PROGRAMKÉSZÍTÉS ÉS GONDOLKODÁS

MAKING PROGRAM AND THINKING

Szlávi Péter
Eötvös Loránd Tudományegyetem Informatikai Kar

Összefoglaló

A programkészítés során számos *gondolkodási művelet*et hajtunk végre. Ezek felismerése és tudatos alkalmazása nemcsak az éppen készülő program szempontjából bír jelentőséggel, hanem az oktatásban a legfontosabb cél, *a problémamegoldó gondolkodás fejlesztése* szempontjából is. Előadásomban a legnagyobb jelentőségű gondolkodási műveletek közül kiragadok néhányat, és kifejtem ezek miben létét példákkal illusztrálva. Részletezem a következő, programozás közben mozgósított mentális eszközeinket: nyelvi absztrakció, információ-konverzió, dekompozíció. Rövid bemutatásként hadd szóljak e fogalmakról egy-egy mondatban. A *nyelvi absztrakció* kiinduló pontja: egy nyelv elsajátítása nagyfokú absztrakciót kíván, hisz minden nyelv elemei, struktúrái elvonatkoztatások. A *dekompozíció és szuperpozíció* két, antagonista hozzáállás a feladatmegoldás során, amelyek a feladat–részfeladat viszonyának ellentétes irányú motorjai. Az általam *információ-konverzió*nak elnevezett gondolkodási eszköz célja, hogy segítségével a programkészítés adott lépésében a feladat megoldásából addig kinyert információ minél nagyobb részét mentsük át a további lépésekbe, takarékoskodva szellemi energiánkkal; vagyis a már meglévő információkat „konvertáljuk” a következő fázisban alapul szolgáló információkká.

Kulcsszavak

Programozásmódszertan, gondolkodási tevékenység, gondolkodási folyamatok, problémamegoldó gondolkodás, algoritmikus és nyelvi absztrakció

Abstract

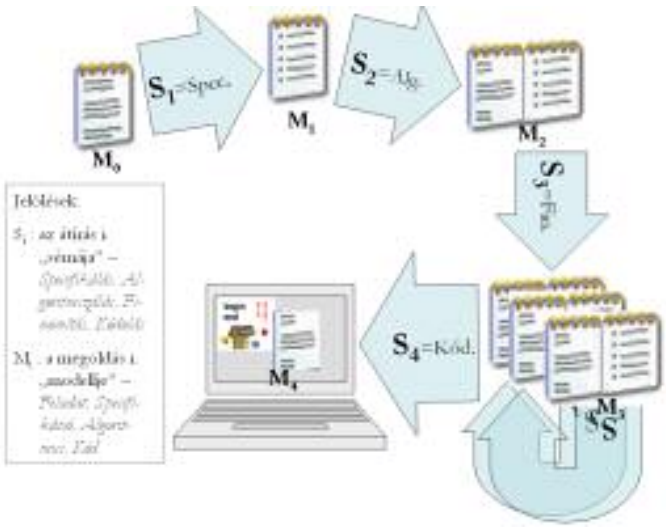
While making a program we perform numerous *cognitive operations*. To recognize and consciously apply them is relevant not only in making that very program but also in developing *problem-solving thinking*, which is the central aim of education. My presentation picks out several of the most important cognitive operations to demonstrate them with explanations and examples. *Language abstraction, information-conversion*, and *decomposition*, as some main mental tools involved in program making, will be discussed. For a brief introduction let me present these terms in one sentence. *Language abstraction* implies that acquiring a language requires a large degree of abstraction, for all linguistic units and structures are themselves abstract. *Decomposition* and *supercomposition* are two antagonistic approaches to solving a task, serving as the opposing motors in the task-subtask relation. The goal of the mental operation I denoted as *information-conversion* is to make it possible that in any stage of the program making process we could bring forward and apply in the next stages the maximum information previously gained, saving our mental energy. In other words, the information we have already gotten will be “converted” into the information serving as the basis of the next phase.

Keywords

Programming methodology, cognitive processes, thought processes, problem-solving, algorithmic abstraction, language abstraction, information-conversion, decomposition programs

1. A programozás gondolkodási eszköztára

Miközben a programozó a feladat megoldásán fáradozik, tudatosan vagy sem, számos *gondolkodási művelet*et használ. Ahogy a feladatból kiindul, többszörösen szelídíti azt a saját, meglévő tapasztalatain alapuló sémáihoz, körkörösen haladva egyre pontosabban fogalmazza újra a feladatot. Úgy is mondhatjuk, hogy ***a programozás a feladat egyre finomabb*** (sémákon alapuló) ***modelljeinek sorozata***, amiben ***addig kell eljutni, ahol*** már a választott ***programozási nyelv szókincse*** (utasítás-sémáinak halmaza) jelenti a modell alapját.



1. ábra: A programkészítés folyamata

> A modellről: a valóság megfogalmazódása a jelölésben = szintaxis, a jelölés értelmezése = szemantika (azaz a valóság tükröződése). A sémáról: modellek közötti átalakítás „kaptafája”, szabálya.

