tanulmányok 124/1981

MTA Számitástechnikai és Automatizálási Kutató Intézet Budapest

MAGYAR TUDOMÁNYOS AKADÉMIA
SZÁMITÁSTECHNIKAI ÉS AUTOMATIZÁLÁSI KUTATÓ INTÉZETE

LOGIKAI ALAPÚ PROGRAMOZÁSI MÓDSZEREK ÉS ALKALMAZÁSAIK SZÁMÍTÓGÉPPEL SEGÍTETT ÉPÍTÉSZETI TERVEZÉSI FELADATOK MEGOLDÁSÁHOZ

Irta:

MÁRKUSZ ZSUZSANNA

Tanulmányok 124/1981

A kiadásért felelős

DR VÁMOS TIBOR

ISBN 963 311 121 8

ISSN 0324 2951

TARTALOMJEGYZÉK

OLDAL

1. BEVEZETÉS

A tanulmány tárgya a számitógéppel segitett épitészeti tervezés egy alapjaiban uj, a matematikai logika eszközeit használó módszere, és annak alkalmazása. A PROLOG uj elvü, logikai alapu programozási nyelv segitségével kidolgoztuk többszintes lakóépületek tervezésének modellezését, számitógéppel megoldottunk eddig nem formalizált épitészeti bonyolultságának vizsgálatával pedig olyan programozás-módszertani elveket fejlesztettünk ki, amelyek hatékonyan alkalmazhatók a gyakorlati programozási munkában.

A bevezetés további részében a tárgykört érintő alapfogalmakról irunk, ismertetjük az eddig elért legfontosabb eredményeket. A második fejezetben megmutatjuk, hogy mit és miért lehet PROLOG-ban programozni a számitógéppel segitett épitészeti tervezés területén, a harmadikban pedig azt, hogy hogyan érdemes. A negyedik fejezetben ismertetünk egy, a gyakorlati igényeket kielégitő komplex programrendszert.

* * *

A 60-as években a hardware eszközök rohamos fejlődésével párhuzamosan egyre összetettebb, magasabbrendű tudásreprezentációt és feladat-megoldó képességet kívántak a számitógépre ráruházni, ami a rendelkezésre álló software eszközök segítségével igen nehézkesnek, néhol lehetetlennek bizonyult. Ezért Nyugat-Európa és az Egyesült Államok több számitástudományi központjában kezdtek el dolgozni olyan emberközeli, "nagyon magas szintű" (új elvű) programozási nyelvek definiálásán és megvalósitásán, amelyek alapjaiban különböznek a hagyományos algoritmikus nyelvektől. A PROLOG ilyen új elvű, a mesterséges intelligencia kutatások keretein belül kifejlesztett programozási nyelv.

Az új elvű programozási nyelvekkel szemben támasztott követelményeket a következő öt pontban foglaljuk össze [47]:

1. Az új elvű programozási nyelven csak a lényeget kelljen megfogalmazni, vagyis a nyelv adjon lehetőséget arra, hogy a programozó csak a számára lényeges kérdéseket döntse el, és a lényegteleneket hagyja nyitva - a "mit" kényelmes megfogalmazása;

2. A nyelv biztositson lehetőséget a programozónak arra, hogy a megoldás algoritmusára vonatkozó módszertani, azaz vezérlési információkat adhasson - lehetőség a "hogyan" megadására;

3. A fenti két funkció ne keveredjen, azaz világosan elkülönithető legyen a feladat definiálása, a "mit" és a megoldás algoritmusa a "hogyan".(A jelenleg elterjedt algoritmikus nyelveknél pl. ALGOL, FORTRAN, PL/1 ez a két funkció teljesen összeolvad. A programozó a megoldás algoritmusán keresztül definiálja a megoldandó problémát.);

4. A nyelvnek legyen explicit, precizen definiált, matematikai eszközökkel jól kezelhető szemantikája;

5. A nyelv legyen "hivatkozási átlátszó", azaz a nyelv szintaxisa megfelelően tükrözze a szemantikát, azaz a "jelentés" minden árnyalatát.

Több ilyen új elvű, vagy "nagyon magas szintű" nyelv született, amelyek közül a legjelentősebbek az USA-ban elterjedt LISP alapu nyelvek, a lambda kalkulusra épülő QLISP [42], PLANNER [3], QA4 [9], POPLER [8], és az Európábam kidolgozott logikai alapu nyelvek, mint a ABSYS [lo], ABSET [11], és a PROLOG.

A LISP alapu nyelvek kissé nehézkes szintaxis-sal rendelkeznek, nincs mindegyiknek kellő pontossággal definiált szemantikája, s alkalmazási területeik is jobbára a mesterséges intelligencia kutatások keretei között maradtak.

A legsikeresebb, és a legigéretesebb uj elvü nyelvek a PLANNER és a PROLOG.

A PROLOG a matematikai logika legjobban kidolgozott nyelvére, az elsőrendű predikatum kalkulusra épül. Egy PROLOG program speciális elsőrendű formulák, a u.n. Horn-klózok halmaza, igy a nyelv szintaxisa a predikatum kalkulus szintaxisának egy leszűkitése, és szemantikája precízen definiált. A PROLOG egy olyan programozási nyelv, amely szinte teljesen eleget tesz az új elvű nyelvekkel szemben támasztott a fent emlitett öt pontban összefoglalt követelményeknek. Alkalmazási területe egyre szélesedő, kérdés-válasz rendszerek, programgenerálás, természetes nyelv megértés feladatain kivül egyre több müszaki alkalmazásával is találkozhatunk. [7, 29, 3o, 31, 43]. Azok a feladatok, amelyekre nem komplikált matematikai formulák kiszámitása, hanem a logikai döntések sorozata jellemző (pl. meghatározott funkcionális és strukturális igényeket kielégitő lakásalaprajz tervezése), tömören és pontosan megfogalmazhatók a logika nyelvén. Éppigy leirható a feladat világa is, amely azon adottságok és követelmények rendszere, amelyeket figyelembe kell venni a feladat megoldása során. A feladat világát axiomákkal, ill. definiciókkal lehet megadni, míg maga a feladat bizonyitandó tételnek fogható fel. Az igy kialakitott rendszer elsőrendű predikatum kalkulusban megfogalmazva alkotja magát a számitógépes programot.

Tehát egy feladat kijelölése és a feladat világának kellő pontosságu logikai definiálása elegendő a számitógépes program irásához. Ez azonban még csak a probléma megoldás "mit" része. A "hogyan", tehát a feladatmegoldás

algoritmusának megadásához is segitséget nyújt a logika eszköztára. A logikai állitásokon működő következtetési szabályokat a számitógép számára érthetővé tették, a nyelv interpreterébe vagy fordítójába beépitették. A feladat megoldását az igy kialakitott automatikus konstruktiv tételbizonyitási eljárás állitja elő. S mivel az elsőrendü predikatum kalkulushoz van teljes következtetési szabályrendszer (a PROLOG esetében a lineáris stratégiával ellátott u.n. rezolluciót használják) a tételbizonyitási eljárás matematikai logikai háttere is megalapozott.

A PROLOG elméleti alapjait R. Kowalski, M. Emden és L. Colmerauer dolgozták ki [21, 22, 5]. Az ötletet először 1974-ben, Marseille-ben implementálták [2], később Edinburgh-ban [45, 46], majd 1975-ben Budapesten, a NIM Ipargazdasági és Üzemszervezési Intézet ICL 19o5/A tipusu számitógépén is megvalósitották. Azóta újabb és gyorsabb változatok müködnek több magyar számitóközpont SIEMENS 7.7.55 ICL System 4-7o, IBM 3o31 és R-2o tipusu gépein [44, 23, 43].

A PROLOG programozási nyelv implementálásával lehetőség nyílt olyan összetett feladatok gyors, számitógépes megoldására, amelyek nagyon nehézkeseknek bizonyultak a hagyományos algoritmikus programozási nyelveken. Ide tartozik az épitészeti tervezés modellezésével kapcsolatos néhány CAD probléma is.

Az épitészeti tervezés modellezése alatt azt a tevékenységet értjük, amellyel formalizáljuk azokat a meggondolásokat, amelyeket egy tervező mérnök végez, miközben egy meghatározott funkciót ellátó épület terét kitöltő egységek méreteit, funkcionális kapcsolatait, közlekedési lehetőségeit, különböző szerkezeti szempontokat kielégitő elrendezési sémáját kialakitja. Az épitészeti tervezés olyan bonyolult, sok komponensből összeálló kreativ szellemi munka, hogy teljes formalizálását, illetve automatizálását nem tudjuk, de nem is akarjuk megvalósitani. Célunk lehet azonban számitógépes rendszerek létrehozása, amelyek a tervező mérnök munkáját segitik, hatékonyabbá teszik. Az ilyen számitógéppel segitett tervezés (az angol elnevezésből "Computer Aided Design" röviditve: CAD) alapvető problémája a munkamegosztás az ember és a gép között. A használt számitógép kapacitásától, a meglévő hardware és software eszközök színvonalától függ, hogy mely tervezési szakaszokat biznak a számitógépre és melyekhez szükséges a tervező mérnök döntése.

A nemzetközi szakirodalomban publikált épitészeti CAD rendszerek közös vonása, hogy a tervezői munka teljes folyamatából kiemelnek egy vagy több szakaszt, s ezek számitógépes automatizálásával foglalkoznak. A számitógéppel segitett épitészeti CAD rendszerekkel foglalkozó szakirodalom a következő témák szerint csoportositható:

1. Automatikus terület felosztási probléma [39, 15, 28];

2. A képi reprezentáció által felvetett önálló grafikai problémák, mint a három-dimenziós testek takartvonalas ábrázolása, perspektivikus ábrázolás kérdései stb. [27, 37, 34];

3. Mérnöki rajzok, vázlatok számitógépes feldolgozása, digitalizálása [25, 48, 41];

4. Az épitészeti tervezés dokumentációjához szükséges müszaki rajzok automatikus elkészitése, és a gazdasági számitások elvégzése [16, 24, 4];

5. Az épitészeti tervezés általános aspektusainak elemzése, a kialakitandó integrált CAD rendszer elvi felépitése [1, 12];

6. A matematikai logikán alapuló uj software eszközök (pl. QLISP, PROLOG) és formális rendszerek (pl. FUZZY halmazok) alkalmazása különböző CAD feladatok megoldásához [13, 26, 43, 29, 3o, 31, 32] .

C.M. Eastman a "The Representation of Design Problems" [12] c. cikkében az ember-gép kapcsolat két alapvető módszerét különbözteti meg. Az egyik fajta kapcsolat a mérnöki tudást a rajzokon, vázlatokon keresztük kivánja a számitógép számára átadni és érthetővé tenni. Ez természetesen sok speciális alakfelismerési problémát vetett fel, amelyek megoldása legalább olyan nehéz, mint a

természetes nyelvek megértése számitógéppel. A másik irányzat a tervezési információkat a gép által érthető kódokban tárolja, a mérnök számára jól hozzáférhető, könnyen módositható alakban. Ezt a fajta kapcsolatot könnyebben kivitelezhetőnek, fejleszthetőbbnek tartja, és rámutat a két különböző irányzat kombinálásának előnyeire. A CAD rendszerek fejlődését a magasabbrendű információ és tudásreprezentációtól, és hatékonyabb, speciálisabb probléma-megoldó rendszerektől várja.

Hasonló kívánságokat fejez ki M. Henrion [15], aki az automatikus terület-felosztási programok gyakorlati alkalmazásának hiányáról ir, miután elemzi a létező legfontosann módszereket. Szerinte a kidolgozott rendszerek nem adekvátak a feladathoz, ezért nem találtak széleskörü alkalmazásra a tervező mérnökök között. Egy módszertanilag megfelelő rendszernek nemcsak a probléma megoldását, hanem a probléma definiálását és a kettő közötti kapcsolatot is tartalmaznia kellene jól érthető formában. Javasolja a tudásreprezentáció IF< feltétel> THEN <cselekvés> formáját, melyet O. Akin is megfelelőnek talál a tervező mérnök gondolkodásmódjához. [1]. Ez utóbbi kívánságot a PROLOG teljes mértékig kielégiti. Nem véletlen, hogy L.M. Pereira az általa kidolgozott automatikus területfelosztási probléma ujszerű, gráfokon alapuló elméletének megvalósitásához a PROLOG programozási nyelvet javasolja [39].

A tételbizonyitáson alapuló probléma-megoldás egy érdekes példáját láthatjuk G. Lafue " A Theorem Prover for Recognizing 2-D Representations of 3-D Objects" c. cikkében [25]. Az ORTHO program sokszög lapokkal határolt három

dimenzós testeket azonosit két dimenziós nézetek alapján. Az input kétértelmüségeit egy tétel-bizonyitó segitségével szürik ki a "generáld és ellenőrizd" elv alapján. A program generálja a test lehetséges nézeteit, majd egy automatikus tételbizonyitó segotségével az inkonzisztens nézeteket kizárja.

A fuzzy halmazok egy ujszerü CAD alkalmazásával találkozhatunk J.S. Gero és M.Vofneuk "Building Fuzzy CAD Systems" c. cikkében [13]. Az épület tervezéséhez figyelembe vett követelmény-rendszerbe úgy épitik be a szubjektivitást, hogy a feltételeket fuzzy tulajdonságoknak tekintve súlyozzák. Igy formalizálják a "pontatlanságot", az egyéni ízlést, s ezzel lehetőséget adnak arra, hogy egy és ugyanazt a CAD rendszert különböző típusú feladatokra a különböző felhasználók más és más szubjektiv döntésnek megfelelően használják.

Láthatjuk tehát, hogy a logikai eszközök CAD alkalmazását az összetettebb, magasabb szintű tudásreprezentációt és probléma-megoldó képességet igénylő feladatok váltották ki. A PROLOG csak egy eszköz a sok között, bár jelenleg a leghasználhatóbbnak tűnik.

Az első CAD PROLOG program Szőts Miklós nevéhez füződik, aki egyszintes csrnokok méretezését oldotta meg 1975-ben [43]. Az általa kidolgozott program előregyártott elemekből készülő, egyszintes, daruzatban ipari csarnokot tervez.

Kiinduló adatként a csarnok geometriai méreteit, és a födémet terhelő egyenletesen megoszló teher intenzitását kell megadni. A program megtervezi az alaprajzi rasztert (födémpanel kiosztást), és kiválogatja a geometriai és statikai feltételeknek megfelelő elemeket.

Egy másik érdekes PROLOG alkalmazás Holnapy Dezső munkája, aki a szoliter-alaptervezés egy problémáját fogalmazta meg PROLOG programmal.[17]. A feladat: épületpillérek alá megfelelő alaptestek beválogatása az eleve adott rendszer-komponens készletből. Bemenő adatok az erőrendszer és az alapok közötti távolságok listája; és az erők helyén alkalmazandó alaptestek azonositói képezik a végeredményt.

A jelen dolgozat szerzője által kidolgozott PROLOG alapú építészeti CAD feladatokkal a tanulmány részletesen foglalkozik [29, 3o, 31, 32] . Külön fejezetet szentelünk azoknak az elvi-módszertani meggondolásoknak, amelyek a CAD programok egyszerű szerkezeti felépitését, könnyű módosithatóságát célozzák [18, 19]. A programlistákat, a komplexitás-számitás részletes adatait tartalmazó táblázatokat, a futási eredményeket, és az őket illusztráló ábrákat külön kötetben, a FÜGGELÉK-ben közöljük.

Itt szeretnék köszönetet mondani dr.Ágnes Kaposi-nak, aki a háromhónapos londoni kutatómunkámat lehetővé tett és irányitotta, s aki nagyban segitségemre volt

az ujszerü komplexitáselmélet kialakitásában. Köszönettel tartozom dr. David Warren-nek, aki lehetőséget adott arra, hogy az Edinburgh-i Egyetem DEC-10 tipusu számitógépén a programjaimat tesztelhessem, s aki véleményével, tanácsaival segitette munkámat, dr. Dömölki Bálint-nak, aki lehetőséget nyujtott arra, hogy a Számitástechnikai Koordinációs Intézet számitógépén dolgozhassak. Külön szeretnem megköszönni Köves Péter-nek az IBM 3031-es számitógép PROLOG interpretének installálását, s azt a segitséget, amelyet a kezdeti nehézségek áthidalására nyujtott. Köszönet illeti Rákossy István-t a többszintes lakóépületet tervező programrendszer épitészeti koncepciójának kidolgozásáért. Végül hálával tartozom Szőts Miklósnak, aki a tanulmány anyagához füzött értékes megjegyzéseivel segitette munkámat.

2.A LOGIKAI PROGRAMOZÁS ÉS AZ ÉPITÉSZETI CAD

2.1. A PROLOG RÖVID ISMERTETÉSE

Egy PROLOG program elsőrendü formulák egy részosztályának, a Horn-klózoknak véges sorozata. A Horn-klóz olyan implikáció, amelynek alakja megfelel az alábbi lehetőségek valamelyikének:

1. $A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n.$
2. $A \leftarrow .$
3. $\leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n.$

ahol $A, B_1, B_2, \ldots, B_n$ atomi formulák,
$\leftarrow, \wedge$ pedig rendre az implikáció, a logika "és" jele.

Atomi formula: egy $P(t_1, \ldots, t_K)$ alaku formula, ahol P egy k argumentumu relációjel, és t_i-k $O< i \leq K$ termek.

Term: egy változó, vagy egy $f(t_1, \ldots t_K)$ alaku kifejezés, ahol f egy k argumentumu függvényjel és t_i-k $O \leq i \leq K$ termek. A O argumentumu függvényjel konstans.

A relációjelek, függvényjelek, és változójelek halmazai páronként diszjunktak.
A változókat minden klózban univerzálisan lekötöttnek tételezzük fel, a függvény és relációjelek, valamint

az $\wedge$, $\leftarrow$ jelek jelentése a szokásos.Egy PROLOG programmal valamilyen vizsgálandó univerzum leirását adhatjuk meg, majd a leirt világra vonatkozóan kérdéseket tehetünk gel, illetve megoldandó célokat tüzhetünk ki. A kérdések megválaszolása illetve a célok megoldása jelenti a PROLOG program futását.

Az 1. és 2. tipusu klózokkal irhatjuk le a feladat világát, melyeket rendre reduktoroknak illetve tényállitásoknak hivunk. A célok megadására a 3. tipusu klózokat használhatjuk, melyeket célállitásoknak nevezünk. A PROLOG interpretere tekinthető SL rezolución alapuló automatikus tételbizonyitónak. Mükődésének pontos ismertetését mellőzzük, részletes leirása megtalálható [44]-ben.Kiemeljük azonban két fontos tulajdonságát, a mintaillesztést (pattern matching) és a visszlépést (backtracking). A mintaillesztés lehetővé teszi, hogy a feladatot először általános problémamegoldási sémákban fogalmazzuk meg, amelyek csak a program futása során fognak konkrét adatokkal aktivizálódni. A visszalépés pedig arra ad lehetőséget, hogy ha nem a megfelelő sémát aktivizáltuk, sorra vegyük a még lehetséges sémákat, amig meg nem oldjuk a feladatunkat, vagy amig kiderül, hogy a program által definiált univerzumban a feladatunkat nem tudjuk megoldani.

A program futása ugy kezdődik, hogy a feladat megfogalmazására szolgáló célállitás első atomjához az

interpreter keres egy olyan reduktort, vagy tényállitást, amellyel az mintaillesztéssel egyesithető. A mintaillesztés egy célállitás - a jelenlegi szintaktikus konvencióknak megfelelő elnevezéssel - egy negativ atomja és egy reduktor "feje"(a implikáció konkluziója) vagy egy tényállitás, mindkét esetben pozitiv atomja közötti illeszthetőséget ellenőrzi, és ha lehetséges, a két klóz egyesitését végrehajtja. Egy negativ és egy pozitiv atom illeszthető, ha

- relációjelük azonos, tehát argumentumaik száma is megegyezik;
- az egymásnak megfelelő argumentum párok rendre megfelelnek az alábbi három eset egyikének;
  1. mindkét argumentum változó;
  2. az egyik argumentum változó, a másik összetett term (azaz konstans, vagy függvények, változók és konstansokból generált kifejezés)
  3. mindkét argumentum összette term, és illeszthetők.

A hagyományos programozási nyelveknél megszokott "értékadás" művelete itt mintaillesztéssel történik, ha a fent emlitett 2. esetben az egyesités után a változó értéke az összetett term lesz. Vegyük észre, hogy teljesen mindegy, hogy a negativ, vagy a pozitiv atom tartalmazza-e az összetett termet, az illesztéssel az értékadás mindkét irányban megtörténhet.

Egy célállitás és egy reduktor egyesítése ugy történik, hogy a reduktor negativ atomjai átveszik azokat az uj értékeket, amelyeket a mintaillesztés során az argumentumok kaptak,(ha szerepeltek a negativ atomokban olyan változók, amelyek a pozitiv atomban is szerepeltek). A továbbiakban ezek a negativ atomok önálló célállitásokká válnak, és újabb mintaillesztési eljárásokkal kielégithetőek. A feladat megoldása akkor fejeződik be, ha az összes eleve kitüzött és generálódott célállitás a mintaillesztéssel kielégitést nyert.(a tételt sikerült bebizonyitani). A program futásának másik megállási módja az, hogy a mintaillesztés sikertelensége miatt a feladatot - az adott feltételek mellett - nem lehet megoldani(a tételt nem tudjuk bizonyitani). Mivel az elősrendű logika nem eldönthető, nem tudjuk biztositani, hogy a bizonyitási eljárás minden esetben végetérjen. Viszont a teljesség miatt igaz tétel esetén az eljárás sikeresen befejeződik.

2.2. A PROLOG SAJÁTOSSÁGAINAK FELHASZNÁLÁSA AZ ÉPITÉSZETI TERVEZÉS MODELLEZÉSÉBEN

Az épitészeti tervezés nem determisztikus folyamat, a kivánt feltételek nem határozzák meg magát az épületet. Az épitészeti tervezésre a próbálkozások, variánsok előállitása a jellemző. A különböző változatok kiértékelése és a végső döntés újabb - esetleg szubjektiv - követelmények figyelembevételével történik. Egy automatikus rendszer, amely a fenti folyamatot kívánja modellálni csak ugy müködhet, hogy bizonyos alapvető szabályok

szeti CAD problémák, amelyek elvileg számitógépre orientáltak (pl. lakásalaprajz variációk előállitása) nagyon nehezen programozhatónak bizonyultak a hagyományos eszközökkel. A PROLOG nyelv használata lehetővé tette, hogy bővüljön azoknak a feladatoknak a köre, amelyeket a számitógépre lehet bizni, s ezáltal az ember és a gép közötti munkamegosztás a magasabb fokú automatizálás irányába tolódjon el.

Ebben a fezetben egy konkrét CAD program bemutatásán keresztül sorba vesszük és elemezzük azokat a PROLOG-ban rejlő új lehetőségeket, módszereket, programozás - technikai megoldásokat, amelyek megkönnyitették, vagy lehetővé tették, eddig nem formalizált tervezési folyamatok számitógépes automatizációját.

