Iterativitatea și recursivitatea

Bujor Claudiu, clasa a 11-a „C”

Profesor: Maria Guțu

2.4.2020

Cuprins:

1 Introducere

Pe parcursul dezvoltării informaticii s-a stabilit că multe probleme de o reală importanță practică pot fi rezolvate cu ajutorul unor metode standard, denumite **tehnici de programare:** recursia, treirea, metoda reluării, metode euristice ș.a. [1]

Iterarea și recursia sunt tehnici cheie utilizate în informatică în scopul crearii algoritmilor și în dezvoltarea software-ului. În termeni simpli, o funcție iterativă este cea care încearcă să repete o parte a codului, iar o funcție recursivă este cea care se numește de mai multe ori pentru a repeta codul. [3]

După cum se cunoaște, orice algoritm recursiv poate fi transcris într-un algoritm iterativ și invers. Alegerea tehnicii de de programare – iterativitate sau recursivitate – ține, de asemenea, de competența programatorului. Evident, această alegere trebuie făcută luând în considerare avantajele și neajunsurile fiecărei metode, care variază de la caz la caz. [1]

2 Iterativitatea

2.1 Aspecte teoretice

Iterația - reprezintă repetarea unui anumit procedeu de calcul, prin aplicarea lui la rezultatul calculului din etapa precedentă. [4]

Iterativitatea este procesul prin care rezultatul este obținut ca urmare a execuției repetate a unui set de operații, de fiecare data cu alte valori de intrare. Numărul de iterații poate fi necunoscut sau cunoscut , dar determinabil pe parcursul execuției. Metoda de repetivitate este cunoscută sub numele de ciclu (loop) și poate fi realizată prin utilizarea următoarelor structure repetitive: ciclu cu test initial, ciclu cu test final, ciclu cu număr finit de pași. Indiferent ce fel de structură iterativă se folosește este necesar ca numărul de iterații să fie finit. [5]

Pascalul prezintă următoarele structuri iterative: ciclul **for**, ciclul **while... do** și ciclul **repeat... until**. [6]

2.1.1 Instrucțiunea *for*

Instrucţiunea **`for`** indică execuţia repetată a unei instrucţiuni în funcţie de valoarea unei variabile de control. Sintaxa instrucţiunii în studiu este:

<*Instrucţiune* **`for`**> ::= **`for`** <*Variabilă*> **`:`** = <*Expresie*> <*Pas*> <*Expresie*> **`do`** <*Instrucţiune*>

<*Pas*> ::= **`to`** | **`downto`**

Diagramele sintactice sînt prezentate în *figura 3.9*.

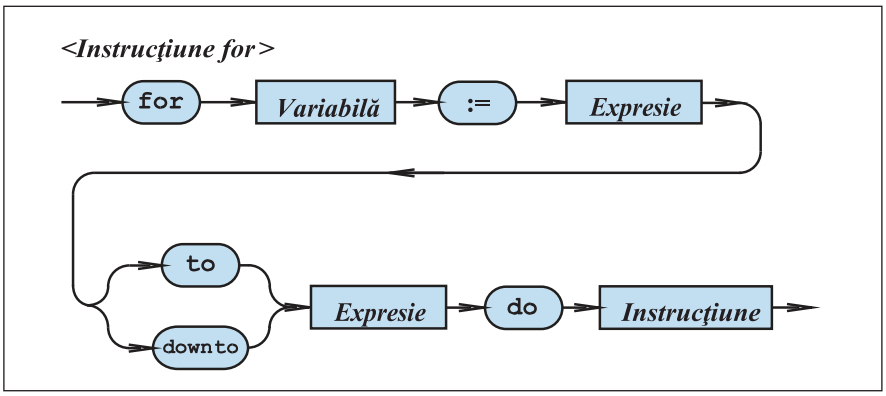

Fig. 3.9. Diagrama sintactică a instrucţiunii **`for`**

Variabila situată după cuvîntul-cheie **for** se numește **variabilă de control** sau **contor**. Această variabilă trebuie să fi e de tip ordinal.

Valorile expresiilor din componenţa instrucţiunii **for** trebuie să fi e compatibile, în aspectul atribuirii, cu tipul variabilei de control. Aceste expresii sînt evaluate o singură dată, la începutul ciclului. Prima expresie indică valoarea iniţială, iar expresia a doua – valoarea fi nală a variabilei de control.

