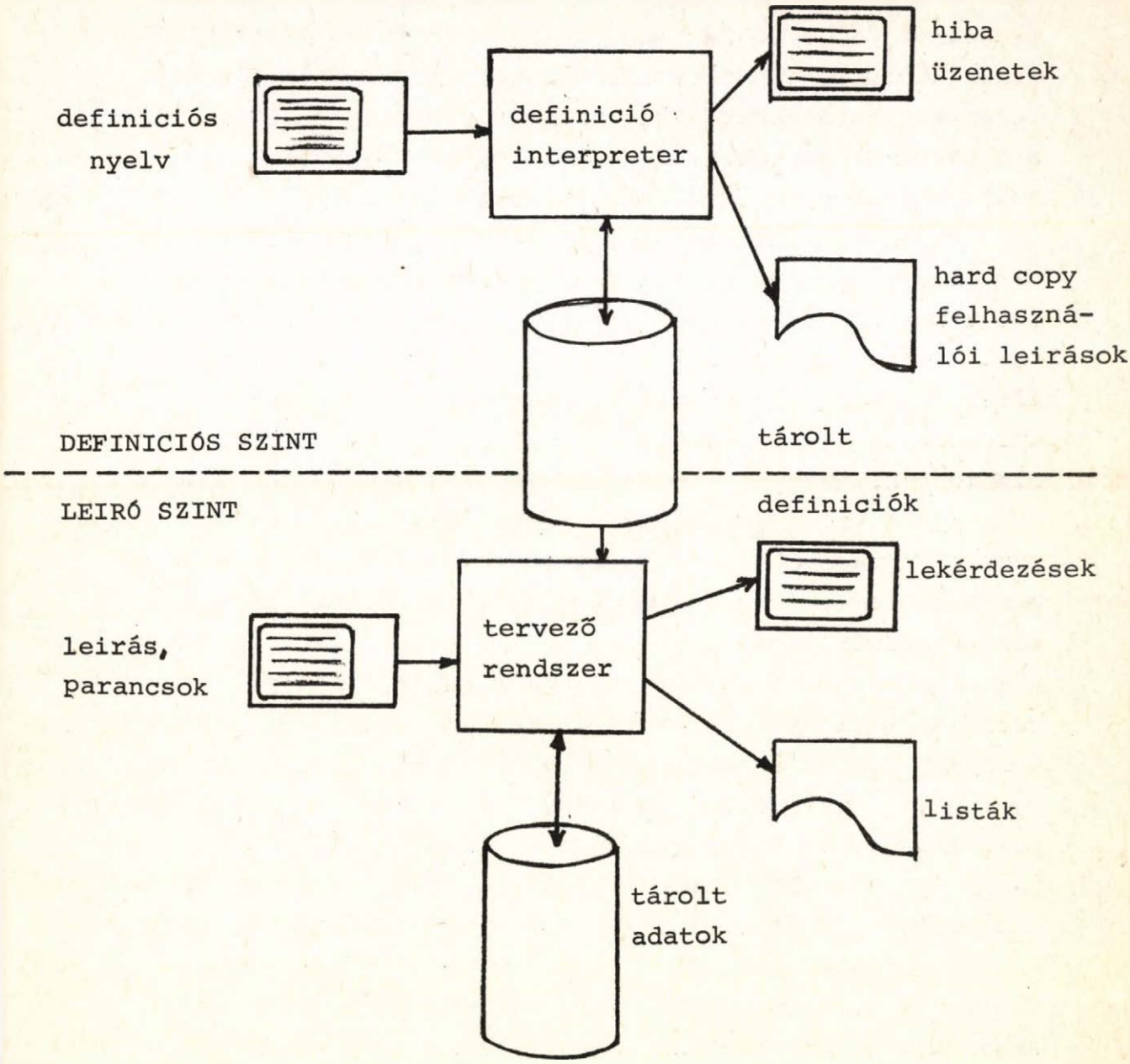
Némileg másképpen fest mindez a generált rendszerek

esetében. A felhasználó itt jövendő tervező rendszerének csak a keretét kapja meg, és ezt neki kell tartalommal megtölteni. Nincsen kész leiró nyelv, helyette adva van egy definiciós lehetőség, melynek segitségével magának kell meghatároznia, milyen nyelvet kiván a leiráshoz használni. A leiró nyelv megadása után azonban, már abban a helyzetben van, mintha kész tervező rendszert kapott volna rögzitett leiró nyelvvel.

A felhasználó tehát nem csak a szokásos számitógépes technológiával segitett rendszerszervezéssel járó feladatokat látja el, hanem őt terheli a tervező rendszer definiciójának jelentős része is. Ha a rendszerszervezést mint modellalkotást tekintjük, elmondható, hogy a metamodellt, vagyis a modellalkotást lehetővé tevő és meghatározó eszközrendszert is a felhasználó teremti meg. Nemcsak a valóság jelenségeit kell modelleznie, hanem a modellben használható terminusokat, a modellezés eszközeit is ő határozza meg. /Természetesen ez nem csak elvégzendő feladatot, hanem - és ez a fontos - a modellezés során nagyobb szabadságot biztositó lehetőséget is jelent./

A két feladat - a leiró nyelv definiálása és annak használata - időben elkülönül. A definiálás pl. információs rendszer fejlesztési folyamat viszonylag rövid szakasza, és ezt követi a kész tervező rendszernek az egész életciklust követő müködése.

A generálás elvi vázlata a 16. ábrán látható. Jól láthatóan válik szét a definiciós és a leiró szint. A felhasználó által adott definiciókat interpreter dolgozza fel, és azokat táblázatok formájában, valamilyen szervezéssel tárolja. Megjegyzendő, hogy a definiciók megadása több lépésben, több alkalommal is történhet, az inter-




16. ábra

A generálás elvi vázlata

preternek emlékeznie kell a korábban közölt definiciókra, és képesnek kell lennie összeegyeztetni azokkal az ujabbakat /a 16. ábrán erre utal a tárolt definiciók és a definició interpreter közötti kétirányu nyil/. Az interpreter interaktiv hibajavitási lehetőséget biztosit a felhasználó számára. A definiciós szakasz végén készül el a tervező rendszer dokumentációja.

A leiró szinttel dolgozó rendszerszervező számára már csupán egy rögzitett leiró nyelvü tervező rendszer létezik. Ez a 16. ábrán látható módon a definició interpreter által létrehozott táblákkal - tehát a definiciók által - meghatározott módon müködik - természetesen anélkül, hogy a tervező rendszer felhasználójának erről tudnia kellene.

A két szint különválása lehetőséget ad a feladatok szétválasztására. A leiró szintet használó szervezőnek nem kell ismernie a definiciós szint eszközeit, a tervező rendszer létrehozásának körülményeit, csak használnia kell tudnia a tervező rendszert az 1.3-ban leirtak szerint. Ez lényeges, ugyanis ez teszi lehetővé, hogy a tulajdonképpeni célfeladatra, a méreteiben általában igen nagy, sok résztvevőt igénylő rendszerszervezői, programozói munkára ráállithatók kevésbé képzett, vagy kisebb gyakorlattal rendelkező szervezők, programozók is, olyanok, akik a definiálásban nem vettek részt.

A definiciós szinten végzett tevékenység a leiró szint - a tulajdonképpeni tervező rendszer - használatához képest méreteiben jelentéktelen. Ezzel szemben jelentősége igen nagy, hiszen már ezen a szinten eldől, hogy használható lesz-e a tervező rendszer. A definiálást érdemes tehát magasan képzett, gyakorlott szakemberekre bizni, semmiképpen nem tekinthető rutinfeladatnak.

Vegyük sorra a definiciós szint felhasználójának feladatait.

A legfontosabb természetesen a tervező rendszer definiálása. Ez elsősorban a leiró nyelv megadását jelenti, de elképzelhetőek pl. listadefiniciós lehetőségek is. A leiró nyelv formális definiciója ahhoz hasonlit, amikor a tervező rendszer felhasználója formális nyelven irja le a modellezett információs rendszert. Ugyanerről van szó itt is, de magasabb szinten metamodellt kell készitení, a modellezés eszközeinek formális leirásával.

Elképzelhető, hogy szükség van kiegészitő programokra a jövőbeni tervező rendszer használatához. Ezek lehetnek pl. szabványt kielégitő listákat készitő eljárások, vagy bizonyos konvenciók betartására kényszeritő programok, jogosultságot ellenőrző, adatvédelmet biztositó rutinok, stb.

A leiró szint felhasználóinak betanitása, a leiró nyelv és a tervező rendszer müködésének ismertetése feltétlenül elvégzendő, bár az előzőekhez képest rutinjellegü feladat.

Az eddigiekben a két szint elkülönülő használatára, szétválaszthatóságára helyeztük a hangsulyt. Nagyon fontos megjegyezni azonban, hogy valójában a leiró nyelv definiciója nem választható el magától a leirástól, a modellezendő rendszertől. A definició a modellezendő rendszer tanulmányozásával, kisérleti leirások készitésével alakitható csak ki. Tulajdonképpen iterativ folyamatról van szó. Definiálni kell a leiró nyelvet, majd megkisérelni használni azt. A tapasztalatokon okulva kell változtatni a definición, majd további próbáknak vetni alá az uj változatot, ismét változtatni, és igy tovább. Mindehhez lehet számitógépet

használni - a definició alapján generálni a tervező rendszert és a kisérleti leirást gépre vinni, listákat kérni róla, stb. - de nem feltétlenül szükséges.

A definiciós folyamat hosszának, az iterációk, kisérletezgetések számának meghatározására nehéz becslést adni. Egy általános kijelentés tehető csupán: a definiciós folyamatnak véget kell érnie valamikor, nem szerencsés megoldás, ha a tervező rendszer leiró nyelve állandóan változik, - erről a jelen fejezet elején már volt szó - azért sem, mert a már tárolásra került adatok ilyen esetben általában elvesznek.

## 2.3. A tervező rendszer formális leirása

A tervező rendszerek létezését az teszi lehetővé, hogy a különböző információs rendszerekben - melyek tervezéséhez felhasználhatóak - van annyi közös, hogy különféle alkotóelemeik osztályokba /tipusokba/ sorolhatóak, közös tulajdonságokkal /attributumokkal/ birnak, és egymás közötti kapcsolataik is tipizálhatóak. Egy-egy adott tervező rendszer leiró nyelvének kialakitása absztrakciós folyamat eredménye, melynek során a modellezni kivánt valós világbeli rendszerek osztályának elemeit vizsgálgatva a közös tulajdonságok felismerhetőek. Az osztályozás teszi aztán lehetővé a pontosabb, formális leirást, hiszen a kötetlen formátumu szöveg helyett, melyben lényeges és lényegtelen keveredhet, a leiró nyelv a maga viszonylag kisszámu tipus, tulajdonság és kapcsolat választékával rákényszeriti a leirás készitőjét a lényeg - kötött szintaktikáju - kiemelésére.

Amikor a tervező rendszereket kivánjuk általánosságban leirni, ugyanezt az absztrakciót kell végrehajtani

magasabb szinten. A tipusok, tulajdonságok és kapcsolataik általános, közös vonásait kell megkeresni és formalizálni. A modellnek elég általánosnak, ugyanakkor elég erős strukturával birónak kell lennie. A PSL/PSA leiró nyelve pl. nem elég általános [46], mig pl. a szövegszerkesztő a bevitt szöveget csak formájában kezeli, a szemlélete mint modell, céljainkra tul strukturálatlan.

A tervező rendszer alkotóelemei a leiró nyelv, a tárolt adatok és a lekérdező, listázó, módositó parancsok /ld. 1.2./. Feladata a leiró nyelven bevitt információ tárolása, a tárolt adatokhoz való hozzáférés biztositása. Felépitését és feladatait tekintve tehát a tervező rendszer tulajdonképpen speciális információfeldolgozó rendszernek tekinthető.

Az információs /adatbáziskezelő/ rendszerek modellezésére az ANSI/SPARC bizottság dolgozott ki javaslatot [47]. Nem nehéz megfeleltetést találni három sémából álló tipus adatbáziskezelő rendszerük és az általános tervező rendszer között.

Az ANSI/SPARC javaslat szerint a felhasználó az adatbáziskezelő rendszerrel a külső sémán keresztül tart kapcsolatot, ezen keresztül történik az adatok tárolása, módositása, elérése. A tervező rendszereknél ezt a felhasználói interface szerepet a leiró nyelv, illetve a módosító, lekérdező és listázó processzorok töltik be.

Az adatok fizikai tárolásáról és eléréséről az ANSI/SPARC szerint a belső séma gondoskodik. Ennek közvetlen megfelelője a tervező rendszerek adatkezelő része, melynek ez a kizárólagos feladata. Ezen a szinten a felhasználó által adott leirás pusztán adat, melyet tárolni, ill. elérni kell, és az egyetlen cél ezeknek a feladatoknak a hatékony elvégzése, a tárolás részleteinek kezelése. Kevéssé kell törődni

a külső séma esetén elsődleges szemponttal, a könnyen olvasható, szemléletes fehasználói nyelv biztositásával, a tárolt adatok csak tartalmukban, de nem formájukban egyeznek meg a leiró nyelv állitásaival.

A külső és a belső séma között a kapcsolatot a konceptuális séma teremti meg. Ez nem más, mint a valós világ modellezni kivánt részének absztrakciója. A külső és a belső séma elemei közötti megfeleltetés ugy jön létre, hogy mind a kettő a konceptuális sémában szereplő fogalmakra hivatkozik. Tulajdonképpen a konceptuális séma az a modell, melyhez egyrészt a felhasználó igényeit kielégitő interface-t /külső séma/, másrészt pedig az adatait hatékonyan tároló és elérő rendszer /belső séma/ kell illeszteni. Nyilvánvaló, hogy a konceptuális séma az egész koncepció központi eleme.

A tervező rendszereknél a konceptuális séma megfelelője a valóság modellezésének módja. A PSL/PSA esetében ez konkrét objektumtipusok, kapcsolattipusok, szemantikai ellenőrzések ill. következtetések, melyeket a rendszer végez /ld. 17. ábra/.

```
type  input;
type  output;
type  process;
type  interface;
        .
        .
        .
relation generates(interface, input);
relation receives(interface,output);
relation uses (process,input);
        :
        :
constraint uses-to-derive (input,process,output) implies
           uses(process,input) and derives(process,output);
```

17. ábra

A PSL/PSA konceptuális sémájának részlete

A tipusok és kapcsolatok leirása eléggé egyszerü, nem szorul magyarázatra. Nehezebb formalizálni viszont a szemantikát. A PSL nyelvben pl. a

```
process P;
    uses  to derive O;
```

leriásrészlet, azaz a

```
uses-to-derive ( I,P,O);
```

kapcsolat automatikusan implikálja a

```
process P;
    uses I;
    derives O;
```

leirásrészletet, azaz a

```
uses(P,I);     derives(P,O);
```

kapcsolatokat tetszőleges P,I és O objektumokra. Ezt a következtetést irja le általánosságban, a konceptuális séma szintjén a 17. ábra "constraint" kijelentése.