2.2.1. Paneles lakásalaprajz variációkat tervező program

A programrendszer adatbázisa, panelek, cellák

A program célja, hogy a budapesti házgyárak által gyártott panelekből adott méretű, szobaszámú és félszobaszámú lakások alaprajzi variánsait előállítsa. Jelen program a 2.sz. Házgyár paneleinek méreteivel működik, de az eljárás más házgyárak paneleire is alkalmazható, amennyiben az adatbázisban rögzitett panelelemek méreteit megváltoztatjuk.
Mivel a lakás összes helyiségének sarokpontjai derékszögű négyszöghálóra, un. modulhálóra illeszkednek, a programban a méreteket nem méterben, hanem modulban adjuk meg. A modul a teherhordó szerkezetek méreteinek

legnagyobb közös osztója. Ebben a rendszerben
1 modul = o,3 m.
A házgyár által gyártott födémelemek szériája adott (n db különböző méretü téglalap). A szükséges számú és méretű téglalapok egymás mellé illesztéséből adódó sikidom-változatokat nevezhetjük alaprajzi konturvariánsoknak.
A program 3 födémpanellal dolgozik:
F 1 - 2,7 m x 4,2 m
F 2 - 2,7 m x 5,4 m
F 3 - 2,7 m x 2,7 m
A födémpanelok határozzák meg a teherhordó egységek (cellák) méreteit. A program adatbázisába a cellákat vettük be. 4 teherhordó cella adott a következő formában:

```
CELLA (NAGY,        14, 18, 252).
CELLA (KIS,          9, 18, 162).
CELLA (FELNAGY,      9, 14, 126).
CELLA (FELKIS,       9,  9,  81).
```

A zárójel utáni első paraméter a cella azonositója, a második szélessége, a harmadik a hossza, a negyedik a területe modulban.
A NAGY cella a nappali szoba, a KIS cella egy kisebb méretü, pl. gyermek- vagy hálószoba, a FELNAGY és FELKIS cellák félszobák kialakitására alkalmasak. Egy NAGY cellából alakitható ki válaszfalak és a vizesblokk beállitásával az un. OSZTOTT cella, amely a lakás összes egyéb helyiségét - az előszobát, konyhát, fürdőszobát, WC-t, gardrob-folyosót - tartalmazza.

A különböző OSZTOTT cellatipusok szintén a program adatbázisához tartoznak. Programrendszerünk kétféle OSZTOTT cellát kezel: egy étkezőkonyhás és egy konyha-étkezőfülkés változatot. A konyha mindkét esetben ablakos, tehát belső szellőztetésű konyhát nem engedünk meg. Az étkezőfülkés megoldás helykihasználás szempontjából előnyösebb, ezért ahol lehet, ennek a változatnak van prioritása. A program adatbázisában egy OSZTOTT cellát, pl. a következő módon lehet reprezentálni:

OSZTCEL (ETKFULKES, ABLAK, 13, NYILT, 8, 14).

A 6 argumentum értelme rendre a következő:

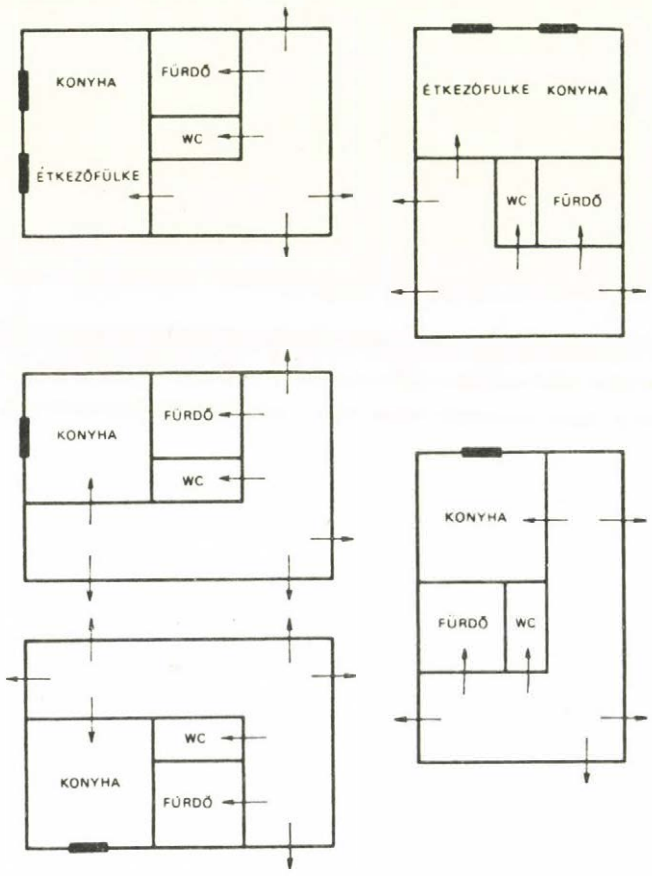
1 - A konyha típusa.

2,3,4,5 - Az északi, keleti, déli és nyugati fal jellemzése.
A példában a konkrét paraméterek jelentése a következő:
ABLAK: északon van a konyhaablak.
13: keleten 13 modulnyi zárt falrész van, a konyha és a vizesblokk keleti fala.
A többi lehet zárt és nyílt, azaz ajtóval ellátott falrész.
NYILT: A déli fal nyílt, lehet rajta ajtó.
8: A nyugati falon 8 modulnyi falrész van, a konyha nyugati fala.

6 - Az OSZTOTT cella szélessége a példánkban 14 modul.

A rendszer adatbázisában 5 különböző OSZTOTT cellavariáns szerepel, ugyanis kétféle konyhatípus van, de az

ablak elhelyezésének változtatása és az OSZTOTT cella 90^{o}-os elforgatása újabb variánsokat eredményez. (1.ábra).



1. ábra

A program a lakásalaprajz-konturnak megfelelő OSZTOTT cellát ugy tervezi meg, hogy az adatbázisban rögzitett OSZTOTT cellák szimmetriaváltozataiból kiválasztja

azokat, amelyek a következő feltételerendszernek megfelelnek:

1. A konyha ablakos legyen.
2. Legalább 2 szoba külön bejáratú legyen (azaz az előszobából vagy a folyosóról nyíljon).
3. Biztositva legyen a lakásba való bejárási lehetőség.

A feladat megadása

A feladatot a következő formában jelöljük ki:
- LAKÁS (73,2,1) - FAIL.
Ez azt jelenti, hogy szeretnénk megtervezni egy maximum 73 m^2-es, 2 szobás + 1 félszobás házgyári lakás összes lehetséges változatát. Ha csak egy változatra vagyunk kiváncsiak, azaz azt szeretnénk eldönteni, hogy lehet-e ilyen lakást tervezni, a - FAIL beépített predikátum alhagyható.
A szoba- és félszobaszám összege legfeljebb 5 lehet.

A programrendszer részei

Jelenleg a rendszer 2 szegmensből áll: az előkészitő programból és a főszegmensből.

Az előkészitő program

A program inputja a tervezendő lakás maximális alapterülete m^2-ben, valamint a szobák és félszobák száma.
Az output:

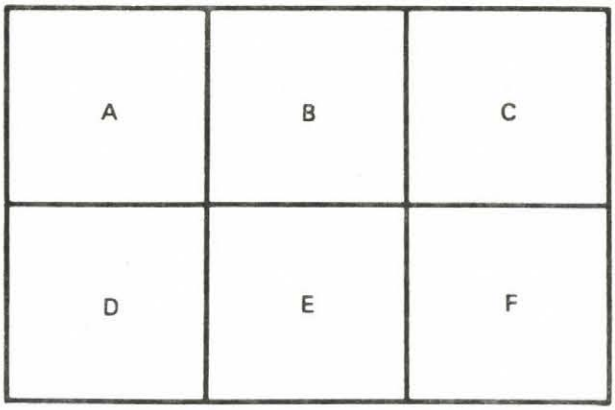
a) A program által tervezett lakás alapterülete m^2-ben (ez nem nagyobb az inputként megadott alapterületnél).
b) A lakáshoz tartozó födémszükséglet, azaz hogy a különböző födémpanelokból külön-külön mennyire van szükség.
c) Egy cellalista vagy cellalista-sorozat. A cellalista elemei az adatbázisban rögzitett teherhordó cellák azonositói. Egy ilyen a tervezendő lakáshoz szükséges cellákat tartalmazza. Például: NAGY, KIS, FELNAGY.NIL.

A program egy feladatra több lehetséges változatot készit el. Futása után a tervezőmérnöknek kell eldöntenie, hogy melyik változat a számára legmegfelelőbb. A kiválasztott változat valamelyik cellalistája fog input adatként szolgálni a következő program, a főszegmens müködéséhez.

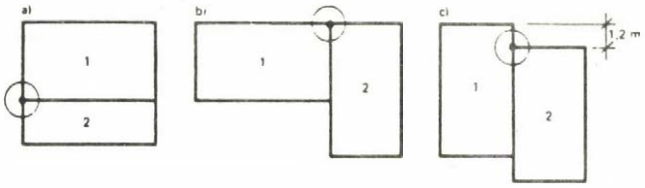
A főszegmens

Inputja a cellalista.
Outputja:

a) A teherhordó cellák geometriai elhelyezkedését megadó sarokpontok koordinátái modulban. Például: NAGY 0; 0. 0; 14. 0; 18. 18; 14.
b) A cellákhoz tartozó szobák ablakainak tájolása
c) Az egyéb helyiségeket tartalmazó OSZTOTT cella falainak koordinátái
d) A konyha típusa
e) A konyhaablak tájolása



2. ábra



3. ábra

f) A konyha sarokpontjainak koordinátái
g) A vizesblokk sarokpontjainak koordinátái
h) A bejárati ajtó helye
i) A külön bejáratu szobák száma.

A főszegmens futása után megrajzolható a program által elkészitett alaprajz. Ha rajzoló periféria is rendelkezésünkre állna, közvetlenül rajzos output készülne. (5, 6. ábra).

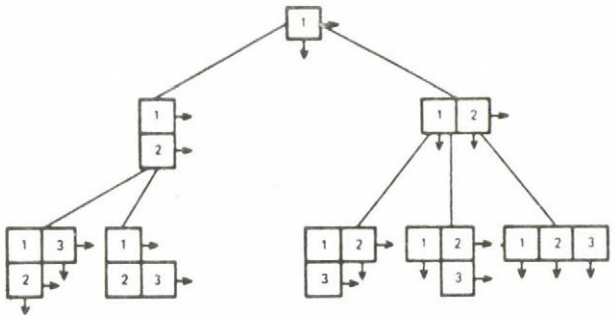
A főszegmens - funkciójat tekintve - 3 lényeges részre tagolódik:

1. A teherhordó cellák geometriai egymás mellé illesztése, az OSZTOTT cella kijelölése, az ablakok helyének rögzitése.
2. Az OSZTOTT cella és a szobák falainak metszete, az un. KÖZÖS LISTA előállitása, amely az átjárásai és bejárási lehetőségek megadásához szükséges.
3. Az OSZTOTT cella felosztása konyhára, vizesblokkra, előszobára, gardrobe-folyosóra.

A különböző cellák (vagy téglalapok) egymáshoz illesztése alkotja a tervezési feladat gerincét. Ezért ezt a részt szeretnénk részletesebben kifejteni. Egy cella alaphelyzetének az adatbázisban rögzitett helyzetét nevezzük. Emellett felhasználjuk a 90^{o}-kal elforgatott helyzetet is. Egy lakás maximum 6 teherhordó cella összekapcsolásából állhat, a 2 ábrán bemutatott elrendezésben.

A cellák összekapcsolására 3 illesztési szabályt vezettünk be (3. ábra) :

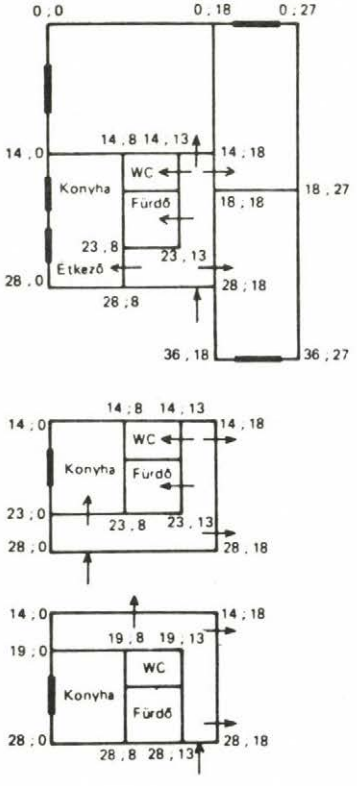
a) Két azonos szélességü cella kerülhet egymás alá, mégpedig ugy, hogy az 1. cella bal alsó sarka illeszkedjék a 2. bal felső sarkához.
b) Két bármilyen méretü cella összekapcsolható ugy, hogy az 1. cella jobb felső sarka illeszkedjék a 2. bal felső sarkához, és a 2. alaphelyzetben van.
c) Két alaphelyzetben levő cella illeszkedhet ugy, hogy a 2. cella bal felső sarka 1,2 m-rel(azaz 4 modullal) lejjebbb van, mint az 1. jobb felső sarka.



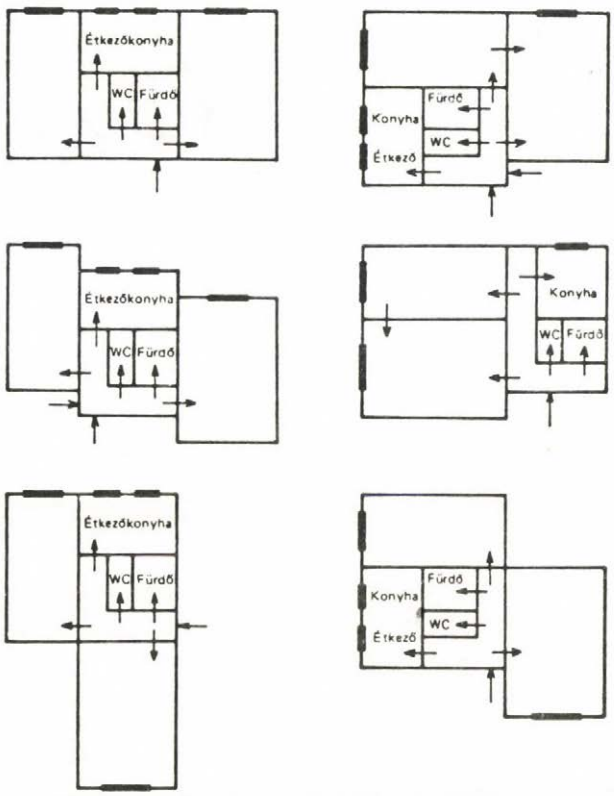
4. ábra

Ha nincs külön letiltva, az elemeket alaphelyzetben és 9o°-kal elforgatva is illeszthetjük egymáshoz. Az illesztés a következő algoritmus szerint megy végbe. A CELLALISTA első elemét kijelöljük kezdőcellának. Ekkor a cellához két irányban: jobbra vagy alá lehet másik cellát kapcsolni. A jobbra kapcsolás a c) vagy b) illesztési móddal, az alákapcsolás az a) illesztési móddal történhet. Ha a 2. elemet már összekapcsoltuk az 1-vel, akkor a 3. elemet kapcsolhatjuk az 1. vagy 2. elemhez a 3. illesztési mód valamelyikével, és igy tovább(lásd a 4. ábrát). Az algoritmusból látható, hogy a CELLALISTA sorrendje befolyásolja a kialakuló sikidom formáját, De mivel a kötött sorrenddel is már elég nagy a variánsok száma (a feladattól függően kb. 2o-5o db), ezért ugy döntöttünk, hogy a főszegmensen belül nem változtatunk a CELLALISTA sorrendjén, ellenben az előkészitő szegmensben előállitjuk a CELLALISTA összes egymástól különböző permutánsát. Igy a főszegmens többszöri futtatásával megkaphatjuk az összes lehetséges változatot.

```
PANELES LAKOHAZ EGY LAKASANAK ALAPRAJZA

CELLAFOSZTAS
NAGY   0 : 0 . 0 : 18 . 14 : 0 . 14 : 18   ABLAKA NYUGATON
KIS    0 : 18 . 0 : 27 . 18 : 18 . 18 : 27   ABLAKA ESZAKON
OSZTOTT   14 : 0 . 14 : 18 . 28 : 0 . 28 : 18   ABLAKA NYUGATON
KIS    18 : 18 . 18 : 27 . 36 : 18 . 36 : 27   ABLAKA DELEN


AZ OSZTOTT CELLA CSOPONTJAINAK KOORDINATAI:
     ESZAKI FAL:  14 : 0 . 14 : 18
     KELETI FAL:  14 : 18 . 28 : 18
     DELI FAL:    28 : 0 . 28 : 18
     NYUGATI FAL: 14 : 0 . 28 : 0

AZ OSZTOTT CELLA HELYSEGEI: KONYHA, FURDOSZOBA, WC, ELOSZOBA, GARDROB FOLYOSO.
A VIZESBLOKK A WC-T ES A FURDOSZOBAT TARTALMAZZA.


A KONYHA TIPUSA:  ETKFULKES
A KONYHAABLAK NYUGATON VAN.
A KONYHA KOORDINATAI:  14 : 0 . 14 : 8 . 28 : 0 . 28 : 8
A VIZESBLOKK KOORDINATAI:  14 : 8 . 14 : 13 . 23 : 8 . 23 : 13
A LAKASRA VALO BEJARAS, AZAZ AZ ELOSZOBAAJTO DELEN VAN.
A KULON BEJARATU SZOBAK SZAMA:  3.

A KONYHA TIPUSA:  ETKKONYHAS
A KONYHAABLAK NYUGATON VAN.
A KONYHA KOORDINATAI:  14 : 0 . 14 : 8 . 23 : 0 . 23 : 8
A VIZESBLOKK KOORDINATAI:  14 : 8 . 14 : 13 . 23 : 8 . 23 : 13
A LAKASBA VALO BEJARAS, AZAZ AZ ELOSZOBAAJTO DELEN VAN.
A KULON BEJARATU SZOBAK SZAMA:  3.

A KONYHA TIPUSA:  ETKKONYHAS
A KONYHAABLAK NYUGATON VAN.
A KONYHA KOORDINATAI:  19 : 0 . 19 : 8 . 28 : 0 . 28 : 8
A VIZESBLOKK KOORDINATAI:  19 : 8 . 19 : 13 . 28 : 8 . 28 : 13
A LAKASRA VALO BEJARAS, AZAZ AZ ELOSZOBAAJTO DELEN VAN.
A KULON BEJARATU SZOBAK SZAMA:  3.
```



5. ábra. A főszegmens outputja



6. ábra

Lakásváltozatok

2.2.2. Programozási elvek, megoldások, fogások

1. A feladat felbontása

Egy eléggé komplex tervezési feladatot ritkán lehet egyetlen programmal megoldani, azaz teljesen automatizálni. Meg kell keresni azokat a pontokat, ahol emberi beavatkozás, döntés szükséges, és a feladatot ilyen módon kell szegmentálni. Az előző fejezetben említett feladat esetén két programra bomlott a rendszer, az előkészitő és a fő programra. Erre azért volt szükség, hogy elkerüljük a "kombinatórikai robbanást". Célunk a feladat világába lévő összes lehetséges megoldás (alaprajzi variáns) előállitása. Ha a feladatot egy programon belül akarnánk megoldani, a megoldások száma sokszorosa lenne a reálisan értékelhető nagyságrendeknek, ráadásul sok azonos, fölösleges változat születne. Ezért egy ponton leszűkiettük a döntési fát, kiválasztva egyetlen cellalista változatot a következő program számára input adatként.

2. A feladat megadása

A feladatot kijelölő célállitás formája tükrözi a feladat tipusát. A

- LAKÁS (7o, 2, 1).

célállitással azt a feladatot adjuk a programrendszernek, hogy tervezzen egy maximum 7o m^2-es, 2 szobás 1 félszobás lakást, avagy ellenőrizze, hogy egyáltalán kielégithető-e ez a kivánság. A

- LAKÁS (7o, 2, 1) - FAIL.

célállitással a megoldás összes lehetséges változatát kivánjuk megkapni. Ez a trükk a PROLOG-nak azt a tulajdonságát használja ki, hogy az interpreter a visszalépés segitségével "próbálkozni" tud. Az interpreter a program futása során a leírt szabályok segítségével döntési fát épit fel. Amikor a program először végigfut, azaz a döntési pontok sorozatán keresztülhaladva elér a fa leveléig, előáll egy lakásalaprajz variáns. A program le is állna, ha a feladat megadásakor nem szereplne a - FAIL predikatum, amely sohasem sikerül (azonosan hamis). A -FAIL predikátumot a következtetési mechanizmus úgy értelmezi, hogy zsákutcába jutott, vissza kell lépnie, és meg kell próbálnia egy másik változatot. Így előállitja a következő alaprajzot, majd a FAIL-lel találkozva ismét visszalép stb. Igy a program bejárja az egész döntési fát, és minden lehetséges változatot előállit.

A előző fejezetben ismertetett feladatban ugyan nem használtuk ki, de másutt jól hasznositható a PROLOG azon tulajdonsága, hogy egy reláció argumentumai input és output funkciót is betölthetnek, és igy egy azon reláció különböző feladatok megfogalmazására lehet alkalmas. Tekintsünk pl. egy olyan programot, amely egy kórház épületcsoportjában lévő helységek funkcióit irja le. Legyen a célállitásunk

- BENNEVAN (FOEPULET, IGAZGATOSAG).

Ez azt a kérdést fogalmazza meg, hogy vajon igaz-e, hogy az igazgatóság a főépületben van. A

- BENNEVAN (*X, IGAZGATOSAG) -OUTPUT (*X).

azt kérdezi, hogy melyik épületben van az igazgatóság. (a * -al kezdődő szimbólumok változókat jelölnek). A

- BENNEVAN (PAVILON1,*Y). - OUTPUT (*Y).

azt kérdezi, hogy milyen helység van az 1. Pavilonban. A

- BENNEVAN (PAVILON1,*Y) - OUTPUT (*Y) - FAIL.

célállitással az 1. Pavilon összes helységét megkaphatjuk.

Mig a

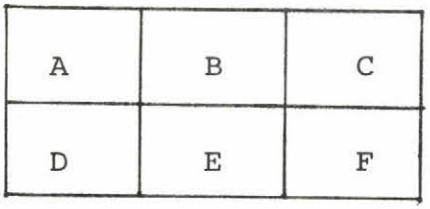
- BENNEVAN (*X, *Y).

célállitással azt tudhatjuk meg, hogy van-e egyáltalán olyan helységünk, amely az adatok között felsorolt épületek egyikében van.

3. Strukturák reprezentációja

Épitészeti tervezési feladatok modellezése során szükség van olyan program-beli strukturákra, amelyek több épitészeti egység (pl. szobák) kapcsolódási lehetőségeit irja le. A strukturák PROLOG programbeli egyfajta lehetséges reprezentációi olyan "termek" azon logikai kifejezések, ahol a függvényjelek az operátorok. Bonyolult strukturákat több operátor alkalmazásával épitünk fel, amelyek zárójelezési szabályait a prioritási szám megadásával lehet definiálni.

Tekintsük a lakások általános sémáját megadó szerkezetet:



A. B. C ! D. E. F

6. ábra

A fenti strukturában két operátort, a "!" és a "." operátorokat használjuk. Kötési prioritásukat az operátor deklarációban definiáljuk:

```
+ OPERATOR (!,  RL, 4).
+ OPERATOR (.,  RL, 5).
```

ahol a RL azt jelenti, hogy a zárójelezés "right to left", jobbról-balra történik, lásd A.(B.(C. NIL))), és a magasabb prioritásu (5) "." operator köt erősebben, mint a "!" operátor (4). Tehát az A.B.C! D.E.F összetett strukturában a "!" a "fenti" és "lenti" cellákat választja el, amelyek maguk is listák. A "fent" és "lent" csak a strukturában elfoglalt helyre és nem a lakás vertikális szerkezetére utal. Mind a hat teherhordó cella egy szinten helyezkedik el.

