



Conceptul de ALGORITM - abordare MODERNĂ

Marin Vlada

În cadrul acestui nou articol, dedicat unei abordări moderne a conceptului de algoritm, vom prezenta detalii referitoare la elaborarea și reprezentarea algoritmilor. Vom prezenta modul în care pot fi descriși algoritmi, folosind limbajul pseudocod.

Evoluția sistemelor de calcul (SC) și a limbajelor de programare (LP) a determinat varietate și evoluție în descrierea și elaborarea algoritmilor. De altfel, reprezentarea algoritmilor și limbajele de programare s-au influențat reciproc.

Până la apariția și răspândirea structurilor de control (anii '70), reprezentarea algoritmilor s-a realizat prin limbajul schemelor logice și "prin pași" care necesitau utilizarea instrucțiunii de salt (*go to <etichetă>*).

Sintaxa și semantica limbajelor de programare din acea perioadă se supuneau acestei restricții (apariția și dezvoltarea unui număr mare de limbaje de programare - la un moment dat existau peste 200 de limbaje de programare - se datorează și căutării eliminării salturilor din procesele de calcul).

Cercetările, experiențele și implementările au durat peste 20 de ani. *Algoritmica* (reprezentarea și elaborarea algoritmilor) și *programarea* (elaborarea și implementarea programelor) au fost nevoite să creeze un "front comun" pentru implementarea structurilor de control și construirea primelor limbaje de programare moderne: limbajul *Pascal* (1972, Niklaus Wirth, Universitatea din Zurich) și limbajul *C* (1972, Dennis Ritchie, AT&T Bell Laboratories).

Apariția limbajelor *Pascal* și *C* au impus limbajul pseudocod în domeniul reprezentării și elaborării algoritmilor.

Mai târziu, evoluția rețelelor de calculatoare și dezvoltarea rețelei *Internet*, precum și implementarea unei versiuni perfecționate a limbajului *C* (versiunea *C++*), au condus la apariția și utilizarea pe scară largă a limbajului de programare *Java*.

Limbajul pseudocod

Limbajul pseudocod are o sintaxă și semantică asemănătoare limbajelor de programare moderne, având o anumită flexibilitate în ceea ce privește sintaxa, în ideea că, prin codificarea unui algoritm într-un limbaj de programare, operația să fie cât mai comodă. De aici, iese în evidență dependența reciprocă dintre *algoritmicitate* și *programare*, ambele fiind influențate de sistemele de calcul și de sistemele de operare actuale.

Un algoritm reprezentat în limbajul pseudocod este constituit dintr-o secțiune în care se declară variabilele și tipul de date asociat acestora, precum și definirea procedurilor sau funcțiilor apelate de algoritm, și corpul algoritmului, care este o succesiune finită de *instrucțiuni executabile*:

```
algorithm <nume_algoritm>
  <declarare_variabile> // tipul variabilelor
  <declarare_rutine> // definirea de proceduri / funcții
begin
  <procesul_de_calcul_P> // instrucțiunile algoritmului
end
```

Din definiția dată conceptului de algoritm se poate observa că structura acestuia conține următoarele elemente de bază:

- *date* - variabile și tipuri de date utilizate pentru accesul la memorie și generarea de valori conform calculelor implementate în procesul de calcul prin intermediul instrucțiunilor;



- **expresii** - formule de calcul asemănătoare expresiilor matematice utilizate pentru *calcul aritmetice, logice (booleene)*, operații asupra valorilor de tip *caracter* sau *șir de caractere*;
- **instrucțiuni** - *instrucțiuni sau comenzi executabile* pentru operații de *intrare / ieșire* și operații de prelucrare a datelor din memorie conform procesului de calcul;
- **proceduri / funcții** - *subprocese de calcul* cu o structură asemănătoare unui algoritm care pot fi executate prin așa-numitele *instrucțiuni de apelare*.