A továbbiakban először a tervező rendszerekre mutatunk be két különböző konceptuális sémát, majd a külső és a belső sémára vonatkozóan teszünk néhány általános megjegyzést.

### 2.3.1. Az ERA konceptuális séma

Az ERA - Entitás, Reláció, Attributum /Entity, Relationship, Attribute/ - séma a P. Chen által [48]-ban bevezetett azonos nevü általános adatmodellen alapul. Nagyon népszerü, a [49]-ben ismertetett szinte valamennyi nem rögzitett leiró nyelvü tervező rendszerben ez a modell szolgál - kisebb nagyobb módositásokkal - konceptuális sémaként. Mi a következőkben a Michigan Egyetemen az ISDOS projekt keretében, a PSL/PSA folytatá-

saként kidolgozott Generalized Analyser /GA/ [50] rendszer ERA-tipusu konceptuális sémáját mutatjuk be.

A GA modellje lényegében a PSL/PSA általánositása. Alapvető fogalmai az objektum és a kapcsolattipus. A 17. ábrán látható "type" definiciók objektum, a "relation" definiciók pedig kapcsolattipusokat adnak meg. Az objektumtipus az ERA entitáshalmazának /entity set/, a kapcsolattipus pedig a relációhalmaznak /relationship set/ felel meg. A definiált tipusoknak a leiró nyelvben tetszőleges számu előfordulása - a konkrét objektumok és azok konkrét kapcsolatai - létezik. Ezt illusztrálja a 18. ábra.

| Definició | Leirás |
|---|---|
| type process; | process account job;<br>process béradatfeldolgozás; |
| type input; | input feladások;<br>input bizonylatok; |
| relation uses(process,input); | uses(account job, feladások);<br>uses(béradatfeldolgozás,bizonylatok) |

18. ábra
A tipusdefiniciók és az objektum ill. kapcsolatdefiniciók megfeleltetése

A GA tehát, a generálandó tervező rendszer konceptuális sémájának formális definiciójához az általános objektum és kapcsolattipus fogalmakat biztositja eszközként. A felhasználó a definiciós szinten ezek segitségével adja meg a leirás során használni kivánt tipusait,

majd a leiró szinten a definiált tipusok előfordulásait, konkrét objektumokat és kapcsolatokat generál a leirásban.

A GA objektumtipus és az ERA entitáshalmaz fogalma között a leglényegesebb eltérés, hogy az objektumtipusnak nem lehetnek attributumai. Az ERA modellben pl. definiálható lenne a

```
type process(input,output);
```

entitáshalmaz, vagyis olyan, amely előfordulás szinten azonnal, a definiálás pillanatában a konkrét objektumhoz rendelné annak inputját és outputját. Ez leiró szinten tehát igy nézne ki:

```
P:process(I,O);
```

A GA ezt a

```
type process;
type input;
type output;
relation uses to derive(process,input,output);
```

tipus definiciókkal és a

```
process P;
input I;
output O;
uses to derive(P,I,O);
```

objektumdefiniciókkal éri el.

Az objektum tehát atomi, tovább nem bontható. Egyedi azonositó neve, és meghatározott tipusa van. Az objektum nevéhez szinonima definiálható, az erre való hivatkozás a továbbiakban egyenértékü lesz az

objektum nevére való hivatkozással.

A kapcsolattipus megegyezik az ERA relációhalmazával. Minden kapcsolattipusnak meghatározott számu attributuma lehet, ezek objektumtipusok. Előfordulás szinten az attributumoknak - lévén objektumok - nevük van, a kapcsolatnak magának viszont nem adható név. Egy kapcsolattipus egy előfordulásában az attributumok - ezek objektumok lesznek - tipusainak meg kell egyezniük a kapcsolattipus definiciójában attributumokként szereplő objektumtipusokkal.

A kapcsolattipusok attributumai között a GA megengedi az 1:N vagy 1:1 kapcsolat megadását. A relációs adatkezelő rendszerek adatmodell terminológiája szerint ez a funkcionális függőségnek felel meg [26]. A definiált 1:N vagy 1:1 kapcsolat betartását a rendszer a leirás szintjén ellenőrzi. Igy pl., ha a

relation osztályvezetés(osztályvezető,osztály);

kapcsolattipusban az osztályvezető és az osztály attributumok között 1:N kapcsolatot adunk meg, az

osztályvezetés(Nagy Pál,Programtervezési Osztály);
osztályvezetés(Kis Péter,Programtervezési Osztály);

kapcsolatok közül a másodikat már hibásnak minősiti a tervező rendszer

### 2.3.2. Relációs referencia szemléletü konceptuális séma

A relációs referencia szemléletü konceptuális séma egységesebb az ERA modellnél. Egyetlen alapvető fogalommal, a relációval operál, azonban ez nem teljesen azonos

az adatbáziskezelő rendszereknél megszokott relációval [25]. Ezen a konceptuális sémán alapul az MTA SZTAKI-ban létrehozott SDLA /System Descriptor and Logical Analyzer/ rendszer [51].

Definiciós szinten a felhasználó fogalmakat /concept/ ad meg. A fogalom felfogható relációs sémának, vagyis intuitive egy üres táblázat megadásának. A táblázatnak neve van - ez a fogalom neve - és meghatározott számu oszlopból áll. Az oszlopok is mind saját névvel rendelekeznek, ezek a fogalom definiciójában szereplő attributumneveknek felelnek meg. A táblázat a definiciós szinten üres - a leirás tölti fel sorokkal. A fogalomdefiniciós mechanizmust a 19. ábra szemlélteti:

concept dolgozó (munkahely,fizetés,születési év),

| dolgozó | munkahely | fizetés | szül.év |
|---|---|---|---|
| Kovács József | programfejlesztés | 5400 | 1949 |
| Nagy Károly | üzemeltetés | 6000 | 1951 |
| : | | | |

19. ábra

Az SDLA fogalomdefiniciós mechanizmusa

Az ábrán látható táblázat a "dolgozó" fogalom konkrét előfordulásait tárolja /csupán logikai tárolásról van szó!/. Nyilvánvaló a hasonlóság a fenti táblázat és a relációs adatmodell jól ismert, hasonló felépitésü táblája között, azonban két lényeges különbséget ki kell emelni.

Az SDLA sémájában az egyes soroknak saját nevük lehet

/az ábrán "Kovács József", "Nagy Károly"/. A "klasszikus" relációs adatmodellben ilyen nincs, noha az ujabb javaslatok /pl. [52]/ tartalmaznak ilyen utalást, bár nem ebben a formában. A leiró szint felhasználója az általa névvel definiált sorokra a név szerint hivatkozhat, a nevet uj sor definiálásához attributumértékként felhasználhatja /pl. "programfejlesztés"/. A relációs adatmodellben a sorok kiválasztása csak attributumértékeik alapján történik, a sornak legfeljebb belső azonositója van, de ezt az adatbáziskezelő rendszer saját használatára tartja fenn, nem bocsájtja a felhasználó rendelkezésére [25]. /Megjegyzésre érdemes, hogy az SDLA is lehetővé teszi az attributumértékek szerinti hivatkozást./

A másik emlitésre méltó különbség a relációs ill. az SDLA konceptuális séma között az attributumok jellegében mutatkozik. Az SDLA reláció attirbutumai maguk is fogalmak - pontosabban fogalmakra történő hivatkozások -, vagyis relációk -,ill. ezekre mutató referenciák. A relációs adatmodellben az első normálformáju relációk attributumai kötelezően atomosak, azaz nem rendelkezhetnek saját attributumokkal, nem lehetnek relációk.

Ez a különbség a két modell között tartalmuk, célkitüzéseik eltéréséből adódik. A relációs adatmodell nagy adatmennyiség redundanciától mentes tárolására és elérési uttól független logikai elérésére alkalmas eszközöket tartalmaz. Ha megengedné a nem atomos attributumot, vagy az első, vagy a második célkitüzés csorbát szenvedne. Az SDLA esetében más a helyzet. A relációk jellege nem homogén, minden SDLA fogalomnak szemantikai tartalma van. Az elsőrendü cél nem a hatékony tárolás és a kényelmes lekérdezés, hanem a valóság minél pontosabb leirására alkalmas általános modell létrehozása könnyen használható eszközökkel.

Az SDLA konceptuális séma lényegesen különbözik az ERA modelltől is. Homogénebb formailag, hiszen az ERA entitáshalmazát és relációtipusát összeolvasztja a fogalommá, nem különbözteti meg a kettőt.

A fogalom az entitáshalmaztól abban különbözik, hogy a fogalmaknak lehetnek attributumaik. /Nem kötelező! Legálisak az atomi, attributumok nélküli fogalmak is/. A fogalomnak ez a tulajdonsága megkönnyiti a modellezést, ezt a 19. ábrán látható definició példája jól szemlélteti. A GA konceptuális sémában ennek a fogalomnak a leirásához kapcsolattipust kellene bevezetni, noha éppenséggel entitás jellegü.

A kapcsolattipus és a fogalom között a különbség a konkrét előfordulások szintjén mutatkozik meg. Mig egy kapcsolat nem rendelkezhet névvel, a fogalom minden előfordulásának természetesen /ld. 19. ábra/ saját neve lehet. Nem kötelező azonban nevet adni az egyes előfordulásoknak. Lehetnek olyan - kapcsolat jellegü - fogalmak, ahol egyetlen előfordulásnak sincs neve, másoknál - entitás jellegü fogalmak - az előfordulások lényegi jellemzője a név, sőt a rendszer megengedi azt is, hogy egy fogalom egyes előfordulásai névvel, mások pedig név nélkül létezzenek.

A relációs jellegü konceptuális séma definiciójához a szemantikus ellenőrzések, feltételek megadása is hozzátartozik. Az SDLA pl. a GA-hoz hasonlóan ellenőrzi az előirt 1:N kapcsolatok betartását a leiró szinten. Homogén szemlélete következtében a GA tipusellenőrzésénél /a kapcsolatokban résztvevő objektumok tipusának ellenőrzéséről van szó/ erősebb tipusegyeztetési lehetőségeket biztosit. Ezekről a kérdésekről a 3. fejezetben lesz részletesen szó.

### 2.3.3. A leiró nyelv

A leiró szint felhasználója a tervező rendszert olyannak látja, amilyennek a leiró nyelv mutatja. Számára a konceptuális séma - legyen az ERA, relációs referencia, vagy bármi más - nem létezik, nem kell tudnia róla. Ez meghatározza a leiró nyelv, továbbá az adatokat módosító és lekérdező lehetőségek fontosságát a tervező rendszerben. /Nem szabad azért megfeledkezni arról sem, hogy a konceptuális séma determinálja a leiró nyelv szerkezetét, hiszen a leirásnak arra leképezhetőnek kell lennie./

A programozási nyelvek, a forditóprogramok fejlődése során sokféle elméleti és gyakorlati eszköz jött létre a nyelvek általános leirására. Elégséges a különféle nyelvtanokat, és az univerzális forditóprogram generátorokat /pl. [44],[45]/ emliteni.

A leiró nyelvek definiciója történhet a gyakorlati eszközök bármelyikével. A módszer a következő: definiciós szinten a nyelv - rendszerint Backus-Naur formáju /BNF/ - leirását a forditóprogram generátor feldolgozza, ennek alapján táblázatokat készit, melyek a nyelv felismeréséhez és elemzéséhez szükséges adatokat tartalmazzák. A leiró szinten a generátor megfelelő rutinjai a táblázatok alapján felismerik és elemzik a leiró nyelvü állitásokat. Lényegében véve ezt a megoldást alkalmazták a PSL/PSA alkotói, akik az XPL [44] által gyártott táblázatokhoz a generátor megfelelő rutinjainak FORTRAN-ra való átirásával készitették el a nyelvi elemző részt. /Egyébként pl. az INGRES relációs adatbáziskezelő rendszer nyelvi elemzője sem más, mint az YACC univerzális forditóprogram alkalmazása [25]/. Külön kell emlitést tenni a minket elsősorban érdeklő SDLA rendszer definiciós mechanizmusáról. Itt a generálandó

tervező rendszer leiró nyelvének megadása sajátos eszköz, az un. formák segitségével történik [51]. Ennek lényege az az egyszerü felismerés, hogy a leirások lényegi része szabványmondatokban megfogalmazható, pl.

- ⟨valami⟩ használja ⟨valami⟩-t;
- amikor ⟨valami⟩ bekövetkezik ⟨valami⟩ indul;

stb. A mondatokba "paraméterek", a ⟨valami⟩-k helyébe a konkrét objektumokat irva könnyen olvasható magyar /vagy megfelelő definicióval bármilyen/ nyelvü szöveget kapunk.

A definiciós szinten a leiró nyelv ilyen mondatokkal adható meg, leiró szinten pedig ezeket konkrét objektumok közötti összefüggések megadására használhatjuk, pl.

- a programgenerátor használja a definició -t;
- amikor a megszakitás bekövetkezik a feldolgozó rutin indul;

stb.

A formák megadásának mechanizmusát részletesebben a 3. fejezet tárgyalja, itt csak a módszer két előnyös tulajdonságára hivjuk fel a figyelmet.

Az elsődleges ok, amiért ezt a megoldást választottuk, annak egyszerüsége és szemléletessége. Ez lényegesnek tünő szempont a definiciós szinten is, - a leiró szinten, ahol gyakran nem számitógépes szakemberek használják a rendszert elsődleges fontosságu - ugyanis az egy-egy feladat megoldására legalkalmasabb leiró nyelv megtalálása nehéz feladat, esetleg sok lépéses iteráció eredménye /ld. 2.2/. Célszerü tehát, ha olyan mechanizmust

alkalmazunk, melynek használata mellett a definiciós és a leiró szint kapcsolata "első ránézésre" nyilvánvaló. A formák - noha speciális esetként nyilvánvalóan gyengébb leiró erejüek - sokkal olvashatóbbak, mint egy BNF vagy vele ekvivalens definició.

Másik lényeges előnye a módszernek, hogy közvetlenül képezhető le a konceptuális sémára. Az egyes mondatok "paraméterei" a mondat által kifejezett fogalom attributumai, pl.

concept bekövetkezés(esemény, folyamat);
amikor esemény bekövetkezik folyamat indul;

Ez a nyilvánvaló kapcsolat megkönnyiti ugy a nyelv szerkesztését a konceptuális séma alapján - vagy akár forditva - mint leiró szinten az állitások interpretálásának, konkrét fogalomelőfordulások sorozatává alakitását.

### 2.3.4. Adatkezelés

Az adatkezelés funkcióját és helyét a tervező rendszeren belül 1.2.-ben igyekeztünk tisztázni, igy ennek részletes ismertetésére itt nem érdemes kitérni. A generált tervező rendszerek esetében egy általános észrevétel azonban némileg módositja a kialakult képet.

