Limbaje de programare

Recursivitate

6 octombrie 2009

Recursivitate: definiție, exemple

Din matematică cunoaștem *șiruri recurente*:

progresie aritmetică: $x_0 = a$, $x_n = x_{n-1} + p$, pentru $n > 0$

progresie geometrică: $x_0 = b$, $x_n = a \cdot x_{n-1}$, pentru $n > 0$

⇒ nu calculează x_n *direct*, ci *din aproape în aproape*, folosind x_{n-1}.

Un obiect (noțiune) e recursiv(ă) dacă e *folosit în propria sa definiție*.

Alte exemple: combinări C_n^k, șirul lui Fibonacci, ... (scrieți relațiile!)

Recursivitate: definiție, exemple

Recursivitatea e fundamentală în informatică:
reduce o problemă la un caz mai simplu al *aceleiași* probleme

obiecte: un *șir* e $\begin{cases} \text{un singur element} & \bigcirc \\ \text{un element urmat de un } \textit{șir} & \bigcirc\underbrace{\bigcirc\bigcirc\bigcirc}_{\text{șir}} \end{cases}$
ex. cuvânt (șir de litere); număr (șir de cifre zecimale)

acțiuni: un *drum* e $\begin{cases} \text{un pas} & \longrightarrow \\ \text{un } \textit{drum} \text{ urmat de un pas} & \underbrace{\longrightarrow\longrightarrow\longrightarrow}_{\text{drum}}\longrightarrow \end{cases}$
ex. parcurgerea unei căi într-un graf

O *expresie*: număr (7), sau identificator (x), sau *expresie* + *expresie*, sau *expresie* - *expresie*, sau (*expresie*), ...

Exemplu: funcția putere

$$x^n = \begin{cases} 1 & n = 0 \\ x \cdot x^{n-1} & \text{altfel } (n > 0) \end{cases}$$

```
#include <stdio.h>
double pwr(double x, unsigned n) {
  return n==0 ? 1 : x * pwr(x, n-1);
}
int main(void) {
  printf("-2 la 3 = %f\n", pwr(-2.0, 3));
  return 0;
}
```

Obs.
Funcția standard putere (cu 2 argumente `double`) este `pow`, din `<math.h>`

Tipul `unsigned` reprezintă întregi fără semn (numere naturale)

Antetul funcției `pwr` reprezintă o *declarație* a ei
deci putem mai târziu folosi funcția în propriul corp (apelul recursiv)

Chiar dacă scriem `pwr(-2, 3)`, *întregul* `-2` va fi *convertit la real*,
(se cunoaște tipul necesar pentru fiecare parametru)

Mecanismul apelului recursiv

Funcția `pwr` face două calcule:
– un *test* (`n == 0` ? a ajuns la *cazul de bază* ?) dacă da, `return 1`
– dacă nu, o *înmulțire*; pt. operandul drept trebuie un *nou apel, recursiv*

```
pwr(5, 3)
    apel↓ ↑125
    5 * pwr(5, 2)
        apel↓ ↑25
        5 * pwr(5, 1)
            apel↓ ↑5
            5 * pwr(5, 0)
                apel↓ ↑1
                  1
```


Mecanismul apelului recursiv (cont.)

În calculul recursiv al funcției putere:

Fiecare apel face *"în cascadă" un nou apel*, până la cazul de bază

Fiecare apel execută *același cod*, dar cu *alte date*
(valori proprii pentru parametri)

Ajunși la cazul de bază, toate apelurile *începute* sunt încă *neterminate*
(fiecare mai are de făcut înmulțirea cu rezultatul apelului efectuat)

Revenirea se face *în ordine inversă* apelării
(apelul cu exponent 0 revine primul, apoi cel cu exponent 1, etc.)

Elementele unei definiții recursive

1. *Cazul de bază* (*NU* necesită apel recursiv)
= cel mai simplu caz pentru definiția (noțiunea) dată, definit direct
  termenul inițial dintr-un șir recurent: x_0
  un element, în definiția: șir = element sau șir + element

E o *EROARE* dacă lipsește cazul de bază (apel recursiv infinit!)