Nézzünk egy példát a fenti struktura program-beli alkalmazására. Definiáljuk az ALATTA relációt, amely a fenti struktura két egymás "alatti" elemét definiálja(pl: B és E):

```
+ALATTA (*X.  *L1 ! *Y. *L2,  *X,  *Y).
+ALATTA (*Z.  *L1 ! *W. *L2,  *X,  *Y) !
      -ALATTA  (*L1 !  *L2,  *X,  *Y).
```

Az ALATTA reláció három argumentumából az első a lakás strukturáját, a második egy "fenti" cellát, a harmadik az "alatta" lévő cellát reprezentálja.

A két klóz értelme a következő:

1. *X cella alatt *Y cella van, ha a ! operátorral elválasztott listákból (*X. *L1 és *Y. *L2) rendre ők az első elemek.
2. *X cella alatt *Y cella van, ha ugyan a *Z. *L1 és a *W. *L2 listákban nem ők az első elemek, de az első elemeket leválasztva az *L1 és *L2 maradék listákban *X alatt *Y van.

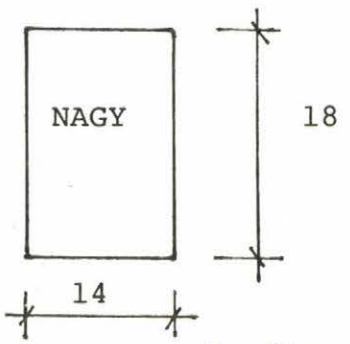
A megfogalmazásból látható, hogy az ALATTA relációt rekurzivan definiáltuk. Három elemü listák esetén ez a definició fölöslegesen általánosnak tünhet, de feltéve, hogy az alapvető szerkezeteket esetleg bővithetjük, ez a tömör és általános megfogalmazás a változtatás után is érvényes maradna.

4. Elforgatás, szimmetriai viszonyok

Az épitészeti tervezés modellezésében alapvető fontosságu a különböző sikidomok (többnyire téglalapok) geo-

metriai transzformációknak ábrázolása. Ez a feladat PROLOG-ban igen frappánsan megoldható a megfelelő relációk argumentumaival való manipulálással. Néhány példa:

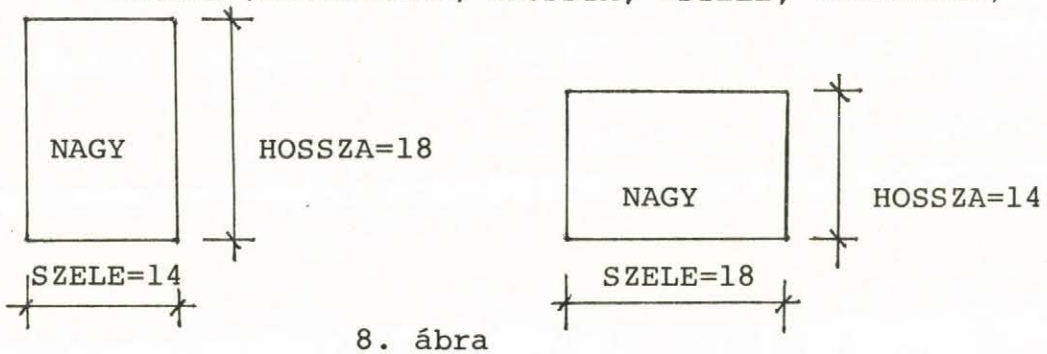
4.1. Az adatbázisban szereplő "teherhordó cellák", téglalapok egy rögzitett geometriai elrendezést tükröznek: a
+CELLA (NAGY, 14, 18, 252).
tényállás azt jelenti, hogy a NAGY azonositóval ellátott teherhordó cella "széle" 14 modul, "hossza" 18 modul, területe 252 modul.



7. ábra

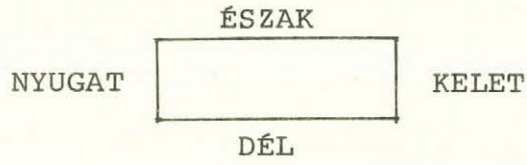
Ezt a téglalapot a tervezés során az adatbázisban rögzitett helyzetében és 90°-al elforgatva is szeretnénk használni. Ezt a szabályt a következő 2 klóz segitségével tudjuk megfogalmazni:

+CELLA1 (*AZONOSITO, *SZELE, *HOSSZA, *TERULETE)
-CELLA (*AZONOSITO, *SZELE, *HOSSZA, *TERULETE)
+CELLA1 (*AZONOSITO, *SZELE, *HOSSZA, *TERULETE)
-CELLA (*AZONOSITO, *HOSSZA, *SZELE, *TERULETE)



8. ábra

4.2. Ha a programban tögzitett téglalap egyik oldalát ÉSZAK-i-nak definiáljuk, a többi irányt a szokásos módon jelölhetjük ki. Igy beszélhetünk északra, vagy délre nyiló ablakokról, a lakás bejáratának irányáról.



9. ábra

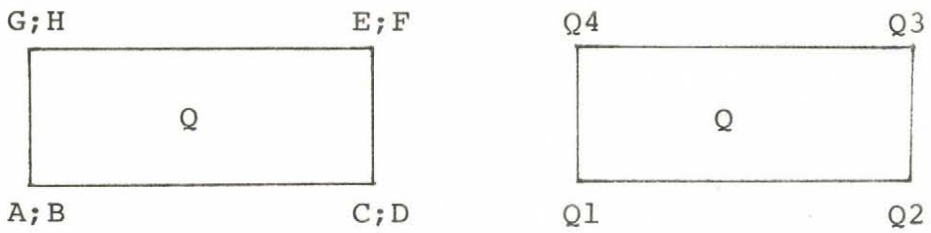
Definájuk az egymással szemben lévő irányokat, és azokat az iránypárokat, amelyek egy szoba sarkaiban futnak össze.

```
+IRANY (ESZAK).
+IRANY (KELET).
+IRANY (DEL).
+IRANY (NYUGAT).

+SZEMBEN (ESZAK, DEL).
+SZEMBEN (KELET, NYUGAT).
+SZEMBEN ( *X, *Y)
    -SZEMBEN (*Y, *X).
+SAROK  (*X, *Y)
    -NOT( SZEMBEN  (*X, *Y)).
```

Látható, hogy a SZEMBEN reláció szimmetrikus voltát külön definiáltuk. A "NOT" beépitett eljárás jelentése a következő: a NOT (R (t_1,...,t_n))

eljárás akkor sikerül, ha R(t_1,...,t_n) eljárás nem sikerül, és forditva, ha a R (t_1,...,t_n) eljárás sikerül, a NOT R (t_1,...,t_n)) nem sikerül.

4.3. Ha a téglalap sarokpontjait egy koordináta-rendszerben elhelyezett koordináta értékpárokkal adjuk meg, akkor a transzformációkat a következőképpen definiálhatjuk. Tekintsünk egy Q téglalapot:



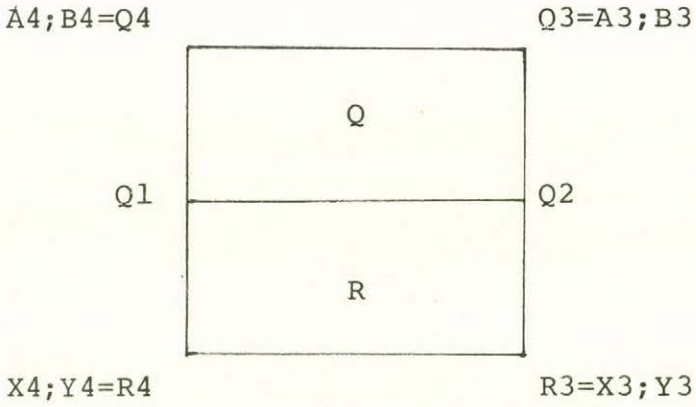
lo. ábra

A +TEGLA (A;B, C;D, E;F, G;H).
tényállás négy paramétere rendre a négy sarokpont koordináta párja. Az abszcissza és ordináta értékek a ";" operátorral vannak elválasztva.

A sarokpontokból listát is készithetünk; s igy egy téglalap helyzetét ez a lista pontosan definiálja:

Q1. Q2. Q3. Q4. NIL

Nézzük meg, hogy hogyan kapjuk meg egy Q téglalap Q1-Q2 egyenesre vonatkozó R tükörképét:



11.ábra

```
+TUKROZ (*Q1.*Q2.*Q3.*Q4.NIL, *R4.*R3.*Q2.*Q1.NIL)
     -TAVOLSAG( Q1,*Q4,*T)
     -TUK (*Q3,*T,*R3)
     -TUK (*Q4,*T,*R4).
+TAVOLSAG (*X1;*Y1, *X4;*Y4,*T)
     -MINUS (*Y4,*Y1,*T).
+TUK (*A;*B, *T, *A; *Y)
     -TIMES (*T,  2, *Z)
     -MINUS (*B, *Z, *Y).
```

Bármely egyéb transzformációt a fent definiált strukturák segitségével hasonló módon kaphatunk meg.

5. Adatbázis - kezelő eljárások használata

A legtöbb PROLOG interpreter változat rendelkezik olyan beépitett eljárásokkal, amelyek módositják az adatbázist, illetve magát a programot, hiszen PROLOG program esetén

az adat és a program egységes szerkezetű(mindkettő Horn-klózokkal reprezentált). Ilyen eljárás az ADDCLAUSE, amelynek hatására új klózzal bővül a program, a DELCLAUSE, amely kitöröl egy klózt, és a SUPCLAUSE, amely időlegesen kitöröl egy klózt, de visszalépés esetén a klóz ujra érvényes lesz.

Az ADDCLAUSE jól használható arra a célra, hogy a futás egy részeredményét ne egy új argumentumban, hanem egy újonnan felvett klózban tároljuk. Nézzünk egy példát az ADDCLAUSE használatára:

A paneles lakásalaprajz tervező program előkészitő alprogramjának outputja a teherhordó cellák azonositóiból álló cellalista. Ez a lista fog input adataként szolgálni a következő programhoz. De mivel a cellalista elemeinek sorrendje befolyásolja a lehetséges alaprajzi variánsokat, az összes lehetséges megoldást csak ugy kaphatjuk meg, ha generáljuk a cellalista összes permutációját. Ezt a funkciót a PERM nevü eljárás végzi, amelynek két argumentuma van. Az első a permutálandó lista, a második az elsőnek egy permutációja:

- PERM (*CEL1, *CEL2).

A PERM eljárás pontos definicióját és magyarázatát később közöljük, most rátérünk arra, hogy hol és miért használtunk az ADDCLAUSE adatbázis kezelő eljárást. A cellalista elemei között több azonos is lehet, s igy a PERM eljárás sok felesleges, egymástól nem különböző permutációt generálhat. Számunkra azonban csak a külön-

böző permutációk érdekesek. A RESZ nevü eljárás segitségével generáljuk a *CEL1 lista egy permutációját, ellenőrizzük, hogy az uj permutáció azonos-e valamelyik régebben generálttal, és csak akkor nyomtatjuk ki, ha különböző.

```
+RESZ (*CEL1, *CEL2)
      -ADDCLAUSE (+ (CLAUSE (*CEL1)). NIL)
      -OUTSPACES (13)  - OUTPUT (*CEL1)
      -PERM  (*CEL1, *CEL2)
      -KULONBOZ  (*CEL2).

+KULONBOZ (*CEL2)
      -CLAUSE (*CEL2) -/.
+KULONBOZ (*CEL2)
      -ADDCLAUSE (+ (CLAUSE (*CEL2)). NIL)
      -NEWLINE  -OUTSPACES (13) - OUTPUT (*CEL2).
```

A RESZ eljárás a következőképpen müködik:
ADDCLAUSE segitségével felveszünk a programba egy új klózt, amelynek neve CLAUSE, és egyetlen argumentuma van, a *CEL1-ben lévő permutálandó lista. Ezt a listát az OUTSPACES és OUTPUT beépitett eljárások segitségével ki is nyomtatjuk. Majd meghivunk a PERM eljárást, amely a *CEL1 lista egy CEL2 permutációját fogja generálni.
A következő KULONBOZ eljárásnak két esete van. Az első, hogy az újonnan generált permutáció megegyezik egy előző, az adatbázisban a CLAUSE nevű klózokban tárolt listák egyikével. Ekkor a -CLAUSE (*CEL2) célállitás illeszkedik valamilyen +CLAUSE(*X). alaku klózhoz, és

a KULONBOZ első változata sikerül. Ebben az esetben nem nyomtatjuk ki a listát, hiszen ez ismétlés volna.

A másik esetre akkor kerül sor, ha az első nem teljesül. Ekkor az újonnan generált permutáció eltér az összes előzőtől, ezért felvesszük ADDCLAUSE segitségével egy uj +CLAUSE(*CEL2). alaku tényállításba, és ki is nyomtatjuk. Mivel az egész program -LAKAS (7o, 1, 2) -FAIL. célállitásokkal van meghiva, azaz az összes lehetséges megoldást keressük, ezért a visszalépések következtében a PERM eljárás az összes lehetséges permutációt generálni fogja, de kinyomtatni csak az egymástól különbözőket fogjuk (12. ábra).

```
                                                    89 =LAKAS(70, 1, 2) =FAIL.

AZ ADOTT ALAPLERULET:  70 NM
A PROGRAM ALTAL TERVEZETT LAKAS ALAPTERULETE:  69 NM

FODEMSZUKSEGLET:
F1 4,2 X 2,7:  6
F2 5,4 X 2,7:  0
F3 2,7 X 2,7:  0
CELLALISTA:  NAGY . NAGY . FELNAGY . FELNAGY . NIL
             NAGY . FELNAGY . NAGY . FELNAGY . NIL
             NAGY . FELNAGY . FELNAGY . NAGY . NIL
             FELNAGY . NAGY . NAGY . FELNAGY . NIL
             FELNAGY . NAGY . FELNAGY . NAGY . NIL
             FELNAGY . FELNAGY . NAGY . NAGY . NIL

AZ ADOTT ALAPLERULET:  70 NM

A PROGRAM ALTAL TERVEZETT LAKAS ALAPTERULETE:  65 NM

FODEMSZUKSEGLET:
F1 4,2 X 2,7:  5
F2 5,4 X 2,7:  0
F3 2,7 X 2,7:  1
CELLALISTA:  NAGY . NAGY . FELNAGY . FELKIS . NIL
             NAGY . NAGY . FELKIS . FELNAGY . NIL
             NAGY . FELNAGY . NAGY . FELKIS . NIL
             NAGY . FELNAGY . FELKIS . NAGY . NIL
             NAGY . FELKIS . NAGY . FELNAGY . NIL
             NAGY . FELKIS . FELNAGY . NAGY . NIL
             FELNAGY . NAGY . NAGY . FELKIS . NIL
             FELNAGY . NAGY . FELKIS . NAGY . NIL
             FELNAGY . FELKIS . NAGY . NAGY . NIL
             FELKIS . NAGY . NAGY . FELNAGY . NIL
             FELKIS . NAGY . FELNAGY . NAGY . NIL
             FELKIS . FELNAGY . NAGY . NAGY . NIL

AZ ADOTT ALAPLERULET:  70 NM
A PROGRAM ALTAL TERVEZETT LAKAS ALAPTERULETE:  61 NM

FODEMSZUKSEGLET:
F1 4,2 X 2,7:  4
F2 5,4 X 2,7:  0
F3 2,7 X 2,7:  2
CELLALISTA:  NAGY . NAGY . FELKIS . FELKIS . NIL
             NAGY . FELKIS . NAGY . FELKIS . NIL
             NAGY . FELKIS . FELKIS . NAGY . NIL
             FELKIS . NAGY . NAGY . FELKIS . NIL
             FELKIS . NAGY . FELKIS . NAGY . NIL
             FELKIS . FELKIS . NAGY . NAGY . NIL
89.23 A (*TIME: 160 SEC 6951 CALLS *)
```

12. ábra

Az előkészitő program célállitása és az output.

Egy lista összes lehetséges permutációinak generálása olyan általános feladat, amely más programban is használható, s amelynek rekurziv definiciója PROLOG-ban igen tömör és szemléletes:

```
+PERM (NIL, NIL).
+PERM(*A, *E. *Y)
    -ELEME (*A,*E)
    -KIHAGY( *A, *E, *B)
    -PERM (*B, *Y).
+ELEME ( NIL,  NIL).
+ELEME ( *X. *L,  *X).
+ELEME ( *X. *L,  *Y)
    -ELEME (*L, *Y).

+KIHAGY (*X. *L,  *X, *L).
+KIHAGY (*Y. *L,  *X, *Y. *Ll)
    -KIHAGY (*L, *X,  *Ll).
```

A fenti programrészletet a következő módon értelmezhetjük:

PERM:

Az üres lista (NIL) permutációja az üres lista.
Egy *A lista permitációja egy olan *E. *Y lista lesz, ahol

- *E a lista első eleme és *Y a maradék lista;
- *E az *A lista egy eleme;
- az *A listából az *E elemét kihagyva *B listát kapunk;
- *B lista egy permutációja *Y lista.

ELEME:
Az üres lista eleme az üres lista.
Egy *X.*L alaku lista eleme az *X. (az első eleme)-
Egy *X.*L alaku lista eleme az *Y, ha eleme az *L maradéklistának.

KIHAGY:
Egy *X.*L alaku listából ha kihagyjuk az *X elemét, akkor *L listát kapunk.
Egy *Y. *L alaku listából ha kihagyjuk az *X elemét egy *Y. *Ll alaku listát kapunk, mégpedig úgy, hogy az *L listából kihagyva az *X elemet *Ll listát kapunk.

6. Hierarchikus, strukturált programozás

A PROLOG nyelv felépitéséből adódik, hogy szinte csak strukturált programokat irhatunk, és a programszervezés hierarchikus. PROLOG-ban ugyanis ugy kell programozni, hogy a feladatot részfeladatokra bontjuk reduktorok segitségével, majd ezeket tovább bontjuk stb.
A részfeladatokra bontás egészen addig történik, amíg a részfeladatok olyan egyszerűnek nem lesznek, hogy megoldhatók tényállásokkal, vagy az interpreterbe beépitett eljárásokkal. Nagyon megkönnyíti a programozási technikát, hogy bizonyos részfeladatok a program egészétől függetlenül megfogalmazhatók.

7. Prioritás

A PROLOG következtetési mechanizmusának egyik alapvető tulajdonsága, hogy ha egy szabályból több változat

van, akkor mindig a lehetséges elsőt választja. Ezáltal lehetőséget ad bizonyos prioritások érvényesitésére. A prioritásokon egyszerű sorrendcserével lehet változtatni. Igy amikor a program előállitja az összes alaprajzi variánst, először a számunkra kedvezőbb, később a kevésbé kedvező, de lehetséges változatokat állitja elő.

8. Módosithatóság

A PROLOG nyelven irt programot rendkivül egyszerü módositani. Egy PROLOG program Horn-klózok listája, ahol a klózok ponttal vannak elválasztva. Ha valamely szabályból több változatot akarunk készitени, egyszerüen felsoroljuk a szabályokat egymás után. Ha bőviteni akarjuk e szabályok körét, akkor anélkül, hogy a már megirt utasitásokon bármit is változtatnánk, a meglévő szabályok mögé beirjuk az uj szabályokat. Éppen ilyen egyszerű bizonyos szabályok elhagyása, Ennek igen nagy a jelentősége, hiszen például kellően nagy feladat esetén a variációk száma olyan naggyá válhat, hogy bizonyos megszoritások, és letiltások nélkül a program nem lenne gazdaságos.

3. LOGIKAI ALAPU PROGRAMOK KOMPLEXITÁSA

3.1. A KOMPLEXITÁS SZÁMITÁS CÉLJA

Jelenleg több időt és energiát igényel a meglévő software eszközök karbantartása illetve módositása, mint az új programok fejlesztése. Több tudományos központban megindult a kutatás olyan, számmal mérhető mértékek definiálására, amelyek valamilyen módon a programok egyszerűségét, érthetőségét és főleg megbizható módosithatóságát mérnék. Ilyen mértékek birtokában objetiv módon lehetne értékelhetni a különböző programvariánasokat, és a változó szempontoknak megfelelően optimális mutatókat lehetne javasolni.

Bill Curtis és társai cikkükben [6] a programok kétfajta komplexitását különböztetik meg:

- számitási (computational) és
- pszichológiai komplexitást.

A számitási komplexitás a feladat megoldásának kiszámitásával kapcsolatos kvantitatív aspektusokat veszi figyelembe, mint például a megoldó algoritmusok gyorsasága, vagy a lehetséges programágak száma.
A pszichológiai komplexitás a software eszközök azon tulajdonságaira utal, amelyek a program érthetőségét, egyszerű és gyors módosithatóságát befolyásolják.

Az utóbbi években kifejlesztett komplexitáselméletek közül jelentősnek mondható M.H. Halstead munkássága [14]. Halstead a programok generálására fordított munka nagyságát négy egyszerü paraméterből vezeti le:

η_1 - a programban előforduló különböző operátorok száma;

η_2 - a programban előforduló különböző operandusok száma;

N_1 - az összes operátor teljes előfordulásainak összege;

N_2 - az összes operandus teljes előfordulásainak összege,

ahol operátor minden alapszó (pl. GOTO, IF THEN ELSE, BEGIN...END, stb) , cimke, zárójelek, matematikai műveleti jelek (pl. +, -, *, :), értékadás jele, az utasitásokat elválasztó jel (pl. ;), és egyéb az előzőektől szintaktikusan megkülönböztethető jelek;

operandus minden változó és konstans.

A program hossza N=N1+N2;

A program szótárának nagysága $\eta=\eta_1+\eta_2$;

A program"köbtartalma" (volume) $V= N \log_2 \eta$;

A program minimális vagy potenciális köbtartalma

$$V = \eta^* \log_2 \eta^*$$

ahol V^* a program olyan változatának köbtartalma, amelyben az operátorok és az operandusok nem ismétlődnek, tehát

$$N_1^* = \eta_1^* \,, \quad N_2^* = \eta_2^* \quad \text{és} \quad \eta^* = \eta_1^* + \eta_2^* \,.$$

A program szintje $L = \frac{V^*}{V}$; közelitõleg

$$L \sim \frac{2\eta_2}{\eta_1 \cdot N_2} \,;$$

A programozási munka mértéke (programming effort)

$$E = \frac{V}{L} = \frac{V^2}{V^*}$$

közelitõleg $$E = \frac{V}{L} \sim \frac{\eta_1 \, N_2 (N_1+N_2) \, \log_2(\eta_1+\eta_2)}{2\eta_2}$$

Ez utóbbi formulával lehet kiszámitani közvetlenül a programlistából a program "pszichológiai" komplexitását.

Az irodalomban publikált komplexitás mértékek közül a McCabe ötletének volt a legnagyobb sikere [33]. McCabe a programok blokkdiagramjait irányitott gráfoknak tekinti, s a gráfok ciklomatikus számának kiszámitásával az alapvetõ programágak számát határozza meg.

V(G)= élek száma - csucsok száma +2

ahol - G egy P program blokkdiagrammja által
meghatározott irányitott grág,
-V(G) a P program McCabe szerinti komplexitás
mértéke.

McCabe javasolja, hogy egy önálló program egység, egy modul V(G) értéke ne haladja meg a lo-et. Szerinte íly módon biztositani lehet az egyszerű és áttekinthető programszerkezetet.

McCabe ötletének finomitásával G.J. Myers [35], és módositásával G. Oulsnam [38] foglalkozik. Halstead és McCeb módszereinek kisérleti alkalmazását és részletes értékelését B. Curtis és kutatócsoportja végezete el [6]. Szerintük a McCabe féle komplexitási mérték inkább a programok számitási, mintsem a pszichológiai komplexitását méri, habár figyelemreméltó összefüggés tapasztalható a kisérleti eredményekkel alátámasztott intuitív pszichológiai és a számitási komplexitás között. Vannak azonban a pszichológiai komplexitásnak olyan aspektusai, amelyeket sem McCabe, sem a Halstead féle mérték nem vett figyelembe. Ezért javasolják olyan új komplexitás-mértékek definiálását, amelyek jobban tükrözik a programozó információ-feldolgozási és probléma-megoldó képességét.