Instrucţiunea situată după cuvîntul-cheie **do** se execută pentru fi ecare valoare din domeniul determinat de valoarea iniţială și de valoarea fi nală.

Dacă instrucţiunea **for** utilizează pasul **to**, valorile variabilei de control sînt incrementate la fi ecare repetiţie, adică se trece la succesorul valorii curente. Dacă valoarea iniţială este mai mare decît valoarea fi nală, instrucţiunea situată după cuvîntul-cheie **do** nu se execută niciodată.

Dacă instrucţiunea **for** utilizează pasul **downto**, valorile variabilei de control sînt decrementate la fi ecare repetiţie, adică se trece la predecesorul valorii curente. Dacă valoarea iniţială este mai mică decît valoarea fi nală, instrucţiunea situată după cuvîntul-cheie **do** nu se execută niciodată. [2]

2.1.2 Instrucţiunea *while*

Instrucţiunea **while** conţine o expresie booleană care controlează execuţia repetată a altei instrucţiuni. Sintaxa instrucţiunii în studiu este:

<*Instrucţiune* **while**> ::= **while** <*Expresie booleană*> **do** <*Instrucţiune*> Diagrama sintactică este prezentată în *figura 3.11*.

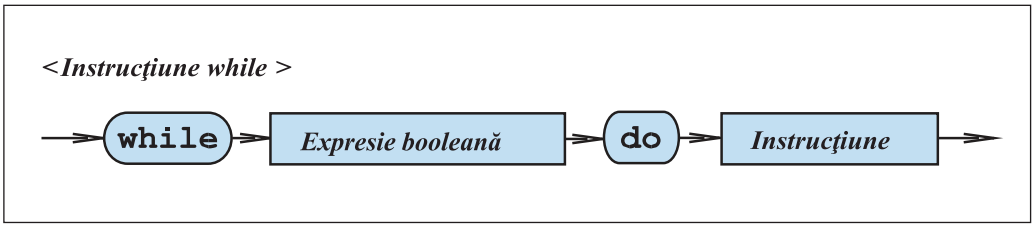

Fig. 3.11. Diagrama sintactică a instrucţiunii **while**

Instrucţiunea situată după cuvîntul-cheie **do** se execută repetat atîta timp, cît valoarea expresiei booleene este `true`. Dacă expresia booleană ia valoarea `false`, instrucţiunea de după **do** nu se mai execută. Se recomandă ca expresia booleană să fi e cît mai simplă, deoarece ea este evaluată la fi ecare iteraţie.

În mod obișnuit, instrucţiunea **while** se folosește pentru organizarea calculelor repetitive cu variabile de control de tip `real`.

În programul ce urmează, instrucţiunea **while** este utilizată pentru afi șarea valorilor funcţiei $y = 2x$. Argumentul x ia valori de la x_1 la x_2 cu pasul Δx.

Instrucţiunea **while** se consideră deosebit de utilă în situaţia în care numărul de execuţii repetate ale unei secvenţe de instrucţiuni este difi cil de evaluat.

2.1.3 Instrucțiunea *repeat*

Instrucțiunea repeat indică repetarea unei secvențe de instrucțiuni în funcție de valoarea unei expresii booleene. Sintaxa instrucțiunii este:

<Instrucțiune **repeat**> ::=**repeat** <Instrucțiune> {;<Instrucțiune>} **until** <Expresie booleană>.

În mod obișnuit instrucțiunea repeat se utilizează în locul instrucțiunii while atunci când evaluarea expresiei care controlează repetiția se face după executarea secvenței de repetat. Aceasta este utilă în situația în care numărul de executări repetate ale unei secvențe de instrucțiuni este dificil de evaluat. [2]

3 Recursivitatea

3.1 Aspecte teoretice

Recursivitatea este procesul iterativ prin care valoarea unei variabile se determină pe baza uneia sau a mai multora dintre propriile ei valori anterioare. Structurile recursive reprezintă o alternativă de realizare a proceselor repetitive fără a utiliza cicluri. [5]

Recursia presupune execuția repetată a unui modul, însă în cursul execuției lui (și nu la sfârșit, ca în cazul iterației), se verifică o condiție a cărei nesatisfacere , implică reluarea execuției modulului de la începutul său. [7]