2. *Relația de recurență* propriu-zisă
– definește noțiunea, folosind un caz mai simplu al aceleiași noțiuni

3. Demonstrație de *oprire a recursivității* după număr finit de pași
(ex. o mărime nenegativă care descrește când aplicăm definiția)
– la șiruri recurente: indicele (nenegativ; mai mic în corpul definiției)
– la obiecte: dimensiunea (definim obiectul prin alt obiect mai mic)

Sunt recursive, și corecte, următoarele definiții ?

? $x_{n+1} = 2 \cdot x_n$
? $x_n = x_{n+1} - 3$
? $a^n = a \cdot a \cdot \ldots \cdot a$ (de n ori)
? o frază e o înșiruire de cuvinte
? un șir e un șir mai mic urmat de un alt șir mai mic
? un șir e un caracter urmat de un șir

O definiție recursivă trebuie să fie *bine formată* (v. condițiile 1-3)
  ceva nu se poate defini doar în funcție de sine însuși NU: $x = f(x)$
  se pot utiliza doar noțiuni deja definite
  nu se poate genera un calcul infinit (trebuie să se oprească)

Exemplu: cel mai mare divizor comun

$$cmmdc(a,b) = \begin{cases} a & a = b \\ cmmdc(a-b,b) & a > b \\ cmmdc(a,b-a) & a < b \end{cases}$$

```
unsigned cmmdc(unsigned a, unsigned b) {
  return a == b ? a
                : a > b ? cmmdc(a-b, b)
                        : cmmdc(a, b-a);
}
int main(void) {
  printf("cmmdc(20, 8) e %u\n",
                        cmmdc(20, 8));
  return 0;
}
```

Numerele unsigned se tipăresc folosind formatul `%u`

Calculul e corect doar cu a și b nenule. Pentru a trata și cazul zero:

```
return a == 0 ? b
       : b == 0 ? a
         : a > b ? cmmdc(a-b, b) : cmmdc(a, b-a);
```


Factorialul, calculat recursiv

```
unsigned fact1(unsigned n)
{
  return n == 0 ? 1 : n * fact1(n-1);
}
```

Corespunde scrierii: $5! = 5 \cdot (4 \cdot (3 \cdot (2 \cdot (1 \cdot 1))))$

Calculul (*) făcut la sfârșitul funcției, *după* revenirea din apelul recursiv

Calcule succesive: `1*1` (1), `2*1` (2), `3*2` (6), `4*6` (24), `5*24` (120), etc.

Factorialul: varianta 2, cu *acumulator*

```
// acumulează în res înmulţirile deja făcute
unsigned fact2(unsigned n, unsigned res)
{
  return n == 0 ? res : fact2(n-1, res*n);
}
```

Corespunde scrierii: $5! = ((((1 \cdot 5) \cdot 4) \cdot 3) \cdot 2) \cdot 1$
$n! = n \cdot (n-1)! \Rightarrow$ rezultatul va conține n ca factor
  actualizăm un *acumulator* / rezultat parțial
  îl transmitem ca *argument* pentru următorul apel
  în cazul de bază, *rezultatul e complet*, îl returnăm
valori `res`: 1, 5 (`5*1`), 20 (`4*5`), 60 (`3*20`), 120 (`2*60`), 120 (`1*120`)

Am scris o funcție mai generală: calculează *res·n!* Vrem *res*=1
⇒ definim `unsigned fact(unsigned n) { return fact2(n, 1); }`

Factorialul: secvența de apeluri

```
fact1(3)
   apel↓↑6
       3 * fact1(2)
           apel↓↑2
               2 * fact1(1)
                   apel↓↑1
                       1 * fact1(0)
                           apel↓↑1
                               1
```

calcul: $3 \cdot (2 \cdot (1 \cdot 1))$

Apelul: în calculul rezultatului

Înmulțirea: după revenirea din apel

```
fact2(3, 1)
apel↓  ↑6
   fact2(2, 3)
     apel↓  ↑6
       fact2(1, 6)
         apel↓  ↑6
            fact2(0, 6)
              apel↓  ↑6
                   6
```

calcul: $(((1 \cdot 3) \cdot 2) \cdot 1)$

Calculul: înainte de apel
(rezultatul parțial acumulat transmis ca parametru)
La revenire: nici un calcul;
valoarea returnată nemodificat