A klasszikus "hatékonyság kontra rugalmasság" problémát generált rendszerek esetében elsősorban a rugalmasság javára kell eldönteni. Arról van szó ugyanis, hogy mig a rögzitett leiró nyelvü tervező rendszereknél a nyelv sajátosságaihoz alkalmazkodva speciális elemeket is tartalmazó, /grafika, központi jelentőségü objektumtipusok köré csoportosuló lekérdezések, dokumentáló parancsok, stb./ kellőképpen nagy számu listából álló rendszer állitható össze, a generált rendszereknél, ahol a leiró nyelv

előre nem ismert, ez nem tehető meg. Természetesen itt is szükséges előregyártott parancskészletet, általános listagenerálási, lekérdező programokat biztositani, de - mint ezt az SDLA használata során tapasztaltuk - mig a leiró nyelv definiálásához megadható jól használható séma, - esetünkben a formák - addig az egyes felhasználók különleges igényeit követni listadefiniciós lehetőségekkel már jóval nehezebb. /Mellesleg a rögzitett nyelvü PSL/PSA listakészletét is bőviteni kellett [33]/. Valószinü tehát, hogy szükséges az egyes konkrét felhasználásokhoz, nyelvekhez speciális igényt kiegészitő lekérdező, esetleg módositó programok készitése, ezekhez pedig biztositani kell az adatok elérését. Nyilvánvaló, hogy ezt a munkát nagymértékben megkönnyiti a jól definiált, rugalmas - tehát a legkülönbözőbb elérési utakat biztositó - adatkezelő interface, vagyis fokozottabban előtérbe kerül a rugalmasság, a már kialakult adatbáziskezelő elmélet és gyakorlat felhasználása.

A használandó adatmodellt a konceptuális séma határozza meg, hiszen a leiró nyelv állitásai, illetve a lekérdezések ezen keresztül képezhetők le adatkezelő utasitásokká. Ezt a leképezést könnyiti meg, ha a két adatmodell - a konceptuális sémáé és az adatbázisé - elég közel van egymáshoz. Nyilvánvaló, hogy a relációs jellegü konceptuális sémák esetében /2.3.1., 2.3.2./ a relációs adatmodell a megfelelő. Az SDLA adatkezelést irja le a 4. fejezet.

## 2.4. A generátor architekturája

A 16. ábrán a generálás menetét mutattuk be, most az egyes komponensek felépitéséről, a szoftver müködéséről lesz szó az ábra alapján. A leirás elég általános

lesz, a fontosabb részletekre az SDLA egyedi megoldásaival a 3. és a 4. fejezet megfelelő részeiben utalunk.

A definiciós interpreter lényegében véve forditóprogram generátor. Feladata a definiciós nyelv állitásainak elemzése, és ennek alapján a leiró szinten a tervező rendszer használatához szükséges definiciós táblázat generálása, és a felhasználó dokumentáció - vagy legalább egy részének - elkészitése.

A definiciós táblázat tárolása külső adathordozón természetesen generátoronként változó. Az SDLA rendszerben egyszerü szekvenciális fájl, melyet a tervező rendszer moduljai futásaik elején a memóriába olvasnak. A Generalized Analyzer a tervező rendszer vezérlő definiciós adatok kezelésére a leirásokat tároló adatbáziskezelő rendszert használja. A megoldásnak - elegánciája mellett /relációs adatbáziskezelő rendszereknél használnak azonos formátumu rendszerleiró és tárolt adatokat [25]/ - előnye, hogy a vezérlő adatok adatbázisából bármikor könnyen nyerhetők uj listázások /itt az adatbázis a teljes definiciós leirást tartalmazza, az minden kiegészitő információ nélkül visszaállitható belőle - az SDLA megfelelő táblázatával ellentétben/. Hátránya viszont, hogy ezeknek az adatoknak az elérése kevésbé hatékony, és ugyanakkor tulajdonképpen az adatbáziskezelő rendszer legfőbb előnye, a sokféle és könnyen változtatható elérési mód nem használható ki igazán.

Lényeges alkotórésze a programnak a dokumentáció generátor. Ez - szintén a leiró nyelv definiciója alapján - elkésziti a generált rendszer használatához szükséges dokumentációt. Célszerü többfajta segédletet biztositani a felhasználók számára: rövid, csak a megengedett állitásokat tartalmazó összefoglalót, magyarázó

megjegyzésekkel ellátott felhasználói kézikönyvet, stb.

A tervező rendszer - felépitését és feladatait tekintve - lényegében megegyezik az 1.2.-ben és 1.3-ban leirtakkal, csupán általánosabb, hiszen a leiró nyelvtől függetlenek kell lennie, pontosabban a definiciós szinten megfogalmazott leiró nyelv a müködését - a tárolt definiciókon keresztül - paraméterként vezérli. Vizsgáljuk meg most 1.2. 14. ábrája alapján az egyes alkotórészeket! Előljáróban megjegyezzük, hogy - mint általában a tervező rendszereknél [33],[50] - célszerü minden felhasználói parancsot - leirás felvitele adatbázisba, módositás, különböző listázások, stb. - külön modulban realizálni.

Az egyes modulok az adatbázissal az adatkezelő rendszeren keresztül tartanak kapcsolatot. A nem adatkezelő részek feladata a leiró nyelvi állitások "leforditása" adatbázis müveletekre /felvitel, módositás/, vagy forditva, az adatbázis tartalmából leirás generálása /lekérdezés/. Ez tulajdonképpen nem sokkal bonyolultabb, mint a rögzitett leiró nyelvü rendszereknél, még ha belesoroljuk az előirt szemantikai jellegü ellenőrzéseket és müveleteket is. A nehézségek főként technikai jellegüek, viszonylag összetett, sok információt tartalmazó táblázatokat kell elhelyezni a memóriában /mágneslemezről alkalomszerüen beolvasni őket reménytelenül lelassitaná a rendszer müködését/, és meg kell szervezni hatékony elérésüket. A modulokat alkotó rutinokat ugy kell szervezni, hogy a táblázatokból nyert adatok, és ne a program szerkezete vezérelje müködésüket.

Az adatkezelő rendszernél némileg bonyolultabb a

helyzet. Minőségileg más feladat előre ismert szerkezetü adatok tárolására és elérésére felkészülni - ez a helyzet akkor, ha előre rögzitett leiró nyelvvel van dolgunk - és megint más olyan rendszert készitени, amely ugyanezeket a feladatokat általánosságban, az adatszerkezet speciális vonásainak kihasználása nélkül oldja meg.

Elvileg két megoldás kinálkozik. Az egyik az lenne, hogy a definiciós interpreter állitsa elő egyéb táblázatok mellett a konkrét adatszerkezet leirását is, és az adatkezelő szoftver ezt interpretálva, ez által vezérelve müködjön /hasonlóan a szintaktikus elemzőhöz/. Ha az adatkezelést pl. CODASYL tipusu adatbáziskezelő rendszerrel kivánjuk megoldani, ez azt jelentené, hogy az aktuális sémát kell generálni. Ez a megoldás elég bonyolult és nehézkes, ezen kivül a változó adatszerkezethez alkalmazkodni képes programok valószinüleg elég lassan müködnének.

Jobbnak tünik a másik megoldás: a konceptuális sémából kiindulva rögziteni az adatbázis szerkezetét. Igaz, ugyan, hogy pl. a Generalized Analyzer esetében mindig más és más objektum és kapcsolattipusokat definiál a felhasználó, de ha az adatkezelés felkészül az objektum és kapcsolattipusok tárolására általában, akkor a konkrét tipusok csupán a tárolt adatrekordok egy-egy mezőjeként, és nem az adatszerkezetet meghatározó tényezőként jelentkeznek.

Az adatbáziskezelés "klasszikus" adatmodellje - hierarchikus, hálós, relációs - közül az utóbbi kinál előre rögzitett séma nélküli ad-hoc adatdefiniciós, tárolási és lekérdezési lehetőségeket. A másik oldalról - az adatkezelés belső problémái felől - tehát

ugyanoda jutottunk, ahová 2.3.4.-ben az adatkezelés "felhasználói" /ez tulajdonképpen a konceptuális séma és az adatkezelés közötti interface/ megközelitése elvezetett: általános relációs, illetve ahhoz közeli adatszerkezetet használó adatbáziskezelő rendszerhez. A 4. fejezet tárgya egy ilyen rendszer gyakorlati megvalósitása az SDLA adatkezelésének megoldására.

# 3. AZ SDLA RENDSZER

Ennek a fejezetnek a tárgya az MTA SZTAKI-ban 1979 és 1982 között kifejlesztett tervező rendszer generátor, az SDLA. Már az eddigiek során is több alkalommal volt szó róla, most rendszerezetten - bár nem pl. egy felhasználói kézikönyv részletességével - is megkisérelem bemutatni.Az ismertetés nem szoritkozik a müködő rendszer tulajdonságainak egyszerü felsorolására, igyekszem megindokolni egyes lehetőségek rendszerbe illesztését, mások kihagyását, esetenként emlitést teszek az érdekesebb implementációs részletekről, elvi és gyakorlati problémákról, stb. Szeretném kiemelni, hogy az alábbiak nem feltétlenül tükrözik szerzőtársaim véleményét, noha sok mindent veszek át [51],[53],[54] és [55] közös dolgozatainkból.

Az SDLA létrehozása a PSL/PSA-val szerzett tapasztalatokon alapult. A különféle felhasználásokhoz szükség volt egy általánosabb rendszerre, olyanra, ahol az aktuális problémához lehet igazitani a tipuskészletet, nem "belemagyarázni" az egyes objektumtipusokba azt az értelmet, amit tulajdonitani kivánunk neki, és aminek a kapcsolattipusai sem felelnek meg tökéletesen /ld. 1.3./.

Az SDLA előnyös tulajdonságai mellett azonban ki kell emelni egy hátrányos vonást, melyről 2.2.-ben már volt szó, de itt - a PSL/PSA-val való összevetése során - hangsulyozni szeretnék. A PSL/PSA - amellett, hogy számitógépes technológia - jól átgondolt tervezési filozófia is egyben. A rögzitett tipuskészlet - noha korlátozó tényező - segitség a felhasználónak, hiszen általános információs rendszer modellnek tekinthető, megoldási sémának, melyre - természetesen csak optimális esetben - a konkrét probléma ráhuzható.

A "csupasz" SDLA nem rendelkezik ezzel a tulajdonsággal. A fogalomkészlet kialakitása a felhasználó feladata. Az optimális megoldás nem az SDLA-nak - vagy bármilyen más tervező rendszer generátornak - az átadása a felhasználónak, hogy az majd kialakitja a maga modelljét, hiszen ez sok - és párhuzamossága miatt felesleges - munkát, és komoly tapasztalatot igényel. Célszerünek látszik egy modellkészlet kialakitása feladatosztályokra, és a modellek SDLA definiciós szintü leirása. Egy-egy konkrét feladat esetén a megfelelő SDLA modell legalábbis jó kiindulási alap lehet a megoldást jelentő fogalomrendszer tervezésénél.

Ennek a modellkészletnek a létrehozása elég nehéz, és valószinüleg időigényes feladat, hiszen egy-egy javasolt modell használhatósága csak konkrét feladatok megoldásával mérhető le. [53] az első lépés ebben az irányban.

Néhány terminológiai jellegü megjegyzés: az SDLA konceptuális séma szintjén a felhasználó fogalmakat definiál, mint ezt 2.3.2.-ben láttuk. A fogalomelőfordulásokat - leiró szinten - objektumoknak fogjuk hivni. Minden objektumnak meghatározott tipusa van - ez nem más, mint az a fogalom, melynek előfordulása.

## 3.1. A definiciós szint

A fogalmak, formák és a szemantikai összefüggéseket megadó állitások sorozata alkotja az SDLA-ban a tervező rendszer definicióját. A definició alapegysége a fejezet /szekció/, mely egy fogalmat ad meg az emlitett tipusu állitások segitségével.

### 3.1.1. Fogalom definiálása

A fogalmat definiáló állitás általános alakja:

concept fogalom(attributum 1:fogalom,...,attributum n:fogalom);

A 2.3.2.-ben tárgyalt példáknál nem szerepelnek attributumnevek. Ez a definiciós nyelvben megengedett, ilyenkor a rendszer az attributumnevet a megfelelő fogalomnévvel egyezőnek tekinti, pl. a

concept dolgozó(munkahely,fizetés,születési év);

ekvivalens a

concept dolgozó(munkahely:munkahely,fizetés:fizetés,születési év:
születési év);

állitással. Az attributumnevekre azért van általában szükség, hogy az azonos tipusu attributumokra külön-külön hivatkozni lehessen, mint pl. a

concept fa(bal:fa,jobb:fa);

fogalomnál.

A definiciós szinten attributumként adott fogalom a

leirás készitése során a fogalomelőfordulásokban attributumként szolgáló objektumok tipusának ellenőrzésére szolgál. Az SDLA tipusellenőrzési mechanizmusát 3.1.3. ill. 3.2. tárgyalja részletesebben, egyelőre annyit érdemes megjegyezni, hogy az olyan objektumokat, melyek tipusa összeegyeztethetetlen az attributumként adott fogalommal, a leirást felvivő és módositó parancsok /programok/ nem fogadják el.

A definició zártságának ellenőrzése a definiciós interpreter szemantikai jellegü funkcióinak egyike. Ez annyit jelent, hogy minden fogalomként használatos névnek szerepelnie kell explicit /concept/ fogalomdefinicióban, tehát az attributum tipusaként megadott fogalomnévnek is. A feljebb bevezetett "dolgozó" fogalom mellé, tehát definiálni kell a "munkahely", "fizetés", "születési év" fogalmakat is a definició zártsága érdekében, ha valamelyiknek ezek közül szintén vannak attributumai, akkor azokat is, és igy tovább. Nem kell definiálni három előre definiált /érték/ fogalmat, ezek az "integer", "real" és "text" /jelentésük nyilvánvaló/.

A definició zártságának megkövetelése a felhasználótól egyes programozási nyelvek /ALGOL, PASCAL, ADA, stb./ kötelező deklarációjának analógja. Ezeknél a nyelveknél a programban nem használható deklarálatlan változó, szemben az ezeket elfogadó és ezeknek konvenciók alapján jellemzőket /tipus, hossz, stb./ tulajdonitó automatikusan deklaráló nyelvekkel /FORTRAN, PL/1, stb./. Mind a két megoldásnak megvan az előnye: a zárt definició nagyobb biztonságot ad, véd az elirások ellen, az automatikus deklaráció pedig sokszor kényelmes, a triviális dolgok leirása megtakaritható /az SDLA esetén pl. az atomi - attributumok nélküli - fogalmak definicióját tenné szükségtelenné/. Az SDLA esetében tulajdonképpen kompromisszumos megoldás született: a definiciós szinten - ennek fontosságára való tekintettel - a leirásnak zártnak

kell lennie, mig a leiró szinten - mint ezt látni fogjuk - nem kötelező az explicit deklaráció.

### 3.1.2. Relativ és abszolut formák

A formák apparátusa a leiró nyelv állitásainak szintaktikáját adja meg, mint ezt 2.3.3.-ban láttuk. A nyelv konstruálásánál a cél az, hogy a definiált fogalmak előfordulásai a beszélt nyelvhez minél közelebb álló mondatokkal legyenek megadhatók.