A programozó legfontosabb *gondolkodási műveletei* a finomodó modellek felállításánál:

- a nyelvi absztrakció,
- az analógiás gondolkodás,
- az algoritmikus absztrakció,
- a dekompozíció, szuperpozíció,
- az információ-konverzió,
- az intuíció,
- a variáció.

Az egyes eszközöket sokszor, sokféle céllal veti be a programozó. Az alábbiakban a fentiek közül három, nem közismert gondolkodási műveletet elemzek: a nyelvi absztrakciót, a dekompozíciót és az információ-konverziót.

2. Nyelvi absztrakció

Abból indulok ki, hogy egy ***nyelv*** elsajátítása ***nagyfokú absztrakció***t kíván, hisz minden nyelv elemei, struktúrái elvonatkoztatások. Ezek ***szintaktikus*** és ***szemantikus viszonyai***nak ismerete nélkül reménytelen bármely nyelv helyes alkalmazása.

Az első absztrakciót maga a ***feladatmegfogalmazás*** kívánja (S_1). A feladatmegfogalmazás során először is a feladat szempontjából *lényeges és a lényegtelen dolgok elválasztása* zajlik. Az informálisan megfogalmazott feladatszövegből kihámozandó lényeges mondanivalók a következők: mik a *kiinduló adatok*, melyek a megoldás során *meghatározandók*; milyen *feltételek*et lehet figyelembe venni a kiinduló adatok között, és *hogyan függnek az eredményadatok a kiindulóktól*.

> Ez az információk „merev” felosztása az M_1 modell (a specifikáció) szintaxisának leglátványosabb tulajdonsága. Az információ kinyerése maga az S_1 séma alkalmazása.

Ezt követi az adatok jellemző attribútumainak konkrétumoktól elvonatkoztatása: a tipizálás, azaz a konkrét adatok *értékhalmazokká általánosítása*, majd ezek adatokhoz rendelése. Itt néhány *alaphalmaz* (Szlávi, 2001) közül egy –vagy összetett adat esetén: több– megfelelőt választunk ki, majd –szükség szerint– megengedett *halmazműveletek* felhasználásával hozzuk létre az összetett adat alaphalmazát.

> Alaphalmazok: Z (egészek), R (valósak), K (karakterek), S (szövegek)… Alapvető halmazkonstrukciók: $\cup$ (egyesítés), $\times$ (direktszorzás), * (iterálás), = (definiálás)…

Ezt a lépést azért soroltam a nyelvi absztrakciós lépések közé, mert a programozónak még formalizált eszközök alkalmazása nélkül is, *intuitíve* tisztában kell lennie azzal a „nyelvi kerettel”, amelyben gondolkodnia kell az adatokról. E nyelvi keret minimálisan a következőket jelenti: a *halmaz fogalma*, a *kiindulásul választható halmazok készlete* és *konstrukciója*. E nyelv Chomsky értelembe véve *nyelv*nek tekinthető (Chomsky, 1999/15), amely igen egyszerű grammatikával írható le.

> Chomsky nyelvfogalma: „*... nyelvnek tekintem a mondatok valamely (véges vagy végtelen) halmazát; minden egyes mondat véges hosszúságú, és elemek véges halmazából épül fel. ... valamely formalizált matematikai rendszer „mondatai” is nyelvnek tekinthetők.*”

Megjegyzem, hogy e nyelv „szintaktikájának” ismerete csupán annyira feltétele a nyelv ismeretének, amennyire egy természetes nyelv esetében az írás (a nyelvtan) ismerete.

A következő nyelvi absztrakciót igénylő helyzetbe a ***tervezés*** során (S_2) kerül a programozó. Ekkor a tervezéshez választott *leíróeszköz* ismerete igényli a nyelvi absztrakciós készséget.

Úgy az adatleírásban, mint az algoritmizálásban föl kell ismerni a *jellegzetes és elegendő* struktúrákat, amelyekhez célszerű *egyértelmű*, de kellően *rugalmas* nyelvet kidolgozni. (Tulajdonképpen ugyanezt a célt látjuk megfogalmazódni a programozási nyelvek fejlődésének korai időszakában is, és természetesen a létrejött kevés számú algoritmizáló eszközben is.)

Pár gondolat ezekről a jelzőkről: „*jellegzetes és elegendő*” – természetes módon lehessen vele dolgozni, és minden várható probléma megtervezhető legyen segítségével; „*rugalmas*” – ne arra kelljen fordítani az energia nagy részét, hogy felidézzük azokat a szintaktikai kötöttségeket, amelyeket tiszteletben kell tartani a munka során; „*egyértelmű*” – stabilan azt jelentse egy idő múlva is a papírra vetett gondolat, mint a leírásakor.