Éppen ez volt a célunk, amikor a PROLOG programok szerkezeti bonyolultságát vizsgálva bevezettünk új komplexitás mértékeket. Mivel a PROLOG "nagyon magas szintű" nyelv, a probléma-megoldás általános szerkezete sokkal jobban tükröződik magukban a programokban, mint a hagyo-

mányos algoritmikus nyelvek esetén. Így a logikai programozás eleve strukturált, hierarchikus szervezettsége megkönnyítette számunkra, hogy olyan programegységeket találjunk, amelyek mérete, és mennyisége befolyásolja a programok érthetőségét, módosithatóságát, azaz a "pszichológiai komplexitást".

3.2. PRIMLOG - A PROLOG EGY MEGSZORITÁSA

A strukturált programozás elveinek megfelelően be lehet vezetni a PROLOG-ban olyan szintaktikus megszoritásokat, amelyek csak a legegyszerűbb programszerkezetk használatát engedik meg(PRIMLOG = PRIMitiv proLOG). Ezen egyszerű programszerkezetek definiálásával és egy erre épülő programtervezési módszer bevezetésével az volt a célunk, hogy a programok jól áttekinthetőek, könnyen módosithatóak legyenek, s ezzel párhuzamosan a szemantikus hibák csökkenését, a tesztelési idő leröviditését vártuk [18].

Az alapvető programszekezeteket particióknak neveztük. A TASK particiót kivéve - amely a feladat megadását definiáló célállitás - egy partició olyan klózok sorozata, amelyek pozitiv atomjai ugyanazt a relációt tartalmazzák. A szintaktikus megszoritás kulcsszáma a a kettő (2)

- a partición belüli klózok száma,
- egy partición belüli negativ atomok száma,
- egy atomon belül az argumentumok száma,
- és egy argumentumon belül az operátorok száma

maximálisan kettő lehet. (Kivételt képez a DATA BASE partició, amely olyan egyszerű, hogy a benne lévő klózok számára nem tettünk kikötést).

Egy PRIMLOG program particiók sorozata. 5 különböző tipusú particiót definiáltunk: TASK, AND, OR, RECURSION és DATA BASE. Minden program egy TASK particióval kezdődik, és ezen kivül még legalább egy DATA BASE particiót kell hogy tartalmazzon.

A particiók definiciója*:

TASK: $-R(\delta_1, \delta_2) \; - P(p_1, p_2).$

AND: $+R(\tau_1, \tau_2) \; - A(\delta_1, \delta_2) \; - B(p_1, p_2).$

OR: $+R(\tau_1, \tau_2) \; - A(\zeta_1, \zeta_2).$

$+R(\delta_1, \delta_2) \; - B(p_1, p_2).$

RECURSION: $+R(\tau_1, \tau_2).$

$+R(\delta_1, \delta_2) \; - A(\zeta_1, \zeta_2) \; - R(p_1, p_2).$

vagy

---

* *Egy A ←B1 ∧B2 formáju implikációt a PROLOG magyar verziójának megfelelő szintaktikus konvenció szerint +A -B1 -B2 formával fejezünk ki.*

$+R\ (\delta_1,\delta_2)\ -A\ (\zeta_1,\zeta_2)$

$-R\ (p_1,p_2).$

$+R\ (\tau_1,\tau_2).$

DATA BASE: $+R\ (\tau_1,\tau_2).$

$+R\ (p_1,p_2).$

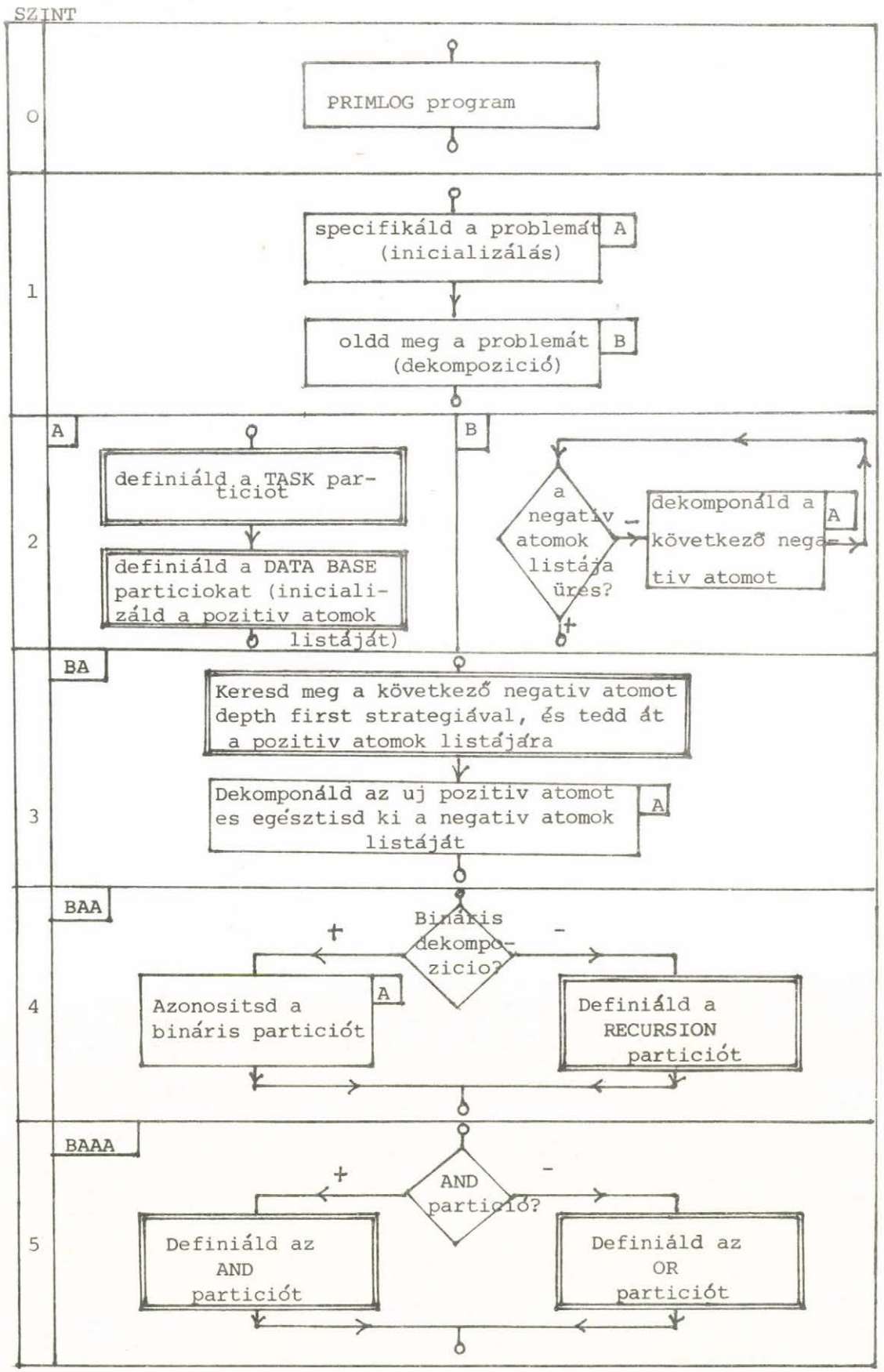
.
.
.

$+R\ (\delta_1,\delta_2).$

ahol R, P, A, B különböző bináris relációjelek;

$\tau_1,\tau_2,\zeta_1,\zeta_2,\delta_1,\delta_2,p_1,p_2$ termek, amelyekben legfeljebb két különböző függvényjel szerepelhet.

Egy PRIMLOG program irásának algoritmusát a következő blokk diagramban reprezentáljuk (13. ábra)



JELÖLÉS: ▭ - az akció tovább bomlik

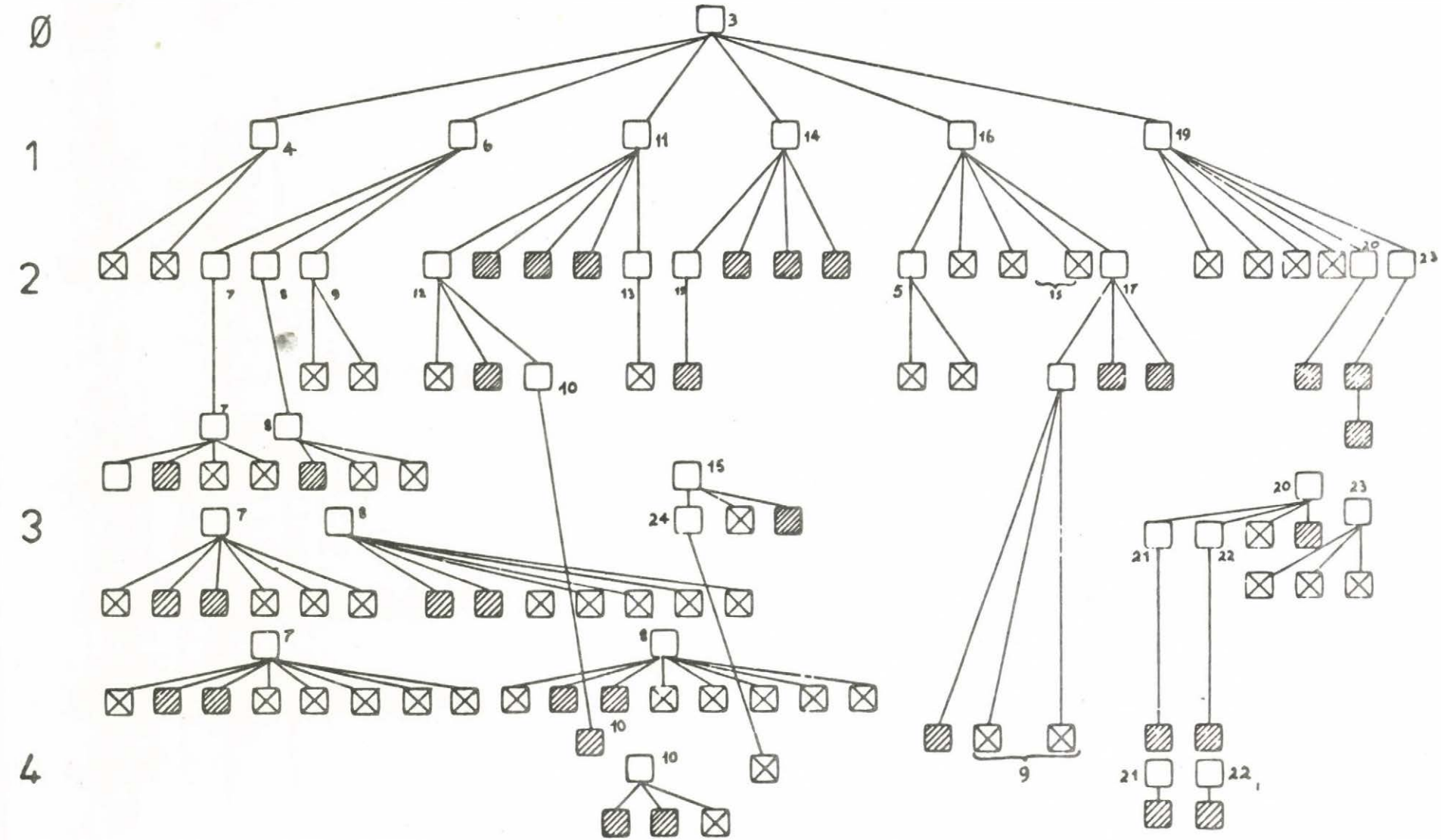
▭ - terminális akció

13. ábra

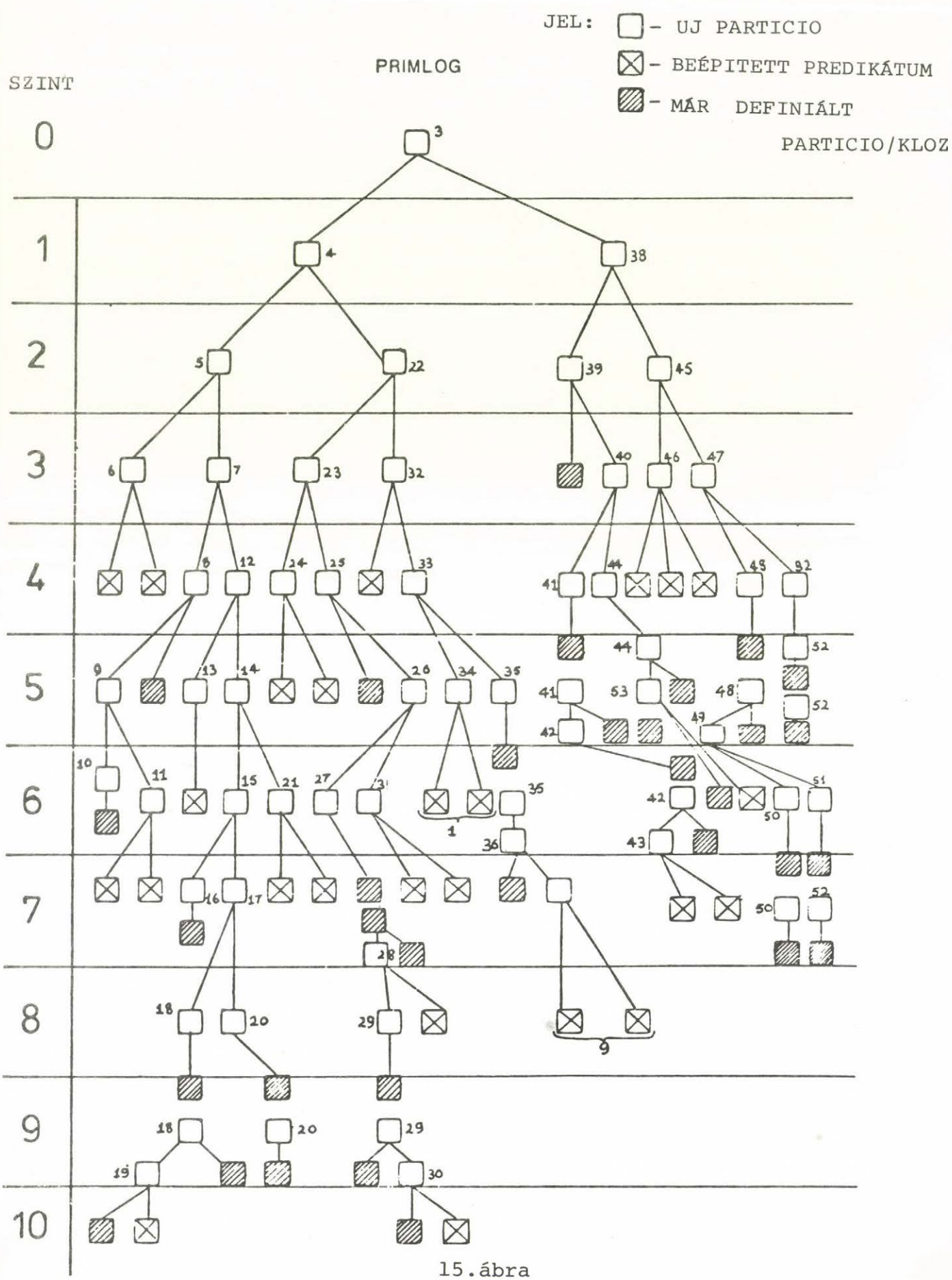
A PRIMLOG programfejlesztési módszert úgy hasonlitottuk össze az Ad-hoc PROLOG programozással, hogy elkészitettük a 2.2.1. fejezetben tárgyalt paneles lakásalaprajz variációkat tervező program előkészitő alprogramjának PRIMLOG változatát. A két programot ugyanazon a számitógépen futtattuk, a Polytechnic of the South Bank, London DEC-lo tipusu számitógépen. Azonos input adatra a két program változat ugyanazt az outputot adta, habár a programok szerkezetei erősen elütöttek egymástól, lásd. 14, 15. ábra. Megvizsgáltuk a programok hosszát, a futási időt és memória igényt is, s a következő konkluzióra jutottunk:

1. A PRIMLOG program szerkezete egyenletesen áttekinthető, világosan látszik rajta a bináris dekompozició szisztematikus használata, ugyanakkor a PROLOG program szerkezete kiegyensúlyozatlan, néhol egyszerű, néhol nagyon bonyolult dekompoziciós strukturát tartalmaz.

2. A PRIMLOG változat tesztelési ideje rendkívül rövid volt, mindössze 3 szemantikus hibát kellett korrigálni. A PROLOG program szemantikus hibáira nem tudunk pontos adatot mondani, mivel annak belövése három évvel előbbre nyulik vissza, de számuk jóval több volt, kb. lo-12. S habár nem szabad figyelmen kivül hagyni, hogy a PROLOG változat született először és a PRIMLOG változat másodszor, és egy második program mindig jobb mint az első, a szemantikus hibák csökkenésére bizonyára erős hatással volt az egyszerű és áttekinthető programszerkezet is.


SZINT
Ø
1
2
3
4
JEL: - UJ PARTICIO
- MÁR DEFINIÁLT PARTICIO/KLOZ
- BEÉPITETT PREDIKÁTUM

14.ábra


JEL: - UJ PARTICIO
- BEÉPITETT PREDIKÁTUM
- MÁR DEFINIÁLT PARTICIO/KLOZ
PRIMLOG
SZINT
0
1
2
3
4
5
6
7
8
9
10

15.ábra

3. A PRIMLOG program hossza több mint a kétszerese a PROLOG programnak, az előbbi 53, az utóbbi 24 particiót tartalmaz. Ennek megfelelően a memória-igény is megnövekedett 1699 szóról 28o3 szóra.

4. A PRIMLOG program hierarchikus szintjeinek száma lo, több mint kétszerese a PROLOG program megfelelő adatának, amely 4. Ezt egy kissé soknak találtuk, hiszen a globális áttekinthetőséget rossz irányban befolyásolja a hierarchikus szintek ilyen mértékü növekedése.

5. A vizsgált feladat futási idejében nincs lényeges különbség, PROLOG: 15,13 sec. PRIMLOG: 17,45 sec.

6. A PRIMLOG program önálló dekompoziciós egységeit, a particiókat tulságosan egyszerűnek találtuk. Egy partició néhol igen kevés gondolatot realizált, s úgy éreztük, hogy a programozó munkáját sok esetben nem könnyitették, hanem nehezitették a rákényszerült szigoru dekompoziciós szabályok.

Az 1. és 2. pontban felsorolt előnyöket és a 3, 4. és 6. pontban összefoglalt nyilvánvaló hátrányokat egybevetve arra a meggondolásra jutottunk, hogy a PRIMLOG alapötlete jó, de előnyösebb megvalósitásához a definició némi korrekciójára lenne szükség. Több gondolatot

tartalmazó, nagyobb (ha ugy tetszik komplexebb) particiókat is meg kellene engedni, s ezzel párhuzamosan a hierarchikus szintek száma, és a program hossza is csökkenne.

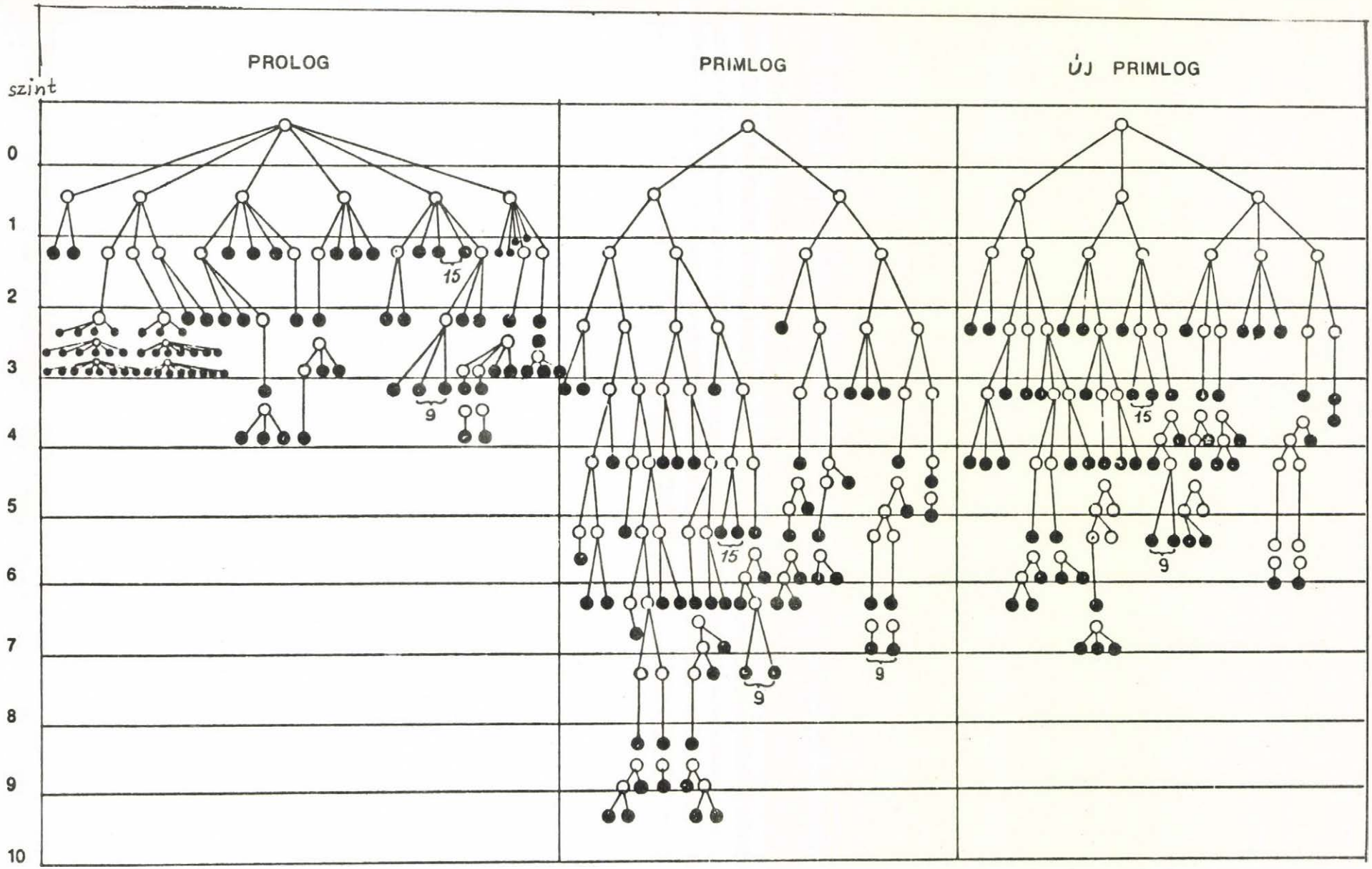
Az UJ PRIMLOG változatához a következő változtatásokat javasoltuk:

1. Az AND partició bővitése.
   +A - B1 - B2 - B3.
   ha a B1, B2, B3 negatív atomok közül az egyik beépitett eljárás, vagy egyesithető egy DATA BASE klózzal.

2. Az OR partició bővitése.
   +A - B1 - B2.
   +A - C1 - C2.
   ha a B1, B2, C1, C2 negatív atomok közül az egyik beépitett eljárás, vagy egyesithető egy DATA BASE klózzal.

