Greşeli frecvente în predarea programării în liceu

M. Frenţiu

(Didactica Matematicii, Turda, 14 mai 2005)

1. Introducere

De peste 25 de ani predau studenţilor din anul întâi primul curs de programare. La primele ore de seminar, în orele de laborator şi apoi la examenele de sfârşit de semestru am observat deprinderile greşite cu care vin unii absolvenţi de liceu. Am avut destule exemple de elevi (consideraţi) foarte buni în liceu care nu au reuşit să promoveze examenul din prima încercare.

Analizând însă manualele şcolare ne putem da seama că nu numai elevii sunt vinovaţi pentru deprinderile greşite cu care vin din liceu. Aceste manuale sunt pline de greşeli, sunt scrise uneori de profesori care nu ştiu ce este programarea, iar programa şcolară este cam aceeaşi cu cea concepută cu peste 30 de ani în urmă. In aceşti ani situaţia s-a schimbat radical, concluzia că simplitatea şi claritatea sunt atribute foarte importante în programare este general acceptată, iar necesitatea respectării unor reguli este consecinţa ultimelor decenii de programare.

Personal mi-am pus de mai multe ori întrebarea dacă e bine ca elevii să înveţe programarea încă din prima clasa de liceu. Intrebarea rămâne, pentru că deprinderile greşite se înlătură greu, de cele mai multe ori însoţesc persoana în cauză în întreaga sa activitate. Cred că e bine ca elevii să ştie programa, dar este necesară înlăturarea multor neajunsuri din programa şcolară, din manualele şcolare şi din activitatea la clasă.

Vom încerca în cele ce urmează să prezentăm acele aspecte pe care le considerăm negative în procesul de învăţare a programării de către elevi. Aceste aspecte vor fi şi titlurile secţiunilor care urmează. Ele sunt:

- rolul proiectării în programare;
- importanta conceptului de subalgoritm;
- simplitate versus complexitate;
- greşeli grave în manualele şcolare răsfoite.

2. Rolul proiectării în programare

Una din cele mai rele deprinderi cu care vin elevii din liceu este fuga la calculator pentru a scrie programul pentru rezolvarea problemei primite. Nu-şi pun întrebarea dacă cunosc bine şi ştiu rezolva problema, dacă metoda aleasă este cea mai potrivită, dacă algoritmul e corect. De fapt nu au algoritm ci scriu direct în limbajul de programare instrucţiuni care, speră ei, vor forma programul dorit. Puţină atenţie se dă în liceu proiectării programelor.

Se ştie că majoritatea programelor realizate azi pe plan mondial conţin erori la predarea lor la beneficiar [Bro87, Fre05], iar majoritatea erorilor provin dintr-o proiectare greşită. Iar una dintre regulile importante de programare spune [Led70, Ker74, Gri90]:

"Gândeşte mai întâi, programează pe urmă"

Ea presupune definirea completă a problemei, deci precizarea datelor de intrare şi a rezultatelor dorite precum şi a condiţiilor pe care trebuie să le satisfacă aceste date, apoi alegerea metodei de rezolvare şi scrierea algoritmului corespunzător şi abia apoi traducerea algoritmului în limbajul de programare, adică programarea propriu-zisă. Pentru că altfel ne vom întâlni cu o altă lege a lui

Murphy: "**The sooner you start coding your program (instead of thinking) the longer it will take to finish it"** [Led70].

Considerăm că un mai mare accent trebuie pus pe proiectarea algoritmilor, independentă de limbajul de programare folosit, pe scrierea lor pe hârtie înainte de a atinge calculatorul. Pentru studenţii anului întâi condiţia de intrare la orele de laborator este redactarea în caietul de laborator a algoritmului pe care-l vor folosi în lucrarea respectivă.

Evident, pentru scrierea algoritmilor se va folosi un pseudolimbaj special dedicat descrierii algoritmilor. La începuturile programării se foloseau în acest scop schemele logice, utile pentru începători, dar neadecvate programării structurate şi conceperii unor programme reale, mult mai complexe decât exemplele didactice simple. El nu este recunoscut de calculator, are propoziţii asemănătoare celor ale limbajului natural şi trebuie să poată reda structurile de calcul necesare rezolvării problemelor. E cunoscut sub numele de Pseudocod şi există mai multe variante, care pot diferi de la programator la programator. Oricare ar fi varianta aleasă ea trebuie să conţină un minim de propoziţii care corespund structurilor de calcul strict necesare şi care să respecte metodologia programării.