Számos ilyen nyelvet dolgoztak ki az idők során. Több szempont figyelembe vételével alakítottuk ki, az ELTE-n, azt a pszeudokódos formalizmust, amellyel az algoritmusainkat készítjük. (Szlávi et al., 2000/6-11)

A programíró a nyelvi absztrakciós képességét mozgósítja, amikor felismeri, hogy pl. ciklust kell a probléma megoldására alkalmazni (valaminek az ismétlését kell megszerveznie), majd a feladat konkrétumai alapján meghatározza a ciklust definiáló „paramétereket" (ciklusfeltételt, ciklusmagot stb.). A nyelv szintaxisát meghatározó szerkezet a tervező eszköz „kötött" elemei alkotják (pl. pszeudókód esetén a kulcs-szavak), amíg a „konkrétumok" a feladat adatain nyugvó résztevékenységek. Nevezhetjük ezt a nyelvi absztrakciós szintet **utasítás-szint**űnek. A jól ismert *finomítások*ban gondolkodás a nyelvi absztrakció azon speciális esete, amely során a programozó az alkalmazott nyelvet bővíti –egyelőre– új alapszavakkal, helyesebben „paraméterezett szerkezetekkel". Itt alkalmazzuk az S_3 sémát. (L. még a Dekompozícióról szóló fejezetet!)

A tervezés magasabb szintjein további nyelvi absztrakciókat alkalmaz a programozó. Gondoljunk például a **programozási tétel**eken nyugvó módszerre! Ezen az absztrakciós szinten a nyelv elemei a tételek konkretizált megtestesülései. Itt tehát nem az „elemi" utasítások, hanem –az ilyenekből felépülő– tételek felismerése jelent kihívást. (A lényeg természetesen nem a tételek elnevezésének ismerete –ez a formalizálásnál szükséges–, hanem létének és szemantikájának ismerete.)

Elsősorban a programozás fejlesztési stratégiájának mássága miatt mondhatjuk, hogy a **moduláris programozás** a nyelvi absztrakció következő szintje. Itt már markánsan válik el a nyelvbővítő rész az alkalmazó résztől. Nem igen képzelhető el formalizálás nélkül, azaz ezen a szinten nem elegendő a nyelvet „csak beszélni", ezt már „írni is tudni kell". A lényeges eltérés az absztrakció fokában az, hogy itt nem (vagy nem akkor) foglalkozunk a nyelvbővítő szókincs kifejtésével, hanem az új „szó" jelentését és alkalmazási formáját készként elfogadjuk, amik azonban időben máskor, vagy más által valósítandók meg.

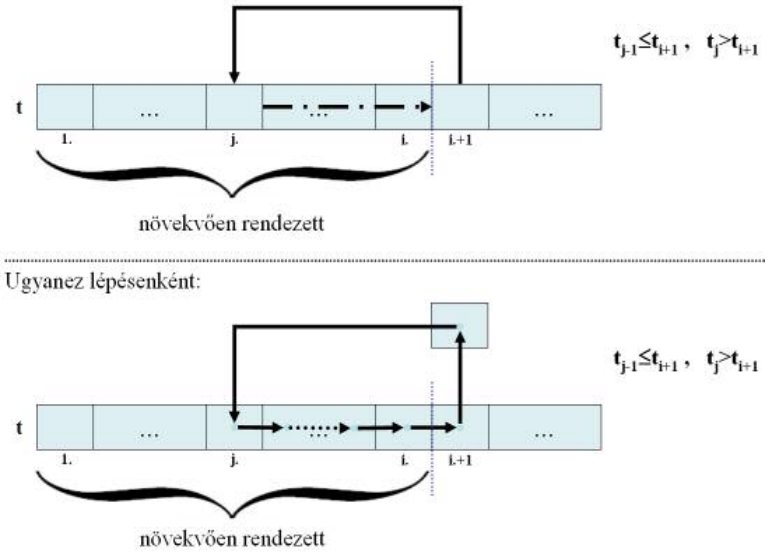
Itt említjük meg azokat a grafikus „praktikákat", grafikus *nyelvi* megoldásokat, amelyekkel a programozó tevékenysége során sokszor él. Nem az algoritmuskészítéshez kitalált számos „rajzos" eszközre, úgymint blokk-diagramra, struktogramra vagy Jackson-féle diagramra gondolok[1], hanem pl. azon ábrákra, amelyekkel a programozó tervezés közben megpróbálja *a transzformáció menetét ábrázolni*, ill. *az adatokat a memóriába képezni*. Egy kellően összetett algoritmus vagy (láncolt) szerkezet esetén kikerülhetetlen ilyen rajzok készítése, amelyeken követni lehet az egyes algoritmusdarabkák működését. De még egyszerűbb feladatok esetén is, annak jobb megértéséhez gyakorta készül grafikus vázlat; mondhatjuk így: a probléma vizualizációja. (L. a 2. ábrát!)