Elementele lexicale ale limbajului sunt următoarele:

- **identificatori** - secvențe de caractere folosite pentru definirea *numelor variabilelor* și a *numelor procedurilor și funcțiilor*;
- **expresii** - forme lexicale asemănătoare expresiilor matematice care se construiesc folosind operanzi (constante, nume de variabile, apel de funcții), operatori (operații) și, eventual, paranteze pentru definirea priorităților operațiilor;
- **cuvinte cheie** - cuvinte din limba engleză care identifică un *tip de date* sau descriu o *instrucțiune*.

Date

Eficiența unui algoritm depinde atât de metodele și tehnicile implementate în procesul de calcul, cât și de tipurile de date utilizate. *Complexitatea algoritmilor* implică o complexitate atât a *metodelor utilizate*, cât și a *organizării datelor* în vederea prelucrării lor. De modul în care sunt *structurate datele* depinde eficiența algoritmilor.

O **structură de date** este o colecție de date înzestrată cu o structură care precizează componentele și procedeele de reprezentare, identificare și înregistrare ale acestora.

Limbajele de programare moderne oferă o mare varietate de *tipuri de date*, începutul l-a realizat limbajul *Pascal* care ulterior a implementat și tehnica OOP. Un *tip de date* este sistemul $T = (D, O)$, unde D este *domeniul de valori*, iar O este *mulțimea operatorilor / operațiilor* care acționează asupra valorilor din D .

Tipurile de date - care utilizează static memoria unui sistem de calcul - sunt clasificate astfel:

- **simple / elementare** - *data / informația* apare ca o entitate indivizibilă atât din punct de vedere al valorii, cât și în raport cu *unitatea centrală de prelucrare (UCP)*; o *dată simplă / elementară* este specificată prin: *identificator, atribut* (domeniul de valori, modul de reprezentare în sistemul de calcul, precizia reprezentării), *valori* (enumerare sau indicate printr-o proprietate comună); din punct de vedere al domeniului de valori asociat, se disting următoarele *clase de date simple*: *integer* (numere întregi), *real* sau *float* (numere raționale), *boolean* (valori logice: *true, false*), *char* (caractere ASCII: cifre, litere, semne, simboluri etc.);
- **structurate** - *data / informația* este organizată într-o *structură* descrisă prin componente de același tip sau de tipuri diferite; componentele pot fi de tip *simplu / ele-*

mentar sau pot fi, la rândul lor, structuri; componentele vor fi identificate prin *nume* sau prin *poziția* pe care o ocupă în structură; din punct de vedere al structurii, se disting următoarele *clase de structuri*: *string* (*șir de caractere ASCII*: concatenarea datelor de tip *char*), *array* (*tablouri n-dimensionale*: structură cu componente de același tip care ocupă locații succesive în memoria sistemului de calcul, identificate prin poziție), *record* (*înregistrare*: structură cu componente de diverse tipuri, identificate prin nume); *file* (*fișier*: structură cu componente de tip *record*).

Tipurile de date prezentate mai sus utilizează memoria sistemului de calcul în *mod static*, nu întotdeauna eficient pentru algoritmul în cauză. Pentru utilizarea *dinamică* a memoriei sistemului de calcul, în scopul utilizării ei eficiente, este definită o organizare specială a datelor: *liste liniare* și *liste circulare*.

În același scop unele limbaje de programare moderne oferă tipul *referință (pointer)*. Evident, în spiritul celor de până acum, conceperea și elaborarea unui algoritm trebuie să țină seama în detaliu de limbajul de programare utilizat, doar în final, pentru codificarea algoritmului.

O **listă liniară** este o structură de date omogenă, secvențială, formată din elementele unei mulțimi de date. O listă poate fi *vidă* (nu conține nici un element), sau *plină* (nu mai poate stoca alte elemente). Orice *listă liniară* are două elemente speciale, unul de început (baza listei) și altul de sfârșit (vârful listei).