În [Păt98] există un limbaj Pseudocod gândit de autor, dar care nu respectă cerinţele de mai sus. Si alte manuale au un astfel de Pseudocod. Limbajul Pseudocod nu e al meu sau al unui autor oarecare. El este consecinţa unei evoluţii istorice, a unei experienţe şi a unor rezultate obţinute în aceşti ani. Manulalul menţionat chiar vorbeşte de programarea structurată şi de teorema Bohm-Jacopini conform căreia ştim că trei structuri de calcul sunt suficiente pentru a scrie un algoritm compus numai din aceste structuri şi echivalent cu un algoritm dat. Limbajul Pseudocod este consecinţa acestei evoluţii şi trebuie să respecte regulile de programare general recunoscute [Gri90, Led70, Ker74, Fre05].

Subliniez de la început că Pseudocodul este un limbaj cât mai simplu, apropiat limbajului natural, dar fără ambiguităţi şi expresii inutile. Voi sublinia importanţa clarităţii şi simplităţii în secţiunea 4. Un algoritm trebuie să fie cât mai clar şi concis, fără litere şi cuvinte în plus. Regulile programării structurate cer folosirea celor trei structuri de calcul deja menţionate, dar şi prezenţa structurii în textul scris, pentru asigurarea clarităţii. Trebuie "să sară în ochi" începutul şi sfârşitul oricărei propoziţii, prin scriere să asigurăm această claritate.

De aceea, consider că un minim de propoziţii care definesc cel mai redus Pseudocod, trebuie să conţină următoarele propoziţii:

CITESTE listaDeVariabile
TIPARESTE listaDeVariabileSiMesaje
[FIE] variabila := expresie
DACA conditie ATUNCI s1 [ALTFEL s2] SFDACA
CATTIMP conditie EXECUTA s1 SFCAT
SUBALGORITMUL nume(l.p.f.) ESTE: s1 SF-nume
CHEAMA nume(l.p.a.)

unde l.p.f. reprezintă lista parametrilor formali iar l.p.a. reprezintă lista parametrilor actuali. Ultimele două propoziţii sunt strâns legate de folosirea subalgoritmilor, care va fi discutată în secţiunea 3.

Aici am scris cuvintele fixe cu litere mari, dar în textul algoritmilor ar trebui să scriem după regulile limbii române, prima literă are rolul de a sublinia începutul unei propoziţii iar punctul de a marca sfârşitul ei. Doar că în programare punctul are un alt rol, motiv pentru care trebuie evitată folosirea lui în alte scopuri, iar pentru marcarea sfârşitului propoziţiilor există alte convenţii. Astfel, cuvântul *Câttimp* (unul singur, nu două cuvinte) este şi numele structurii repetitive, special ales în acest scop, iar *sfcât* e un cuvânt ce marchează sfârşitul acestei propoziţii

şi înlocuieşte punctul din limbajul natural. Nu este nevoie de nici un alt cuvânt pentru a marca începutul secvenţei *s1*, secvenţă care începe după cuvântul *execută*. Este nevoie de un astfel de marcaj de sfârşit întrucât secvenţa *s1* este o secvenţă de oricâte propoziţii Pseudocod, separate prin caracterul ";". Un rol asemănător îl are cuvântul *sfdacă*, care marchează sfârşitul secvenţei *s2* (respectiv *s1* în cazul în care lipseşte alternativa ALTFEL), secvenţe ce pot conţine oricâte propoziţii.

Evident, limbajul Pseudocod poate conţine şi alte propoziţii (în mod deosebit menţionez propoziţiile REPETĂ ... , respectiv PENTRU ...).

Unele manuale folosesc un Pseudocod cu cuvinte din limba engleză [Nic01]. Considerăm nepotrivită o astfel de alegere pentru că proiectarea trebuie făcută independent de limbajul de programare. Cuvintele respective apropie Pseudocodul ales de limbajul de programare, în loc să accentueze această independenţă.