Calculul sumei unei serii

Forma: $s_0 = t_0$, $s_n = s_{n-1} + t_n$, pentru $n > 0$
(t_n = termenul general, pentru care avem o formulă)

Exemplu pentru seria armonică ($t_n = 1/n$)
$s_n = 1/1 + 1/2 + \ldots + 1/n$
recursiv: $s_0 = 0$, $s_n = s_{n-1} + 1/n$ pentru $n > 0$

În cuvinte: știm să răspundem direct cât e s_0 : 0.
Nu putem calcula direct s_n (pentru $n > 0$),
dar dacă aflăm cât e s_{n-1} mai trebuie să adunăm $1/n$.
$\Rightarrow$
Funcția care calculează pe $s(n)$ răspunde 0 dacă $n = 0$
iar altfel, calculează $s(n-1)$, adună $1/n$ și returnează rezultatul.

Calculul sumei unei serii

```
#include <stdio.h>
double suma_rec(unsigned n) {
  return n == 0 ? 0 : suma_rec(n-1) + 1.0/n;
}
int main(void) {
  printf("suma pana la 1/100: %f\n", suma_rec(100));
  return 0;
}
```

Termenii se adună începând de la 1/1 la 1/100, la revenirea din apel

`1.0 / n` : operație între real și întreg : întregul convertit la real
ATENȚIE: `1/n` dă valoarea `0` când `n > 1` (împărțire intreagă)

Suma unei serii – variantă cu rezultat acumulat

În $s_n = s_{n-1} + 1/n$, trebuie adunat $1/n$, dar nu știm încă s_{n-1}
$\Rightarrow$ folosim un rezultat parțial `rez` la care adunăm $1/n$
$\Rightarrow$ apelăm recursiv cu valoarea `rez + 1.0/n`

```
double suma_inv(unsigned n, double rez) {
  return n == 0 ? rez : suma_inv(n - 1, rez + 1.0/n);
}
```

Când $n = 0$, totul e adunat deja în `rez`, care e returnat ca rezultat

În apelul inițial, rezultatul acumulat e zero: `suma_inv(100, 0.0)`

`rez` e un detaliu de implementare, nu face parte din enunțul problemei
$\Rightarrow$ definim o funcție cu un singur parametru, care apelează `suma_inv`

```
double serie_armonica(unsigned n) { return suma_inv(n, 0.0); }
```


Calculul cu aproximații: rădăcina pătrată

Din matematică: $a_0 = 1$, $a_{n+1} = \frac{1}{2}(a_n + \frac{x}{a_n})$

Formulăm recursiv:

Calculul *aproximației dorite* (ex. cu $\epsilon = 10^{-3}$) de la o *aproximație dată*:
ce se dă (parametri): x și aproximația curentă
ce se cere = val. funcției (o aproximație suficient de bună)

Dacă precizia e bună $|a_{n+1} - a_n| < \epsilon$ returnăm *aproximația curentă* a_n
Altfel, returnăm valoarea *calculată recursiv* cu *noua aproximație* a_{n+1}

Dezvoltăm: $|a_{n+1} - a_n| < \epsilon \Rightarrow |a_n - x/a_n| < 2 \cdot \epsilon$

Calculul cu aproximații: rădăcina pătrată

```
#include <math.h>
// pentru declaraţia double fabs(double x);  (val. abs. nr. real)

double rad(double x, double a_n) {    // rad.lui x, se da aprox.a_n
  return fabs(a_n - x/a_n) < 2e-3 ? a_n : rad(x, (a_n + x/a_n)/2);
}

double radacina(double x) { return x < 0 ? -1 : rad(x, 1.0); }
```

Soluția dorită e funcția `radacina`: apelează `rad` cu aprox. inițială `1`

Pentru argument negativ, returnează `-1` (îl interpretăm ca eroare)

Calculul sumei unei serii cu precizie dată

Calculăm $s_n = s_{n-1} + t_n$ ($n \geq 0$), cu $s_0 = 0$
până când valoarea absolută a termenului $t_n = x^n/n!$ e neglijabilă.