Operațiile asupra elementelor unei liste sunt exprimate de *extragerea / accesarea* unui element din listă, *inserarea / adăugarea* unui element în listă, precum și *eliminarea / ștergerea* unui element din listă. Aceste operații vor fi implementate de utilizatori prin proceduri / funcții particulare, elaborate special pentru un anumit limbaj de programare (*Pascal, C++* etc.).

Stocarea listelor în memoria unui sistem de calcul se poate realiza în două moduri:

- **secvențial** - stocarea elementelor listei se face în *locații succesive* de memorie conform ordinii elementelor din listă; pentru a putea utiliza lista se reține primul element al listei (adresa de bază) care este baza listei; structura elementară adecvată reprezentării secvențiale a listelor este *tabloul unidimensional*;
- **înlănțuit** (*simplu sau dublu înlănțuit*) - lista înlănțuită este organizată secvențial, elementele fiind *dispersate în memorie*; fiecare element este înlocuit cu o celulă (nod) formată dintr-o *componentă de informație* (corespunzătoare elementului listei) și o *componentă de legătură*; în cazul listei simplu înlănțuite, partea de legătură conține *adresa celulei următorului element* din listă și se va reține adresa de bază a listei, iar ultima celulă va indica o valoare specială (*NIL*) care nu poate desemna o legătură; în cazul listei dublu înlănțuite, partea de legătură are o componentă care indică succesorul și o componentă care indică predecesorul.



Tipurile de date care utilizează dinamic memoria unui sistem de calcul sunt următoarele:

- **stack** (*stiva*) - listă liniară la care inserarea și extragerea elementelor se face prin vârful listei (de exemplu, stiva se utilizează pentru parcurgerea în adâncime (*DF - Depth First*) a arborilor / grafurilor, pentru generarea soluțiilor unei probleme folosind metoda *Backtracking* etc.);
- **queue** (*coada*) - listă liniară la care inserările se fac la baza listei, iar extragerile se fac prin vârful listei (de exemplu, *coada* se utilizează pentru parcurgerea pe lățime (*BF - Breath First*) a arborilor / grafurilor).

Listele liniare pot fi transformate în liste circulare dacă se consideră că *legătura ultimului element* indică adresa bazei listei. Utilizarea înlănțuirii permite o ușoară manipulare a listelor circulare prin considerarea unui singur *nod fictiv*, numit *bază*.

Declararea variabilelor

O **declarare** este constituită dintr-o parte în care se indică *tipul de date* - prin cuvintele cheie corespunzătoare - și o parte care reprezintă o *listă de nume de variabile* - prin identificatori corespunzători:

```
<tip_date> <listă_identificatori>.
```

Un **identificator** (*nume*) de variabilă reprezintă o secvență de caractere *ASCII* (maxim 256 caractere), primul caracter fiind *literă* (pentru unele limbaje de programare poate fi și caracterul "_"), iar celelalte fiind *litere*, *cifre* sau caracterul ".". Evident, lungimea secvenței de caractere trebuie să fie rezonabilă, suficient de mică pentru ca semantic să sugereze un aspect privind funcții, operații, metode etc. implementate în procesul de calcul.

Există libertate și flexibilitate mare în definirea identificatorilor pentru variabilele utilizate în conceperea și reprezentarea unui algoritm.

Exemple

```
integer a, b, c, max_X, x(100);
real x, y, z, m_produ(20,20);
boolean ind_1, ind_2;
char ch1, ch2;
string s1, s2, s3;
stack S;
queue Q1, Q2;
record (integer cod; string nume; real media)
        x, y, z, t;
```

Din punct de vedere sintactic, instrucțiunile sunt delimitate prin caracterul ";" și pot fi scrise pe rânduri separate sau pe același rând.

Proceduri și funcții

Rezolvarea multor probleme complexe necesită executarea repetată a aceluiași calcul / operații pentru date de intrare diferite.