Bár e „rajzos" nyelvek mindenkiben ösztönösen kialakulnak, mégis célszerű a „módszertanával" az oktatásban foglalkozni. Pl. fel kell hívni a figyelmet arra, hogy egy-egy rajz kell tartozzon az összes tipikus állapothoz (kezdőállapothoz, egy „általánoshoz", s más, „szélsőséges" állapotokhoz is). Hasonló a helyzet, mint pl. a matematikaoktatásban a függvénydiszkusszió esetében.

Az informatika szakos hallgatóim „Programozásmódszertan" tárgy első gyakorlatán rendszerint kapnak egy feladatot, amelynek megoldásán (helyesebben: amelyhez való „hozzáállásán") mérem le a hallgatók „hozott" programozási tudását. Azt tapasztaltam –ami némi meglepetést okozott nekem–, hogy a programozásban teljesen járatlanok között is jó néhá-

[1] amelyek éppúgy nyelvek, mint az általunk alkalmazott pszeudokód, mégha a nyelvi *alapelemeik grafikus jelek*, a *grammatikai szabályaik* ezen *elemek* szabályozott *kapcsolatai*nak halmaza.

nyan egy rajzzal próbálták megközelíteni a feladatot. Sőt nem ritkán, azok, akik „rutinból" álltak a megoldásnak, mindenfajta rajzos segítség nélkül, teljesen elvétették a feladatot.



2. ábra: A beillesztéses rendezés egy lépésének rajzos „lenyomata"

Pólya György következő gondolatai könnyen adaptálhatók a megfelelő adatábrázolás megtalálásának programozásbeli problémájára (Pólya, 1977/71):

„*Az ilyen típusú feladat beható elemzését azzal kezdjük, hogy felrajzolunk egy olyan ábrát, amely tartalmazza az ismeretlent és az adatokat, mégpedig olyan elrendezésben, ahogy azt a feladat kikötése előírja. Ahhoz, hogy a feladatot világosan megértsük, minden adatot, a kikötés minden egyes részét külön-külön szemügyre kell vennünk; akkor azután egyetlen képbe egyesítjük az összes részleteket, a kikötést mint egészet vizsgáljuk meg, és megpróbáljuk együtt látni a feladat által előírt különféle összefüggéseket. Mindezeket a részleteket papirosra felrajzolt ábra nélkül aligha tudnánk kézben tartani, szétválasztani és ismét összerakni.*"

Összegezve a fentieket: a tervezés során a programozó a szabványos algoritmus leíró nyelvek mellett ***nem rögzített szintaxisú nyelv***et (a grafikus „kapaszkodókhoz") is alkalmaz.

A következő nyelvi kihívás maga a ***kódolás*** (S_4). Itt komoly absztrakcióra már kevés szükség van akkor, ha a programozó eljut arra a felismerésre, hogy egy jól végiggondolt algoritmus után az adott konkrét programozási nyelvre való átültetés, azaz a ***kódolás*** nagyrészt ***mechanikusan*** elvégezhető.

Ennek feltétele, hogy ki legyenek dolgozva azok a *kódolási szabályok*[2], amelyek többé-kevésbé egyértelmű kapcsolatot létesítenek az algoritmus nyelvi elemei és a programozási nyelv struktúrái között. A „többé-kevésbé" jelzi, hogy mindenképpen jelen van az absztrakció, legfeljebb a foka jellemzi a programozó gyakorlottságát, tudatosságát.

[2] L. Pascal és ELAN programozási nyelvek esetére: (Szlávi et al., 2000/6-11).

3. Dekompozíció

A dekomponálás a komplex probléma elemibb problémák együttesére **bontását** jelenti. Az elemi illetve komplex probléma viszonyát megfordítva sokszor szuperponálásról beszélnek. Ekkor a komplex probléma elemi feladatokból való **felépítésé**re gondolnak. A lényeg ugyanaz: a komplex probléma és elemibbek kapcsolatba hozása.

Az emberi elme jól ismert pszichológiai korlátjáról így ír –szabadon idézve– Mérő László: az emberi agy rövid távú memóriájában egy időben tartható sémák –azaz gondolati struktúrák– maximális száma egyénenként a 7±2 érték körül mozog. (Mérő, 1989/12) Ez egyszerűen szólva azt jelenti, hogy az ember tervezéskor pl. nem igen képes áttekinteni olyan bonyolultságú tervet, amelynek sémaszáma a fenti értéket meghaladja. Ezek szerint a strukturált programozás *felülről lefelé tervezés*, vagy hétköznapi szavakkal mondva: az „*oszd meg és uralkodj*” elve pontosan ezen emberi „gyengeség” beismerését, sőt elkerülésének módját fogalmazza meg, számszerűsítés nélkül. Tehát egy, programozásból kihagyhatatlan elvről van szó.