3. A CASE partició bevezetése.
   +A - B1.
   +A - B2.
   .
   .
   .
   +A - BN.
   ahol N bármilyen természetes szám.

Ezeknek a bővitéseknek a figyelembevételével elkészitettük a program egy harmadik, UJ PRIMLOG változatát. A várakozásnak megfelelően e harmadik változatnál jelentősen csökkentek azok a tényezők, amelyek a PRIMLOG kedvezőtlenebb tulajdonságait hangsulyozták, megőrizve az előnyként kiemelt áttekinthető programszerkezetet. A három programváltozat szerkezetének összehasonlitó ábrája a 16. ábrán látható, egyéb tulajdonságainak részletes elemzését a következő fejezetben közöljük.



JEL: O - UJ PARTICIO

● - MÁR DEFINIÁLT PARTICIO/KLOZ VAGY BEÉPITETT PREDIKÁTUM

16. ábra

3.3. LOKÁLIS ÉS GLOBÁLIS KOMPLEXITÁS MUTATÓK BEVEZETÉSE

A PRIMLOG programozási módszerrel való kisélet annak a felismeréséhez vezetett, hogy a logikai alapu programozás vizsgálata elősegiti a program tervezés folyamatának analizisét, és a hiba-források feltárását. Ebben a fejezetben javaslunk egy olyan elméletet, amely szerint a program-fejlesztő feladatának komplexitása négy faktorra bontható, amelyek mindegyike számokkal kifejezhető a logikai alapu programozásban. Ennek megfelelően javaslunk a progfejlesztés feladatára egy durva komplexitásmutatót, mint eme négy faktor összegét, amelyből kettő az adat specifikációval, kettő a feladat strukturális dekompozicióval áll kapcsolatban.[19]

3.3.1. A program-tervezés komplexitása, szemantikus hibák

Tegyük fel, hogy egy probléma nehézségi foka mérhető azon valós eszközök számával, amelyek szükségesek a megoldásához . CAD feladatok esetén valós eszközök lehetnek a kész software termékek, mint program-könyvtárak, szubrutin csomagok, vagy a programozási nyelvek utasitás készletei. Egy CAD probléma általában olyan bonyolult, hogy nem igen lehet megoldani a valós eszközök közvetlen kombinálásával.
A 13. ábrán látható algoritmusnak

megfelelően a top-down probléma-megoldás egy egyszerűsitett folyamata a következőképpen irható le:

1. Specifikáljuk a valós problémát azaz definiáljuk a problémateret, és felsoroljuk az elérhető valós eszközöket.
2. A problémateret a valós probléma felbontásával részletesebben definiáljuk, a probléma megoldására absztrakt eszközöket javasolunk, valamint meghatározzuk az absztrakt eszközökön lévő relációkat. Ha egy absztrakt eszköz illeszthető egy valós eszközzel, úgy további dekompozicióra nincs szükség.
3. A fennmaradó abasztrakt eszközöket absztrakt problémaként kezeljük a következő hierarchikus szinten.
4. Mindegyik absztrakt problémát megoldjuk a 2. és 3. pont segitségével, amíg az összes absztrakt eszköz nem illeszthető valós eszközökkel, azaz a valós probléma meg nem oldódik.

A megoldást program formájában reprezentáljuk. A programozó munkája komplex, mivel egyidejüleg sok és különböző fajta döntéssel kell szembenéznie, s mivel egyik-másik döntéshez önmagában is komoly meggondolások kellenek. A gyakorlat azt mutatja, hogy a probléma definiálása után tervezési-strukturális hibák léphetnek fel

1. amikor az absztrakt probléma adatait ujra specifikálják (esetleg bontják) az új hierarchikus szinten,

2. amikor kiválasztják azokat az absztrakt eszközöket, amelyekre lebontják a feladatot,
3. amikor definiálják az absztrakt eszközökön lévő relációt,
4. amikor defiálják az adatokat, amelyek specifikálják ezeket az absztrakt eszközöket.

Vegyük észre, hogy a 2. és 4. a feladat szerkezetének tervezésével, mig az 1. és 3. a rajta áthaladó információ áramlással áll kapcsolatban.

Fontos felismerés, hogy az összes ilyen jellegü hiba két hierarchikus szinthez tartozik, azaz a hierarchia bármelyik K-ik szintjén lévő egyik problémájához, és az ő felbontásából adódó K+1 szint problémáihoz. Ezen megfigyelés alapján bevezetjük a lokális komplexitás fogalmát (jele:ℓ), mint négy paraméter függvényét, amelyek mindegyike megfelel egy-egy hiba-forrásnak:

$$\ell = f\ (p_1,\ p_2,\ p_3,\ p_4),$$

ahol
- p_1 - az új adat entitások száma, amelyek a K-ik hierarchikus szinten az S absztrakt probléma specifikációja következtében keletkeztek,
- p_2 - az absztrakt eszközök száma, amelyre az S probléma lebomlik a K+1-ik szinten,
- p_3 - az absztrakt eszközökön lévő reláció komplexitás mértéke,

p_4 - az új adat entitások száma, amelyek szükségesek az absztrakt eszközök specifikációjához,

és p_1, p_2, p_3, p_4 -et "komplexitás paramétereknek" nevezzük.

Az ℓ komplexitás függvényt első megközelitésben e négy paraméter összegeként javasoljuk. Lehetséges, hogy a jövőben némi alkalmazási tapasztalat alapján ezt az összeget valamilyen polinomális függvény fogja felváltani.

Legyen S egy probléma olyan megoldása, amely absztrakt problémák egy hierarchiája, és S legyen reprezentálva egy hierarchikus strukturált programmal. Ekkor javasoljuk, hogy a probléma "globális komplexitása" legyen az őt alkotó absztrakt problémák lokális komplexitásának összege:

$$G_S = \sum_{i+1}^{n} \ell_i ,$$

ahol G_S - a teljes S probléma globális komplexitása,

ℓ_i - az i-ik absztrakt probléma lokális komplexitása,

n - azon absztrakt problémák száma, amely sesitségével S problémát megoldották.

Amíg egy program globális komplexitásának mértéke összefügg a program hosszával, a memória igényel, és a futási idővel, semmilyen bizonyiték nincs arra, hogy egy programban lévő absztrakt problémák száma, azaz az "n" befolyásolná, hogy a programozó hány szemantikus hibát követ el a program tervezése közben. Arra a megállapitásra jutottunk, hogy a szemantikus hibák száma csökkenthető a programok lokális komplexitásának csökkentésével.

A programot lokális és globális komplexitásának fogalmát ugyan a logikai programozás elveit figyelembevéve definiáltuk, azonban az elmélet éppugy alkalmazható bármilyen hierarchikus rendszere [36,4o], és igy jó kiindulási alapot nyujthat más tipusu komplex rendszerek, mint például algoritmikus programozási nyelveken irt CAD rendszerek egyfajta tervezési metodológiájának magalapozásához.

3.3.2. PROLOG programok komplexitás mértékei

A lokális komplexitás általános definicióját szükitsük le a PROLOG programozás terminológiájára. Egy partició lokális komplexitása

$$\ell = p_1 + p_2 + p_3 + p_4,$$

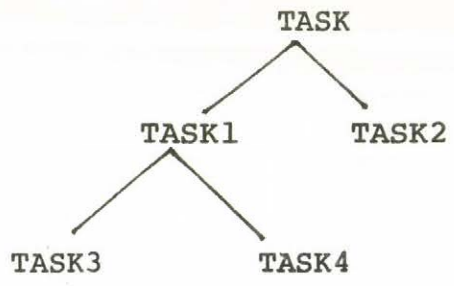
ahol p_1 - a pozitiv atonokban lévő új argumentumok száma,

p_2 - a partició negatív atomjainak száma,

p_3 - a reláció mértékszáma, amely mindig 0, kivéve a RECURSION partició esetét, amikor 2,

p_4 - a negativ atonokban lévő új argumentumok száma.

Az ℓ lokális komplexitás mértékének kiszámitását egy példán illusztráljuk. Legyen a TASK egy probléma, amely felbomlik két alapproblémára, TASK1-re és TASK2-re. A TASK1 ujra felbomlik TASK3-ra és TASK4-re:



17. ábra

Legyen a program-részlet a következő:

```
+TASK (*I, *O)    - TASK1  (*I,*X)
                  - TASK2  (*X,*O).
+TASK1  (*Q. *S,   *X)    - TASK3 (*Q,*Y)
                          - TASK4 (*S.*Y, *X).
```

1. TÁBLÁZAT

A TASK1 partició lokális komplexitás paraméterei:

Komplexitás paraméterek	Megjegyzések
$p_1 = 2$	*I felbomlott *Q-ra és *S-re
$p_2 = 2$	A TASK1 két negativ atonot, a TASK3-at és a TASK4-et tartalmazza
$p_3 = 0$	A TASK1 AND partició
$p_4 = 1$	A negativ atomokban egy uj argumentum, az *Y szerepel

$$\ell = p_1 + p_2 + p_3 + p_4 = 5$$

A számitást a következő módon végeztük el:
A TASK1 először a TASK partícióban szerepel, mint absztrakt eszköz (negativ atom). A lokális komplexitás első paramétere, a p_1 onnan adódik, hogy miközben az absztrakt eszköz absztrakt problémává (pozitív atommá) válik a következő hierarchikus szinten, az *I argumentum felbomlik két új adat entitássá, a *Q-ra és az *S-re. A p_2 =2 azt jelenti, hogy a TASK1 absztrakt problémát két absztrakt eszközzel (negatív atommal) definiáltuk. A p_3 = 0, mivel a TASK1 AND partició. A RECURSION partíciók esetén a p_3 komplexitás paraméter értéke nemcsak azért magasabb, mivel nehezebb megirni egy rekurzív particiót, mint a többit,

hanem azért is, mert egy szemantikus hiba itt könnyen végtelen rekurziót eredményezhet. A p_3 egy büntető paraméter.

Pszichológusok megfigyelései és a tapasztalat azt mutatja, hogy egy átlagos intelligenciájú ember egyszerre legfeljebb hét dologra tud odafigyelni. Ha hétnél több dolgot kell kontroll alatt tartani, megnő a hibák elkövetésének valószínüsége. Ezt figyelembevéve négy lokális komplexitási fokozatot vezettünk be:

2. TÁBLÁZAT

értéke	Komplexitási fokozat	Komplexitás kód
$1 \leq \ell \leq 3$	triviális	T
$4 \leq \ell \leq 7$	egyszerű	S
$8 \leq \ell \leq 17$	komplex	C
$18 \leq \ell$	nagyon komplex	V

Egy PROLOG program átlagos lokális komplexitása

$$\tilde{\ell} = \frac{1}{n} \sum_{i=1}^{n} \ell_i ,$$

ahol $\tilde{\ell}$ - a program átlagos lokális komplexitása,
n - a program particióinak száma,
ℓ_i - az i-ik partició lokális komplexitása.

Az $\tilde{\ell}$ nem függ a program hosszától, kizárólag a program szerkezetétől, azaz a programozási stílustól. Ha $\tilde{\ell} \leq 7$ egyszerű programról,ellenkező esetben komplex programról beszélünk. Minél kisebb az $\tilde{\ell}$, a programozó annál egyszerűbb alapegységekből (particiókból) épitette fel a programját, s a hibák elkövetésére az esélye annál alacsonyabb. Azonban a túl sok triviális partició megnöveli a program hosszát, memória igényét, a hierarchikus szintek számát, mint ahogy látni fogjuk egy példán a következő fejezetben. Ezért célszerű ugy programozni, hogy a programban túlnyomórészt egyszerű particiók szerepeljenek, és igy az átlagos lokális komplexitás 5 és 7 között legyen.

3.3.3. A komplexitás mutatók alkalmazása egy CAD program három változatának kiértékeléséhez

Térjünk vissza a 3.2. fejezetben tárgyalt program PROLOG, PRIMLOG és UJ PRIMLOG változatához. A programok szerkezetét grafikus formában összehasonlithatjuk a 16. ábrán. Azonnal szembeötlik, hogy a PROLOG változat rövid és széles, a PRIMLOG változat keskeny és hosszu ábrája mellett az UJ PRIMLOG változat közepes arányokkal az "arany középutat" jelenti. A programok szerkezetének számbeli kiértékelését a komplexitás mutatók segitségével fogjuk elvégezni. Nézzük az összegző táblázatot:

3. TÁBLÁZAT

	PROLOG	PRIMLOG	ÚJ PRIMLOG
Komplexitási fokozat	particiók száma	particiók száma	particiók száma
triviális (T)	4	21	lo
egyszerű (S)	9	25	27
komplex (C)	8	7	6
nagyon komplex (V)	3	-	-
átlagos lokális komplexitás $\tilde{\ell}$	lo (C)	5 (s)	5,7 (S)
maximális ℓ_i	34 (V)	11 (C)	11 (C)
a particiók száma n	24	53	43
globális komplexitás g	245	266	246
hierarchikus szintek száma	4	lo	7

A táblázat adatainak elemzése során a következő megállapításokat tehetjük:

1. A PROLOG változat particióinak majdnem a fele komplex, ezen belül 3 nagyon komplex. A program átlagos lokális komplexitása lo, azaz jóval 7 fölött van, szintén komplex.
2. A PRIMLOG és az ÚJ PRIMLOG változatban nagyon komplex partició nem szerepel, a komplex particiók az összes partició 14-15%-t teszik ki.

3. Feltünően sok a triviális partició a PRIMLOG változatban, mintegy 4o%-a a teljes programnak. Ezt a nagy számot az ÚJ PRIMLOG változatban sikerült csökkenteni a felére. Ez azt jelenti, hogy feleslegesen sok apró, triviális részfeladatot (absztrakt problémát) kreáltunk a PRIMLOG program tervezése során.
4. A három program közül a PRIMLOG program globális komplexitása a legmagasabb (266), mig a másik kettőé majdnem egyenlő (245, 246). Ez is arra mutat, hogy a PRIMLOG változat feleslegesen sok kis részfeladata valamilyen formában "elbonyolitotta" a megoldandó problémát.
5. Az UJ PRIMLOG program hossza és a hierarchikus szintjeinek száma jelentősen csökkent a PRIMLOG megfelelő adataihoz képest (lo particióval és 3 hierarchikus szinttel), ami mindenképpen az ÚJ PRIMLOG előnyeit bizonyitja.
6. A megfelelő változtatások bevezetésével az ÚJ PRIMLOG program átlagos lokális komplexitása a PRIMLOG-éhoz képest 5 -ről 5,7-re emelkedett, ami még mindig bőven a 7 alatt van. Tehát a tömörebb programszerkesztés nem ment a lokális komplexitás rovására. Sőt, ebből az adatból az is kiderül, hogy még jobban lazithatunk a szigorú szabályokon, s a PRIMLOG-ban definiált szintaktikus megszoritások kulcsszámát, a kettőt, felemelhetjük háromra, és helyenként - pl. az argumentumok maximális számánál - a négyre.

Minden összevetve megállapithatjuk, hogy az UJ PRIMLOG program szerencsésen ötvözi az áttekinthető, egyszerü egységekből álló programszerkezetet a PRIMLOG-énál tömörebb megfogalmazással. az UJ PRIMLOG program 5,7-es átlagos lokális komplexitása pedig arra enged következtetni, hogy az egy fokkal bővebb, gondolatgazdagabb particiók megengedése sem okoz olyan komplex programot, amely valószinüsitené a szemantikus hibák magas számát.

Ez utóbbi gondolatot a következő fejezetben ismertetésre kerülő CAD programrendszer fejlesztésénél teljes mértékig figyelembe vettük, és alkalmaztuk. A programok átlagos lokális komplexitása 6-7 körül mozgott, s a tesztelési idő, valamint az elkövetett szematnikus hibák száma figyelemreméltőan kevés volt. A komplexitás-számitás, valamint a program-tesztelés adatait az összegző táblázatban a programrendszer ismertetése után közöljük.

4. A LOGIKAI PROGRAMOZÁS ALKALMAZÁSA TÖBBSZINTES LAKÓÉPÜLET TERVEZÉSI RENDSZERÉHEZ

4.1. A PROGRAMRENDSZER FELADATA

Az alábbiakban ismertetésre kerülő programrendszer célja többszintes lakóépületek tervezésének számitógépes segitése. A feladat abban áll, hogy a lakásigénylők eltérő, változatos követelményeiket kielégitő különböző alaprajzú lakásaikból egy vagy több épületet kell tervezni,amelyek megfelelnek az adott müszaki és funkcionális feltételeknek. Egy-egy igénylő részére az egyik program több lakásalaprajz változatot állít elő, amelyekre az igénylő prioritási sorrendet állithat fel, és a kevésbé sikereseket kizárhatja. A megmaradt lakásalaprajz változatokból állítja össze egy másik program az épület vagy épületek szintenkénti alaprajzi elrendezését. Különböző épületvariánsokat, a tervező program épületszerkezeti adatainak megváltoztatásával és/vagy a háttéren tárolt lakásalaprajz gépi reprezentációinak (klózoknak) sorrendcseréjével vagy bővitésével kaphatunk.

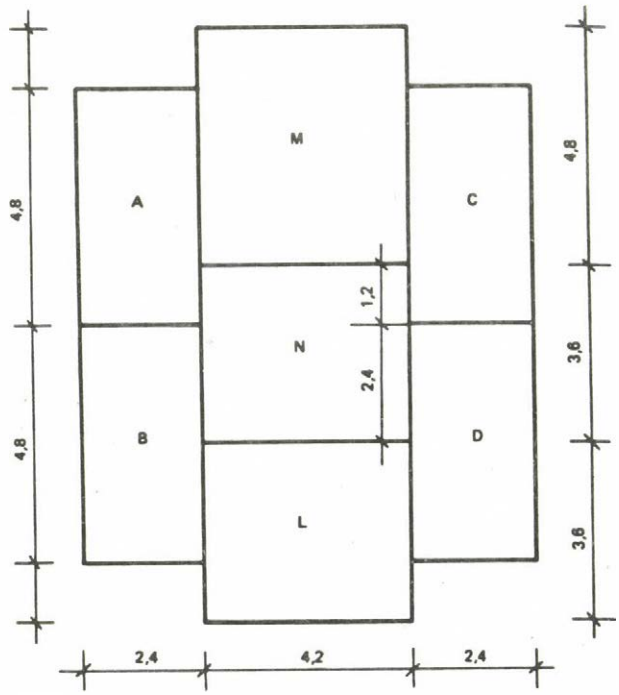
A programokat a Számitástechnikai Koordinációs Intézet Siemens 7.7.55 tipusu gépén fejlesztettük ki, és a Magyar Tudományos Akadémia IBM 3031-es számitógépére áthoztuk, és némileg módositottuk.

4.2. A LAKÓÉPÜLET TERVEZÉSI RENDSZERE, ADATBÁZIS, STRUKTURÁK

A jelen programrendszerben a lakóépület tervezésének alapsejtjei a funkcionális tartalommal felruházott, adott méretü cellák. Három különböző méretű cellát vettünk fel az adatbázisba:

méret	megnevezés
2,4 m x 4,8 m /keskeny/	A, B, C, D
4,2 m x 4,8 m /széles/	M
4,2 m x 3,6 m /széles/	N, L

Minden lakás 3 db széles cella és 1-4 db keskeny cella egymáshoz sorolásából jön létre. Egy általános, a maximális 7 cellából álló lakás geometriai méretei (méterben) a következőek:

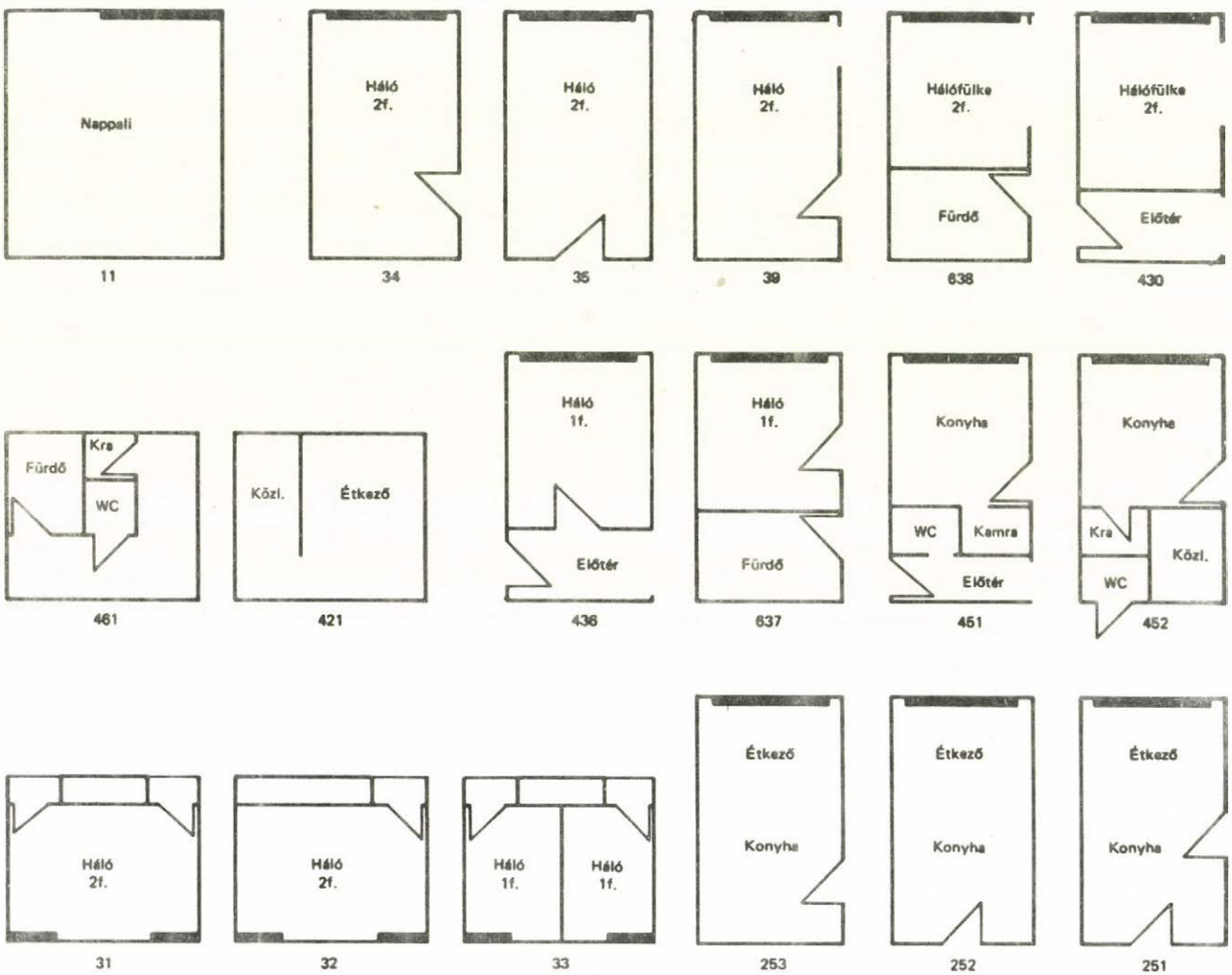


18. ábra
Egy lakás befoglaló konturja

A lakás nagyságától függően a keskeny (A, B, C, D) szeletek közül tetszés szerint 1, 2, 3 elhagyható. A cellák funkciói és azok kódjai a következők:

nappali	1
étkező	2
háló	3
bejárat	4
konyha	5
fürdő	6

Egy cella kettős funkcionális tartalommal is felruházható. Azonos méretű és funkcionális tartalmú cellák az ajtók számától és elhelyezésétől függően különböző változatokat képeznek. A változat száma az adatbázisban szereplő szeletek kódjának utolsó karaktere. A jelen programrendszerbe 18 különböző kóddal ellátott szelet került, amelyek a 19. ábrán láthatóak. Egy szelet kódja két vagy három számjegyű attól függően, hogy egy vagy két funkciót tölt be. Az utolsó számjegy mindig a variációszám, az első vagy az első kettő pedig a fenti funkcionális kódok.