Mivel a leiró nyelvi állitások megadásának módjáról lesz szó, azzal kell kezdeni, hogy az SDLA egy lényeges megszoritást tesz a leiró nyelv szerkezetére vonatkozóan Bármelyik generált tervező rendszerben a leirás szekciókból épül fel. Egy szekció első állitása /a szekció feje/ a legegyszerübb esetben

fogalomnév objektumnév; /pl. process P;/

vagy

objektumnév: fogalomnév; pl. P:process;

alaku. A két forma ekvivalens, jelentésük nem más, mint az objektumnak és tipusának deklarációja. A szekció tartalma az erre az objektumra vonatkozó állitások halmaza, pl. a

```
process P;
    uses I;
    derives O;
    updates F;
    utilized by Q;
```

szekció a P folyamat kapcsolatait /használja I-t, elő-

állitja O-t, stb./ irja le, tehát azokat a dolgokat, amiket erről a folyamatról tudunk.

A formadefiniciós mechanizmus a szekciókból álló leirásszerkezetnek felel meg. Minden fogalomhoz egy vagy több forma definició tartozhat. Ezek bármelyikének előfordulása az adatfelvitel során az illető fogalom egy előfordulását hozza létre, ill. már létező előfordulásra való hivatkozásnak számit. Egy-egy forma a fogalom előfordulásainak egy meghatározott attributum nézőpontjából - ez leiró szinten azt jelenti, hogy egy meghatározott tipusu szekcióból - való definiálására szolgál, pl. a

```
concept usage (process, input);
form process: uses input;
form input: used by process;
```

definiciórészlet leiró szinten a

```
process P;
    uses I;
```

és az

```
input I;
   used by P;
```

leirásrészletek használatát teszi lehetővé. Mind a két szekció eredménye a konceptuális séma szintjén a

```
usage(P,I);
```

objektum lesz. Teljesül tehát az a - nyilvánvaló - követelmény, hogy ha több különböző nézőpontból /attributuma

felől/ definiáljuk ugyanazt a fogalomelőfordulást, az eredmény ugyanaz lesz.

A fenti példából látható, hogy a forma általánosan nem más, mint a

form attributumnév:szövegbe ágyazott attributumnevek;

definiciós állitás. Az ilyen formát relativ /nézőpont, vagyis attributum felől/ formának hivjuk, leiró nyelvi használata csak a nézőpontjukkal összeegyeztethető tipusu szekció belsejéből megengedett.

Használati mechanizmusa igen egyszerü: leiró szinten is a definiciós szinten megadott formát kell használni, csak minden attributumnév helyett annak az objektumnak a nevét kell irni, melyet az ujonnan generálandó fogalomelőfordulás megfelelő attributumaként meg akarunk adni. /Az attributumok maguk is lehetnek uj objektumok./

Kényelmi szempontból - a tapasztalat szerint - lényeges, hogy egy formával egyszerre több különböző objektum is generálható legyen, oly módon, hogy az attributumnevek helyett objektumnevekből álló listát adhasson meg a felhasználó. Ilyenkor minden objektumnév egy-egy uj objektumot generál, pl. a

```
process P;
    uses I1, I2, I3;
```

leirásrészlet a

```
usage(P,I1);  usage(P,I2);  usage(P,I3);
```

objektumokat hozza létre. A jelenleg müködő SDLA változat - technikai okokból - ezt a lehetőséget nem biztositja.

Szükség van olyan állitásokra is, melyek nincsenek nézőponthoz, tehát szekcióhoz kötve. Az ilyen formát abszolutnak nevezzük. Már találkoztunk az abszolut forma egy speciális esetével, az un. előredefiniált abszolut formával. Ez nem más, mint a feljebb már emlitett

fogalomnév objektumnév;

vagy

objektumnév: fogalomnév

alaku forma. Azért nevezzük előredefiniáltnak, mert /leiró szinten/ a definicióban szereplő mindegyik fogalomnévvel használható, definiciós szinten való megadása nélkül.

Az abszolut forma általános alakja hasonló a relatívéhoz, használatának mechanizmusa pedig megegyezik azzal:

form absolute: szövegbe ágyazott attributumnevek;

A feljebb már szerepelt

concept usage (process,input);

fogalomhoz adható meg pl. a

form absolute: process uses input;

abszolut forma, melynek eredményeként a leirásban legálissá válik pl. a

P uses I;

állitás.

3.1.3. Szemantika

Az SDLA szemantikát definiáló vagy a leirás szemantikus helyességét ellenőrző /biztositó/ lehetőségeiről beszélve, először is meg kell mondani, hogy mit értünk szemantika alatt [61]. Meghatározásunk önkényes és SDLA specifikus lesz, noha bizonyos mértékig a leiró nyelvek körére általánositható és az általánosan elfogadott szemantika fogalommal összhangban álló. /Ez eléggé informális, nagyjából az egyszintes nyelvtannal le nem irható szabályokat szokás érteni alatta./

Szemantikusnak fogjuk nevezni tehát az SDLA konceptuális sémájának objektumai - a fogalmak - között a definiciós szinten megadott általános összefüggéseket. Bővebben kifejtve ez a következőket jelenti:

Az ANSI/SPARC modell analógiát /2.3./ használva a leirás ellenőrzésének és tárolásának a folyamata nem más, mint két leképezés - a külső séma nyelvéről /leiró nyelv, formák/ a konceptuális sémáéra /fogalmak előfordulásai/ majd a konceptuális séma nyelvéről a belső sémáéra /adatkezelő rendszer/ - megvalósitása. /A lekérdezés ugyanennek a két leképezésnek a végrehajtása forditott sorrendben és a módositás is ezekkel irható le./ Ennek megfelelően azok az összefüggések szemantikusak, melyek a két leképezés megvalósitása között, tehát pl. uj leirás-készletek felvitelekor a leiró nyelvi állitások objektumokká való leképezésének megvalósulása után, és az adatkezelő rendszer nyelvére való leképezés végrehajtása előtt végrehajtandó akciókat /főleg ellenőrzéseket, de másokat is/ eredményeznek.

Az SDLA esetében a két leképezés során végrehajtandó - tehát nem szemantikai ellenőrzésnek számitó - müveletek elég pontosan leirhatóak. Az első leképzés a

forma felismerését jelenti a beérkező lexikális egységek sorozatának és a definiált formák összehasonlitásának az alapján. A forma felismerése után a fogalommá való átalakitás már triviális. Ide tartozik még egy ellenőrzés - ha a generálandó objektumnak neve van, ellenőrizni kell, hogy szerepel-e már ilyen nevü objektum az adatbázisban.

A második leképezés lényegében az uj leirásrészlet tárolásához szükséges adatkezelő müveletek sorozatának az előállitása. Érdemes megemliteni, hogy ez a valóságban is jól elkülönül a szemantikus jellegü akcióktól, ugyanis nem érdemes addig elkezdeni az adatbázis módositását, amig kiderülhet, hogy az uj leirásrészlet nem konzisztens, akár önmagában, akár a már tárolt adatokkal összevetve.

A fenti meghatározás nyilván minden, az ANSI/SPARC modellel leirható tervező rendszer generátor esetén használható. Az analógia a programozási nyelveknél értelmezett szemantikával elég nyilvánvaló, hiszen a formák leképezése fogalmakra nagyjából megfelel a forditásnak /szintaktikus elemzésnek/, a leirás tárolása - ez önmagában kevés, ld. szövegszerkesztő - és főképpen a tárolt leirás konzisztenciájának biztositása pedig a statikus szemantikai ellenőrzésnek és a végrehajtásnak.

A szemantikus összefüggések megadhatóságának jelentőségét a tervező rendszer használhatósága szempontjából nehéz lenne tulbecsülni. A számitógép igazán az ellenőzések automatikus elvégzésével, a leirás konzisztenciájának ellenőrzésével fizeti vissza azt a többletköltséget, amit a használata a tervezés során jelent. Persze ahhoz, hogy a rendszer ellenőrizni tudjon egy leirást, meg kell adni, hogy mit és hogyan ellenőrizzen.

A szemantikai összefüggés fenti definiciójából már következik megadásának módja. Mivel a konceptuális séma objektumainak kapcsolatairól van szó, a megadásuk ennek terminusaiban történik.

Rögzitett leiró nyelvü tervező rendszer esetében ez általában nem okoz igazán komoly gondot, hiszen a konceptuális sémában konkrét tipusok szerepelnek /2.3./, tehát meg lehet adni konkrét ellenőrzési algoritmust, egészen az adatbázison végrehajtandó müveletekig, vagyis el lehet késziteni az ellenőrzést végrehajtó programot. A PSL/PSA pl. csak akkor fogad el - figyelmeztető üzenet nélkül - "uses" kapcsolatot, ha a használni kivánt objektum korábban egy "derives" vagy "generates" kapcsolattal bekerült a rendszerbe, azaz a

uses(process,input);

kapcsolat csak akkor legális, ha az "input" tipusu objektumra létezik pl. egy

generates(interface,input);

kapcsolat. Nyilvánvalóan készithető olyan program, amely minden "uses" állitásra ellenőrzi ezt a tulajdonságot.

Bonyolultabb a helyzet a generátorok esetében, ugyanis itt a szemantikai összefüggéseket leiró apparátusnak eléggé általánosnak kell lennie. Az egyes kijelentések nem kapcsolódhatnak adott objektumtipusokhoz, hiszen csupán a konceptuális séma objektumaival /az SDLA esetében a fogalmak és attributumaik/ operálhatnak, viszont az összefüggéseket nem általában, hanem konkrét objektumtipusok /fogalmak/ között kell megadni. Olyan formális apparátus kialakitására van tehát szükség, mellyel nem csak egy konkrét leiró nyelv, hanem a leiró nyelvek egy elég nagy családja bármely tagjának a szemantikája megadható.

Általános szemantikadefiniálási módszerek a programozási nyelvekre szép számmal léteznek [35], de ezek

meglehetősen bonyolult, számunkra tul általános, a tervező rendszerek gyakorlatában nem használható eszközök. Járhatóbb utnak tünik a rögzitett nyelvü tervező rendszerekben létező szemantikai funkciók általánositásával, majd pedig a gyakorlati tapasztalatok, a kialakuló módszerek felhasználásával létrehozni egy standard - és a későbbiekben bővithető - apparátust.

Az SDLA esetében is igy történt a jelenleg müködő rendszerbe három szemantikai funkciót épitettünk bele. Ezek felhasználhatóságát pl. [53] vagy [55] illusztrálja. A válogatás - elég sok javaslatot kellett elvetni - részben szubjektiv volt. Az absztrakt adattipusos programozási nyelvek, főleg a SIMULA-67 [13] meggyőzően illusztrálják a tipusszerkezet előnyeit általános rendszerek leirására. A kényszeritésekhez a PSL/PSA szolgáltatta az ötletet. A funkcionális függőségek megadása és ellenőrzése a relációs adatbáziskezelő rendszerek elméletében betöltött döntő jelentőségü szerepköre /ld. pl. [56], [57]/ miatt semmiképpen nem maradhatott ki egy alapvetően relációs szemléletü konceptuális sémát használó tervező rendszer generátorból.

### 3.1.3.1. Fogalmak finomitása - hierarchia és háló

Az egy leirás során felhasznált fogalmak általában nem függetlenek egymástól, összefüggések vannak közöttük, egyik a másik speciális esete lehet. Pl. a

dolgozó(munkahely,fizetés,születési év);

fogalom speciális eseteként /finomitásaként/ definiálható a

vezető (munkahely,fizetés,születési év,beosztás);

fogalom. Ezt a valós világban magától értetődő kapcsolatot igyekszik modellezni az SDLA tipusszerkezete [62].

Mivel a "vezető" maga is "dolgozó", logikusnak tünik, hogy rendelkezzen annak valamennyi tulajdonságával. Ez esetünkben azt jelenti, hogy a "vezető" fogalomnak rendelkeznie kell a "dolgozó" fogalom valamennyi attributumával, és hogy valamennyi leiró nyelvi állitásban, ahol "dolgozó" tipusu objektum szerepelhet, meg kell engedni "vezető" tipusu objektumot is.

Nyilvánvaló, hogy a "finomitás" reláció tranzitiv. Ha A speciális esete B-nek, B pedig C-nek, akkor A nyilván C-nek is speciális esete. Ily módon, tehát kialakul egy meghatározott tipusszerkezet.

A különböző programozási nyelvek tipusszemlélete igen eltérő. A PASCAL például a tipusokat egymástól függetlennek tekinti, nincs finomitási lehetőség [21]. A SIMULA-67 a hierarchikus tipusszerkezetet támogatja [13]. Ez azt jelenti, hogy minden tipus /SIMULA osztály/ legfeljebb egy másiknak lehet közvetlen leszármazottja /finomitása/. Az ALGOL-68 egyesitéssel /union/ alkot uj tipusokat régiekből [14]. Az egyesitésben résztvevő tipusok az ujonnan keletkezett speciális esetei lesznek, és az ahhoz definiált valamennyi müveletnek argumentumai lehetnek. Ez a tipusfilozófia elég kusza gráf-szerkezetü tipusstrukturához vezet.

A programozási nyelveket vizsgálva tehát, szinte bármilyen tipusszerkezethez lehet analógiát találni. Nézzük meg ezért a problémát a minket érdeklő szemszögből: milyen tipusszerkezetet érdemes kialakitani egy leiró nyelv esetében?

Kiindulásképpen a tipusszerkezet elfogadhatóságára a következő kritériumot adjuk meg: a leirásban szereplő valamennyi objektumnak bármelyik pillanatban egyértelmüen meghatározott tipussal kell rendelkeznie. Ennek a

követelménynek a szükségessége eléggé nyilvánvaló, ha figyelembe vesszük azt, hogy a tervező rendszer lényeges feladata listázásokkal, a lekérdezések megválaszolásával segiteni a felhasználót a modellezett rendszer áttekintésében. Mivel a tipus egy objektum leglényegesebb jellemzője, általában lekérdezési szempont is, tehát egyértelmünek kell lennie.