E gondolkodási művelet lényege, hogy a feladatot egyre finomodó műveletkészlet segítségével fogalmazzuk meg újra meg újra, amíg az algoritmikus nyelvünk eleve meglévő utasításáig el nem jutunk. Minden szint a feladatot teljesen megoldja, de a szint egy-egy összetett sémája (finomítása) csak kb. 5-9 még elemibb mikrosémára (azaz finomításra, vagy már definiált műveletre) bomlik.

Cormen és szerzőtársai könyvükben így közelítik meg a lényeget (Cormen et al, 1997/10): „*Az oszd meg és uralkodj paradigma a rekurzió minden szintjén három lépést vesz igénybe: Felosztja a problémát több alproblémára. Uralkodik az alproblémákon rekurzív módon való megoldásukkal. Ha az alproblémák mérete kicsi, akkor közvetlenül oldja meg az alproblémát. Összevonja az alproblémák megoldásait az eredeti probléma megoldásává.*”

Ehhez két észrevételt fontos hozzátenni. Egyrészt: az alproblémák (vagy a pszichológia szerint: sémák) összevonása a megfelelő utasítás-szervezés kiválasztását jelenti. El kell rendezni az alproblémákat a rekurzió érintett szintjén, dönteni kell egymáshoz való „viszonyuk” felöl. Vagyis az alábbi lehetőségek –és esetleges kombinációik– közül választunk:

- *szekvencia* – több alprobléma adott sorrendben kerül egymással mellérendelő viszonyba, amelyek *mindegyike* végrehajtandó;
- *elágazás* – több, önálló *feltételtől függő* alprobléma, amelyek közül az igaz-feltételű hajtandó végre, az elágazás egészéhez képest az említett alproblémák alárendelt viszonyban állnak;
- *ciklus* – egy „*feltételtől függő számszor*” hajtódjék végre a ciklus törzsét alkotó alprobléma, a ciklus egészéhez képest az alprobléma alárendelt viszonyban áll.

Az alá- és mellérendelő viszonynak több szempontból is jelentősége van. Ezek egyike az *algoritmus leírási mód*ját befolyásolja. A másik fontos következménye a *bonyolultság*ot érinti. Míg utasítások mellérendelésével bővítve a programot, annak bonyolultságát (additíve) *lineárisan* növeljük, addig ugyanezen utasításokat alárendelten illesztve a programhoz, pl. egy összetett szerkezetben, (multiplikatíve) *hatványozottan*. Ezt tükrözi az egyik jól ismert bonyolultsági mérték: a *mélységi bonyolultság*, amelynek lényege, hogy a program egyes részstruktúrájához „*azt a kitevőjű kettő-hatványt rendeljük, ahány magasabbrendű struktúra bel-*

sejében van” (Zsakó, 2000/103), s e részstruktúrák bonyolultság összege adja ki a program összbonyolultságát.

A fenti három probléma-összetételi mód közüli választáshoz is van kezdők számára kapaszkodó: a „*struktúra szerinti feldolgozás*” elve (Szlávi, Zsakó, 2000/46). Az elv szoros kapcsolatot állít föl a feldolgozás alatt álló adatszerkezet és a feldolgozást szervező algoritmus-szerkezet között.

1. táblázat: A ’struktúra szerinti feldolgozás’ elve

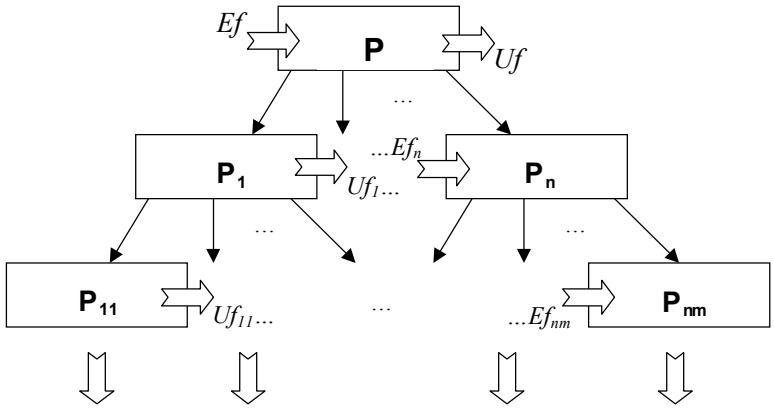
Adatszerkezet	→	Algoritmus-szerkezet
skalár	→	függvény vagy eljárás
direkt-szorzat	→	szekvencia
unió	→	elágazás
sokaság	→	ciklus

Az elv felhasználása abból áll, hogy

1) meghatározzuk a feladat vagy az alfeladat bemeneti és kimeneti adatait, az (Szlávi, Zsakó, 2000) irodalomtól kölcsönzött terminus technicus-szal élve: a feladathoz tartozó bemeneti és kimeneti *szuperstruktúrá*t,
2) egyiket kiválasztjuk „vezérlő” adatszerkezetnek,
3) megfeleltetjük az elv felkínálta adatszerkezetek egyikével, és
4) kiválasztjuk az elv szerint hozzátartozó algoritmus-szerkezetet.