19. ábra

A lakásalaprajz összeállitó program alap egységeinek illusztrációja

Egy lakást a programban egy lista reprezentál, a cellák listái:

*M. *N. *L; *A. *B. *C. *D vagy
*MIDDLE; *OUTSIDE

ahol

*MIDDLE - a középső széles szeletek listája
*OUTSIDE - a szélső keskeny szeletek listája.

A lakás tervezése során a listában szereplő változókat töltjük fel a kiválasztott cellák kódjának konstans értékével. Ha a lakás kevesebb, mint 7 cellát tartalmaz, az üres helyen változó marad a listában.

Két lakás és a lépcsőház alkot egy szekciószintet.
Egy vagy két szekciószint vizszintes sorolása alkot egy dilatációs egységszintet.

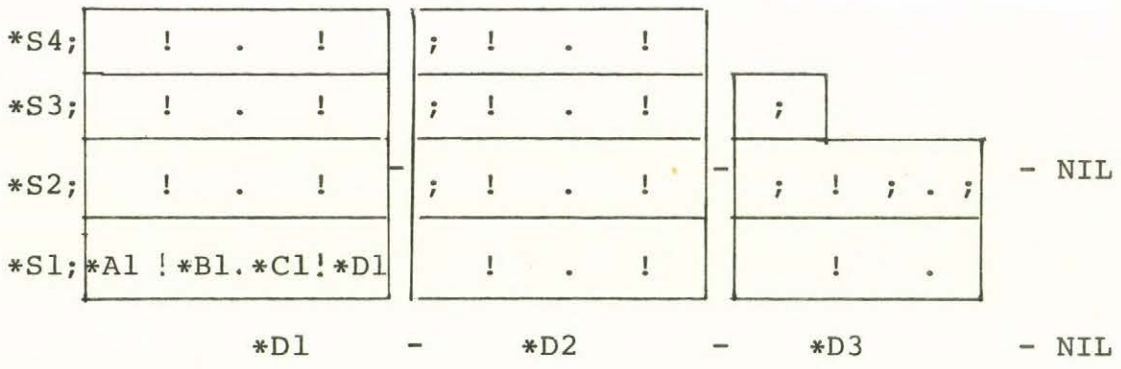
2-4 szekciószint függőleges sorolása alkot egy szekciót, 2-4 dilatációs egységszint függőleges sorolása egy dilatációs egységet.

Tetszőleges számu dilatációs egység vizszintes sorolása alkot egy épületet.

Egy épületet egy összetett lista reprezentál, amelyben négy különböző operátor szerepel. Az operátorok kötési prioritás sorrendjében a következőek:

! lakásokat köt össze (vizszintesen)
. szekciószinteket köt össze (vizszintesen)
; dilatációs szinteket köt össze (függőlegesen)
- dilatációs egységeket köt össze (vizszintesen)

Egy 4 emeletes 3 dilatációs egységet tartalmazó épület például a következő:



ahol - *D1, *D2, *D3 az egyes dilatációs egységek,
- *D1 bontása szintek szerint:
*D1 = *S4; *S3; *S2; *S1; *NIL
- *S1 bontása lakások szerint:
*S1 = *A1 ! *B1 '. *C1 ! *D1 . NIL

2o. ábra

Az épületet reprezentáló lista szerkezete modellezi azt az épitészeti koncepciót, amelyet rögzitettünk a feladat kijelölésével. Ez a programrendszer csak olyan lakásépületeket tud tervezni, amelyek maximálisan 4 szinttel rendelkeznek, egy lépcsőházból egy szinten két lakás nyilik, és egy dilatációs egység kettő, vagy egy lépcsőházat tartalmazhat. A 3. ábrán a ! jellel kapcsolódó lakások között van a lépcsőház, a . szimbólummal elválasztott lakások kapcsolódási sémáit a 7. ábra tartalmazza.

A tervezendő épületet reprezentáló bonyolult lista szerkezetét az egyik szegmensprogram (BUILD) állitja elő, és egyetlen argumentumban tárolja, hogy átadhassa a másik programnak (TOTAL) mint input értéket. Ez a lista

tahát nem kerül kinyomtatásra, csak belső ábrázolás-hoz szükséges. Helyette egy sokkal informálisabb output formát választottunk, lásd. az 22. ábrát, ahol is minden lakás helyére egy X szimbólumot nyomtattunk. Annyi X-ekből álló sor van egymás felett ahány emeletből áll az épület.

Az épületet reprezentáló lista először tehát csak változókat tartalmaz úgyis mondhatnánk, hogy üres szerkezet, váz, amelyet a későbbiek során töltünk fel lakásokkal.

Az épületet ugy tervezzük meg, hogy a lakásokat reprezentáló változók (a példában *A1, *B1, *C1, *D1) helyére a már megtervezett lakások konstansokból álló listáját helyettesitjük.

4.3. A PROGRAMRENDSZER RÉSZEI

A programrendszer a következő 4 programból áll:

1. Egyéni igényeket kielégitő lakásalaprajz variációkat tervező program (FLAT),
2. A lakás variációk prioritási sorrendjének kialakitását végző program (ORD),
3. A tervezendő épület vertikális szerkezetét kialakitó program (BUILD),
4. A lakóépület szintenkénti alaprajzi elrendezését tervező program (TOTAL).

A programlisták a Függelék 2.1.fejezetében találhatók.

4.3.1 FLAT

A lakásigénylő egyéni igényeinek megfelelően ez a program előállitja a lakás összes lehetséges alaprajzi változatát és a variánsokat a megrendelő kódjával ellátva lemezen tárolja. Ez a program annyiszor futtatandó, ahány lakásigénylő van.

INPUT:

A program inputját a - FLAT célállitás három argumentuma tartalmazza, ahol a második a megrendelő cégének a neve, a harmadik a megrendelő azonositó kódja. Az első argumentum egy nyolc elemből álló lista, amely sorrendeben a következő kérdésekre adott válaszokat tartalmazza:

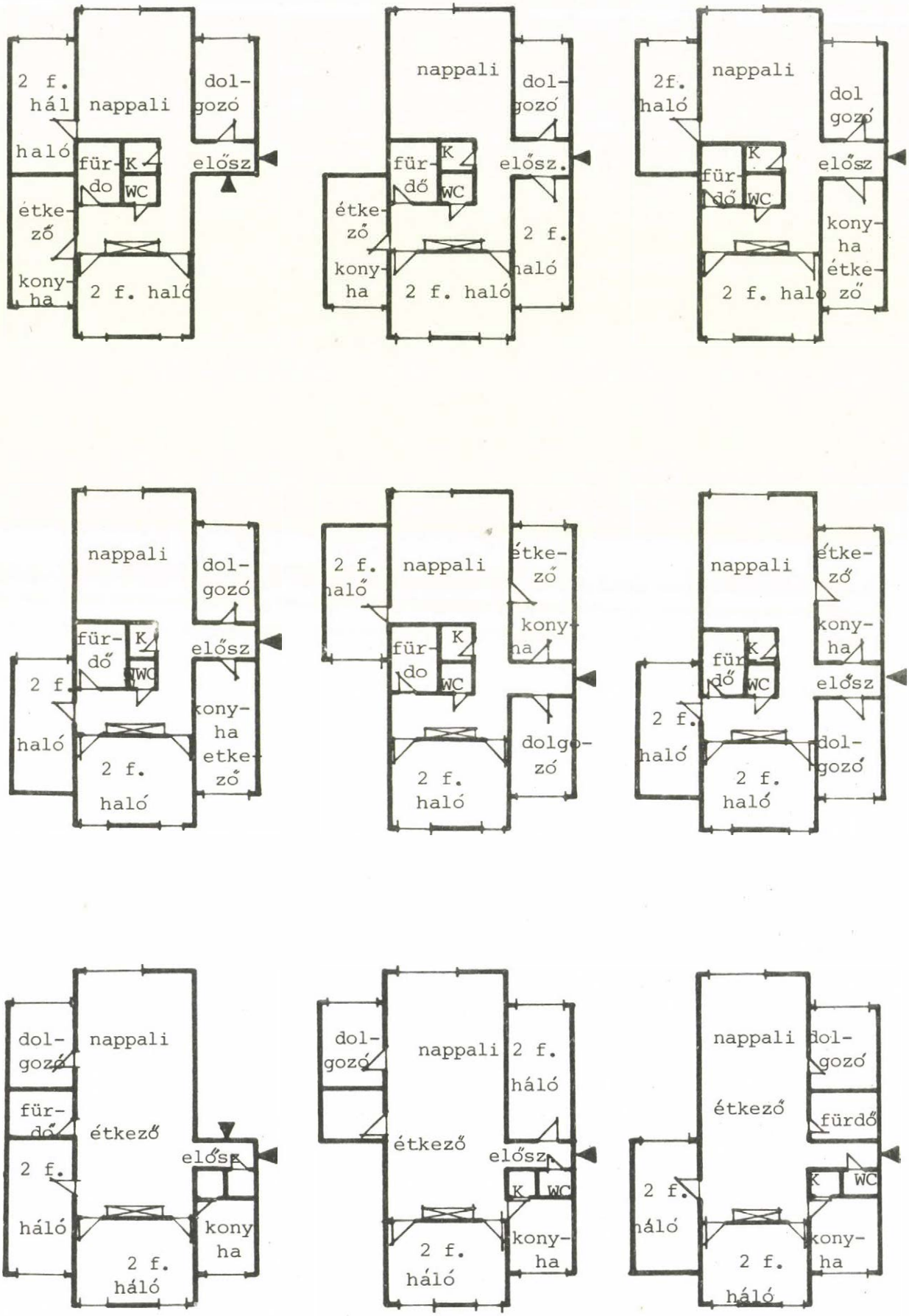
1. A nappali szobán kivül hány darab 2 férőhelyes hálószobát kivánnak? (o-3)
2. Hány darab 1 férőhelyes hálószobát kivánnak?
3. Az alábbi funkcionális kapcsolatok közül melyiket részesitenék előnyben?(Válaszlehetőségek: 1 vagy 2 vagy 3)
   konyha - étkező = 1;
   nappali - étkező = 2;
   nappali - étkező - konyha = 3;
4. Ha megengednek másfajtát is, melyik az? (1,2 vagy 3; Ø , ha nem)

5. Elfogadnak-e harmadik lehetőséget? (1,2 vagy 3; ∅, ha nem)
6. Dupla ágy elhelyezésére alkalmas szülői hálót kérnek-e? (I v. N)
7. Megengedik-e, hogy a nappali lakótérhez kapcsolódó 2 férőhelyes hálófülke létesüljön? (I v. N)
8. Kérnek-e dolgozószobát? (I v. N)

Output: - az input igényeket kielégitő lakásalaprajzok szeletlistái a háttér tárolón (disc)
- sornyomtatón mátrix formájában kinyomtatja a szeletlistákat a hozzájuk tartozó területi és férőhelyi adatokkal együtt:

- a lakás teljes alapterülete;
- a lakás lakóterülete;
- férőhelyek száma;
- a lakó- és lakásterület hányadosa;
- az 1 férőhelyre jutó alapterület

A 21. ábrán látható példában ezek szerint a megrendelő a WORK céghez tartozik, azonositó kódja lol5, és a következő kivánságai vannak: Kér 2 db kétszemélyes hálószobát, egyszemélyest nem kér, A konyha-étkező és a nappali-étkező kapcsolatot kéri, kér dupla ágyas szülői hálószobát, hálófülkére nem tart igényt, de kér egy dolgozószobát.



-FLAT (2.O.1.2.O.I.N.I.NIL, WORK, 1ol5) - FAIL.

21. ábra

Lakásalaprajz változatok

4.3.2 ORD

Ez a program lehetővé teszi, hogy a lakásigénylő a kívánságának megfelelő lakásokat saját véleményes szerint sorolhassa, tehát a lakások sorrendje a legkedvezőbbtől a még megfelelőig terjedjen, és ki tudja zárni azon alternativákat, amelyek szubjektiv itélete szerint neki nem felelnek meg. Ennek megoldási módja, hogy a kilistázott lakásokat sorszámmal látjuk el, majd a megrendelő prioritási sorba rendezi a sorszámokat, számára nem kivánatos lakásokat kihagyja.

Input: A FLAT program outputja.

Output: A program a prioritási sorrendnek megfelelően rendezi a lemezen lévő szeletlisták sorrendjét.

4.3.3 BUILD

A tervezendő épület maximális szintszáma és a kivánt lakásszám függvényében ez a program kiszámitja, hogy az épület hány dilatációs egységből fog állni. Megtervezi az általános és az utolsó - esetleg csonka - dilatációs egység szerkezetét szekciók, szintek, lakások száma . Kialakítja az épület általános vázát, amely bemenő adatként szolgál a következő programhoz. Ez a program csak kijelöli a lakások helyét az épület szerkezetében, de a konkrét lakásokkal még nem foglalkozik.

Input: - a tervezendő épületben lévő lakások száma;
- maximális szintszám;
- az egy dilatációs egységben lévő szekciók száma.

Output: - előállitja és kinyomtatja a teljes épület általános szerkezetét 22.ábra .

4.3.4 TOTAL

Ez a program felhasználja az előző három program eredményét. Célja az, hogy az épület területi és geometriai igényeit figyelembe véve feltöltse a BUILD program által kialakított strukturát a FLAT program többszöri futásával megtervezett lakásokkal. A lakások egymás mellé és egymás fölé sorolását ugy végzi, hogy kiválogatja azokat a lakásokat, amelyek a vízszintes és függőleges geometriai és funkcionális kapcsolatoknak megfelelnek.

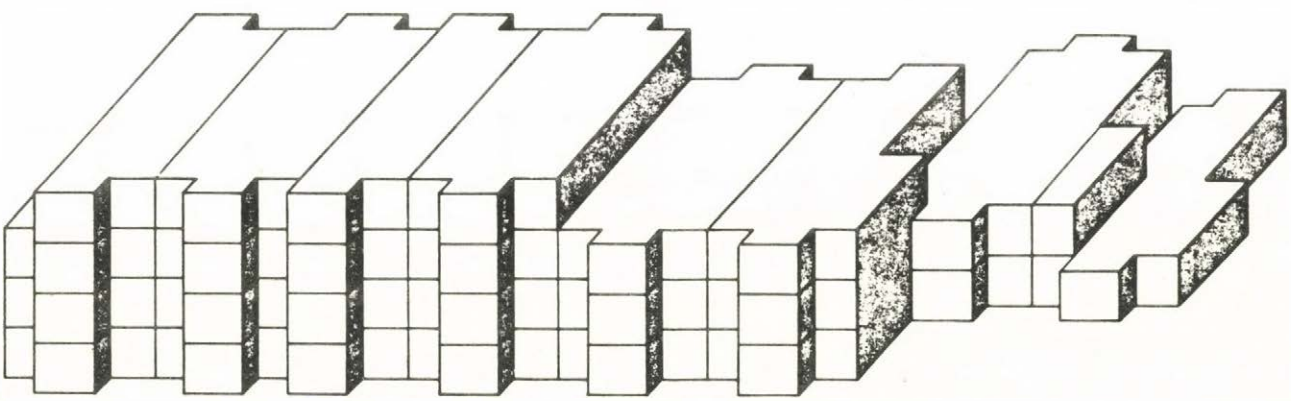
A BUILD program inputja:

-BUILD /25.4.2, S/.

A BUILD program outputja:

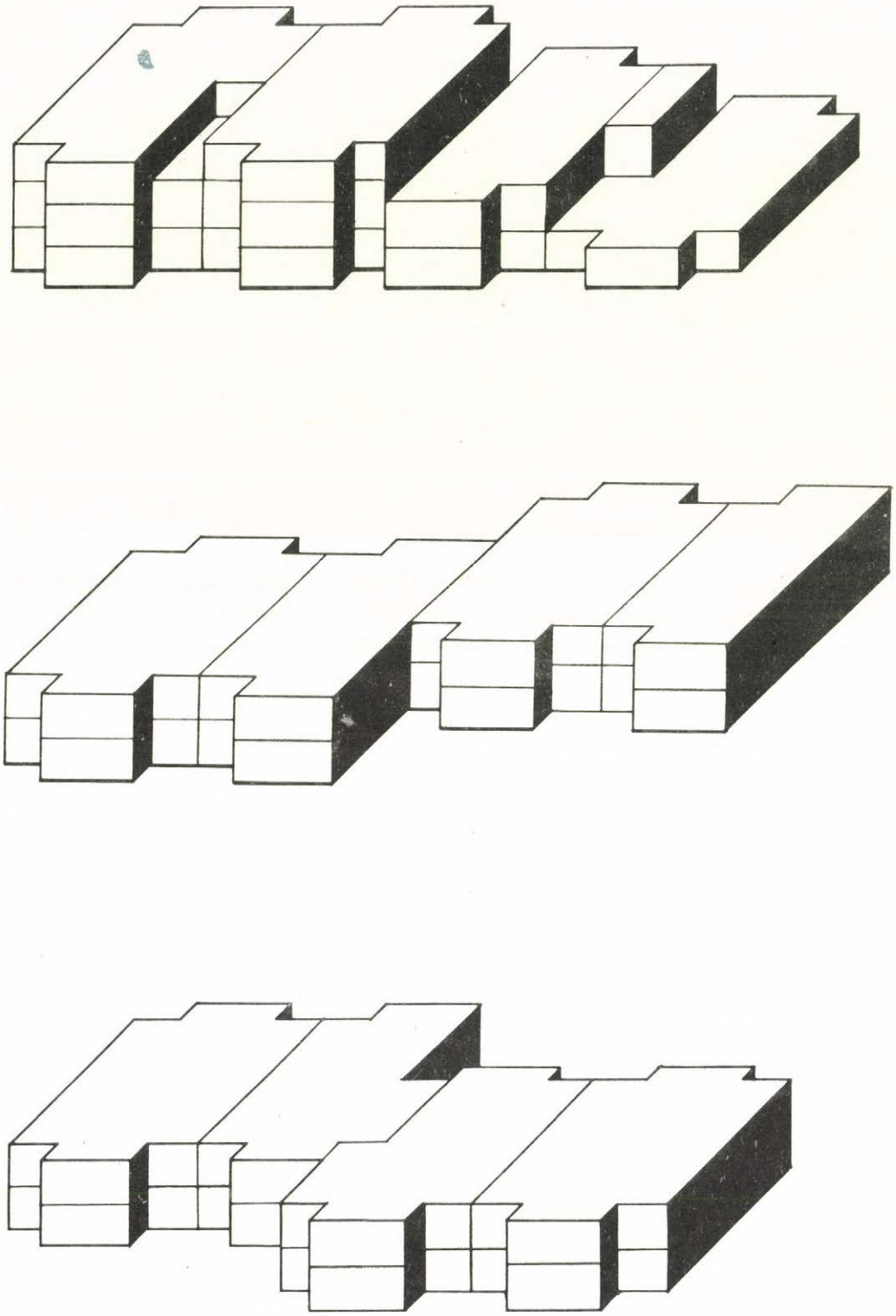
A tervezendő épület
lakásainak száma: 25
szintjeinek száma: 4
szekcióinak száma egy dilatációs szinten: 2
lakásainak száma egy szintszekcióban: 2

X X X X
X X X X X X
X X X X X X X
X X X X X X X X



22. ábra

A fenti outputnak megfelelő a TOTAL program által megtervezett egyik épületváltozat perspektivikus sémája



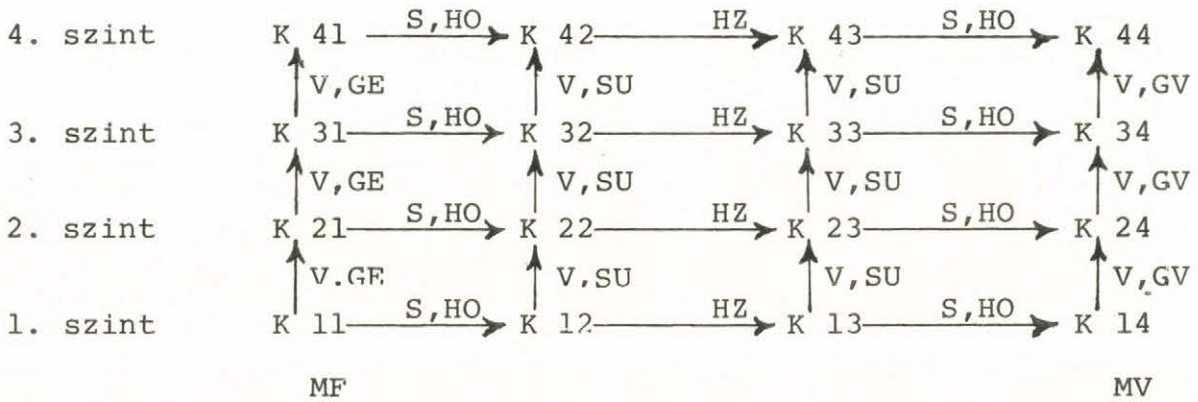
23. ábra

A TOTAL program által tervezett 25 lakást tartalmazó épületcsoport perspektivikus sémája

Input: - dilatációs egység bal végének típusa nyílt vagy zárt ;
- dilatációs egység jobb végének típusa;
- az első lépcsőház lakásai szintben, félszinttel felfelé vagy lefelé eltolva kapcsolódjanak-e?
- ugyanez a második lépcsőházra;
- az első és második lakás csatlakozásánál a keskeny szeletek száma mennyi legyen?
- ugyanez a második és harmadik lakásnál;
- ugyanez a harmadik és negyedik lakásnál;
- a két szekció el legyen-e csusztatva egymáshoz képest vizszintesen ? Jobbra vagy balra?

Output: Dilatációs egységként az épület szintenkénti alaprajzi elrendezése. Az egyes lakások a cellák kódjainak listájával és a lakásigénylő sorszámával adottak. A kódszámok mátrix alakban történő kinyomtatása a funkcionális szeletek és a lakások geometriai elrendezéséről vizuális képet ad.
26. és 27. ábra .

Az épületen belül az egyes lakások közötti kapcsolatokat következőképpen reprezentáljuk:



ahol K nm 1 n, m 4 - a lakásokat jelöli;

K nm $\xrightarrow{P,Q}$ K n m+1 - két lakás vizszintes kapcsolódását jelöli. A K nm jelű lakáshoz kiválasztjuk azt a K n m+1 lakást, amely megfelel a P és Q követelményeknek.

K n+1 m
↑ R,S
K nm - két lakás függőlges kapcsolódását jelöli. A K nm lakáshoz kiválasztjuk azt a K n+1 m lakást, amely megfelel az R és S követelményeknek.

Röviditések:

S - szimmetria,
HO - lépcsőház,
HZ - horizont,
MF - megfelel bal,
MV - megfelel jobb,
V - vertikális,
SU - támasz,
GE - geo eleje,
GV - geo vége.

Minden kapcsolathoz egy-egy eljárás tartozik, melyek funkciói a következőek:

1. SZIMMETRIA

Előállitja a FLAT program által tervezett lakás Y tengelyre vonatkozó szimmetria variánsát.

2. LÉPCSŐHÁZ

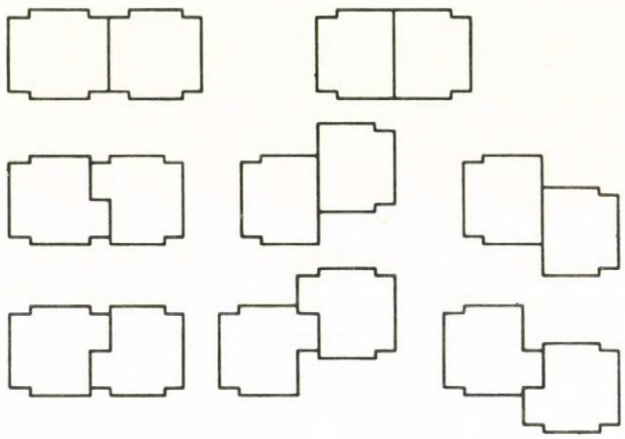
Ellenőrzi, hogy két egymás melletti lakás közé beilleszthető-e a lépcsőház ugy, hogy a lépcsőházból az előszobákba legyen a bejárat.