4 Avantaje și Dezavantaje

4.1 Studiul Comparativ al iterativității și recursivității

Nr. Crt.	Caracteristici	Iterativitate	Recursivitate
1.	Necesarul de memorie	mic	mare
2.	Timpul de execuție	același	
3.	Structura programului	complicată	simplă
4.	Volumul de muncă necesar pentru scrierea programului	mare	mic
5.	Testarea și depănarea programelor	simplă	complicată

5 Exemple de programe (Iterativitate)

5.1 Scrieți un program care ia un număr întreg pozitiv n și afișează toate numerele de la 1 până la n. Exemplu:5→12345. [8]

```
Program iterativitate_1;

Var i, n : integer;

begin

readln(n); {Se citește valoarea variabilei n de la tastatură}
```

for i:= 1 **to** n **do** **write**(i); **{Variabila i va lua, pe rând, câte o valoare de la 1 la n, cu pasul de 1, iar valorile variabilei vor fi afișate într-un rând, fără spații}**

end.

5.2 Scrieți un program care ia un număr întreg pozitiv n apoi calculează și afișează numărul cu poziția n din șirul lui Fibonacci. Primele două numere ale șirului vor fi definite ca 1 și fiecare număr ulterior reprezintă suma a numere două anterioare, deci secvența este 1, 1, 2, 3, 5, 8, 13... Exemplu: n=6 => nr din șir este 8. [8]

Program iterativitate_2;

Var n : integer;

Function Fibonacci (n1:integer): integer; var i, a, b, fib: integer;

Begin

a:=1; b:=1; fib:=1; i:= 3; {a și b reprezintă primele 2 numere din șir, care sunt egale cu 1, iar variabilei fib I se atribuie valoarea 1 pentru cazurile în care poziția numărului este 1 sau 2. Variabila i arată poziția numărului și, întrucât pentru pozițiile 1 și 2 este specificată valoarea variabilei fib, I ia valori începând cu 3} while I <= n1 do begin fib:= a + b; a:= b; b:= fib; inc(i);

end; **{În acest ciclu *while* are loc calcularea fiecărui număr din șir de la cel de pe poziția 3 până la cel de pe poziția necesară. Variabila b reprezintă predecesorul numărului de pe poziți i, iar variabila a predecesorul predecesorului. Valoarea variabilei fib este reprezentată de suma variabilelor a și b, iar valorile acestor variabile sunt actualizate. La ieșirea din ciclu, variabila fib va conține rezultatul pentru i=n1}** Fibonacci := fib; **{Funcției I se atribuie valoarea variabilei fib} end;**

begin

readln(n); {Este citită poziția numărului}

writeln('Al ',n, '-lea numar din sirul lui Fibonacci este ', fibonacci(n));

{Se afișează numărul de pe poziția data de la tastatură, fiind apelată funcția fibonacci}

end.

5.3 Să se scrie un program care citește și afișează caracterul introdus de utilizator și se oprește când caracterul ‘c’ este introdus. [9]

Program iterativitate_3;

Var c : char;

Begin

Repeat

writeln('Introduceti un caracter. Introduceti ‘c’ pentru a opri programul:');

readln(c);

```
write('Ati introdus: ');

writeln(c); {Datorită instrucțiunii repeat, utilizatorului i se va cere să introducă un caracter, iar la ecran se va afișa caracterul introdus, până când utilizatorul va introduce caracterul 'c', condiția de la until va devein adevărată și se va ieși din ciclu}


until c = 'c';

writeln('Stop'); {La introducerea caracterului 'c'are loc ieșirea din ciclu și se trece la următoarea instrucțiune, astfel încât se afișează cuvântul "Stop"}

end.
```

5.4 Calcularea sumei numerelor de la 1 până la N

```
Program iterativitate_4;

Var n, sum, i: integer;

Begin

Readln(n);

For i:=1 to n do begin    {Adunăm numerele de la 1 la N pentru a afla}

  Sum:=sum+i;                          {suma numerelor}

  End;

Writeln(sum);

End.
```

5.5 Calcularea produsului numerelor de la 1 la N

```
Program iterativitate_5;

Var n, i: integer;

     Produs:longint;

Begin

  Readln(n);

  Produs:=1;

For i:=1 to n do begin    {Înmulțim toate numerele de la 1 la N pentru a afla}

            Produs:=Produs*i;                          {produsul numerelor}

  End;
```

```
Writeln(Produs);
End.
```