Un rol important în programare îl are specificarea problemei ce trebuie rezolvată. Specificarea este prima etapă, ce precede proiectarea. Din păcate ea lipseşte din aproape toate manualele şcolare, Excepţie face [Ran03]. Specificarea unei probleme P, deci şi a algoritmului A corespunzător, trebuie să precizeze ce se cunoaşte în problemă şi ce se doreşte să se obţină. De asemenea, trebuie să reflecte tot ce se ştie despre aceste elemente. Ea poate fi făcută prin două propoziţii foarte simple, sub forma:

DATE X
REZULTATE Z

unde X este lista variabilelor de intrare (care marchează datele cunoscute în problemă), iar Z este lista variabilelor care marchează rezultatele dorite (variabilele de ieşire). In plus trebuie spus tot ce se ştie despre variabilele de intrare sub forma unei precondiţii satisfăcute de aceste variabile şi ce legătură există între rezultate şi variabilele de intrare, sub forma unei postcondiţii. Precondiţia şi postcondiţia se vor scrie cât mai simplu, folosind notaţiile cunoscute în matematică şi propoziţii în limba română. Scrierea specificaţiei asigură cunoaşterea problemei şi nu este banală. Cerând studenţilor la începutul anului întâi să scrie specificaţia problemei rezolvării ecuaţiei de gradul doi rareori am obţinut un răspuns corect, Absolvenţii de liceu nu ştiu exact ce rezultate se cer în această problemă (ei spun că rădăcinile x1 şi x2 sunt rezultatele, omiţând o a treia variabilă care reţine dacă au existat rădăcini reale sau nu). La o problemă mai dificilă scrierea specificaţiei este o activitate serioasă.

Ca exemplu la cele de mai sus vom scrie un subalgoritm pentru ridicarea la putere prin înmulţiri repetate. Specificarea acestuia este :

Date a,b ; Precondiţia : $a,b \in N, a \neq 0$
Rezultate p ; Postcondiţia : $p = a^b$

Pentru rezolvare pornim de la condiţia
I ::= $p*u^v = a^b$
pe care o vom menţine adevărată pe timpul execuţiei subalgoritmului. Iniţial ea devine adevărată dacă p=1, u=a şi v=b. Execuţia trebuie să se termine când v=0 deci I devine $p = a^b$. Subalgoritmul este dat în continuare şi el foloseşte atribuirea multiplă, (x,y) :=(f,g) însemnând că se calculează expresiile f şi g şi simultan variabilele x şi y primesc valorile celor două expresii.

Subalgoritmul Putere(a,b,p) **este** : { $p := a^b$ }
Fie (p,u,v) := (1,a,b) ; { $p*u^v = a^b$ este adevărat}

```
    Câttimp v>0 execută
        Dacă v mod 2 = 0
            atunci (u,v) := (u*u, v div 2)      {cazul v e par}
            altfel  (p,v) := (p*u, v-1)         {cazul v e impar}
        sfdacă                          { pe ambele ramuri I rămâne adevărată}
    sfcât                               {aici v este 0 deci  :  p = a^b}
sf-Putere
```

3. Importanţa conceptului de subalgoritm

Deşi folosirea subalgoritmilor/subprogramelor este extrem de importantă în programare, prezenţa subalgoritmilor în procesul de învăţământ liceal lasă foarte mult de dorit. Absolvenţii claselor de informatică vin la facultate fără deprinderi de a folosi subalgoritmi în programare şi acceptă foarte greu să-şi schimbe stilul de programare dobândit în liceu. Am prezentat acest aspect în două articole tipărite în Gazeta de Informatică [Fre00, Fre02], în care arătam greşelile frecvente făcute de elevi la examenul de admitere în facultate.

Din păcate nici editorul acestei Gazete nu cunoaşte acest concept, întrucât şi-a permis să schimbe în titlul articolului cuvântul subalgoritm în algoritm şi să intervină în textul articolului (modificând Pseudocodul folosit !).

Când vorbim de un algoritm ne gândim şi la o problemă pe care o rezolvă acest algoritm. Iar când definim un **subalgoritm** trebuie să explicăm cele două părţi ale acestui cuvânt: **sub** şi **algoritm**. În primul rând el este un algoritm pentru rezolvarea unei probleme P. Numai că această problemă este o parte dintr-o problemă mai complexă C. Iar pentru algoritmul de rezolvare a problemei C, algoritmul de rezolvare a problemei P este un subalgoritm. De aici provine şi diferenţa dintre algoritm şi subalgoritm. În orice problemă există date cunoscute şi rezultate dorite. Algoritmul problemei C trebuie să citească datele cunoscute în problema C şi trebuie să tipărească rezultatele obţinute. Pentru subproblema P datele presupuse cunoscute sunt de multe ori rezultate intermediare obţinute de algoritmul problemei C şi nu sunt cunoscute de către programator, deci nu pot fi citite! De asemenea, rezultatele obţinute de subalgoritmul problemei P nu sunt rezultatele problemei C care ne interesează, ele pot fi rezultate intermediare pentru continuarea rezolvării problemei C. Deci nu trebuie tipărite. De aici rezultă şi semnificaţia listei parametrilor formali din definiţia unui subalgoritm şi, corespunzător, a listei parametrilor actuali. Este obligatoriu ca lista parametrilor formali să conţină variabilele care marchează datele presupuse cunoscute în subproblemă şi variabilele care marchează rezultatele obţinute, deci toate variabilele care intervin în specificarea subalgoritmului. Este sarcina algoritmului care apelează subalgoritmul să decidă ce face cu aceste date şi ce rezultate tipăreşte.