3. HORIZONT

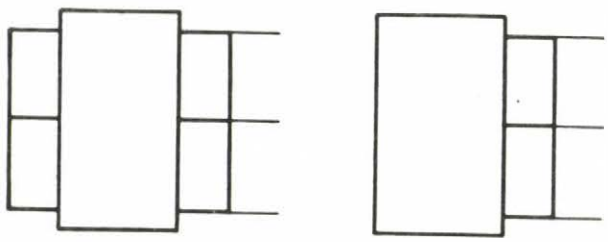
Ellenőrzi, hogy az első lakás jobb szeleteinek illetve a második lakás bal szeleteinek geometriai felépitése alkalmas-e a kapcsolódásra. A lehetséges kapcsolódási sémák egy része a 24. ábrán látható.

4. MEGFELEL BAL

Ellenőrzi, hogy a dilatációs egység bal szélén, az első szinten levő lakás K 11 bal szeletei megfelelnek-e a dilatációs egység bal tipusának. Ha ez a tipus NYILT, akkor bal felől nem csatlakozik egy másik dilatációs egységhez, ezért a szeletek konturja lehet akármilyen. Ha ZÁRT, akkor a csatlakozás biztositása miatt a lakás konturjának bal oldala beharapást nem tartalmazhat, tehát a 25. ábrán látható két változat jöhet szóba.



24. ábra
Horizontális kapcsolódási sémák



25. ábra
Zárt bal dilatációs vég két esete

5. MEGFELEL JOBB

Ugyanaz mint a 4. pontban, csak a dilatációs egység jobb szélén, az első szinten levő lakás jobb szeleteire vonatkozóan.

1. VERTICAL

Minden emeleti lakás kiválasztásához szükséges. Ellenőrzi, hogy - a kiválasztott lakás s az alatta lévő lakás vizesblokkjai egymás fölött vannak-e;
- az előszobák egymás fölött vannak-e;
- a felső szinten lévő lépcsőház az alsó szinten lévő felett van-e.

2. TÁMASZ

Ellenőrzi, hogy a felső lakás szeletei alatt mindenütt van-e alsó szelet.

3. GEO ELEJE

A dilatációs egység bal oldalán lévő emeleti lakások kiválasztásához szükséges. Ellenőrzi, hogy a fenti lakás bal szeletei megfelelnek-e a dilatációs egység bal széle tipusának, és a felső és alsó lakás geometriája összeegyeztethető-e.

4. GEO VÉGE

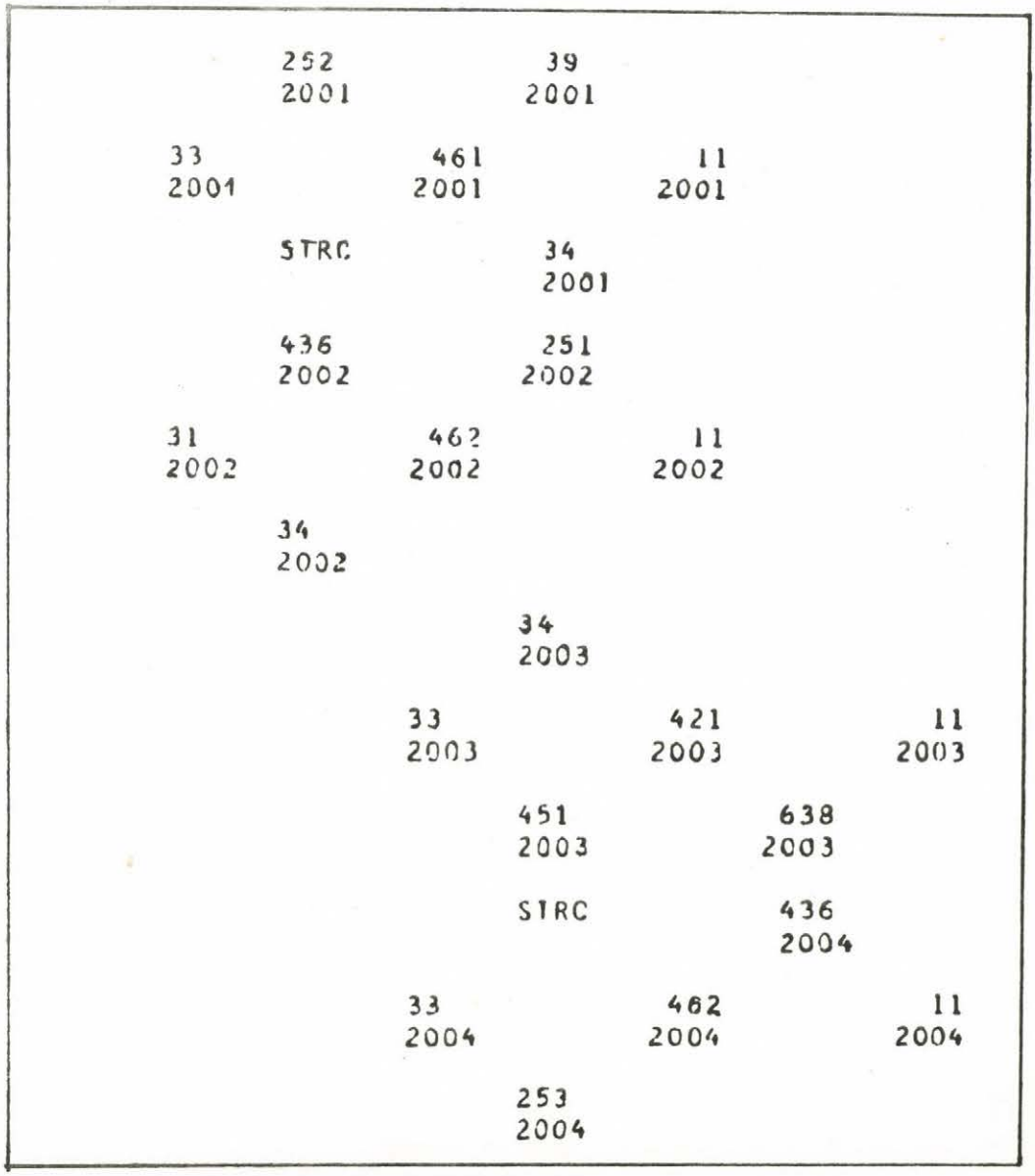
Ugyanaz mint a 3. pontban, csak a dilatációs egység jobb szélén lévő emeleti lakásokra vonatkozóan.

Egy dilatációs egység vázának lakásokkal való feltöltése a következőképpen történik:

- kiválasztjuk az 1.szint bal szélén lévő lakást K 11 a MEGFELEL BAL feltétel alkalmazásával;
- feltöltjük lakásokkal az 1.szintet K 12, K 13, K 14 sorrendben, a vizszintes feltételek figyelembevételével;
- kiválasztjuk a 2. szint bal szélén lévő lakást K 21 ugy, hogy az alatta lévő K 11 lakásra vonatkozóan alkalmazzuk a megfelelő függőleges feltételeket;
- a következő, K 22 lakás beillesztéséhez mind a mellette lévő K 21, mind az alatta lévő K 12 lakás szab feltételeket. A második szint többi lakását K 23, K 24 hasonló módon illesztjük be;
- a 2.szinthez hasonlóan töltjük fel a 3. és 4. szintet lakásokkal.

A 22. ábrán egy 25 lakásos háznak a BUILD program által kinyomtatott vertikális szerkezete látható, valamint a TOTAL program által összeállitott egyik épület perspektivikus sémája. A 26. ábrán ugyanennek az épületnek második dilatációs egységéből az 1.szint alaprajzi elrendezésének számitógépes outputja, a 27. ábrán pedig ez utóbbi rajzi megfelelőjét láthatjuk.

Az épület szintenkénti alaprajzi elrendezését a 28. 29. ábrán illusztráljuk.



Jelmagyarázat egy példán:

a 252 jelcsoport egy szeletet azonosit, ahol
2001

25- funkcionális kód

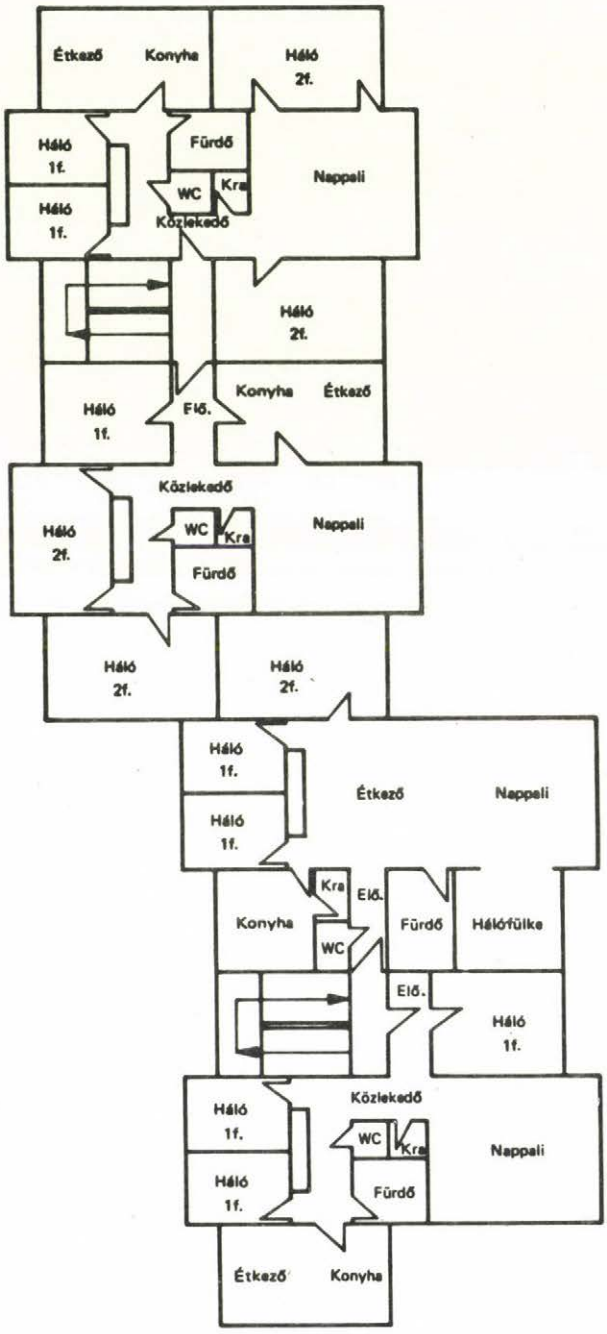
2- variáció szám

2001- a lakás megrendelőjének kódja

STRC- a lépcsőházi szelet jele

26. ábra

A 27. ábrának megfelelő szint alaprajzi elrendezése



27. ábra

A 26. ábrának megfelelő szint alaprajzi elrendezésének rajzi illusztrációja

A 28. 29. ábrákon látható röviditések jelentése:

NAP - nappali

2FH - 2 férõhelyes hálószoba

1FH - 1 férõhelyes hálószoba

2FF - 2 férõhelyes hálófülke

KON - konyha

ETK - érkező

FÜR - fürdõszoba

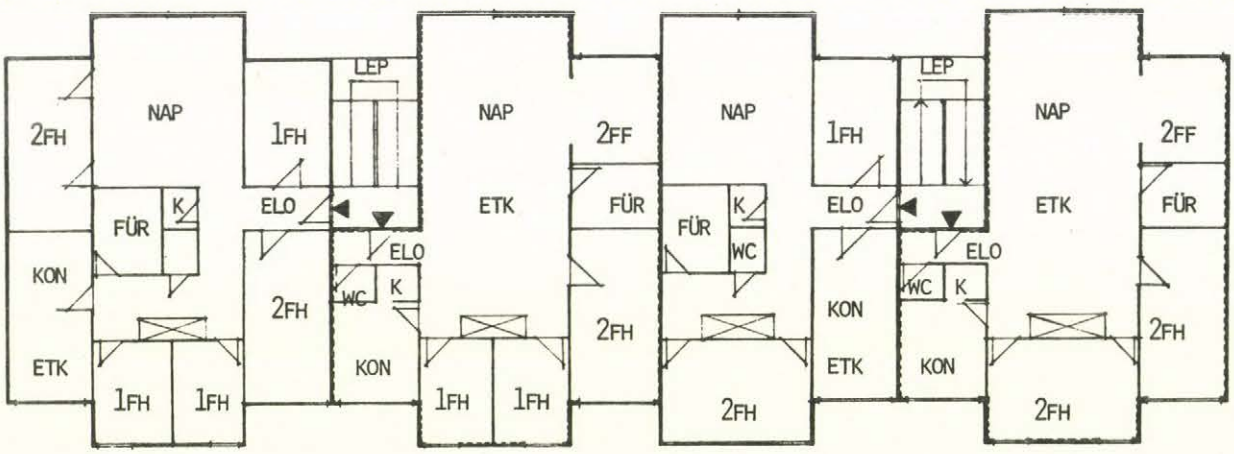
WC - WC

K - kamra

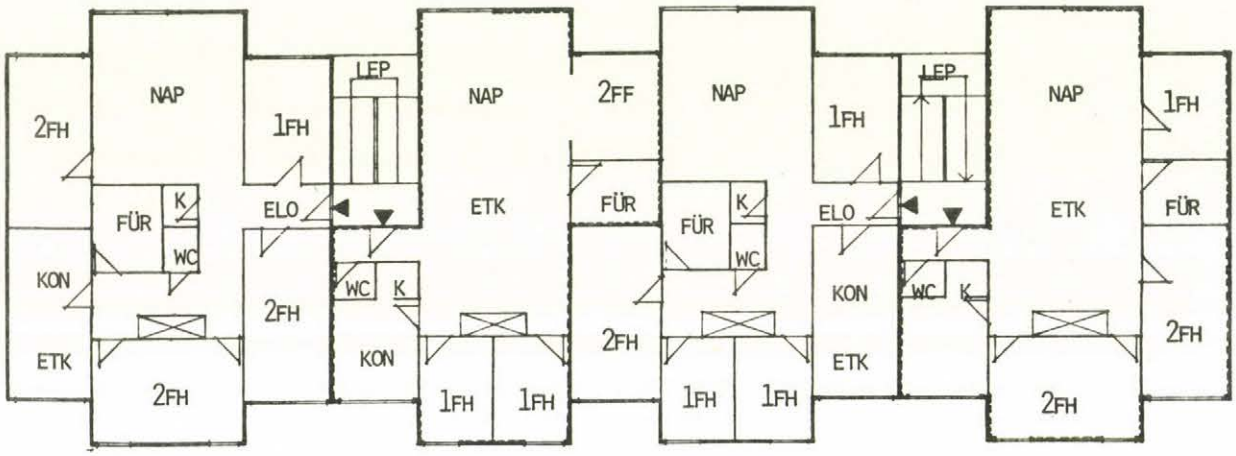
ELO - előszoba

LEP - lépcsõház

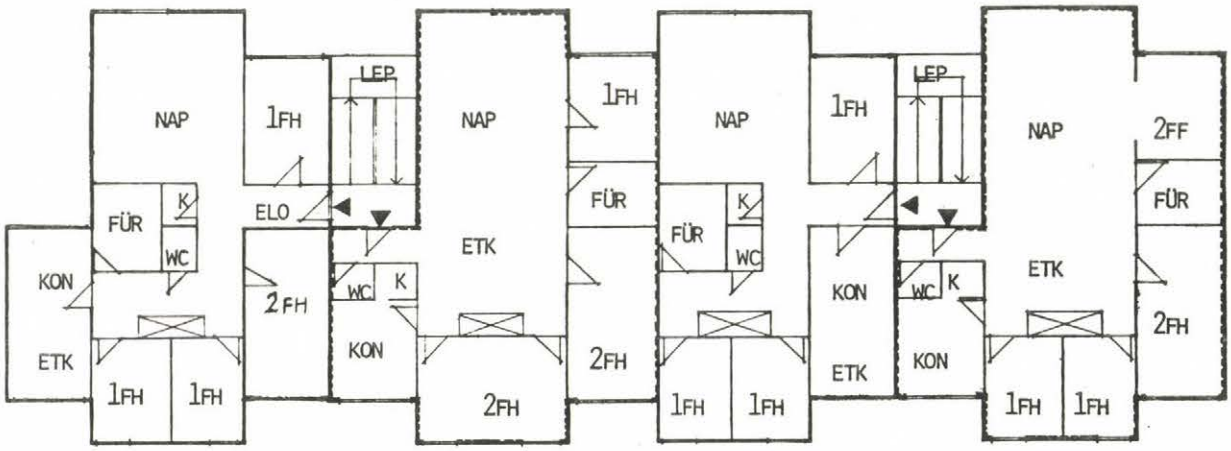
1. SZINT



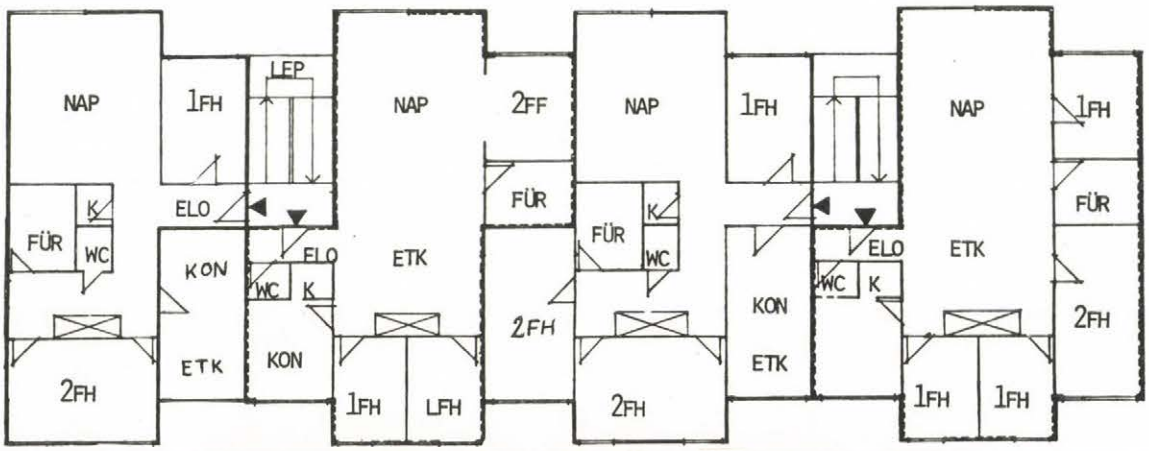
2. SZINT



3. SZINT



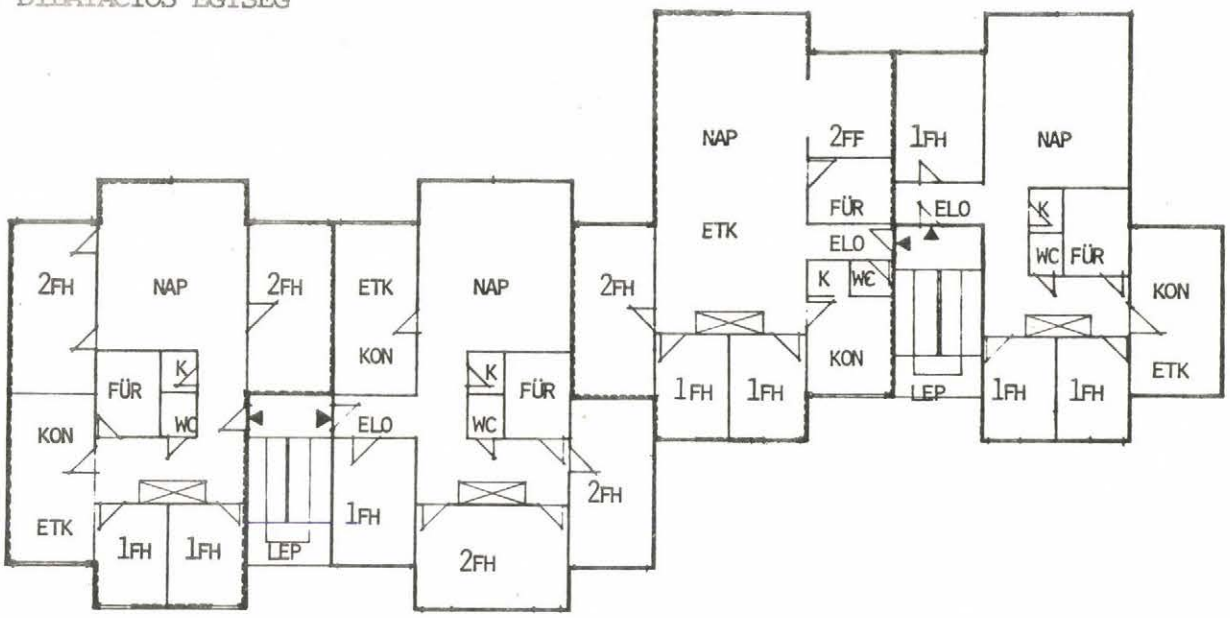
4. SZINT



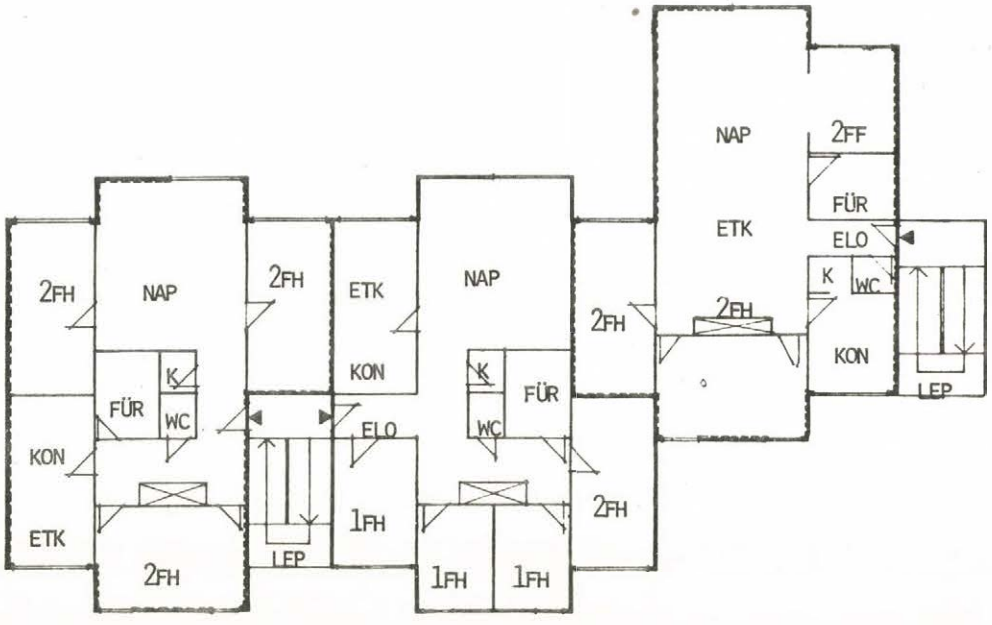
28. ábra

2. DILATÁCIOS EGYSÉG

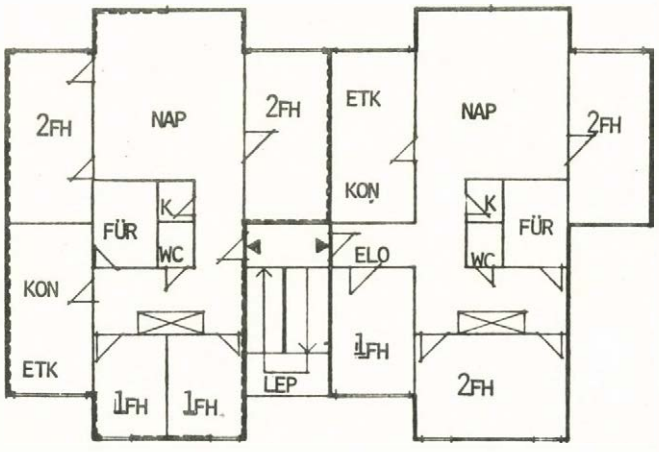
1. SZINT



2. SZINT



3. SZINT



A TOTAL program két futási eredményének rajzi illusztrációit a 28. 29.-es és a 23. ábrán láthatjuk. Az első egy 25 lakásos társasház, a második ugyanazon megrendelői igényeket kielégitő három épület alaprajzi terve, melyek közül egy 9 lakásos és kettő 8 lakásos. A három épület perspektivikus sémája a 23. ábrán látható.