6 Exemple de programe recursive

6.1 Calcularea sumei numerelor de la 1 până la N

```
Program_1;
Var n, sum, i: integer;
begin
Function sum(n:integer):integer;        {Rezultatul sumei este integer}
Begin
  If n=1 then sum:=1 else begin
          Sum:=n+sum (n-1);             {Adunăm N la sumă apoi reapelăm funcția cu valoarea
  End;                                   precedentă (N-1) adăugând numărul la suma...Repetăm acest
End.                                     proces până ajungem la N=1 }
```

6.2 Calcularea Produsului numerelor de la 1 la N

```
Program_2;
Var n,i: integer;
    Produs: longint;
begin
Function produs(n:integer):longint;
Begin
  If n=1 then produs:=1 else begin          {Înmulțim produsul=1 la N apoi reapelăm funcția cu
          produs:=n*produs (n-1);            valoarea precedentă (N-1) înmulțind produsul}
  End;
End.
```

6.3 Calcularea sumei numerelor de la 1 la N ce sunt divizibile cu Q

```
Program_3;
Var q,n,i,sum : integer;
begin
procedure suma(q:integer; divizor:integer; var sum:integer) ;
```

```
begin
if q=0 then sum:=sum+0 else begin
  if q mod divizor = 0 then begin          {dacă q se împarte exact la divizor atunci procedura
        sum:=sum+q;                         se auto-apelează cu valoarea de dinainte(q-1)}
        writeln(q);
        suma (q-1, divizor, sum);
  end else suma(q-1, divizor, sum );
end;
end.
```

6.4 Scrieți un program care calculează produsul P(n)=1*4*7*...*(3n-2) [1]

Program_4

```
Type Natural_nenul=1..MaxInt;
Var n:Natural_nenul;
Function P(n1:Natural_nenul):Natural_nenul;
Begin
If n1=1 then P:=1 else      {Are loc specificatia cazului elementar, care se rezolva direct}
P:=P(n1-1)*(3*n1-2) ș
Begin
Writeln(`Dati m');
Readln(n); {Are loc citirea lui N}
Writeln(`Produsul este',P(n));
End.
```

6.5 Scrieti un program care să calculeze produsul P(n)=2*4*6*2n [1]

Program_5

```
Type Natural_nenul=1..MaxInt;
Var n:Natural_nenul;
Function P(n1:Natural_nenul):Natural_nenul;
Begin
If n1=1 then P:=2 else   {Are loc specificatia cazului elementar, care se rezolva direct}
P:=P(n1-1)*2*n1;
```

```
End;

Begin

Writeln(`Dati m');

Readln(n);  {Are loc citirea lui N}

Writeln(`Produsul este',P(n));

End.
```

Concluzie:

În concluzie din informațiile cercetate dar și din propria experiență pot spune că personal pentru mine modul recursiv este mult mai ușor decât cel iterativ atât timp cât nu ia mult spațiu și este mai usor de redactat.

După cum se cunoaște, orice algoritm recursiv poate fi transcris într-un algoritm iterativ și invers. Alegerea tehnicii de de programare – iterativitate sau recursivitate – ține, de asemenea, de competența programatorului. Evident, această alegere trebuie făcută luând în considerare avantajele și neajunsurile fiecărei metode, care variază de la caz la caz

Bibliografie:

1. Anatol Gremalschi „Informatică. Manual pentru clasa a 11-a” http://ctice.gov.md/manuale-scolare/;
2. Anatol Gremalschi, Iurie Mocanu, Ion Spinei „Informatică. Manual pentru clasa a 9-a” http://ctice.gov.md/manuale-scolare/;
3. Iteration & Recursion https://www.advanced-ict.info/programming/recursion.html;
4. Wikitionary https://ro.m.wikitionary.org/wiki/iterație;
5. http://m.authorstream.com/presentation/aSGuest41792-360322-iterativitate-sau-recursivitate-rodika-guzun-science-technology-ppt-powerpoint/;
6. https://www.cise.ufl.edu/~mssz/Pascal-CGS2462/ifs-and-loops.html;
7. https://prezi.com/qfmfcl_7jdpg/recursivitate-si-iterativitate/;
8. „Iteration vs recursion in Introduction to Programming Classes: An Empirical Study”
9. http://staff.cs.upt.ro/~chirila/teaching/upt/id12-sda/lectures/ID-SDA-Cap5.pdf