Az egymástól teljesen független tipusok esetét tekintjük az egyik végletnek. Ez feltétlenül használható megoldás, az elfogadhatóság kritériumát nyilván kielégiti, de elég merev. Azt jelentené, hogy minden uj fogalomelőfordulás generálásánál az uj fogalom attributumaként szereplő objektumok tipusainak a fogalom definiciójánál megadott tipusokkal pontosan meg kell egyezniük. A gyakorlatban ennél még sulyosabb következmény, hogy a rendszerbe kerülő objektum tipusa egyszer és mindenkorra rögzitve van /illetve csak módositó paranccsal változtatható/. Ez elég kellemetlen, ha meggondoljuk, hogy bármilyen - elég nagy - rendszer csak fokozatosan ismerhető meg, és alakitható ki. Az objektum tipusa annak általános tulajdonságait tükrözi /1.2./, igy az a tervező rendszer adatbázisába kerülésekor általában még nem rögzithető, ugy logikus, hogy jellegének fokozott megismerésével alakuljon ki végleges tipusa. /Gyakran fordul elő pl., hogy egy adatot biztosan használni akarunk, de még nem dőlt el, hogy önálló bizonylatként, adatrekordként, csoportként, vagy más formában./

Most megvizsgáljuk a másik végletet, vagyis azt az esetet, amikor semmiféle előzetes kikötést nem teszünk a tipusszerkezetre vonatkozóan. A tipusok ilyenkor a leirás készitése során változhatnak, a definiciós szinten rögzitett finomitásoknak megfelelően. Mivel a szerkezetre vonatkozóan nem tettünk kikötést, elvben egy fogalomnak akárhány finomitása létezhet, és ő maga is több fogalom

finomitása lehet. Nézzük meg, milyen mechanizmus szerint változhat a leiró szinten egy objektum tipusa! /Részletesen erről 3.2.-ben lesz szó./

Amikor az objektumot létrehozzuk, annak egyértelmüen definiált tipusa van /3.1.2./. Amikor hivatkozunk rá valamilyen állitásban /azaz egy uj objektum attributumaként/, akkor rá vonatkozóan is közlünk uj információt, igy logikus, ha a tipusa ilyen esetekben megváltozhat. Példaként tekintsük a következő definiciórészletet!

```
concept data;
concept compound data is data;
concept contain ( contained:data,container:compound data);
form contained: part of container;
```

A második sor magyarázatra szorul: ebben a "compound data" fogalmat a "data" finomitásaként definiáljuk. A finomitási mechanizmus müködését leiró szinten a következő példa szemlélteti:

```
data B;
  :
data C;
  part of B;
```

A valamikor egyszerüen adatként /"data"/ definiált B objektumról a későbbiek során kiderült, hogy egy másik adatot /C/ tartalmaz. Az ilyen tartalmazási kapcsolatban /concept "contain"/ álló objektum tipusa viszont összetett adat /compound data/ kell, hogy legyen, és mivel ez a tipus definiciós szinten a B objektum eredeti tipusának finomitása, a tervező rendszer a B tipusát automatikusan megváltoztatja, beillesztve ezáltal az összképbe az uj információ B-re vonatkozó részét.

A tipusváltoztatási /nyugodtan irhatjuk, hogy finomitási, hiszen egy objektum tipusa ujabb információ megszerzésével legfeljebb finomodhat/ mechanizmus áttekintése után pontosithatjuk a tipusszerkezet elfogadhatóságára adott kritériumot. Láttuk, hogy az objektum tipusa lépésenként alakul ki, ahogy ujabb és ujabb állitásokban - ezek lehetnek más objektumokkal való kapcsolatai, de pl. előredefiniált abszolut formával explicit tipusdefinició is - szerepel. Az uj tipus minden esetben a régi tipus, és az uj állitásból adódó tipusra vonatkozó információ összevetéséből keletkezik, A tipusszerkezet használhatósági kritériuma olyan esetekben sérül meg, ha létezik olyan szituáció, amikor az összeegyeztetés nem ad egyértelmü eredményt.

Két tetszőleges tipus metszetét /legyenek A és B/ a következő módon definiáljuk:

a/ ha A=B, akkor a metszet A lesz;

b/ ha A tipus B speciális esete /finomitása/, a metszet A lesz;

c/ független A és B esetén tekintsük az olyan tipusok halmazát, melyek ugy A-nak, mint B-nek finomitásai. Ha ez a halmaz üres, a két tipus összeegyeztethetetlen, metszetük üres. Ha a halmaz nem üres, és kiválasztható belőle olyan C tipus, melynek speciális esete /finomitása/ a halmaz összes többi eleme, akkor A és B metszete C lesz. Ha ilyen C tipus nem létezik, akkor A és B metszete meghatározhatatlan.

A definició leiró szinten a tipusfinomitó mechanizmus müködését irja le. Ha egy objektum régi tipusa A, és a tervező rendszerbe érkező állitásban mint B tipusura hivatkozunk rá, akkor az uj tipusa A és B metszete lesz.

A definició a/ pontja értelmében, ha az objektumot az állitás /forma/ hatására létrejövő fogalomban a régi tipusának megfelelően szerepeltetjük, nem változik a tipusa. A b/ pont értelmében, ha az állitásban feltételezett tipus az eredeti finomitása, akkor az objektum tipusa ennek megfelelően finomodik /mint a tipusváltoztatást illusztráló fenti példában/. Ugyancsak a b/ pontból következik, hogy ha az állitásban feltételezett tipus az eredetinél durvább /az eredeti annak finomitása/, a tipus nem változik - ez azt is jelenti, hogy a tervező rendszer egy fogalom attributumaként a definiciós szinten megadott tipusunál finomabb tipusut /tehát az attributumként adott fogalom speciális esetének előfordulását/ is elfogad.

A c/ pont több részesetre bomlik. Először is, ha a két tipus metszete üres, ez a leiró szinten azt jelenti, hogy a felhasználó az objektumot eddigi jelentésével összeegyeztethetetlen módon akarta használni. Ilyen esetben a tervező rendszer a teljes állitást elveti, és hibaüzenetet küld a felhasználónak. Ilyenkor a tipus nem változik.

A c/ pont irja le a legbonyolultabb esetet, amikor az objektum régi tipusa, és az, ami az uj állitásban betöltött szerepének megfelel, függetlenek, de van olyan tipus, mely mind a kettőnek finomitása, tehát mind a kettő tulajdonságaival rendelkezik. Ilyen esetben nyilván nem felhasználói hibáról van szó, hiszen nincs ellentmondás a régi és az uj információ között, csupán annyi történik, hogy az uj ismeretek birtokában bővül az objektumról alkotott kép. A tervező rendszernek ebben az esetben tehát képesnek kell lennie arra, hogy az objektum uj tipusát meghatározza.

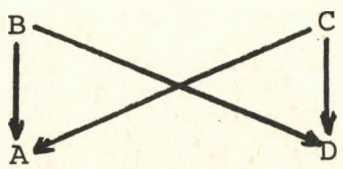
A c/-ben leirt algoritmus szerint az uj tipusként szóbajöhetők közül a legdurvábbat kell választani, ugyanis a tipus meghatározásánál a döntéshez csak a már meglévő biztos információ szolgálhat alapul, és mig ez a tipus éppen a felhasználó által eddig adott információt tartalmazza, ennek finomitása már annál többet.

A c/ pontban lényegében az elfogadhatósági kritérium pontosabb megfogalmazása is benne van: az olyan tipusszerkezetet fogjuk elfogadhatatlannak nevezni, melyben létezik két meghatározatlan metszetü tipus. Ha ugyanis egy tipusszerkezetben pl. A és B metszete meghatározatlan, a felhasználó olyan leirást készithet, ahol egy X objektumot először A tipusuként definiál, majd arra B tipusuként hivatkozik. A tervező rendszernek X tipusát A és B metszetére kellene módositania, ami viszont meghatározatlan, igy X tipusa is az lesz. A jelenlegi SDLA verzió az elfogadhatósági kritériumot kielégitő tipusszerkezetet fogadja el.

A fenti kritérium nem elfogadhatónak minősiti pl. a

```
concept A is B,C;
concept D is B,C;
```

tipusszerkezetet, hiszen B-nek és C-nek meghatározatlan a metszete. Ha a finomitás jelölésére a tipusból annak speciális esetére mutató nyilat használunk, a fenti tipusszerkezetet a 20. ábra szemlélteti.

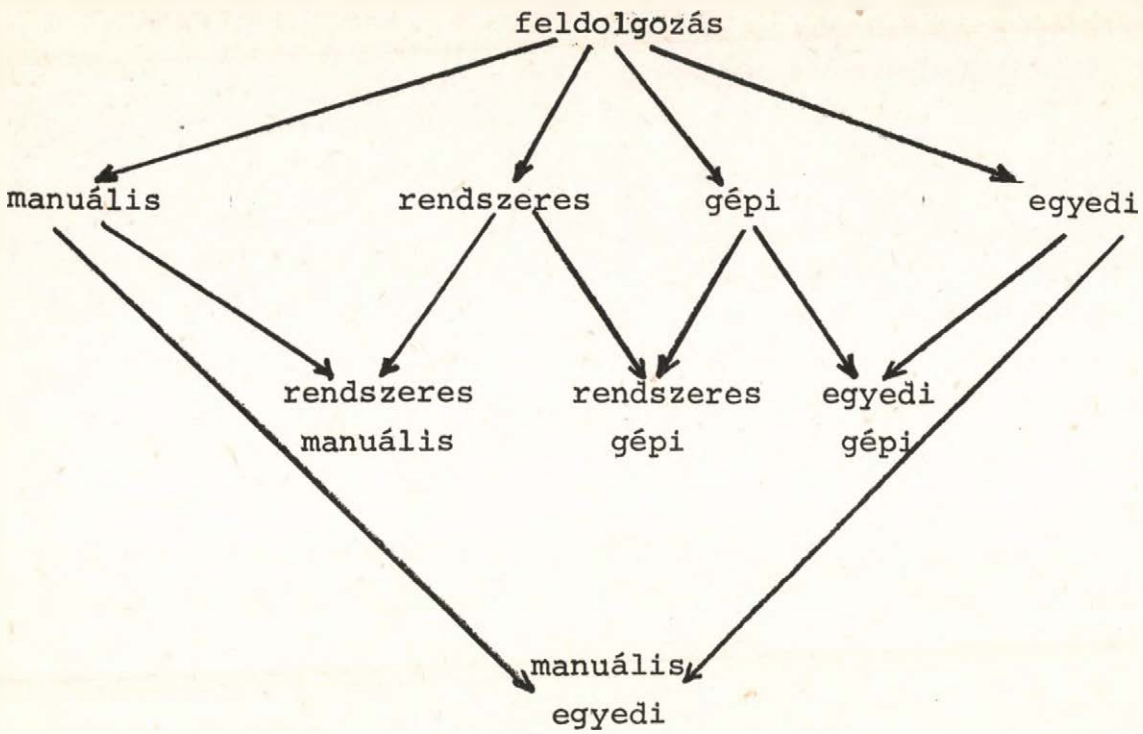



20. ábra
Egy nem elfogadható tipusszerkezet

Első ránézésre ugy tünik, hogy igen nehéz elfogadható tipusszerkezetet konstruálni, ha nem hierarchiát használunk./Minden hierarchikus tipusszerkezet nyilvánvalóan elfogadható lesz./ Valójában ez a benyomás felületes. A csak a példa kedvéért létrehozott, szemantikai tartalommal nem biró tipusszerkezetek valóban ritkán lesznek elfogadhatóak /ha véletlenszerüen felrajzolunk egy irányitott gráfot, valószinütlen, hogy elfogadható tipusszerkezethez jutunk/.

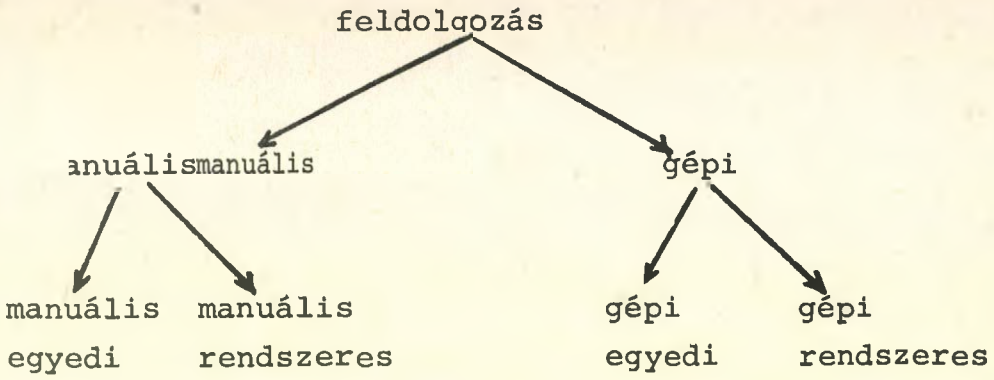
A példában definiált tipusösszefüggéseket megvizsgálva azonnal látható a hibás fogalomépitkezés. Az alapfogalmak B és C. A fogalom szemantikai tartalma - lévén mind a kettőnek finomitása - a következő: az olyan objektumok A tipusuak, melyek rendelkeznek az ugy B-t, mint C-t jellemző tulajdonságokkal. Csakhogy - érthetetlen módon - ugyanez a jelentése D-nek is! Arról van szó tehát, hogy ugyanannak a tulajdonságosztálynak két különböző fogalom felel meg - és ettől elfogadhatatlan a tipusszerkezet.

A hierarchikus tipusszerkezet jól használható olyan esetekben, amikor egy fogalmat valamilyen szempont szerint részfogalmakra kell bontani. Amikor több szempont szerint kell a fogalmat felbontani olyankor érdemes az SDLA hálós tipusszerkezet lehetőségét kihasználni. A 21. ábra erre hoz példát.




21. ábra
Egy elfogadható tipusszerkezet

Egy rendszerben a feldolgozások lehetnek manuálisak és gépiek, ill. rendszeresek és egyediek. Ez két különböző szempont. A hálós szerkezet kellemes tulajdonsága, hogy a két különböző felbontás egyenrangu. Ha összevetjük a 22. ábrán látható ekvivalens hierarchikus felbontással, látható, hogy ott pl. egy egyedi feldolgozásnál először tisztázni kell, hogy manuális vagy gépi, és csak utána vihető be a tervező rendszerbe.




22. ábra
"A feldolgozás" fogalom hierarchikus felbontása

A hierarchikus tipusszerkezet nagyon rokonszenves vonása egyszerüsége /ez a 21. és 22. ábrát összehasonlitva azonnal érzékelhető/. Az is igaz, hogy az esetek tulnyomó többségében elégséges. Mégis ugy tünik, hogy a bonyolultabb alkalmazások, főleg az előregyártott SDLA tipusmodellek /ld. 3./ érdekében érdemes megtartani a hálós tipusszerkezet lehetőségét.

Az SDLA definiciós szintjén az, hogy az A fogalom a B finomitása, az A megadásakor jelezhető a

concept A is B( X1,X2,...Xn);

állitással. Az A objektum automatikusan rendelkezni fog a B valamennyi attributumával, és továbbiak is definiálhatók hozzá /X1,X2,...,Xn/. A hálós szerkezet kialakitása a

concept A is B1,B2,...,Bn;

tipusu állitással történik. Ilyenkor A természetesen valamennyi Bi i=1,2,...,n attributumait örökli.

3.1.3.2. Kényszeritések

A kényszeritás nem ellenőrzés, még csak ilyen jellegü mellékhatása sincs mint pl. a tipusszerkezetnek -,mégis mint látni fogjuk, vitathatatlanul szemantikai tevékenység, példa tehát arra, hogy a szemantikus összefüggések nem korlátozódnak a szintaktikusnál bonyolultabb ellenőrzésekre.

A kényszeritás az adatfelvitel során objektumok automatikus generálását okozza, vagyis nem a leirás konzisztenciájának ellenőrzése, hanem a konzisztencia automatikus biztositása a célja. A pontos müködési mechanizmusa legvilágosabban a már többször emlitett, a PSL leiró nyelvből vett példán keresztül érthető meg.