4.4. FUTÁSI ADATOK

Az IBM 3031-es számitógépen a PROLOG rendszer interpreterrel mükődik. Az interpreter kb. 100 Kbyte helyet foglal el. A 4.5 és 6. TÁBLÁZAT-ban a programok futtatásaiból néhány kiragadott példának részletes adatait közöljük.

4. TÁBLÁZAT

FLAT program futási adatai

Feladat	Változatok száma	CPU IDŐ SEC	HIVÁSOK SZÁMA	VISSZA-LÉPÉSEK SZÁMA	MEMÓRIA-IGÉNY Kbyte
3.0.2.0.0.I.I.N.	4	1.937	2333	475	81
2.0.1.0.0.I.I.I.	8	3.849	4604	931	
3.1.2.1.0.I.I.N.	6	3.463	3805	771	71
2.1.1.0.0.I.I.N.	4	2.110	2329	473	
2.0.1.2.0.I.I.I.	12	6.413	6900	1401	72
2.2.1.0.0.I.N.I.	5	2.879	3144	635	
0.3.2.0.0.N.N.N.	2	1.053	1108	227	
0.3.1.0.0.N.N.N.	4	2.033	2161	437	

5. TÁBLÁZAT

ORD program futási adatai

Feladat	CPU IDŐ SEC	HIVÁSOK SZÁMA	VISSZA-LÉPÉSEK SZÁMA	MEMÓRIA-IGÉNY SZÓ
2.3.4.1.NIL	0.177	100	15	88
1.2.4.6.8.7.NIL	0.296	178	17	
1.2.14.13.12.11.10.9.8.7.6.5.4.NIL	0.546	323	29	

F E L A D A T	CPU IDŐ SEC	HIVÁSOK SZÁMA	VISSZA-LÉPÉSEK SZÁMA	MEMÓRIA-IGÉNY Kbyte
25 lakásos épület (TOTAL)	31.188	47502	8506	321
9 lakásos (TASK1)	2.719	3613	299	
8 lakásos (TASK2)	4.010	5826	915	191
8 lakásos (TASK3)	3.176	4438	562	
10 lakásos (TASK1)	2.613	3575	228	
10 lakásos (TASK2) nem tudta megtervezni	50.024	81821	20235	198
9 lakásos (TASK1)	14.062	22496	5127	
16 lakásos (TASK1)	4.247	5905	423	244
9 lakásos (TASK2)	26.525	41656	8083	
6 lakásos (TASK1)	1.364	1735	73	
6 lakásos (TASK2)	1.640	2224	204	164
6 lakásos (TASK3)	115.816	171906	48699	

6. TÁBLÁZAT

BUILD és TOTAL programok futási adatai

7. TÁBLÁZAT

A különböző feladatok épületszerkezeti adatai:

	T1	T2	L1	Q1	L2	Q2	P	Z
TOTAL								
1.D.EGYS.	SZABAD	ZART	L	2	L	2	1	1
2.D.EGYS.	ZÁRT	SZABAD	L	2	L	2	2	1
TASK1	SZABAD	ZART	L	2	L	2	1	1
TASK2	ZÁRT	SZABAD	L	2	L	2	2	1
TASK3	SZABAD	SZABAD	L	2	L	2	3	1

ahol T1 - a dilatációs egység bal oldalának tipusa

T2 - a dilatációs egység jobb oldalának tipusa

L1, L2 - van-e félszinteltolás a lépcsőházi illesztésnél? (L:nincs).

Q1 - az első lépcsőházi illesztésnél a keskeny szeletek száma

Q2 - a második lépcsőházi illesztésnél a keskeny szeletek száma

P - van-e csusztatás a lépcsőház nélküli illesztésnél

1:nincs

2:felfelé

3:lefelé

Z - a lépcsőház nélküli illeszkedésnél a keskeny szeletek száma.

(Megjegyzés: a Q1, Q2 és Z megadásával lehet az épület hosszát szabályozni.)

4.5. A PROGRAMRENDSZER KOMPLEXITÁSA

Az előző fejezetekben vázolt programrendszerbe egy rögzitett épitészeti koncepciót épitettünk be. A lakások geometriai szerkezete, az őket felépitő alapsejtek és az épület általános váza, azok a szükitő feltételek, amelyek megszabják, hogy végül is milyen tipusú épületeket fogunk tervezni. Hogy ezeket az adatokat könnyen tudjuk módositani - azaz más épitészeti koncepciót adaptálni - olyan programtervezési elveket kellett használni, amelyek biztositják, hogy a programok könnyen áttekinthetőek, egyszerűen módosithatóak legyenek. A PROLOG programok szerkezeti bonyolultságával a 3. fejezetben foglalkoztunk. Az ott tárgyalt elmélet azt tüzte ki céljául, hogy a programszerkezetek bonyolultságának csökkentésével elősegitse a szemantikus programhibák csökkentését, a könnyebb módosithatóságot, és a tesztelési idő leröviditését. Éppen ezért alkalmaztuk az ott bevezetett komplexitás-számitási módszereket az előbbiekben tárgyalt programok fejlesztésénél.

A programokat "top-down"' irtuk, és a hierarchikus dekompoziciónál mindig eldöntöttük, hogy a feladat további bontásához milyen tipusú particióra van szükségünk (AND, OR CASE vagy RECURSION). A particiót úgy fogalmaztuk meg, hogy a klózok és argumentumok száma lehetőleg ne haladja meg a négyet, mivel ezzel lehet valószinűsíteni, hogy a partició komplexitása hét alatt marad. A program megirása után, még géprevitel előtt kiszámitottuk a program lokális és globális komplexitás mutatóit. A magas komplexitású particiókat felbontottuk több egyszerü particióra és ennek megfelelően módositottuk a programokat.

8. TÁBLÁZAT

A programrendszer komplexitás és tesztelési adatainak összesitése

	FLAT	BUILD	ORD	TOTAL	ÖSSZESEN
Particiók száma	84	41	7	119	251
Globális komplexitás	544	252	33	806	1636
Átlagos lokális komplexitás	6,5(S)	6,1(S)	4,7(S)	6,8(S)	
Triviális particiók száma és %-a	16 19%	8 19%	1 14%	24 21%	49
Egyszerü particiók száma és %-a	46 55%	20 48%	5 72%	49 41%	120
Komplex particiók száma és %-a	22 26%	13 33%	1 14%	46 38%	82
Nagyon komplex particiók száma és %-a	-	-	-	-	-
Maximális lokális komplexitás	13	17	8	17	
Hierarchikus szintek száma	13	8	4	13	
Program tervezés (ember-nap)	10	3	1	10	24
Program tesztelés (ember-nap)	13	3	1	32	49
Szemantikus hibák száma	12	5	1	24	42

A programrendszer átlagos lokális komplexitása: 6,1

A programok tesztelési adatait, a szemantikus hibák számát és a komplexitási mutatókat a 8. táblázat tartalmazza. A részletes számitás a FÜGGELÉK 2.2 fejezetében található.

Az összesitő táblázat alapján a következő megállapitásokat tehetjük:

Mind a négy program átlagos lokális komplexitása 4 és 7 között mozog, tehát egyszerű. A legkomplexebb program a TOTAL, amelynek átlagos lokális komplexitása 6,8. Ez a program tartalmazza darabszámra is és százalékos összetételben is a legtöbb komplex particiót. A FLAT programmal összehasonlítva a TOTAL program nem egészen másfélszerese az előbbinek (a FLAT program 84 particiót, a TOTAL 119 particiót tartalmaz) , de kétszer annyi komplex particiója van (FLAT:22, TOTAL: 46). Ez is az oka lehet annak, hogy a TOTAL program fejlesztése közben elkövetett szemantikus hibák száma kétszer annyi, mint amennyit a FLAT programban találtunk.

A programok tervezésére és tesztelésére forditott idő figyelemreméltóan alacsony, mindössze 73 ember-nap, nem egészen 4 ember-hónap. Állitjuk, hogy a komplexitás-számitás nagymértékben hozzájárult, hogy ezt a meglehetősen bonyolult problémát valóban rövid idő alatt, kevés szemantikus hibával megirtuk és működőképes állapotba hoztuk.

5. ÖSSZEFOGLALÁS

A disszertációban leírt eredményeket a következő pontokban foglaljuk össze:

1. Megmutattuk, hogyan lehet az épitészeti tervezés eddig nem formalizált szakaszait modellezni és számitógéppel automatizálni.
   Megtaláltuk azokat a logikai eszközöket, és azt a programozási nyelvet, amelyek segítségével a különböző épitészeti koncepciókat le tudjuk irni, össze tudjuk hasonlitani. Arra a konkluzióra jutottunk, hogy az elsőrendű logikán alapuló programozási nyelv szerkezete igen alkalmas a lakások alapsejtjeinek funkcionális kapcsolatai, és az egyes lakások épületen belüli függőségi relációinak reprezentálására.

2. Részletesen tárgyaltuk a leíró jellegű relációs programozás előnyeit egy CAD program bemutatásán keresztül. Arra a meggyőződésre jutottunk, hogy a hagyományos algoritmikus programozási nyelveken nagyon nehézkesen tudtuk volna megoldani a bemutatott műszaki tervezési feladatokat. Ezáltal feltártuk a feladatok egy olyan osztályát, amelyhez adekvát programozási nyelv a PROLOG.

3. A szintaktikus szabályok megszorításával bevezettünk egy új PROLOG programozási módszertant, amelynek célja az egyszerű és áttekinthető programszerkesztés, és a szemantikus hibák csökkentése volt. Egy példán illusztráltuk a módszer használatát, előnyeit és hátrányait.

Kialakitottuk azokat az elveket, amelyek alkalmazása mellett optimálisnak mondható program-variánst kaphatunk.

4. A PROLOG programok szerkezeti komplexitásának feltárására kialakitottunk egy új elméletet. A lokális és globális komplexitás-mutatók bevezetésével olyan számszerű adatokhoz jutottunk, amelyek a programtervezés bonyolultságának mérésével objektiv értékelést adnak a programozási stílusról, a program szerkezetének egyszerűségéről. Az uj program átlagos lokális komplexitásának ismeretében - és az előző tapasztalatok alapján - a programozó meg tudja becsülni a megírt program tesztelési idejét és a szemantikus hibák számát. Ennek nagy jelentősége lehet a határidők kijelölésében.

5. A programok lokális és globális komplexitásának fogalmát ugyan a logikai programozás elveit figyelembevéve definiáltuk, de az elmélet éppugy alkalmazható bármilyen hierarchikus rendszerre, és így jó kiindulási alapot nyujthat más tipusú komplex rendszerek, mint például algoritmikus programozási nyelveken irt CAD rendszerek egyfajta tervezési metodológiájának megalapozásához.

6. A 3. pontban említett programozási módszer számszerű kiértékeléséhez alkalmaztuk a komplexitás-mutatókat. Javaslatot tettünk olyan programozási stílus kialakitására amely optimálisnak tünő, 6-7 körüli átlagos lokális komplexitású programokat eredményez. A kialakított elveket sikeresen alkalmaztuk egy új CAD program-rendszer tervezésénél.

A többszintes lakóépület tervezését segítő programrendszer leirásával a fent említett programozási elvek és módszerek gyakorlati alkalmazását mutattuk be. A programrendszer kialakitása önmagában is jelentős eredmény, hiszen hasonló jellegű feladat megoldását, ilyen automatizálási fokon, eddig sem külföldi, sem hazai rendszereknél nem láthattunk.

A fent emlitett programrendszer a jövőben konkrét gyakorlati felhasználásra is alkalmas lehet. Nagyobb lakótelepek tervezésénél jó segitője és kiegészitője a mérnöki tervező munkának. Előnyei elsősorban a gyors és több lakás és épületváltozat előállitásában lennének, amelyből a tervező az esztétikai és egyéb nem formalizált feltételeket figyelembevéve válogathat. Elképzelhető a programok olyan funkciója is, amely ellenőrzi, hogy az ember által megtervezett épületváltozat kielégiti-e az összes alapvető tervezési követelményt. Ha a program lefut az előre megtervezett adatokra, nagy hiba már nem lehet a tervben, hiszen a program minden egyes lakásra automatikusan végrehajtja az összes ellenőrző eljárást. A programrendszer gyakorlati alkalmazásához elengedhetetlen lenne a rendszer grafikus outputtal való kiegészitése. Rajzoló gép, vagy grafikus display használatával a programok outputja a kivánt pontosságu és részletességű épitészeti alaprajzok lehetnének.

IRODALOMJEGYZÉK

[1] Akin, O.:
How do Architects Design?
Proceedings of AI and PR in Computer Aided Design,
Grenoble, France, 1978.

[2] Battani, G., Meloni, H.:
Interpeteur du language de programation PROLOG.
Groupe de l'Intelligence Artificielle U.E.R.
de Luming Universite d'Aix, Marseille, 1973.

[3] Bobrow, D.G., Raphael, B.:
New Programming Languages for AI Research Techn.
Note 82. AIC SRI 1973.

[4] Campion, D.:
A Computer Aided Draughting/Scheduling System in
a Medium Sized Architectural Practice.
Proceedings of CAD 8o, Brighton, England.

[5] Colmerauer, A.:
PROLOG un language de communication entre homme-
machine;
Groupe de l'Intelligence Artificielle U.E.R.
de Luminy Universite d'Aix, Marseille, 1972.

[6] Curtis, B., Sheppard, S.B., Milliam, P., Borst, M.A.,
and Love, T.:
Measuring the Psychological Complexity of Software
Maintenance Tasks with the Halstead and McCabe Metrics
IEEE Transaction on Software Engineering, Vol. SE-5,
No. 2, March, 1979.

[7] Darvas F., Futó I., Szeredi P.:
Logic Based Program System for Predicting Drug
Interaction
International Journal of Biomedical Computing.

[8] Davies, D.J.M.:
POPLER 1.5. Reference Manuel TPU Raport No.1.
University of Edinburgh 1973.

[9] Derksen, J.A.C., Rulifson, J.F., Waldinger, R.J.:
The QA4 Language Applied to Robot Planning.
AFIPS Press, Montrale New Jersey, 1972.

[1o] Elcock, E.W.:
Problem-solving Compilers.
University of Edinburgh.

[11] Elcock, E.W., Foster, J.M., és mások:
ABSET: A Programming Language Based on Sets:
Motivations and Examples. MI6. Edinburgh University
Press, 1971.

[12] Eastman, C.M.:
The R-presenation of Design Problems and Maintainance
of their Structure
Proceedings of AI and PR in Computer Aided Design,
Grenoble, France, 1978.

[13] Gero, J.S., Volfneuk, M.:
Building Fuzzy CAD Systems
Proceedings of CAD 8o, Brighton, England.

[14] Halstead, M.H.:
Elements of Software Science.
New York: Elsevier, 1977.

[15] Henrion, M.:
Automatic Space-Planning: A
Proceedings of AI and PR in Computer Aided Design,
Grenoble, France, 1978.

[16] Hashimoto, E., Kuriyama, H., Hirayama, Y.:
Automatic Design and Cost Estimation System for
Costom-Made Houses.
Kézirat.

[17] Holnapy D.:
Az automatizált müszaki tervezés matematikai alapjai.
A rendszerépitéssel izomorf algebrai struktura.
ÉTI Tanulmány, 1979.

[18] Kaposi, A.A., Márkusz Z.:
PRIMLOG: A Case for Argumented PROLOG Programming
INFORMATICA 79, Bled, Jugoszlávia,1979.

[19] Kaposi, A.A., Márkusz Z.:
Introduction of a Complexity Measure for Control of
Design Errors in Logic-Based CAD Programs
Proceedings of CAD 8o. Brighton, England

[2o] Kaposi, A.A., Rzevski, G.:
Research into the Assessment of Complexity of Man-Made Systems
Design and Systems Yearbook of Polish Academy of Sciences, 1979, Wroclaw, Poland.

[21] Kowalski, R.:
Predicate Logic as a Programming Language DCL Memo No. 7o, Edinburgh University, 1973.

[22] Kowalski, R., van Emden, M.H.:
The Semantic of Predicate Logic as a Programming Language
DCL Memo MIP-R. lo3 Edinburgh University, 1974.

[23] Köves P.:
BS2000 PROLOG felhasználói kézikönyv
SZKI Tanulmány, 1978

[24] Kulcsár Gy.:
Tervezésautomatizálási módszer a magasépitésben
Kézirat.

[25] Lafue, G.:
A Theorem Prover for Recognizing 2-D Reprezentation of 3-D Objects
Proceedings of AI and PR in Computer Aided Design, France, 1978.

[26] Latombe, J.C.:
Artificial Intelligence in Computer Aided Design: The "TROPIC" System
Proceedings of the IFIP Working Conference on Computer-Aided Design Systems, Austin, Texas, USA, 1976.

[27] Liardet, M., Holmes, C., Rosenthal, D.:
Input to CAD Systems: Two Practical Examples
Proceedings of AI and PR in Computer Aided Design, France, 1978.

[28] Ligget, R.S.:
A Partitioning Approach to Large Floor Lay out Problems
Proceedings of CAD 8o, Brighton, England

[29] Márkusz Z.:
How to Design Variants of Flats Using Programming Language PROLOG, Based on Mathematical Logic.
Proceedings of IFIP'77, Toronto, 1977

[3o] Márkusz Z.:
A PROLOG programozási nyelv alkalmazása paneles lakóépületek épitészeti tervezési feladatainak megoldására
Információ-Elektronika, XII.évf. 3.szám, 1977

[31] Márkusz Z.:
Application of PROLOG in Designing Many-Storied Dwelling Houses
Proceedings of Logic Programming Workshop, Debrecen, 198o

[32] Márkusz Z.:
A PROLOG alkalmazása többszintes lakóépület tervezési rendszeréhez
Információ-Elektronika, XV.évf. 5.szám, 198o

[33] McCabe, T.J.:
A Complexity Measure
IEEE Transaction on Software Engineering, Vol. SE-2, p. 3o8-32o, December, 1976

[34] Mohr, R., Masini, G.:
Drawing Analysis and Computer Aided Design
Proceedings of AI and PR in Computer Aided Design, Grenoble, France, 1978

[35] Myers, G.J.:
An Extension to the Cyclomatic Measure of Program Complexity
SIGPLAN Notices, October, 1977

[36] Mesarovic, M.D., Macko, D., Takahara, Y.:
Theory of Hierarchical Multilevel Systems
Academic Press, New York, 197o

[37] Nuzzolese, V.:
Obtaining Buildings Perspective Views Complete of Materials' Textures
Proceedings of CAD 8o, Brighton, England

[38] Oulsnam, G.:
Cyclomatic Numbers do not Measure Complexity of Unstructured Programs
Information Processing Letters, Vol. 9, No.5. December, 1979

[39] Pereira, L.M.:
Artificial Intelligence Techniques in Automatic Layout Design
Proceedings of AI and PR in Computer Aided Design, Grenoble, France, 1978

[4o] Popovic, L.M.:
An Algorithmic Approach to Large Scale Problem Solving
Ph. D. Thesis, Kingston, England, 1977

[41] Preiss, K.:
Constructing the 3-D Representation of a Plane-Faced Object from a Digitized Engineering Drawing
Proceedings of CAD 8o, Brighton, England

[42] Reboh, R., Sacerdoti, E.:
A Preliminary QLISP Manual
Techn. Note 81. AIC SRI 1973

[43] Sántha E.:
A hazai PROLOG alkalmazások helyzete 1979-ben.
SZKI Tanulmány, Budapest, 1979

[44] Szeredi P., Futó I.:
PROLOG kézikönyv. NIM IGÜSZI Tanulmány, 1977

[45] Warren, D.:
What is PROLOG? Kézirat, 1974

[46] Warren, D.:
Implementing PROLOG - Compiling Predicate Logic Programs
Dept. of AI, Edinburgh University, 1977

[47] Tóth P.:
Az uj elvü programozási nyelvek összenhasonlitó vizsgálata
NIM IGÜSZI Tanulmány, 1974

[48] Yasky, Y.:
Transforming a Set of Building Drawings into a Consistent Database
Proceedings of CAD 8o, Brighton, England

A TANULMÁNYSOROZATBAN 1980-BAN JELENTEK MEG:

101/1980 Gerencsér László - Hangos Katalin: Diszkrét lineáris sztochasztikus rendszerek önhangoló szabályozása

102/1980 Pásztorné Varga Katalin: Rekurziv eljárás

103/1980 Gerencsér Piroska - Szép Endre - Zilahy Ferenc Marton Zsolt: Robotmegfogók adaptivitása I.

104/1980 Knuth Előd - Radó Péter - Tóth Árpád: A SDLA előzetes ismertetése

105/1980 E. Knuth - P. Radó - Á. Tóth: Preliminary description of SDLA

106/1980 Prékopa András: Sztochasztikus programozási modellek és alkalmazásuk

107/1980 Kelle Péter: Megbizhatósági készletmodellek és alkalmazásuk

108/1980 Almásy Gedeon: Mérlegegyenletek és mérési hibák

109/1980 Békéssy A. - Demetrovics J. - Gyepesi Gy.: Relációs adatbázis logikai szintü vizsgálata funkcionális függőségek szempontjából

110/1980 Gaál A. - Soltész J. - Ruda M. - Ratkó I.: Tanulmányok a statisztikai adatfeldolgozásról

111/1980 Benedikt Szvetlána: Nem ismételhető döntéshozatal analizise kockázattal járó esetekben

112/1980 Verebély Pál: Többprocesszoros, osztott intelligenciáju grafikus rendszerek tervezési és megvalósitási kérdései

113/1980 V. Visegrádi Téli Iskola

114/1980 Demetrovics János: Relációs adatmodell logikai és strukturális vizsgálata

115/1980 Gergely József: Program package for sparse matrices

1981-BEN JELENTEK MEG:

116/1981 Siegler András: Egy 6 szabadságfoku antropomorf manipulátor kinematikája és számitógépes vezérlése

117/181 Knuth Előd - Radó Péter: Principles of Computer Aided System Description

118/1981 Demetrovics János - Gyepesi György: Általános függések és lekérdezéssel kapcsolatos algoritmusok relációs adatmodellekben

119/1981 Sztanó Tamás: REAL-TIME programrendszerek eseményvezérelt szervezése

120/1981 Szentgyörgyi Zsuzsa: A számitástechnika müszaki fejlődése és társadalmi hatásai

121/1981 Vicsek Tamásné (Strehó Mária): Vizsgálatok a kezdeti érték problémák numerikus megoldásával kapcsalatban

122/1981 Andó Györgyi - Lipcsey Zsolt: Sztochasztikus Ljapunov módszerek és alkalmazásaik

123/1981 Márkusz Zsuzsanna: Intelligens interaktiv rendszerek elvi problémái