Az elv alkalmazása látszólag csak az algoritmusváz összeállításában segít. A konkrét feladatok esetén meglévő további információ is sokszor beépíthető a hozzárendelt algoritmusba. (Pl. a ciklusfajták közüli választáshoz.) Az elv gyümölcsöző voltát fejtegetjük a már hivatkozott (Szlávi, Zsakó, 2000) irodalom II. fejezetében.

A másik kiegészítésem a Cormen-ék féle körülírásban szereplő rekurzióhoz kapcsolódik. Itt a rekurzió meglehetősen általános értelmű: a probléma-alprobléma analóg voltát jelenti. Ezek szerint viszont, ahogy a problémát fogalmaztuk meg eddig, pontosan úgy kell az alproblémák megfogalmazásával is eljárni. Ha a problémához eddig specifikációt rendeltünk, akkor az alproblémák esetében is ezt kell tennünk. Így tehát a probléma specifikációját is dekomponáltuk alproblémák specifikációinak egymásba fűzött sorozatára. E specifikációknak „hézagmentesen” kell egymáshoz illeszkedniük. (L. 3. ábrát!) Íme egy újabb ok az algoritmikus diszkutációra, a tudatos, megfontolt programozásra, problémamegoldásra.



3. ábra: A felülről lefelé tervezés (Szlávi, 1999/6)

A felülről lefelé tervezés elv fontosságának hangsúlyozása miatt a programkészítési elveink legmagasabb szintjére helyeztük, és *stratégiai* elvként említettük. (Szlávi, Zsakó, 2002/ 93)

A dekompozíciós gondolati eszközhöz továbbiak is kapcsolódnak. Ezeket egyfajta hierarchiába szervezve tárgyaljuk a (Szlávi, Zsakó, 2002) jegyzetünkben, amelyben *programkészítési elv*eknek aposztrofáljuk. Szem előtt tartásuk is nagyban befolyásolják a készülő program „jóságát", születési idejét. Ezek nem mind a programozás kezdetén levőknek jelentenek szempontokat (elsősorban az ergonómiaiakra gondolok), de a gyakorlás során hamar eljutnak a kezdő programozók is arra a szintre, hogy saját bőrükön tapasztalják meg az alábbi elvek célszerű voltát.

2. táblázat: Programozási elvek csoportjai

Elvcsoport	Mire vonatkoznak az elvek?
Taktikai	– a tervezés során határozza meg a továbblépés irányát, módját stb.,
Technológiai	– a tervezés nyelve, ahhoz közelálló problémákra vonatkozó elvek,
Technikai	– kifejezetten a kódolást érintő elvek,
Ergonómiai	– a tervezéssel és/vagy a kódolással kapcsolatos elvek, amelyek azonban magas szinten fogalmazzák meg elsősorban a program és felhasználó közötti kommunikáció elveit.

A fentiek közül a *finomítás*okkal –tehát a dekompozíció gondolkodási műveletével– állnak kapcsolatban a *taktikai* elvek. A finomításokat ugyancsak érintik az *ergonómiai* elvek is. Módszertani szempontból az utóbbiak elsősorban mint típusok specifikációi bírnak jelentőséggel.

4. Információ-konverzió

A programozás egyes lépései között szoros logikai kapcsolat áll fenn. Nyilvánvalóan annyival több energiánk marad a következő modell lényegi aspektusainak kibontására, minél többet tudunk kiaknázni a már kidolgozott eddigi modellekből. Ezt támogatja az ***információ-konverzió***nak nevezett gondolati eszközünk. Lényege, hogy ismerjük föl, melyek azok a *korábbi modellbeli fogalmak*, *objektumok*, amelyek jószerivel *mechanikusan ültethetők át*, konvertálhatók a készülő új modellbe, majd vigyük tovább ezeket.

Természetes törekvés az egyes modellekhez tartozó nyelvek kialakításánál, hogy ezek között minél több kapcsolódási pont legyen, és minél kevesebb olyan részlet, amely félrevezetheti a programépítőt a továbblépésben.

A következő kapcsolatokat vizsgáljuk meg (itt jelezzük az információ továbbvitelére alkalmazható *speciális módszer*t):

1) Specifikáció → adatleírás,
2) Specifikáció → algoritmus – *programozási tételek*, *programtranszformációk*,
3) Adatleírás → a kód adatdeklarációja – *kódtranszformáció*,
4) Algoritmus → kódtörzs – *kódtranszformáció*,
5) Specifikáció + adatleírás → kódtörzs – *kódtranszformáció*.

A *specifikáció*ban szereplő állapottér-leírás (értsd: a bemeneti és kimeneti adatok együttese) kiindulópontja a tervbeli adatleírásnak. A specifikációban az alábbi halmazkonstrukciós műveletekkel szoktunk élni: direktszorzat-, unió-, iterált-képzés, halmaz-átnevezés, új alaphalmaz definiálás. Ezeknek feleltetjük az adatleírásban –elsősorban a típusdefinícióban– a rekord, alternatív rekord (általánosabban itt is uniónak nevezett művelet[3]), sorozat-képzések (számos ilyent ismerünk: tömb, fájl, lista, sor, verem stb.), felsorolás-típus. A tömör 3. táblázattal igyekszem világossá tenni a kapcsolatokat.