La examenele de admitere aproape toţi candidaţii care au ştiut scrie subalgoritmii ceruţi au citit şi tipărit în subalgoritmi şi aproape toţi au avut subalgoritmi fără listă de parametri formali, variabilele globale fiind singura comunicaţie între modulele programului. Este una din deprinderile grave cu care vin marea majoritate a elevilor de liceu.

Cele scrise mai sus provin şi din scopul pentru care s-a inventat conceptul de subalgoritm. Nu este aici spaţiul suficient să arătăm importanţa subalgoritmilor în programare. Ne limităm să punctăm doar câteva avantaje:

- Reutilizabilitate. Subalgoritmii se pot refolosi ori de câte ori apare subproblema respectivă în alte probleme;
- Mărirea productivităţii în programare şi prin refolosirea subalgoritmilor existenţi şi prin
- Simplificarea activităţii de programare. Fiind doar o parte din problema de rezolvat este mult mai uşor şi mai sigur să proiectăm subalgoritmii respectivi;
- Reducerea complexităţii programului. Activitatea de programare este o activitate intelectuală dificilă [Bro87], cu un efort deosebit necesar pentru testarea programelor şi depanarea lor. Descompunerea problemei complexe în părţi mai mici, cât mai simple, uşurează rezolvarea ei, permite conceperea unui algoritm verificabil, compus din subalgoritmi cât mai simpli [Flo67, Fre01a, Gri81, Sch90]. Unele tratate sugerează o margine superioară pentru lungimea unui subalgoritm, ca fiind 50 de propoziţii. Nu există o explicaţie pentru această limită, dar există o demonstraţie pentru ca nici un subalgoritm să nu conţină mai mult de 20 de condiţii (foarte multe) altfel el nu mai poate fi testat de om!!! Se recomandă ca în orice subalgoritm să existe cât mai puţine condiţii, zece sugerând că testarea subalgoritmului poate cere 2^{10} execuţii! Cum s-ar testa un algoritm care nu apelează subalgoritmi şi conţine 50 de condiţii? Sunt necesare până la 2^{50} execuţii!
- Apare posibilitatea ca în rezolvarea unei probleme să lucreze mai mulţi programatori, fiecare dintre ei scriind câteva proceduri. Fără subalgoritmi programarea modulară nu există!

4. Simplitate versus complexitate

Chiar şi pentru autorul unui program, înţelegerea acestui program după un an sau doar câteva luni de la scriere poate fi dificilă. Intrucât reutilizarea sau întreţinerea cer înţelegerea unor proceduri scrise anterior este important să uşurăm această înţelegere prin orice mijloace. Iar azi, când se concep programe uriaşe, realizate de echipe compuse din zeci de programatori, deci multe persoane trebuie să descifreze texte scrise de alţii, când echipele de inspectare [Fre04] fac zilnic aşa ceva, claritatea textelor sursă este foarte importantă.

De aceea, prin orice mijloace, trebuie să asigurăm această claritate. Iar una din aceste posibilităţi este folosirea comentariilor în textul sursă. Din păcate puţine manuale fac acest lucru. Chiar şi manualele bune conţin programe fără nici un comentariu [Nic01]. E uşor de înţeles că nu e timp să scriem comentarii pe tablă, dar nu e admisibilă lipsa acestora într-un manual, la care profesorul face apel ca model şi, cel puţin, aminteşte de utilitatea comentariilor.

Claritatea textului şi simplitatea subprogramelor uşurează înţelegerea lor şi sunt atribute de calitate importante. Inspectarea, testarea, depanarea, reutilizarea, sau întreţinerea cer unor persoane să înţeleagă texte sursă scrise de alţi programatori. Această înţelegere depinde în primul rând de modul în care a fost proiectat programul, apoi de claritatea textului şi de simplitatea subalgoritmilor folosiţi. Atât depanarea, cât şi întreţinerea sunt activităţi foarte costisitoare ca preţ, dar şi ca efort intelectual. Artificiile făcute atunci când nu sunt necesare îngreunează fără rost înţelegerea acestor programe, fac dificilă modificarea/adaptarea unor proceduri la alte structuri de date, sau adăugarea unor funcţii noi.