Procedurile și funcțiile (în domeniul limbajelor de programare se numesc *subprograme*) oferă posibilitatea definirii și descrierii acestor calcule o singură dată. Prin intermediul subprogramului aceste calcule se execută prin apelarea lor ori de câte ori este necesar.

Orice subprogram este *identificat* printr-un *nume / identificator de procedură / funcție* și o *listă de parametri (argumente)*.

O procedură poate fi declarată astfel:

```
Procedure <nume> [ (<ListăParamFormali> ) ]
    [<DeclarareVariabile>] // tipul variabilelor locale
    [<DeclarareProceduri>] // definirea de proceduri / funcții

begin
    <ProcesulDeCalcul> // corpul / instrucțiunile procedurii
end
```

Declararea unei funcții:

```
Function <nume> [ (<ListăParamFormali> ) ]:
    <tip_rezultat>
    [<DeclarareVariabile>] // tipul variabilelor locale
    [<DeclarareProceduri>] // definirea de proceduri / funcții

begin
    <ProcesulDeCalcul> // corpul / instrucțiunile funcției
end
```

Un algoritm este complet elaborat dacă este descrisă și definiția procedurilor și funcțiilor folosite. Definiția presupune descrierea instrucțiunilor pentru efectuarea calculelor proprii și modul cum se transmit *rezultatele* între programul principal (*programul apelant*) și *subprogramul apelat*.

În cazul unei proceduri, *<ListăParamFormali>* conține atât *parametri de intrare*, cât și *parametri de ieșire* (în urma apelării procedurii, vor stoca rezultatele calculate de instrucțiunile procedurii).

În cazul unei funcții, *<ListăParamFormali>* conține doar *parametri de intrare*, în urma apelării funcției, rezultatul va fi stocat în identificatorul (*numele*) funcției. De aceea este obligatoriu ca printre instrucțiunile funcției să existe o *instrucțiune de atribuire* în care membrul stâng să fie chiar identificatorul funcției.

Această deosebire dintre procedură și funcție face ca modul de apelare a acestora să fie diferit:

- **apelarea unei proceduri** este realizată prin instrucțiunea specială "*apel de procedură*" care are forma:

```
<nume> [<ListăParamActuali>];
```

în momentul în care este necesară execuția unei instrucțiuni de *apel de procedură*, se realizează o *corespondență* între parametrii formali (*argumente*) și parametrii ac-



tuali și se execută toate instrucțiunile specificate în definiția procedurii; după execuția ultimei instrucțiuni a procedurii se continuă execuția cu instrucțiunea apelului de procedură; în limbajele de programare, dar și în limbajul pseudocod se utilizează instrucțiunea **return** care provoacă părăsirea corpului unui subprogram;

- **apelarea unei funcții** se face prin considerarea ca *operand* într-o expresie a formei:

`<nume> [<ListăParamActuali>];`

în momentul în care se evaluează expresia respectivă, operandul considerat determină o *corespondență* între parametrii formali și parametrii actuali și se lansează în execuție toate instrucțiunile specificate în cadrul definiției funcției.

Instrucțiuni executabile

Limbajul *pseudocod* oferă următoarele categorii de instrucțiuni executabile:

- **simple**
 - ♦ de intrare / ieșire (**read**, **write**);
 - ♦ de atribuire / memorare;
 - ♦ de transfer (**return**, **exit**);
 - ♦ de operare liste (*inserare*, *extragere*);
- **structurate**
 - ♦ *secvențiale* / *secvența* (**begin ... end**);
 - ♦ *decizionale* / *decizia*:
 - *selecția simplă* (**if ... then ... else ...**);
 - *selecția multiplă* (**case**);
 - ♦ *repetitive* / *repetiția*:
 - *ciclu cu test inițial* (**while ... do ...**);
 - *ciclu cu test final* (**repeat ... until ...**);
 - *ciclu cu contor* (**for ... do ...**);
 - ♦ *apel de procedură* / *funcție*.