Formulăm recursiv: calculul *sumei dorite*, dată fiind *suma curentă* s_{n-1}:
– dacă termenul curent t_n e suficient de mic, returnăm suma curentă
– altfel, returnăm suma calculată *recursiv*, de la *noua sumă* $s_{n-1} + t_n$

Exemplu: seria $1 + 1/2^2 + 1/3^2 + \ldots = \pi^2/6$ $t_n = 1/n^2$ ($n > 0$):

```
double sum_2(unsigned n, double s_n_1) {
  return 1./n/n < 1e-6 ? s_n_1 : sum_2(n+1, s_n_1 + 1./n/n);
}             // 1. == 1.0 = 1 real, forteaza impartire reala
```

și folosim apelul inițial   `sum_2(1, 0)`   (pornind de la $n = 1$, $s_0 = 0$)

Exemplu: seria Taylor pentru e^x

$e^x = x^0/0! + x^1/1! + x^2/2! + \ldots$ cu $t_n = x^n/n!$ $(n \geq 0)$
Pentru a nu recalcula inutil în t_n pe x^{n-1} și $(n-1)!$
exprimăm recursiv $t_n = t_{n-1} \cdot x/n$, pentru $n > 0$, $t_0 = 1$.
⇒ la pasul curent, avem s_{n-2} și t_{n-1}, calculăm t_n și $s_{n-1} = s_{n-2} + t_{n-1}$

```
#include <math.h>
#include <stdio.h>
double e_xr(double x, unsigned n, double s_n_2, double t_n_1) {
  return fabs(t_n_1)<1e-6 ? s_n_2
                          : e_xr(x, n+1, s_n_2+t_n_1, t_n_1 * x/n);
} // apelam cu valori initiale potrivite in e_x cu 1 parametru, x
double e_x(double x) { return e_xr(x, 1, 0.0, 1.0); }
int main(void) {
  printf("e^-1 = %f\n", e_x(-1.0));
  return 0;
}
```


Recursivitate și inducție

Recursivitatea e strâns legată de inducția matematică; ambele:
– au un *caz de bază*
– leagă o *noțiune* de *ea însăși* (relatia de recurent / pasul inductiv)

Diferă al treilea element, *sensul* în care se face raționamentul:

– *crescător* la principiul inducției matematice:
O afirmație $P(n)$ e valabilă pentru orice n (*crescând* spre infinit) dacă:
$P(0)$ e adevărat și
$P(n) \Rightarrow P(n+1)$ dacă $P(n)$ adevărat atunci $P(n+1)$ adevărat

– *descrescător* la recurență: definim ceva *mai mare* prin ceva *mai mic*
se oprește când noțiunea definită ajunge la cazul de bază (suficient de simplu)

Recursivitatea în sintaxa limbajelor de programare

Programele pot fi oricât de complexe, dar au structură riguros definită
⇒ se pretează la definiții recursive
– înșiruiri liniare: un program are oricâte funcții,
o funcție are oricâte argumente și instrucțiuni, etc.
– structuri mai complexe, ex. expresie formată din operator și 2 expresii

Structura (sintaxa, *gramatica*) limbajului se reprezintă uzual
într-o notație standard numită BNF (Backus-Naur Form). Ex.
antet-funcție ::= *tip identificator* (*parametri*)
parametri ::= `void` | *lista-parametri*
lista-parametri ::= *tip identificator* | *tip identificator* , *lista-parametri*

unde ::= denotă *definiție* iar | *alternativă* (alegere)

Cazuri particulare: recursivitate *la stânga* și *la dreapta*,
după locul în care apare noțiunea recursivă în corpul definiției