Relaţia dintre complexitate şi simplitate în programare este abordată în foarte multe articole. Menţionez doar câteva [Cap03, McG04, Ven03, XXX05]. Celebrul Meyer spunea [Ven03] :

"The single biggest enemy of reliability and perhaps of software quality in general is complexity."
Trebuie subliniat de la început că există două înţelesuri ale cuvântului complexitate, două tipuri de complexitate. E complexitatea calculului făcut de un algoritm, complexitate prezentă în manualele şcolare şi complexitatea structurală, redată de textul algoritmului, care uşurează sau îngreunează înţelegerea acestui text. Căutând să micşorăm complexitatea calculului facem artificii şi mărim complexitatea structurală. Si de multe ori complexitatea calculului este mai puţin importantă decât complexitatea structurală.

Fiindcă orice program serios este folosit de mai multe persoane şi trebuie citit şi înţeles de aceste persoane. Dar nu am întâlnit în nici un manual menţiuni despre simplitatea textului. Dimpotrivă, se găsesc destule exemple de algoritmi complicaţi, de tendinţa de a complica lucrurile şi chiar conceptul de "stil în programare" e incorect definit [Păt98, pg.13].

5. Greşeli de înlăturat din manualele şcolare răsfoite

Mă voi referi şi la erori în unele manuale şcolare. Nu pentru că aceste manuale ar fi exemple de manuale greşite. In orice carte există greşeli şi cred că în orice manual se găsesc greşeli de tot felul (tehnoredactare, exprimări greşite etc). Si nu am răsfoit decât câteva dintre multele manuale existente.

Nu mă voi referi la erori minore, ci la definiţii şi concepte greşite, la exemple nepotrivite, la încălcarea unor principii general acceptate în ingineria programării. Sunt convins că unele concepte şi reguli nu pot fi explicate elevilor, dar profesorii trebuie să le cunoască şi să le respecte implicit.

Astfel, în [Mat00] se spune "Scopul organizării în subprograme este acela de a apela un subprogram de mai multe ori" !? In consecinţă, dacă nu e nevoie de cel puţin două apeluri nu trebuie să folosim subprograme. Oare autorul ştie ce este programarea top-down, reutilizarea, care sunt atributele de calitate ale unui program? Sigur, ele nu figurează în programa şcolară, dar absolvenţii unei facultăţi trebuie să le ştie şi să le respecte. Şi cu atât mai mult ele trebuie respectate de autorii unor manuale. De altfel, eu consider mai vinovaţi de aceste erori pe referenţii manualelor, care şi-au dat girul lor unor astfel de texte greşite. Şi cred că n-ar fi rău ca aceşti referenţi, pentru a nu mai semna referate fără a verifica manuscrisele, să fie penalizaţi material pentru greşelile grave întâlnite în manualele şcolare girate de ei.

Şi mai puţin se spune despre parametrii unui subprogram: "Există însă şi subprograme fără parametri". Dar cine trebuie să fie parametrii nu se explică nicăieri.

În aproape toate manualele folosirea variabilelor globale este pe prim plan! Iar variabilele globale trebuie evitate în programare [Fre93, Fre00c, Fre01a, Gri81].

Nu se cunoaşte bine conceptul de subalgoritm de toţi cei ce citesc date şi tipăresc rezultate în interiorul subalgoritmilor care nu au acest rol. Cine cunoaşte rolul subalgoritmilor va respecta regula

"Nu citi şi nu tipări într-un subalgoritm, decât dacă specificaţia sa o cere"!

De exemplu, in [Mat00, pag. 23] subprogramul calculMedia conţine instrucţiunea

```
writeln(' M=', M:8:2)
```

In plus, procedura menţionată nu are parametri formali. In tot manualul nu se explică cine trebuie să fie parametrii formali. Se insistă pe prezentarea limbajului Pascal, pe faptul că parametrii formali pot fi parametri valoare sau parametri referinţă, dar nu e clar ce trebuie să conţină lista parametrilor formali.

Astfel, la pagina 30 se scrie o altă procedură “calcul_media(a,b:real)” care nu are parametru formal pentru rezultat. Care e specificarea acestei proceduri, în ce scop se va folosi?