În continuare, fiecare instrucțiune va fi descrisă și prezentată prin *sintaxa* și *semantica* sa.

Instrucțiuni de intrare / ieșire

Sintaxa *instrucțiunilor de intrare / ieșire* este:

read <ListăVariabile>

write <ListăVariabile>

unde <ListăVariabile> este o listă de nume de variabile de tip elementar.

În continuare este descrisă semantica *instrucțiunilor de intrare / ieșire*. Se presupune că citirea valorilor (datelor) se va face de la un mediu de intrare (de exemplu: *tastatură*, *suport magnetic* etc.), iar scrierea rezultatelor (valori de ieșire) se va face la un mediu de ieșire (de exemplu: *display*, *imprimantă*, *plotter*, *suport magnetic* etc.).

Execuția instrucțiunii **read** determină citirea de pe mediul de intrare a valorilor corespunzătoare tipului de variabile ale căror nume sunt indicate în <ListăVariabile>. Ordinea citirii valorilor este dată de ordinea numerelor de variabile din <ListăVariabile>.

Execuția instrucțiunii **write** determină scrierea pe mediul de ieșire a valorilor corespunzătoare tipului de va-

riabile ale căror nume sunt indicate în <ListăVariabile>. Ordinea scrierii valorilor este dată de ordinea numerelor de variabile din <ListăVariabile>.

Instrucțiunea de atribuire

Pentru instrucțiunea de atribuire / memorare se poate folosi una dintre următoarele două sintaxe:

`<var> ← <expresie>` sau

`(<v1>, <v2>, ..., <vn>) ← (<e1>, <e2>, ..., <en>)`, unde <var>, <v₁>, <v₂>, ..., <v_n> sunt *nume de variabile*, iar <expresie>, <e₁>, <e₂>, ..., <e_n> sunt expresii de același tip (aritmetic, logic, caracter etc.) cu tipul variabilelor corespunzătoare.

Simbolul "←" reprezintă operatorul (operația) de atribuire / memorare. O expresie se construiește folosind operanzi (constante, nume de variabile, apel de funcții), operatori adecvați și eventual paranteze.

Execuția instrucțiunii în prima variantă are ca efect *evaluarea valorii expresiei* din membrul drept, după care în locația corespunzătoare variabilei <var> se memorează valoarea obținută în urma evaluării expresiei <expresie>.

În cazul celei de-a doua variante, execuția instrucțiunii este echivalentă cu execuția secvențială / paralelă a instrucțiunilor de atribuire / memorare:

`<vari> ← <ei>`, unde $i \in \{1, 2, \dots, n\}$.

Instrucțiuni de transfer

Sintaxa instrucțiunii de transfer este:

return [<expresie>] sau

exit

Instrucțiunea **return** poate fi utilizată în cadrul subprogramelor (proceduri și funcții) având rolul în execuție să provoace părăsirea corpului (procesului de calcul) unui subprogram și revenirea în unitatea de program din care a avut loc apelul, și anume la instrucțiunea ce urmează imediat după acest apel, în cazul unei proceduri.

În cazul în care cuvântul **return** este urmat de o expresie, valoarea expresiei obținută prin evaluare este folosită ca valoare de retur a subprogramului.

Instrucțiunea **exit** este utilizată în cazul ieșirii forțate din corpul unei instrucțiuni, și anume, controlul este preluat de instrucțiunea care urmează după instrucțiunea respectivă.

Instrucțiuni de operare cu liste

Sintaxa instrucțiunilor de operare cu liste este următoarea:

`<var> ⇒ <lista>`

`<var> ← <lista>`,

unde <var> este un *nume de variabilă* de același tip cu tipul elementelor listei <lista>, iar <lista> este un *nume de listă*.

Prima instrucțiune reprezintă *instrucțiunea de inserare* a elementului <var> în lista <lista>, iar a doua instrucțiune reprezintă *instrucțiunea de extragere* a elementului curent din lista <lista> și memorarea lui în variabila <var>.