A PSL a

```
process P;
    uses I to derive O;
```

leirásrészlet hatására nem csak a P folyamat, I input és O output közötti "felhasználja az előállitásához" /uses to derive/ kapcsolatot tartja nyilván, hanem ettől függetlenül még két másik kapcsolatot is; a P az I objektummal "használja" /uses/, a O-val pedig "előállitja" /derives/ viszonyba kerül.

A rendszer képes annak felismerésére, hogy ha a P folyamat az I objektumot felhasználja az O előállitásához, akkor a P által használt objektumok között

/a P-vel "uses" kapcsolatban álló objektumok között/ szerepelnie kell az I-nek, az előállitottak között pedig az O-nak. A PSL - kötött fogalomkészlettel biró nyelv lévén - természetesen nem teheti általánositható-vá tetszőleges fogalmakra ezt a szemantikus összefüggést, csupán a "felhasználja az előállitáshoz" kapcsolat mellékhatásaként kezeli.

Az SDLA-ban, ha két fogalom a fenti példához hasonló jellegü kapcsolatban áll, azt kényszeritésnek nevezzük. Megadása definiciós szinten a

```
concept A(B1,B2,...,Bn);
    implies  C(Bi1,Bi2,...,Bik);
```

állitással történik. A definicióban szereplő A-t kényszeritő, a C-t pedig kényszeritett fogalomnak nevezzük. Hatására leiró szinten minden A tipusu objektum létrehozatalakor egy B tipusu objektum is automatikusan létrejön. Ez az objektum attributumai értékét a kényszeritės definiálásánál megadott módon - tehát a kényszeritő objektum attributumai részhalmazának valamilyen permutációjaként - veszi fel. A kényszeritő ciklusok /pl. A kényszeriti B-t, B pedig A-t/ elkerülése érdekében a kényszeritett objektumnak a kényszeritőénél kevesebb számu attributummal kell rendelkeznie.

A fenti példával bemutatott szemantikai tevékenység az SDLA-ban tehát a következő kényszeritésekkel irható le:

```
concept uses(process,input);
concept derives(process,output);
concept uses to derive(process,input,output);
    implies uses(process,input);
    implies derives(process,output);
```

/Felhivjuk a figyelmet arra, hogy a definició zártságának elve itt is érvényesül. A "uses" és"derives" fogalmakat explicite definiálni kell, ahhoz, hogy kényszeritett fogalmakként legálisan lehessen hivatkozni rájuk./

Érdemes összehasonlitani a kényszeritést és a tipusfinomitást. Ugyanis pl.

concept uses to derive is uses (output);

definició hatására is elérhető, hogy a

uses to derive(P,I,O);

objektum a

uses(P,I);

nyilvántartását maga után vonja /A "uses" tipusu objektumok lekérdezésénél pl. a ez az objektum is megjelenik./ A lényegi különbség ott van a két mechanizmus müködése között, hogy a kényszeritès hatására kényszeritőtől független uj objektum keletkezik, mig a tipusszerkezetnél pusztán arról van szó, hogy az egy létrejövő objektumot minden olyan helyzetben használni lehet - tehát le is lehet kérdezni - ahol a nálánál durvább tipusuak használhatóak. /A kényszeritett tipusu objektum nem fogadható el a kényszeritő tipusu helyett, ha a két tipus összeegyeztethetetlen./

Különösen szembeötlő a két mechanizmus különbsége törlés esetén. Ha egy objektumot törlünk a rendszerből, az természetesen semmiféle tipusuként nem lesz többé megtalálható, viszont az általa kényszeritett objektumok

- lévén születésük után tőle teljesen függetlenek - maradnak. /Ez igy logikus: abból, hogy a "P folyamat felhasználja I-t O előállitásához" információ nem igaz, még nem következik sem az, hogy a P nem használja fel - esetleg valami más előállitásához - az I-t, és az sem, hogy nem állitja elő az O-t./

Szükség lenne tulajdonképpen a kényszeritéshez némileg hasonló másik mechanizmusra is. A kényszeritésnél arról van szó, hogy egy X objektum létezése feltételezi egy másikét is, tehát amikor a felhasználó X-et definiálja, a rendszer automatikusan generálja azt. Elképzelhető olyan eset is, amikor generálás helyett célszerübb lenne hibaüzenettel felszólitani a felhasználót, hogy X létrehozásának - noha önmagában véve korrekt módon kérte - akadályai vannak. Jó példa erre a használt, de létrehozni elfelejtett objektumok példája. Itt arról van szó, hogy a leirásban időnként olyan objektumok szerepelnek más objektumok forrásaként, melyeknek a rendszerbe kerülését elfelejtette /vagy majd később szándékozik/ leirni a felhasználó. Némileg formalizálva:

```
concept uses(user: process,used:input);
    presumes derives(anything,used:output);
```

A jelenlegi SDLA ilyen lehetőséget egyelőre nem tartalmaz, elvileg semmi akadálya nincs annak, hogy bekerüljön a rendszerbe.

### 3.1.3.3. Funkcionális függőség

Ez tisztán ellenőrzés jellegü funkció. Formálisan három tipusát különböztetjük meg:

function;
function of attributum$_1$,...,attributum$_n$;
define attributum$_1$,...,attributum$_k$, as function of
attributum$_{k+1}$,...,attributum$_n$;

Az első állitás azt deklarálja, hogy a fogalom előfordulásai között nem szerepelhet két különböző nevü, de azonos attributumértékekkel rendelkező objektum. Például a

concept város ( hosszusági fok: real,szélességi fok:real);
function;

fogalom esetében az attributumok - a földrajzi elhelyezkedés - nyilván meghatározzák az objektum nevét, egy helyen /bizonyos pontossági szint felett/ két különböző nevü város nem lehet /Leszámitva persze azt a lehetőséget, hogy ugyanaz a városnév különböző városok neve lehet./

A második állitás az első általánositása, előirja, hogy a felsorolt attributumok értékei meghatározzák az objektum nevét. A

concept elsődleges kulcs(reláció,tipus,hossz:integer);
function of reláció;

a definiciórészlet a relációs adatmodell "elsődleges kulcs" fogalmának azt a tulajdonságát fejezi ki, hogy egy relációhoz csak egy adható meg belőle. Ez azt jelenti, hogy a reláció neve - tehát a "reláció" attributum értéke - meghatározza az elsődleges kulcs - tehát a fogalomelőfordulás - nevét.

A harmadik állitás az attributumok közötti funkcionális függőségek megadására szolgál. Megadja, hogy a

második attributumcsoport értékei meghatározzák az első csoportban szereplő attributumokét, A

```
concept codasyl halmaz(név,tulajdonos:rekord,tag:rekord);
  define tulajdonos as function of név;
```

definiciórészlet a CODASYL halmaznak azt a tulajdonságát irja le, hogy csak egy tulajdonos rekordtipusa lehet, azaz a halmaz neve meghatározza a tulajdonos nevét.

Ez utóbbi tipusu funkcionális függőség azonos a relációs adatmodellben használt azonos nevü fogalommal, formájában és szemantikai tartalmában egyaránt. /Az első két tipusnak a relációs adatmodellben nem lehet megfelelője, hiszen ott a sorok névtelenek./ Ez jó példa arra is, hogy a számitástudomány általános fejlődésének eredményei jól használhatóak a tervező rendszerek fejlesztésénél is: a funkcionális függőségek kutatásának eredményei /pl. [56], vagy[57] / az SDLA és általában a tervező rendszerek konceptuális sémája tervezésénél, az előirt ellenőrzések végrehajtási stratégiájának kialakitásánál alkalmazhatóak.

Az egymástól függetlennek látszó szemantikai összefüggések között első ránézésre nem nyilvánvaló kapcsolatok vannak. Ezt illusztrálja pl. az a tény, hogy egy fogalomra kimondott funkcionális függőségeknek annak valamennyi finomitására is vonatkozniuk kell. Ez a rendszer logikájából szükségszerüen adódik, hiszen a finomabb fogalom egy előfordulása bármikor használható a durvább fogalomelőfordulás helyett, és vigyázni kell arra, hogy ilyenkor se sérüljenek meg az előirt függőségek.

## 3.2. A leiró szint

Az SDLA leiró szintje a definiciók rögzitése és a táblázatok generálása után önálló tervező rendszernek tekinthető. Ennek ellenére beszélhetünk általában az SDLA leiró szintjéről, mert a különböző definiciós szintek alapján müködő tervező rendszerek leiró nyelvében közös, valamennyiüket egyaránt jellemző vonások vannak. Ezekről lesz a következőkben szó.

### 3.2.1 Objektum létrehozása és azonositása

A felhasználó relativ és abszolut formákat használva viheti be leirását a tervező rendszerbe /3.1.2./. A leiró szinten használt formákat mondatnak fogjuk nevezni. Minden mondat vagy létrehozza a formának megfelelő fogalom egy uj előfordulását, vagy egy már régebben létező előfordulásra hivatkozik. Ha a

```
process P;
   ⟨a P folyamatra vonatkozó állitások⟩
      :
process P;
   ⟨a P folyamatra vonatkozó további állitások⟩
```

leirásrészlet első mondata P első előfordulása a rendszerben, hatására létrejön az objektum. A második alkalommal a rendszer felismeri, hogy már létező objektumra való hivatkozásról van szó, nem hoz létre uj objektumot, viszont a szekcióban következő mondatokat a régi P folyamathoz tartozónak tekinti /a hatásukra létrejövő objektumok egyik attributuma P lesz, a relativ formák

használati módjának megfelelően/,

Ha egy fogalom valamelyik előfordulása a fogalomhoz definiált valamelyik relativ, vagy abszolut formával jön létre, akkor explicit definicióról beszélünk. Ilyen esetekben a létrejövő objektum tipusa nyilvánvalóan az a fogalom, melynek előfordulásaként definiáljuk.

Az explicit mellett lehetőség van az objektumok implicit definiálására is. A

```
concept process;
concept input;
concept usage ( process,input);
form  process:uses input;
```

definiciórészletnek megfelelően a rendszer korrektnek tekinti a

```
process P;
    uses I;
```

leirásrészletet, akkor is, ha az I objektum nem szerepelt előzőleg explicit definicióban. A második mondat hatására létre kell hozni a

```
usage(P,I);
```

objektumot, melyhez a rendszer automatikusan generálja az I "input" tipusu objektumot. Implicit definicióról tehát olyankor beszélünk, amikor az uj fogalomelőfordulás nem valamelyik megfelelő formával, - tehát explicite - hanem más módon, pl. egy explicite definiált objektum attributumaként automatikusan jön létre. /Nem ez az implicit definició egyetlen lehetősége: a

kényszerités hatására létrejövő objektumok is implicite definiáltak./ Az uj objektum tipusa ilyenkor a definiciós szinten megadott lesz.

3.1.1-ben szó volt a definició zártságának elvéről. Az a döntés, hogy a leiró nyelvben lemondunk a zártság megköveteléséről és megengedjük az implicit definiciót, a kényelmesebb felhasználást segiti elő, elég sok irást tesz szükségtelenné. A másik oldalról természetesen csökkenti a hibaellenőrzést, nem véd az elirások ellen. Jelen esetben a kényelem mellett dontöttünk, abból a meggondolásból, hogy az elirás jellegü hibák a leirás fokozatos kiépitése, az áttekintő és ellenőrző listázások során elég hamar előbukkannak.

Egy beérkező mondat tehát vagy explicit definició, vagy létező objektumra történő hivatkozás. Ahhoz, hogy a két eset egymástól elválasztható legyen, szükség van egy azonositó mechanizmusra, amely két objektumról eldönti, hogy azonosak-e. Az SDLA-ban a következő szabály érvényes:

- a névvel rendelkező objektumokat a nevük,
- a névtelen objektumokat tartalmuk /tipusuk és attributumaik értéke/ azonositja.

Egy korábban definiált objektumra való hivatkozásnál változhat az objektum tipusa. Ennek mechanizmusáról elég részletesen volt szó 3.1.3.1-ben, igy most csak egy példával illusztráljuk. Legyenek érvényesek a

```
concept sorted file;
concept random file;
concept vsam file is sorted file, random file;
concept random use(program,random file);
form program: requires random access of random file;
concept sorted use (program,sorted file);
form program: requires sequential access of sorted file;
```

definiciók. Az

```
account checking: program;
        requires random access of account file;
program  account listing;
        requires sequential access of account file;
```

leirásrészlet hatására az "account file" nevü objektum "random file" tipusu objektumként létrejön, majd amikor a negyedik mondatban hivatkozás történik rá, a tipusa "vsam"-ra finomodik.

### 3.2.2. A nézőpont_verem

3.1.2-ben a formákkal kapcsolatosan volt szó a nézőpontról. Akkor azt mondtuk, hogy a szekciókból álló leiró nyelvben a szekció első állitása /a szekció feje/ meghatároz egy objektumot, és a szekcióba tartozó állitások erre az objektumra vonatkoznak. Valójában az SDLA leiró nyelvének szerkezete ennél valamivel általánosabb, minden uj mondat uj nézőpontot hoz létre, anélkül, hogy az őt létrehozó nézőpontot megszüntetné.

A már sokszor használt

```
process P;
   uses I;
```

leirásrészletet vizsgálva láthatjuk, hogy az első mondat a

```
P:process;
```

objektum, a másik pedig a

usage(P,I);

objektum explicit definiciója. AZ explicit definició egyben nézőpontot is létrehoz, tehát a leirásrészlet végén két nézőpont - a "P"-é és a névtelen" usage(P,I)"-é - is érvényes. Ha pl. a

```
concept update (usage,data);
form usage: to update data;
```

definiciót használjuk, ugy a leirás, pl. igy egészithető ki:

```
process P;
   uses I;
     to update O;
   uses J;
```

A harmadik mondat a "usage" nézőpontját, a negyedik ismét a "P" objektumét használta.

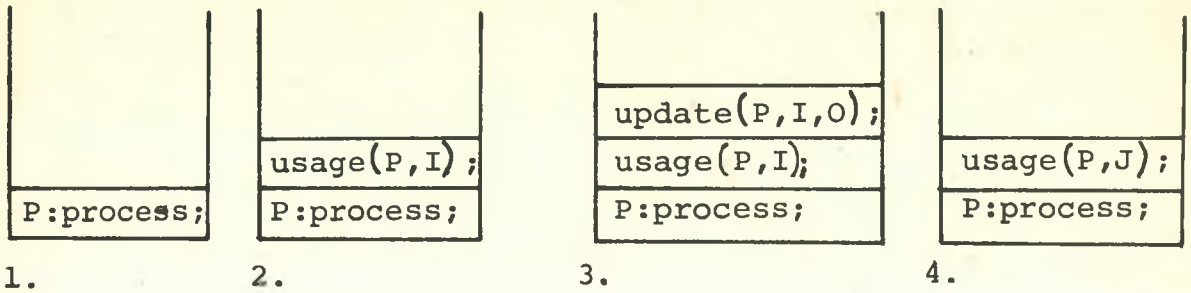
Az érvényes nézőpontok egymásra következését az uj mondatok értelmezését a nézőpont verem szabályozza. Ennek elemei objektumok, és a következő módon épül fel.