Accentul e pe limbaj nu pe activitatea de programare. De aici deprinderile greşite, chiar manualul punând un mare accent pe variabilele globale. Credem că mai bine fără ele decât cu asemenea deprinderi. Viitorii programatori trebuie să înveţe regulile de programare corecte şi abia după aceea excepţiile posibile.

Consider nepotrivită introducerea conceptului "programarea structurată" în primele ore din clasa a noua. In plus, e dificil a defini ce este programarea structurată. Iar definiţia din [Păt98] care spune că programarea "devine o artă, atunci când se folosesc cele trei elemente de structurare şi se numeşte programare structurată" este departe de a fi o definiţie a acestui concept. Elevii pot învăţa foarte bine programarea structurată fără să ştie că se numeşte astfel dacă cei ce predau o cunosc, dar nu înţeleg nimic atunci când acest concept nu este înteles de profesor. Imi amintesc că în 1977 am citit un material intern ICI în care se dădeau peste 70 de definiţii-caracterizări ale programării structurate [Teo76]. Definiţia ar trebui să reunească cele mai importante caracteristici şi aici aleg un minim fără de care nu putem vorbi de programare structurată [Dah72, Fre94, Văd78]:

- folosirea celor trei structuri de calcul;
- lipsa instrucţiunii GOTO;
- prezenţa structurii în text pentru asigurarea clarităţii;
- practicarea programării modulare şi top-down, bine cunoscute în momentul în care Dahl-Dijkstra-Hoare [Dah72] au introdus programarea structurată. Deci, nu există programare structurată fără folosirea subalgoritmilor.

Deşi programarea structurată o presupune, programarea descendentă (top-down) nu e prezentă în programa sau manualele şcolare. Fără a o prezenta, menţionăm doar că programarea descendentă presupune [Led75]:

- definirea exactă a problemei;
- proiectarea iniţială independentă de calculatorul şi limbajul de programare folosite;
- proiectarea pe nivele (etape). Fiecare etapă se obţine din precedenta prin rafinarea unei părţi din etapa precedentă;
- amânarea detaliilor pentru etapele ulterioare, târzii;
- asigurarea corectitudinii la fiecare etapă;
- rafinarea în paşi succesivi (a subalgoritmilor).

Nu există proiectarea programelor scrise în nici unul din manualele referite! Deşi Tudor [Tud00] prezintă un limbaj Pseudocod, nu foloseşte acest limbaj în capitolele următoare. Cu ce scop a fost prezentat? Descrie însă metoda (ideea) folosită, fără ca textul dat să fie un algoritm Pseudocod [Tud00, pg.201, 202], dar foloseşte exprimarea “Algoritmul este:”, iar ceea ce urmează nu este un algoritm.

Nici conceptul de variabilă nu e suficient de bine explicat. Intrucât el contribuie foarte mult la claritatea textului sursă considerăm că ar trebui mai mult insistat asupra acestui concept. In primul rând că fiecare variabilă e folosită într-un anumit scop, deci are o semnificaţie a sa. Si, ca regulă importantă în programare, nu trebuie să folosim acelaşi nume pentru semnificaţii diferite!!! Ca exemplu, în [Tud00, pg.112-3] se cere câtul şi restul împărţirii întregi a două numere naturale, n1 la n2. Este nevoie de variabile care să marcheze rezultatul şi c este folosit pentru cât, dar nu există un nou nume pentru semnificaţia “rest”, folosind în acest scop pe n1, care are deja o altă semnificaţie. Aparent e un lucru banal pentru unii, dar o greşeală gravă pentru o programare serioasă!

Există şi greşeli de neatenţie, erori corectabile, dar totuşi prezente. Incă odată spun că referenţii ar trebui penalizaţi pentru erorile grave, de conţinut, şi nici un manual nu ar trebui publicat fără cel puţin doi referenţi.

Menţionez câteva erori găsite în manualele referite, convins că orice carte poate conţine erori inerente, dar nu toate erorile sunt la fel de grave. Conceptele nu trebuie să fie greşit definite!

Astfel, în [Păt98, pg.12] se spune că problema "trebuie să fie înţeleasă şi de către calculator", apoi "calculatorul va trebui să cunoască metoda de rezolvare". Oare ce înţelege calculatorul?