3. táblázat: Specifikáció→adatleírás – megfeleltetési példák

Specifikációbeli példa	Adatleírásbeli példapárja
N∈**N** , VAN∈**L**	**Változó** N:Egész; VAN:Logikai
Mag∈**R***	**Konstans** MaxN:Egész(???[4]) **Típus** TMag=**Tömb**(1..MaxN:Valós) **Változó** Mag:TMag
NyilvTart∈(Név×Nem×SzülIdő)*, Név=**S**, Nem=(Ffi,Nő), SzülIdő=Év×Hó×Nap, Év,Hó,Nap=**N**	**Konstans** MaxN:Egész(???) **Típus** TNév=Szöveg, TNem=(Ffi,Nő) TÉv=Egész, THó=Egész, TNap=Egész TSzülIdő=**Rekord**(Év:TÉv,Hó:THó,Nap:TNap) TNyilvTart=**Rekord**(Név:TNév, Nem:TNem, SzülIdő:TSzülIdő) TNyilvTartK=**Tömb**(1..MaxN: TNyilvTart) **Változó** NyilvTart:TNyilvTartK

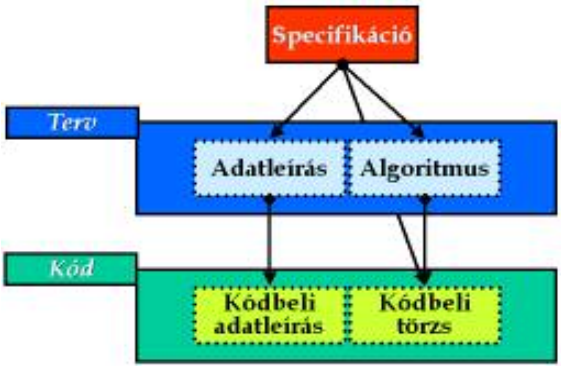
Megjegyzés: azt, hogy szemlátomást alulról felfelé haladva szerepelnek a típusok egymásra épülése, ne tekintsük lényegi vonásnak, hiszen az adatleíró nyelvünk nem olyan szigorú, mint egy konkrét programozási nyelv. (Bár e haladási irány szokásosnak mondható a programozási nyelveknél.) A példában egyébként sugalljuk a módszertani tanácsot, hogy összetett adatok esetén kezdjük a *legegyszerűbben* elintézhetőkkel, majd a *specifikációban* többnyire *visszafelé* haladva, a már definiált típusokkal építkezzünk!

[3] L. az (Szlávi, Zsakó, 2000) a 48., 57. oldaltól kezdődő a példáiban.
[4] Ide természetesen egy konkrét egész szám kerül.

A *terv* és a *kód adatleírás*ának közelsége függvénye a választott programozási nyelvnek. Az általunk a Programozásmódszertan tárgyban „beszélt nyelv” a Pascal. Szerencsére roppant közel állnak egymáshoz, így nagyon egyszerű *kódtranszformáció*s szabályokkal megadható az egymáshoz rendelés. (Szlávi et al., 2000/6-17)

A *tevékenységek terv*beli és *kód*beli leírása közötti áttérésre az előbbiek nagyvonalakban megismételhetők. A Pascal nyelv esetén egyetlen említésre érdemes különbség van: az alapfilozófia következetesen fordított volta. Amíg a tervezés javasolt stratégiája a *felülről lefelé* haladás, addig –pusztán fordítóprogramot egyszerűsítő okok miatt– a Pascal-ban *alulról felfelé* kell építkezni, azaz minden fogalmat előbb definiálni, s csak azután felhasználni. E sarkalatos ponton való eltérést mégsem kell nagy didaktikai problémának venni, hiszen a kódolást a mai programfejlesztő környezetek tetszőleges sorrendben engedik elvégezni, s így már nincs különösebb jelentősége. Nem beszélve arról, hogy a kódoláshoz –ideális esetben– már csak kész algoritmus birtokában kezd neki a programozó. Így a begépelés sorrendjét érinti csupán a Pascal szerencsétlen elvárása.

A *specifikáció* nemcsak az őt közvetlenül követő programozási lépésre hat, hanem a *kód* egyes részeibe is beleszól. Ugyanis a specifikációbeli *előfeltétel* a bemeneti adatokkal szembeni elvárásainkat tartalmazza, amit a program lényegi, transzformációs részének jogában áll elvárnia. Így tehát a kódnak garantálnia kell azt, hogy a bekerült adatok a megengedett tartományban lesznek. Ebből következik, hogy a *beolvasási rész*nek bizony időnként a lényegi részt meghaladó bonyolultságúra kell híznia: tartalmaznia kell minden elvárás ellenőrzését elvégző kódot. (L. 4. ábrát!)