Vom prezenta acum sintaxa și semantica unor instrucțiuni care pot fi utilizate în limbajul pseudocod.

Instrucțiuni secvențiale

Sintaxa instrucțiunilor secvențiale este următoarea:

begin

<instrucțiune₁>

...

<instrucțiune_n>

end

unde <instrucțiune_i>, $i \in \{1, 2, \dots, n\}$ este o instrucțiune executabilă oferită de limbajul pseudocod.

Execuția acestei construcții (secvență de instrucțiuni) înseamnă *execuția secvențială (step by step)* a celor n instrucțiuni, după care controlul procesului de calcul este dat următoarei instrucțiuni de după cuvântul cheie **end**.

Instrucțiuni decizionale

Există două tipuri de instrucțiuni decizionale, și anume, selecția simplă (**if ... then ... else ...**) și selecția multiplă (**case**).

Selecția simplă

Sintaxa instrucțiunii de selecție simplă este următoarea:

if <expresie_bool> **then** <corp₁>

[**else** <corp₂>],

unde <expresie_bool> este o *expresie booleană* (logică), iar <corp₁> și <corp₂> sunt *secvențe de instrucțiuni executabile*.

Inițial se evaluează valoarea logică a expresiei booleene (logică) <expresie_bool>. În cazul în care valoarea obținută este **true**, atunci (**then**) se trece controlul pentru execuția instrucțiunilor din <corp₁>; altfel (**else**) se trece controlul pentru execuția instrucțiunilor din <corp₂>, în situația când această secvență este prezentă.

În cazul în care valoarea expresiei booleene este **false** și secvența de instrucțiuni <corp₂> nu apare, controlul este trecut la următoarea instrucțiune de după **if**, acțiune realizată și după execuția instrucțiunilor din <corp₁>, respectiv <corp₂>.

Selecția multiplă

Sintaxa instrucțiunii de selecție multiplă este următoarea:

case <selector> **of**

C_{11} [, C_{12} , ..., C_{1m}] : <modul₁>

...

[C_{n1} [, C_{n2} , ..., C_{nm}] : <modul_n>

[**else** <modul_{n+1}>]

end,

unde <selector> este o *variabilă sau o expresie simplă* de tip diferit de tipul *real*, care va putea avea ca valori constantele distincte C_{ij} , unde $i \in \{1, 2, \dots, n\}$, $j \in \{1, 2, \dots, m\}$, iar <modul_i>, $i \in \{1, 2, \dots, n, n+1\}$ este o *secvență de instrucțiuni executabile*.

Prin execuția acestei instrucțiuni, dacă <selector> este o expresie simplă, aceasta va fi evaluată obținându-se

o *valoare* care va fi comparată cu constantele C_{ij} , $i \in \{1, 2, \dots, n\}$, $j \in \{1, 2, \dots, m\}$.

În cazul în care <selector> este o variabilă se va considera *valoarea variabilei*. Dacă valoarea este identică unei constante C_{ij} , atunci se trece controlul pentru execuția instrucțiunilor din <modul_i> după care se trece la executarea primei instrucțiuni de după **end**.

În cazul în care valoarea nu este identică nici uneia dintre constantele C_{ij} , $i \in \{1, 2, \dots, n\}$, $j \in \{1, 2, \dots, m\}$, atunci se trece controlul pentru execuția instrucțiunilor din <modul_{n+1}>, după care se trece la executarea primei instrucțiuni de după **end**.

Instrucțiuni repetitive

Vom prezenta în continuare cele trei tipuri de instrucțiuni repetitive care pot fi utilizate: ciclul cu test inițial (**while ... do ...**), ciclul cu test final (**repeat ... until...**) și ciclul cu contor (**for ... do ...**).

Ciclul cu test inițial

Sintaxa ciclului repetitiv cu test inițial este următoarea:

while <expresie_bool> **do** <corp>,

unde <expresie_bool> este o *expresie booleană* (logică), iar <corp> este o *secvență de instrucțiuni executabile*, inclusiv repetitive.