Tot în [Păt00a, pg.16], în prima lecţie despre algoritmi se vorbeşte despre complexitate. Elevul nu ştie încă scrie algoritmi pentru rezolvarea unei probleme, dar se consideră important să ştie ce este complexitatea. E definită? Nici un concept n-ar trebui să fie folosit înainte de a fi definit!

Ideea de a compara doi algoritmi între ei este importantă, dar conceptul complexitate e greu de predat elevilor de liceu. Intre doi algoritmi care fac 2n, respectiv 3n operaţii pentru rezolvarea aceleaşi probleme elevii vor înţelege uşor că primul este mai bun. Dar numărul mediu de operaţii executate de un algoritm este greu de evaluat pentru marea majoritate a algoritmilor. Ca exemplu, pentru a calcula numărul mediu de comparaţii făcut de algoritmul Quicksort la nivelul elevilor de clasa a X-a, în [Mat00, pag.203-4] se face apel la cunoştinţe de Analiză matematică (clasa a XI-a).

In [Mat00, pg.186] se dă o funcţie care calculează maximul dintr-un şir de numere, ca exemplu de aplicare a metodei Divide et Impera. E un exemplu nepotrivit, algoritmul dat fiind mai complicat decât algoritmul bazat pe parcurgerea secvenţială a celor n numere. Nu e nevoie să forţăm astfel de exemple în dauna simplităţii.

Un exemplu similar este determinarea celui mai mare divizor comun a n numere [Mat00, pag.208], care forţează din nou folosirea metodei Divide et Impera. De ce trebuie să complicăm algoritmii ?

6. Concluzii

Am prezentat în paginile anterioare câteva păreri personale privind predarea programării în liceu. In mai multe articole [Fre84-Fre05] m-am referit la corectitudinea algoritmilor şi la regulile importante ce trebuie respectate pentru asigurarea ei. Aceste articole se bazează pe o vastă bibliografie care este folosită indirect şi aici, fără a mai fi rescrisă. Chiar dacă nu pot fi predate elevilor, aceste reguli trebuie cunoscute de profesori ! Sigur, nu poate exista o unanimitate în privinţa celor prezentate. Iar o schimbare radicală nu se poate produce fără modificări în programa şcolară şi în pregătirea profesorilor care predau programarea.

In ordinea importanţei aceste modificări ar fi :

- o clară definire şi folosire a conceptului de subalgoritm ;
- proiectarea algoritmilor înaintea scrierii programelor ;
- respectarea unor reguli importante în programare :
  . evitarea folosirii variabilelor globale ;
  . scrierea indentată pentru asigurarea clarităţii;
  . folosirea comentariilor în autodocumentarea programelor;
  . alegerea unui Pseudocod adecvat;
- schimbarea locului subalgoritmilor în programa şcolară. Subalgoritmul trebuie să se predea mult mai devreme decât în actuala programă.

Cred că nu e greu să se cunoască ce este reutilizarea şi elevii ar trebui să înţeleagă acest concept. Elevii trebuie să conceapă orice algoritm cu gândul la reutilizare, deci să-l scrie ca un subalgoritm SP apelabil în orice program, iar algoritmul va avea structura

Citeşte X;
Cheamă SP(X,Z)
Tipăreşte Z

unde X este vectorul de intrare, iar Z este vectorul de ieşire. Este de fapt algoritmul necesar testării subalgoritmului SP. Dacă SP nu e destul de simplu el trebuie ***obligatoriu*** descompus în alţi subalgoritmi!

Bibliografie.

[Bro87] Fred Brooks, No Silver Bullet. Essence and Accidents of Software Engineering, Computer, vol.20(1987), n0.4, p.10-19

[Cap03] Paul Caplan, Complexity is the root of all Evil, Simplicity, 2003, dec.12, jroller.com/page/paulcaplan/

[Dah72] Dahl O.J., E.W.Dijkstra,and C.A.R.Hoare, Structured programming, Academic Press, New York, 1972.

[Fre84] Frenţiu M., On the program correctness, Seminar on Computer Science, preprint 1984, 75-84.

[Fre93] M.Frenţiu, B.Pârv, Programming Proverbs Revisited, Studia Universitatis Babeş-Bolyai, Mathematica, XXXVIII (1993), 3, 49-58.

[Fre94] M.Frenţiu, B.Pârv, Elaborarea Programelor. Metode şi Tehnici Moderne, Promedia, Cluj-Napoca, 1994.

[Fre95] M. Frentiu, Reguli de programare pentru incepatori, in Lucrarile Conferintei "Informatizarea invatamantului", Balti, 4-7 octombrie 1995, pp. .