Kezdeti állapotban a verem üres, ilyenkor csak az abszolut formák használhatóak. Ha egy uj mondat érkezik, a rendszer megkisérli a verem legfelső elemének nézőpontjából értelmezni. Ha a mondat értelmezhető/az illető objektumtipus vagy annak egy finomitása nézőpontjából definiált relativ forma, ill. üres verem esetén abszolut forma/, akkor az általa explicite definiált, ill. hivatkozott objektum a verem tetejére kerül, ellenkező esetben a legfelső elemet ki kell venni a veremből és a következő elemmel próbálkozni, és igy tovább. Ha a verem kiürül, és a mondat abszolut forma, akkor bekerül a

verembe, ha pedig nem az, hibás mondatként elutasitásra kerül, és a verem visszaáll a mondat beérkezése előtti állapotára. A

```
process P;
    uses I;
      to update O;
    uses J;
```

feljebb már használt leirásrészletnek megfelelő veremállapotok az egyes mondatok beérkezése után:




### 3.2.3. Alrendszerek

Gyakran fordul elő, hogy a leirandó rendszer nagy mérete és adott strukturája szükségessé és lehetségessé teszi részrendszerekre bontását, A cél ilyenkor nyilván az, hogy az egyes részek egymástól minél függetlenebbek legyenek, lehetőséget adva a feladat szétdarabolására a szervező team tagjai között.

Az SDLA ezt támogatja a leiró nyelv egyik standard - minden generált tervező rendszerre jellemző - lehetőségével. A felhasználó egymás mellé, illetve alá rendelt független alrendszereket definiálhat, és használhat.

Az elgondolás hasonlit a blokkszerkezetü programozási

nyelv koncepciójára. Ott a program részegységekre bontása a cél. A megoldás a hierarchikusan egymásra épülő blokkok használata. A blokkok függetlensége a változónevek jól ismert lokalitási szabályaival érhető el.

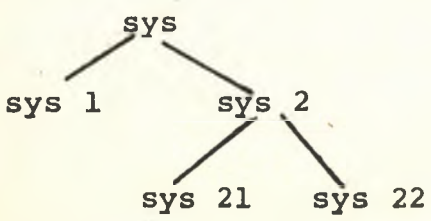
Az SDLA is a rendszer hierarchikus felbontását támogatja. Alrendszerek a leiró nyelvben bárhol használható

create ⟨alrendszer neve⟩;

mondattal hozhatók létre, Létező alrendszerbe való belépés az

enter ⟨alrendszer neve⟩;

mondattal történik. A "create" hatására létrejövő alrendszer a hierarchián belül közvetlenül alá lesz rendelve annak, amelyből létrehozták. A 23. ábrán egy alrendszer hiererchia és az azt létrehozó állitások láthatók.




```
create sys;
create sys 1;
enter  sys;
create sys 2;
create sys 21;
enter sys 2;
create sys 22;
```

23. ábra
Alrendszerek létrehozása

Az alrendszerekben definiált nevek érvényességi körére ugyanazok a szabályok vonatkoznak, mint a programozási nyelveknél az egyes blokkokban definiált azonositókra. Ennek alapján az alrendszerekre bontott rendszer leirása ideálisan a következő módon történik:

A felhasználó minden alrendszerben csupán a közvetlenül alá rendelt alrendszerek globális kapcsolatát specifikálja. Az alárendelt alrendszerek leirásai egymástól függetlenül történnek, de mindegyik esetében a feljebb specifikált kapcsolatból lehet kiindulni,és ehhez alakitani az alrendszer belsejét. Ez a módszer lényegében a felülről lefelé történő tervezésnek felel meg.

### 3.2.4. Törlés, módositás

A PSL/PSA-val ellentétben az SDLA-nál a törlés és módositás nem külön parancsrendszer, hanem - elegánsabb megoldással - mind a két funkció a leiró nyelv része. A törlés a

cancel ⟨mondat⟩

a módositás pedig a

modify ⟨mondat⟩
⟨uj mondat⟩

utasitásokkal történik. Az egyetlen szintaktikai megkötés, hogy a mondatnak valamelyik aktuális nézőpontból érvényesnek kell lennie, és létező objektumra kell hivatkoznia.

Néhány példa:

```
1.      cancel process P;
2.      process P;
3.        cancel uses I;
4.        modify uses O;
5.          uses O1;
6.      modify input I;
7.        input J;
```

Az első mondat a P objektumot törli. A harmadik mondat a

usage(P,I);

objektumot törli, a negyedik és az ötödik a

usage(P,O);

objektumban az O-t O1-re cseréli. A hatodik és hetedik mondat az

I:input;

objektum nevét változtatja meg.

Általános elv a törlés és módositás helyességének ellenőrzésénél, hogy a művelet végrehajtása után a szemantikus összefüggések ne sérüljenek meg. Ez egy sor elég bonyolult ellenőrzést jelent, pl. törlés előtt meg kell vizsgálni, hogy nem kényszeritett objektumot kiván-e a felhasználó törölni, anélkül, hogy a kényszeritőt előzetesen törölte volna, stb.

## 3.3. A lekérdező rendszer

A lekérdező rendszerek rendeltetésük szerint lehetőséget

adnak felhasználójuknak, hogy az információtömegből

- csak az őt érdeklő objektumoknak,
- csak az őt érdeklő kapcsolatait,
- a számára megfelelő formátumban

kiirathassa /sornyomtatóra, vagy terminálra, ez nagyjából mindegy/. A végrehajtandó feladat specifikációja a lekérdező nyelven történik, melynek - elvileg - mind a három igény megadására lehetőséget kell biztositania, és ehhez még könnyen használhatónak, egyszerűnek is kell lennie.

A tervező rendszerek lekérdező nyelve esetében ezt a szempontot különösen fontosnak tartom, ugyanis egyfelől annyira sokrétüek és speciálisak a kiiratásra vonatkozó igények - grafikus listáktól a legkülönfélébb dokumentálási szabványokig, - másfelől annyira képzetlen és a számitógép iránt nem tul nagy érdeklődést tanusitó felhasználókra kell számitani, hogy eléggé erős és elég egyszerű nyelvet még rögzitett leiró nyelvü rendszerhez is nehéz tervezni. Realistább stratégia véleményem szerint, ha az egyszerüséget tartva elsősorban szem előtt olyan nyelv készitésére törekszünk, mellyel az igények minél nagyobb részét ki lehet elégiteni. Emellett segédeszközöket kell biztositani a speciális listák elkészitéséhez/jól használható adatkezelő interface, a lekérdezés eredményeinek mágneslemezre irányithatósága felhasználói programmal feldolgozható formában, stb./, nem pedig az összes speciális eset lefedésére törekedni.

Az SDLA esetében az egyszerüségre törekvés a következő alapelvben nyilvánul meg: a lekérdezések a normális leiró nyelven specifikálhatóak /a formamegadásra emlékeztető módon/,tehát a felhasználónak nem kell

uj nyelvet megtanulni. Természetesen szükség van kiegészitő eszközökre, ezért bevezetjük a "szabad változó" fogalmát. /Nevét a predikátumkalkulussal való analógia alapján kölcsönöztük/ szabad változók lehetnek a következők:

- any,
- any < név>,
- érték tipusok: integer,real,text .

A szabad változók a formáknak megfelelő mondatokban az attributumok helyén állhatnak. Az "any" és az "any név" a fogalom, az érték tipusok pedig az érték tipusu attributumok helyén állhat.

Egy egyszerü lekérdezés specifikáció például a következő:

```
report;
   process any;
       uses any;
equ;
```

A lekérdezés eredménye igy néz ki:

```
process P;
   uses I;
   uses J;
    :
process Q
   uses K;
   uses L;
    :
```

A specifikáció mint a példából is látható - meghatározza az output tartalmát és formátumát is.

A lista generálása logikailag a következő módon történik. A szabad változók helyére sorban behelyettesitődik az adatbázisban tárolt megfelelő tipusu /ez az adott tipust is annak finomitásait jelenti/, összes objektum, és ezzel konkrét adatneveket tartalmazó leirásrészleteket nyerünk. A listába azok a leirásrészletek kerülnek be, melyek egészükben is megfelelnek az adatbázis tartalmának. Ezeket összerendezve /a példánkban pl. egy folyamat által használt valamennyi objektum egy szekcióban van/ irja ki a lekérdező rendszer.

Az "any" és az "any név " szabad változók a helyettesités módjában különböznek egymástól. Valamennyi "any" különböző változónak számit, tehát ezekbe az objektumok behelyettesitése egymástól függetlenül történik. Az "any név" tipusu változók esetében azonban az azonos "név" résszel rendelkező változók azonosak. Pl. a

```
report;
   process any;
      uses any X;
   process Q;
      derives X;
   equ;
```

specifikáció alapján /másodszorra más nem kell az X elé az "any"-t kiirni/ csak azok az "X" adatok kerülnek be a listába, melyekre egyidejüleg fennáll, hogy valamilyen folyamat használja /uses/, és a Q folyamat pedig előállitja /derives/ őket.

A lekérdezésekhez hasonló módon definiálhatók a halmazok /set/ Pl. a

```
set D-users;
   process any;
      uses D;
equ;
```

specifikáció a "D" objektumot használó folyamatok halmazát adja meg. A halmaz generálása a lekérdezéséhez hasonlóan történik, de az eredmény nem leirás lesz, hanem /összeegyeztethető tipusu/ nevek halmaza. A nevek közös tipusa /a tipusok metszete/ lesz a halmaz tipusa.

A definiált halmazok további lekérdezések, ill. halmazok specifikálásához használható. Pl. az

```
input any;
   used by D-users;
```

specifikáció a D-t használó folyamatok által használt adatok megadása.

Be lehetne vezetni - a jelenlegi rendszerben nem szerepel - a "none" szabad változót is. Használatával az

```
input any;
   used by any;
   derived by none;
```

specifikáció az "elfelejtett" /használt, de elő nem állitott/ objektumokat irná ki. A "none" annyiban módositaná a válasz generálásának folyamatát, hogy az "any" változók konkrét objektumnevekkel való helyettesitése után a leirásrészlet akkor kerülne be a listába, ha a "none"

helyére nem irható semmilyen objektumnév ugy, hogy a leirásrészlet továbbra is megfeleljen az adatbázis tartalmának.

## 3.4. Adatkezelés

A tervező rendszerek adatkezelésének problémáiról 1.2-ben és 2.3.4-ben már szó volt, most bemutatjuk az ott kifejtett általános elképzelések egy lehetséges megvalósitását. Igyekeztünk ugy tervezni az SDLA adatkezelő rendszerét [58] hogy a megfogalmazott elveknek megfelelően a rendszer többi részétől független - tehát bármilyen környezetben használható - és hatékony legyen. A felhasználói interface önmagában zárt logikai rendszer - ez a használatát könnyiti meg, nincs szükség az implementációs részletek ismertetésére.

Mivel az SDLA konceptuális sémája relációs táblákkal dolgozik, célszerünek látszott egy relációs interface kialakitása. Ez egy CODASYL tipusu adatbáziskezelő rendszerre [59],[60] épül. Először felhasználói oldalról mutatjuk be az adatkezelési lehetőségeket, majd a megvalósitás logikai sémáját ismertetjük, végül a hatékonyság szempontjából legfontosabb fizikai implementációs részletekről lesz szó.

### 3.4.1. A relációs_interface

Az interface logikai adatmodellje az SDLA konceptuális sémája adatmodelljével /2.3.2./ egyezik meg,

használatához elégséges ennek a modellnek és az adatkezelő rutinok használati módjának az ismerete.

Az azonositási mechanizmusnak megfelelően /3.2.1/ a névvel rendelkező objektumot a neve, a névtelent pedig attributumai értéke és a tipusa /a tartalma/ azonositja. Ez a hivatkozási mód a leiró nyelv felhasználója számára kényelmes, de ezen a szinten már nehézkes és nem eléggé hatékony.

Az adatkezelő rendszer minden névhez és minden /logikailag/ tárolt névtelen relációs referencia sorhoz egyértelmü belső azonositót rendel. Ez az azonositó névvel rendelkező objektum esetén nem a relációs referencia sort azonositja, hanem magát a nevet. A kettő nem egészen ugyanaz, mivel a tipusszerkezet következtében egy névvel azonositott objektumnak több különböző tipusu sor felelhet meg. A névtelen objektumok esetében ilyen probléma nincs, őket a tartalmuk - ebben benne van a tipus is - azonositja. Látható tehát, hogy névvel rendelkező objektumoknál a név és a tipus vagy az azonositó és a tipus, névteleneknél pedig az azonositó egyértelmüen meghatározza a relációs sort.

A relációs sorok manipulálása rutinhivásokkal valósul meg. A rutinok paraméterei általában azonositók. Az interface öt rutinból áll és kb. a relációs adatbáziskezelő rendszerek adatbázis assembler szintjének [25] felel meg. Az öt rutin a következő:

- CREATE - Paraméterként adott relációs táblába uj sort illeszt be. A rutin paraméterei a sort alkotó elemek azonositói.
- ERASE - Azonositóval adott sort töröl adott relációs táblából.
- MODIFY - Azonositóval adott sor adott elemét módositja adott táblában.

- FIND - A névvel vagy azonositóval és tipussal /relációs táblával/ megadott sort olvassa ki az adatbázisból.
- SELECT - A sorok tartalmuk szerinti visszakeresését végzi tetszőlegesen megadható attributumkombinációra, rögzitett vagy rögzitetlen tipus /relációs tábla/ mellett.

### 3.4.2. A CODASYL implementáció

A CODASYL sémában minden névnek egy rekord /NAMREC/ felel meg. A nevet minden olyan alrendszerben, ahol definiálta a felhasználó külön SYSREC rekord képviseli. Az alrendszerek szerkezetét - és ezzel együtt a nevek érvényességének körét - a hierarchia CODASYL ábrázolása tükrözi.

Egy alrendszeren belül ugyanaz a név - noha az általa meghatározott objektumnak a tipusa és attributumainak értékei egyértelmüek - a tipusszerkezet következtében annál durvább tipusuként, esetleg kevesebb attributummal is szerepelhet hivatkozásokban. Az ábrázolás megoldható lenne a legfinomabb tipus fizikai tárolásával, ugyanis durvább tipusok ebből a tipusszerkezetre vonatkozó általános információval előállithatóak. Nem ezt a megoldást választottuk, mert ez megbontotta volna az adatkezelő rendszer függetlenségét a rendszer többi részétől. Ehelyett bevezetjük az OBJREC rekordtipust, mely egy adott részrendszerben szereplő adott tipusu relációs referencia sor reprezentánsa.