4. ábra: Információ-konverziók

A fentiekből kiviláglik: annak tudatosítása a programíróban, hogy ezek a kapcsolatok léteznek és kihasználandók, programfejlesztés szempontjából igen jelentős. Hiszen: hiba- és programfejlesztési időt csökkentő tényező.

5. Összefoglalás helyett

Az előadásban a legjelentősebb 8 gondolkodási művelet közül 3-mal foglalkoztam. A téma lezárásaként hadd adjam a többiek rövid értékelését! A *szuperpozíció*, amely a dekompozíció egyfajta „ellentéte” legfőképpen a moduláris programozás során jut szerephez. E gondolkodási művelet során a programozó az alapelemekből indul ki, és bonyolultabb manipulációs „eszközöket”, pl. típusokat, típuskonstrukciós eszközöket készít. Így magasabb szintű alapot

teremt a feladatot megoldó program dekomponálással létrejövő finomításához. Tehát a dekompozíció és szuperpozíció valódi kiegészítői egymásnak.

Az *algoritmikus absztrakció* legelemibb szintjén a feladat részfeladatokra bontása során keletkező eljárásokat, függvényeket szokták említeni. Ezek olyan absztrakt tevékenységek hivatkozásai, amelyek pontos tartalma a tervezés adott szintjén még nem ismertek, csak elvont „tevékenység-elvárások". Ezen az absztrakción lépünk túl, amikor programozási tételeket alkalmazunk. A programozási tételek –a formalizmustól eltekintve– nem mások, mint rokon feladatokból általánosított feladat és a konkrétumoktól megcsupaszított megoldásuk kettőse.

Az *analógiás gondolkodás* során a feladatmegoldó tapasztalatában létező feladatok között keres hasonlót, ami közben az eredeti és a hasonlónak vélt kapcsolatait deríti föl. Ha megfelelőnek ítéli a hasonlóságot, akkor az ismert feladat megoldását idézi föl, hogy azután a feltárt kapcsolatokra építve megalkossa a vizsgált feladat megoldását. E gondolatmenet során absztrahálódtak a legfőbb vezérlési szerkezetek, sőt a programozási tételek, ill. az adatok birodalmában a típus, később az objektum-osztály fogalma. Tehát az analógiás gondolkodásra az elvont gondolkodás motorjaként is tekinthetünk.

Amikor algoritmikus megoldást keresünk egy feladatra, akkor kimondatlanul számos megoldás-variációból választjuk ki a legmegfelelőbbnek tűnőt. Azaz a *variációk készítése* tudat alatt is hozzájárul a legéletrevalóbb megoldás megtalálásához. E művelet tudatossá tételével javítható úgy a gondolkodás pallérozottsága, mint a megoldás jósága (Zsakó, 2000).

Az *intuíció* legfőbb sajátossága a megfoghatatlansága, a „józanészből" való levezethetetlensége. Az intuíció működését persze lehet, sőt szokás is jól ismert példákkal bemutatni. Éppen a szokásra való utalással egyetlen hivatkozással zárom ezt a passzust: (Szlávi, 2004/27).

Irodalomjegyzék

[1] Chomsky,N. (1999) Mondattani szerkezetek – Nyelv és elme. Osiris Kiadó.

[2] Cormen,T., Leiserson,Ch., Rivest,R. (1997) Algoritmusok. Műszaki Könyvkiadó.

[3] Mérő L. (1989) A mesterséges intelligencia és a kognitív pszichológia kapcsolata. Tankönyvkiadó.

[4] Pap G.né, Szlávi P., Zsakó L. (2000) Módszeres programozás – Szövegfeldolgozás. Mikrológia, 14, ELTE IK.

[5] Pólya Gy. (1977) A gondolkodás iskolája. Gondolat Kiadó.

[6] Szlávi P. (1999) Programok, programspecifikációk. Informatika a felsőoktatásban'99 Konferencia, Debrecen, 576-582.

[7] Szlávi P. (2001) Programozási tételek – összefoglaló. Elektronikus kézirat, http://digo.inf.elte.hu/~szlavi/ProgModsz/Prtetel.pdf.

[8] Szlávi P., Temesvári T., Zsakó L. (2000) Módszeres programozás – Programkészítés technológiája. Mikrológia, 21, ELTE IK.

[9] Szlávi P., Zsakó L. (2000) Módszeres programozás – Adatfeldolgozás. Mikrológia, 12, ELTE IK.

[10] Szlávi P., Zsakó L. (2002) Módszeres programozás – Programozási bevezető. Mikrológia, 18, ELTE IK.

[11] Szlávi P. (2004) A programkészítés didaktikai kérdései – Intuíció. Doktori értekezés.

[12] Zsakó L. (2000) Módszeres programozás – Hatékonyág. Mikrológia, 6, ELTE IK.