[Fre97] M.Frenţiu, On program correctness and teaching programming, Computer Science Journal of Moldova, vol.5(1997), no.3, 250-260.

[Fre00a] M.Frenţiu, On programming style – program correctness relation, Studia Univ. Babeş-Bolyai, Seria Informatica, XLV(2000), no.2, 60-66.

[Fre00b] M.Frenţiu, Conceptul de subalgoritm şi importanţa lui, Gazeta de Informatică, vol.10(2000), nr.6.

[Fre00c] M.Frenţiu, I.Lazăr, Bazele programării. Proiectarea algoritmilor, Ed. Univ."Petru-Maior", Tg.-Mureş, 2000, 184 pagini, ISBN 973-8084-06-7

[Fre01a] M.Frentiu, *Verificarea Corectitudinii Programelor*, Ed. Univ."Petru-Maior", Tg.-Mureş, 2001, 116 pagini, ISBN 973-8084-32-6

[Fre01b] M. Frenţiu,and H.F.Pop, A Study of Licence Examination Results Using Fuzzy Clustering Techniques, Research Seminar on Computer Science, 2001, pp. 99-106.

[Fre02a] M. Frenţiu, The Impact of Style on Program Comprehensibility, Proceedings of the Symposium "Zilele Academice Clujene", 2002, pp. 7-12

[Fre00b] M.Frenţiu, Admitere la UBB 2002, Gazeta de Informatică, vol.12(2002), nr.7.

[Fre02c] M. Frenţiu, H.F.Pop, A Study of Dependence of Software Attributes using Data Analysis Techniques**,** Studia Universitatis Babeş-Bolyai, Informatica, 47(2),2002, 53-60.

[Fre03] M. Frenţiu, On programming style, Babes-Bolyai University, Department of Computer Science, http://www.cs.ubbcluj.ro/ mfrentiu/articole/style.html

[Fre04] M. Frenţiu, Program Correctness in Software Engineering Education, Proceedings of the International Conference on Computers and Communications ICCCM4, pp.154-157, Oradea, 2004, may 27-29.

[Fre05] M. Frenţiu, Correctness: a very important factor in programming**,** Studia Universitatis Babeş-Bolyai, Informatica, 50(1),2005, 11-21.

[Gri81] Gries D., The Science of Programming, Springer-Verlag, Berlin, 1981.

[McG04] Gerry McGovern, Achieving greater simplicity involves managing increasing complexity, New Thinking, 2004, nov.22, www.gerrymcgovern.com/nt/class/overload.htm

[Ker74] Briam Kernigham & P.J.Plauger, The Elements of Programming Style, McGraw-Hill Book Company, New York, 1974.

[Led75] Ledgard H.F., Programming Proverbs for Fortran Programers, Hayden Book Company, Inc., New Jersey, 1975.

[Mat00] Mateescu George Daniel, Pavel Florin Morariu, Otilia Sofron, Informatica cls.IX-a, Ed. Petrion 2000

[Nic01] Stelian Niculescu, Vasile Butnaru, Marius Vlad, Informatica. Manual pentru clasa a X-a, Teora, 2001.

[Păt98a] Bogdan Patrut, Manual de Informatica. Algoritmi si limbaje de programare, Ed.Teora, 1998.

[Păt98b] Bogdan Patrut, Manual de Informatica. Programarea calculatoarelor, Ed.Teora, 1998.

[Ran03] Doina Rancea, Informatica pentru liceu şi bacalaureat, Materia de clasa a IX-a, Editura Donaris, Sibiu, 2003.

[Tud00] Tudor Sorin, Informatica. Varianta C++, Manual pentru clasa a IX-a, Editura L&S Infomat.

[Sch90] Schach S.R., Software Engineering, IRWIN, Boston, 1990.

[Teo76] B.Teodorescu, R.Bercaru, Programare structurată. Suport de studio, ICI, Bucureşti, 1976.

[Văd78] I.Văduva, V.Baltac, V.Florescu, I.Floricică, Programare structurată, Ed.Tehnică, Bucureşti, 1978.

[Ven03] The Demand for Software Quality. A Conversation with Bertrand Meyer, Part I by Bill Venners, October 27, 2003.

[Xxx05] *** Performance and Optimization in the "Real World", TopCoder Feature, March 2, 2005, http://www.topcoder.com/

[Wir71] Wirth N., Program development by stepwise refinement, Comm. ACM 14(1971), 4, 221-227.