A nevek közötti kapcsolat, vagyis a relációs sorok ábrázolása a klasszikus alkatrész problémára emlékeztet, hiszen arról van szó, hogy az A név /mely

maga is több elemü sor neve lehet/ eleme-e a B név által reprezentált sornak. A megoldás erre két halmaz, a tartalmazó /consists, CNSTS/ és a tartalmazott /contained, CNTND/ és a kapcsolatot reprezentáló rekord /connection, CONREC/ bevezetése. Esetünkben ez a

```
SET CNSTS
OWNER OBJREC
MEMBER CONREC

SET CNTND
OWNER SYSREC
MEMBER CONREC
```

sémához vezet. A CNSTS halmaz tulajdonosa azért az OBJREC, mert egy relációs sor nem más, mint ennek a halmaznak előfordulása /a CONREC-ekhez a CNTND halmazban tartozó tulajdonos képviseli az attributum értékét/. Innen a CNTND halmaz jelentése is látható - lényegében mutató azokra a sorokra, melyekben a SYSREC által képviselt név elemként résztvesz. A két halmaz tulajdonosai különbözőek, ugyanis a relációs sor reprezentánsa - mint láttuk - nem lehet a SYSREC, viszont amikor az a kérdés, hogy egy név milyen sorokban szerepel, akkor közömbös, hogy milyen relációs sorként /tipusként/.

### 3.4.3. A fizikai tárolás

Az SDLA rendszer tárolási stratégiája név szerinti elérés orientált, vagyis azt szerettük volna elérni, hogy ha egy név adva van, gyorsan meg lehessen kapni a név által reprezentált /adott tipusu/ sor elemeit, illetve azt, hogy a név milyen sorokban

szerepel elemként. Ezt a hozzáállást a PSL/PSA gyorsitása során szerzett tapasztalataink indokolják, ahol a rendszer lassu müködését éppen a nevek szerinti nehézkes hozzáférési lehetőség okozta. /PSL/PSA nélkül is elég nyilvánvaló persze, hogy egy név beérkezésekor utána kell nézni, hogy benn van-e az adatbázisban, ha a név egy sornak ez eleme, megvizsgálni, hogy a sor nem szerepel-e már, stb./.

Mind a két, minket érdeklő kérdésnél /milyen elemei vannak a név által reprezentált sornak, illetve milyen sorokban szerepel elemként a név/ a névtől indulunk, tehát először a NAMREC-et kell megtalálni. A rendszer ezt hash-eli, tehát a tartalmazó lap egy I/O müvelettel beolvasható. /A tárgyalás egyszerüsitése végett itt, és a továbbiakban is eltekintettünk a tulcsordulás problémájától./ Ugyanazon a lapon /"near to owner" stratégia/ helyezkedik el a SYSREC és az OBJREC is.

Ha a név által reprezentált sor elemeit akarjuk megkapni, a CNSTS halmaz adott előfordulását kell bejárni, tehát egy pointerláncot kell követni, további "keresés" nem szükséges. Megjegyezzük ugyanakkor, hogy minden elemet reprezentáló CONREC beolvasása általában uj I/O müvelet, és a CONREC-ből az elem nevét további lap beolvasásával lehet csak megkapni.

Ennek hatékonyabbá tételénél fontosabbnak tartottuk ugyanis annak a felvitel során nagyon gyakori kérdésnek a gyors megválaszolhatóságát, hogy adott elemek adott tipusu relációban állnak-e egymással. Kedvező számunkra, hogy - mivel a CONREC-ek elhelyezése hash algoritmussal történik /a kulcs a reláció tipusa/ - egy adott név adott tipusu sorokban való részvételét reprezentáló összes CONREC egy lapra kerül. Ezen a ponton kihasználjuk a hash algoritmusnak azt a tulajdonságát,

hogy a kulcs mellett az adott rekord elhelyezéséhez - a felhasználó rendelkezése szerint - paraméterként szolgálhat a rekord valamelyik halmazbeli tulajdonosának adatbázis kulcsa [60]. Ennek következtében nem kerül az összes azonos relációtipusu CONREC egy - a tipusok nagy részénél nyilván gyorsan tulcsorduló - lapra, azonban a CNTND halmazban közös tulajdonosuak - az egy névhez tartozók - igen. Látható, hogy egy név összes azonos tipusu kapcsolatait tároló rekordokhoz két I/O müvelettel juthatunk el /tulcsordulástól eltekintve/. Ezekből a rekordokból természetesen nem derül ki, hogy a reláció melyik soráról van szó, és a sorban még milyen nevek szerepelnek, ez a CNSTS set előfordulásának /nem teljes/ bejárásával, és a CONREC-ekből a nevek elérésével derithető ki.

## Köszönetnyilvánitás

Szeretnék köszönetet mondani az SDLA projekt valamennyi résztvevőjének, akikkel együtt dolgoztunk a rendszer megvalósitásán. Hálás vagyok társszerzőimnek értékes megjegyzéseikért, ötleteikért és az alkotó vitákért. Köszönettel tartozom Knuth Elődnek és Demetrovics Jánosnak, akiknek támogatása és segitőkészsége lehetőséget biztositott számomra e dolgozat megirásához. Végül köszönetet mondok az MTA SZTAKI Számitógéptudományi Főosztálya teljes kollektivájának, ahol kutatómunkámat kellemes körülmények között, jó szellemben folytathattam.

## IRODALOM


[1] B.W. Boehm; Software Engineering. IEEE Transactions on Computers, C-25 /1976/ 12, 1226-1241.

[2] Szentgyörgyi Zsuzsa; A számitástechnika müszaki fejlődése és társadalmi hatásai. MTA SZTAKI Tanulmányok, Budapest, 120/1981 1-229.

[3] D.T. Ross, K.E. Schoman; Strucutred Analysis for Requirements Definition. IEEE Transactions on Software Engineering, SE-3 /1977/ 1, 6-15.

[4] C. Gane, T.Sharon; Structured System Analysis: Tools and Techniques. Prentice Hall, 1979.

[5] T. De-Marco; Structured Analysis and System Specification. Yourdon Inc., 1978

[6] D. Teichroew, E. Winters; Recent Developments in System Analysis and Design. Atlanta Economic Review, November-December 1976, 39-46.

[7] D.M. Sage; Information System: A Brief Look into History. Datamation, November /1968/, 63-69.

[8] C.L. Biggs, E.G. Birks, W. Atkins; Managing the System Development Process. Prentice-Hall, /1980/.

[9] A.I. Wasserman; Information System Design Methodology. Journal of the American Society for Information Science, January /1980/, 5-24.

[10] N. Wirth; Program Development by Stepwise Refinement. Communications of the ACM, 14 /1971/ 4, 221-227.

[11] D. Parnas; On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM, 15 /1972/ 12, 1053-1058.

[12] O. Dahl, E. W. Dijkstra, C.A.R. Hoare; Structured Programming. London Academic, 1972.

[13] O. Dahl, B. Myhrhaug, K. Nygaard; SIMULA 67. Common Base Language. Norwegian Computing Center Publication Oslo, S-2 /1968/.

[14] A. Wijngarden et. al; Revised Report on the Algorithmic Language Algol 68. Acta Informatica, 5 /1975/, 1-236.

[15] B.H. Liskov, S. Zilles; Programming with Abstract Data Types, SIGPLAN Notcies, April 1974, 50-59.

[16] P. Wegner; Programming with ADA. Prentice-Hall, /1980/.

[17] J. Ludewig; Computer-aided Specification of Process Control Systems. Computer, 15 /1982/ 5, 12-20.

[18] B. Shneiderman et.al.; Investigations of the Utility of Detailed Flowcharts in Programming. Communications of ACM, 20/1977/ 6, 373-381.

[19] R.J. Lano; A Technique for Software and Systems Design. Elsevier North-Holland Inc., /1979/

[20] B. Shneiderman; Control Flow and Data Structure Documentation: Two Examples. Communications of ACM, 25 /1982/ 1, 55-63.

[21] N. Wirth; Algoritmusok + adatstrukturák = programok. Müszaki Könyvkiadó, Budapest, /1982/

[22] P. Bernus; Connecting SADT and ISDOS into a System Design System. MTA SZTAKI Working Papers, January /1979/.

[23] F.W. Allen, M.E.S. Loomis, M. Mannino; The Integrated Dictionary/Directory System. ACM Computing Surveys, 14 /1982/ 2, 245-286.

[24] R.J. Abbott, D.K. Moorhead; Software Requirements and Specifications: A Survey of Needs and Languages. The Journal of Systems and Software, 2 /1981/, 297-316.

[25] Radó P.; Relációs adatbáziskezelő rendszerek összehasonlitó vizsgálata. MTA SZTAKI Tanulmányok, Budapest, 156/1984, 1-171.

[26] Demetrovics J.; Relációs adatmodell. MTA SZTAKI Közlemények, Budapest, 20/1978, 21-33.

[27] Halassy B.; Az adatmodellezés elvi alapjai. III. rész: A SZIAM rendszer ismertetése. Információ és Elektronika, Budapest, 1981/3 129-136.

[28] W.P. Stevens, G.J. Myers, L.L. Constantine; Structured Design. IBM Systems Journal, 13 /1974/ 2, 115-139.

[29] G.J. Myers; Reliable Software through Composite Design. Petrocelli/Charter, New York, /1975/

[30] OS/VS1 Supervisor Logic SY24-5155-4. IBM Corporation, /1975/

[31] J.F. Stay; HIPO and Integrated Program Design. IBM Systems Journal, 15 /1976/ 2, 143-154.

[32] D.T. Ross; Structured Analysis /SA/: A Language for Communicating Ideas. IEEE Transactions on Software Engineering, SE-3 /1977/ 1, 16-34.

[33] Kiss O., Radó P.; A PSL/PSA felépitése, módositása, fejlesztése, tapasztalatok. Számitástechnika, Budapest, 11 /1980/ 7-8, 22.

[34] Gyurácz N.T., Radó P.; PSL/PSA. A nyelv eszközei. Számitástechnika, Budapest, 11 /1980/ 6, 13.

[35] P. Wegner; Programozási nyelvek - fogalmak, és kutatási irányok. /ford. Varga L./. Alkalmazott Matematikai Lapok, Budapest, 6 /1980/ 159-211.

[36] D. Teichroew, E.A. Hershey; PSL/PSA: a Computer-aided Technique for Structures Documentation and Analysis of Information Processing Systems", IEEE Transactions on Software Engineering, SE-3 /1977/ 1, 41-48.

[37] J.F. Numaker, B.R. Konsynski; Computer-aided Analysis and Design of Information Systems. Communications of ACM, 19 /1976/ 12, 674-687.

[38] The Software Development Facility /SDF/. ISDOS Ref. 79W10-0214-1, University of Michigan, August /1979/.

[39] A.A. Levene, G.P. Mullery; An Investigation of Requirement Specification Languages: Theory and Practice. Computer, May /1982/, 50-59.

[40] PROTEE IV-EDITION. La lettre de lire SIS, 1 /1978/.

[41] В.Л. Эпштейн, В. И. Сеничкин, Языковые средства архитектора АСУ. Энергия, Москва,/1979/.

[42] D.J. Pearson; The Use and Abuse of a Software Engineering System. National Computer Conference, /1979/, 1029-1035.

[43] R.J. Lauber; Development Support Systems. Computer, 15 /1982/ 5, 36-46.

[44] W.M. Mc Keenan, J.J. Hornig, D.B. Wortman; A Compiler Generator. Prentice-Hall, /1970/.

[45] C.H. A. Koster; A Compiler Generator. MR 127, Mathematish Centrum, Amsterdam, /1971/.

[46] Gyurácz N.T., Hannák L., Szokolov M.; Információs rendszerek tervezését segitő eszköz. Információ és Elektronika, Budapest,/1979/4, 206-208.

[47] ANSI/X3/SPARC Study Group on Database Management Systems. Interim Report, /1975/.

[48] P.P.S. Chen; The Entity-Relationship Model - Towards a Unified View of Data. ACM Transactions on Database Systems, 1 /1976/ 1, 9-36.

[49] T.W. Olle, H.G. Sol, A.A. Verjin-Stuart /ed./; Information Systems Design Methodologies: A Comparative Review. North Holland, Amsterdam, /1982/.

[50] Users Manual for Generalized Analyzer G 1.2. ISDOS REF. 80G12-0284-2, University of Michigan, January /1980/.

[51] E. Knuth, P. Radó, Á. Tóth; Preliminary description of SDLA. MTA SZTAKI Tanulmányok, Budapest, 105/1980 1-62.

[52] E.F. Codd; Extending the Database Relaitonal Model To Capture More Meaning. ACM Transactions on Database Systems, 4 /1979/ 4, 397-434.

[53] E. Knuth, P. Radó; Principles of Computer Aided System Description. MTA SZTAKI Tanulmányok, Budapest, 117/1981, 1-46.

[54] J. Demetrovics, E. Knuth, P. Radó; Specification Meta Systems. Computer, 15 /1982/ 5, 29-35.

[55] E. Knuth, F. Halász, P. Radó; SDLA System Descriptor and Logical Analyzer. [49]-ben, 143-173.
Lásd még MTA SZTAKI Tanulmányok, Budapest, 117/1983 125-152.

[56] J. Demetrovics; A.Békéssy; Contribution to the Theory of Data Base Relations. Discrete Mathematics, 27 /1979/, 1-10.

[57] J. Demetrovics, Gy. Gyepesi; Some Generalized Type Functional Dependencies Formalized As Equatity Set on Matrices. Discrete Applied Mahtematics, 6 /1983/, 35-47.

[58] Radó P., Kiss O., Szilléry A.; Relációs adatbázis interface. MTA SZTAKI Working Paper, Budapest, II/10 /1980/, 1-10.

[59] Radó P.; Az ISDOS DBMS uj szervezése. MTA SZATKI Working Paper, Budapest, II/2 /1979, 1-12.

[60] Kiss O., Radó P.; A CODASYL Modification Efficiency Problem. MTA SZTAKI Tanulmányok, Budapest, 113/1980, 183-193.

[61] P. Radó; On the Semantics of Description Lnaguages. MTA SZTAKI Közlemények, Budapest, 31 /1984/, 7-15.

[62] Radó P.; Tipusszerkezetek leiró nyelvekben. Alkalmazott Matematikai Lapok, /megjelenés alatt/.

## A TANULMÁNYSOROZATBAN 1984-BEN MEGJELENTEK:

155/1984 Deák, Hoffer, Mayer, Német, Potecz, Prékopa, Straziczky: Termikus erőmüveken alapuló villamos-energiarendszerek rövidtávu, optimális, erőmüvi menetrendjének meghatározása hálózati feltételek figyelembevételével.

156/1984 Radó Péter: Relációs adatbáziskezelő rendszerek összehasonlitó vizsgálata

157/1984 Ho Ngoc Luat: A geometriai programozás fejlődései és megoldási módszerei

158/1984 PROCEEDINGS of the 3rd International Meeting of Young Computer Scientists
Edited by J. Demetrovics and J. Kelemen

159/1984 Pertók Péter: A system for monitoring the machining operation in automatic manufacturing systems

160/1984 Ratkó István: Válogatott számitástechnikai és matematikai módszerek orvosi alkalmazása

161/1984 Hannák László: Többértékü logikák szerkezetéről

162/1984 Kocsis J., Fetyiszov V.: Rugalmas automatizált rendszerek: megbizhatóság és irányitási problémák

163/1984 Kalavszky Dezső: Meleghengermüvi villamos hurok-emelő hajtás vizsgálata

164/1984 Knuth Előd: Specifikációs adatbázis modellek

165/1984 Petróczy Judit: Publikációk 1983