Execuția acestei instrucțiuni determină evaluarea repetitivă (în cadrul unui proces dinamic) a valorii logice a expresiei <expresie_bool>.

La prima evaluare, dacă valoarea obținută este **true**, se trece la executarea instrucțiunilor din secvența <corp> și se continuă cu reevaluarea valorii de adevăr (instrucțiunile din secvența <corp> pot schimba starea unor variabile care apar în <expresie_bool>), altfel, dacă valoarea de adevăr este **false**, nu se mai execută instrucțiunile din secvența <corp>.

În cazul în care instrucțiunile din secvența <corp> s-au executat o dată, se repetă execuția atât timp cât (**while**) valoarea de adevăr a expresiei <expresie_bool> rămâne **true**.

Procesul repetării executării instrucțiunilor din secvența <corp> se încheie în momentul în care valoarea de adevăr a expresiei <expresie_bool> devine **false**.

În concluzie, este posibil ca instrucțiunile din secvența <corp> să nu se execute niciodată în cazul în care la prima evaluare (*test inițial*) valoarea de adevăr a expresiei <expresie_bool> este **false**.

De asemenea, se poate face observația că execuția instrucțiunilor din secvența <corp> este *precedată întotdeauna* de testul privind valoarea de adevăr a expresiei <expresie_bool>.

Ciclul cu test final

Sintaxa ciclului repetitiv cu test final este următoarea:

repeat

<corp>

until <expresie_bool>,



unde `<expresie_bool>` este o expresie booleană (logică), iar `<corp>` este o *secvență de instrucțiuni executabile*, inclusiv repetitive.

Execuția acestei instrucțiuni determină execuția repetitivă (în cadrul unui proces dinamic) a instrucțiunilor din secvența `<corp>` și evaluarea repetitivă a valorii logice a expresiei `<expresie_bool>`.

La început, necondiționat se trece la executarea instrucțiunilor din secvența `<corp>` și apoi se continuă cu evaluarea valorii de adevăr (instrucțiunile din secvența `<corp>` pot schimba starea unor variabile care apar în `<expresie_bool>`).

Dacă valoarea de adevăr a `<expresie_bool>` este true, nu se mai execută instrucțiunile din secvența `<corp>`, altfel se repetă execuția instrucțiunilor din secvența `<corp>` până când (**until**) valoarea de adevăr a expresiei `<expresie_bool>` devine true.

Procesul repetării executării instrucțiunilor din secvența `<corp>` se încheie numai în momentul în care valoarea de adevăr a expresiei `<expresie_bool>` devine true.

În concluzie, întotdeauna instrucțiunile din secvența `<corp>` se execută cel puțin o dată și fiecare execuție a instrucțiunilor din secvența `<corp>` este urmată de evaluarea (*testarea finală*) valorii de adevăr a expresiei `<expresie_bool>`.

De asemenea, se poate face observația că execuția instrucțiunilor din secvența `<corp>` este urmată întotdeauna de testul privind valoarea de adevăr a expresiei `<expresie_bool>`.

Ciclul cu contor

Sintaxa ciclului repetitiv cu contor este următoarea:

```
for <var> = <exp_inf>, <exp_sup> [, <pas>] do  
    <corp>
```

unde `<var>` este o variabilă (numită contorul ciclului) de același tip (excepție tipul *real*) cu tipul expresiilor simple `<exp_inf>`, `<exp_sup>` care reprezintă valori (*marginea inferioară*, respectiv *marginea superioară*) dintr-o mulțime de valori ordonate care pot fi generate prin intermediul valorii `<pas>` (numit *pasul de generare - pasul ciclului*), iar `<corp>` este o *secvență de instrucțiuni executabile*, inclusiv **for**.

Execuția acestei instrucțiuni determină execuția repetitivă (în cadrul unui proces dinamic) a instrucțiunilor din secvența `<corp>` prin *modificarea automată* a valorii contorului, având inițial valoarea dată de *marginea inferioară* `<exp_inf>`, iar apoi valori generate prin acțiunea pasului (prin *incrementare* sau *decrementare*) `<pas>` până când, după un număr finit de pași, va lua valoarea dată de *marginea superioară* `<exp_sup>`.

În momentul când valoarea contorului `<var>` va fi valoarea dată de *marginea superioară* `<exp_sup>`, instrucțiunile din secvența `<corp>` vor fi executate pentru ultima oară, iar controlul se va transfera la următoarea instrucțiune de după instrucțiunea **for**.

Bibliografie

1. Albeanu Gr., *Algoritmi și limbaje de programare*, Universitatea "Spiru Haret", Editura România de Măine, București, 2000
2. Apostol C., Roșca I. Gh., Roșca V., Ghilic-Micu B., *Introducere în programare. Teorie și aplicații*, Editura București, 1993
3. Georgescu H., *Programare concurentă. Teorie și aplicații*, Editura Tehnică, București, 1996
4. Kinston J. H., *Algorithms and Structures Design. Correctness, Analysis*, Addison Wesley, Longman, 1998
5. Mitrană V., *Provocarea algoritmilor. Probleme pentru concursurile de informatică*, Editura Agni, București, 1994
6. Odăgescu I., *Optimizarea algoritmilor*, Editura Militară, București, 1991
7. Perjeriu E., Văduva I., *Îndrumar pentru lucrări de laborator la cursul de Bazele Informaticii*, Universitatea din București, 1986
8. Popovici C., Georgescu H., State L., *Bazele Informaticii*, vol. I, Universitatea din București, 1990
9. Skiena S., *The Algorithm Design Manual*, Springer Verlag, New York, Inc., 1998
10. Vlada M., *Informatica*, Universitatea din București, Editura Ars Docendi, București, 1999
11. Vlada M., *Poligoane stelate. Problema lui Hopf și Pannwitz*, Gazeta de matematică, nr. 8/1995, pag. 339-348
12. Zaharia M. D., *Structuri de date și algoritmi. Exemple în limbajele C și C++*, Editura Albastră, Cluj-Napoca, 2002
13. Cristea V., Giumale C., Kalisz E., Păunoiu Al., *Limbajul C standard*, Editura Teora, București, 1992
14. Popovici M. D., Popovici M. I., *C++. Tehnologia orientată spre obiecte. Aplicații*, Editura Teora, București, 2000
15. www.math.gatech.edu/~thomas/FC/fourcolor.html
16. mathworld.wolfram.com/Four-ColorProblem.html
17. spicerack.sr.unh.edu/~student/tutorial/fourColor/FourColor.html
18. mathcentral.uregina.ca/RR/database/RR.09.97/fisher1.html
19. www.uccs.edu/~asoifer/book5.html
20. www.uccs.edu/~asoifer/solve98.html
21. www-groups.dcs.st-and.ac.uk/~history/BigPictures/Guthrie.jpeg
22. mathworld.wolfram.com/Algorithm.html
23. www.cs.rit.edu/~atk/Java/Sorting/sorting.html
24. faculty.juniata.edu/rhodes/cs2/ch129.htm
25. www.scism.sbk.ac.uk/law/Section5/chap1/s5c1p2.htm
26. www.ics.uci.edu/~eppstein/161/people.html
27. www.cs.pitt.edu/~kirk/algorithmcourses/
28. www.gnacademy.org/text/cc/
29. java.sun.com/docs/books/tutorial/java/concepts/
30. www.accu.org/acornsig/public/articles/oop_c.html
31. loki.cs.brown.edu:8081/webae/full.html

Domnul conf. dr. Marin Vlada este cadru didactic la Universitatea București și poate fi contactat prin e-mail la adresa vlada@chem.unibuc.ro.