Programarea calculatoarelor

Recursivitate

3 martie 2009

Recursivitate: definiție, exemple

Recursivitatea e un concept fundamental în matematică și informatică:
⇒ reducem o problemă la o instanță mai simplă a *aceleiași* probleme

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

Exemplu din matematică: *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 $n > 0$
⇒ formula nu calculează x_n *direct*, ci *din aproape în aproape*

Alte exemple: combinări C_n^k, șirul lui Fibonacci, ...

– *obiecte* definite recursiv: un *șir* e $\begin{cases} \text{un singur element, sau} \\ \text{un element urmat de un } \textit{șir} \end{cases}$

ex.: cuvânt (șir de litere); număr (șir de cifre zecimale)

– *acțiuni* definite recursiv: un *drum* e $\begin{cases} \text{un pas, sau} \\ \text{un } \textit{drum} \text{ urmat de un pas} \end{cases}$

(de exemplu o cale într-un graf)

Exemplu: funcția putere

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

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

```
#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;
}
```

– tipul `unsigned` reprezintă întregi fără semn (numere naturale)
– antetul funcției reprezintă o *declarație* a ei, deci poate fi folosită în orice punct ulterior (și în propriul corp — cazul apelului recursiv)
– chiar dacă scriem `pwr(-2, 3)`, întregul `-2` va fi convertit la real, întrucât 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.)

Observăm, pentru exemplul funcției putere:

– fiecare apel face *"în cascadă"* un alt 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* (conditia de oprire)
– cel mai simplu caz pentru definiția (noțiunea) dată, definit direct, *NU* necesită apel recursiv. Exemple:
  termenul inițial dintr-un șir recurent
  un element, în cazul unui șir (v. exemplu p. 2)
  un pas, în cazul unui drum (v. exemplu p. 2)
EROARE dacă lipsește cazul de bază (apel recursiv infinit!)

2. *relația de recurență*
– definește noțiunea, folosind un caz mai simplu al aceleiași noțiuni

3. demonstrație de *oprire a recursivității* după un 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, şi corecte, următoarele definiţii ?

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

O definiţie recursivă trebuie să fie *bine formată* (v. condiţiile 1-3)
– ceva nu se poate defini doar în funcţie de sine însuşi ($x = f(x)$)
– se pot utiliza doar noţiuni 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 şi b nenule. Pentru a trata şi cazul zero:

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


Factorialul: două variante

```
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 sfârşitul funcţiei, *după* revenirea din apelul recursiv
calcule succesive: `1*1` (1), `2*1` (2), `3*2` (6), `4*6` (24), `5*24` (120), etc.

```
unsigned fact2(unsigned n, unsigned res) { // apel iniţial: res=1
  return n == 0 ? res : fact2(n-1, res*n);
}
```

corespunde scrierii: $5! = ((((1 \cdot 5) \cdot 4) \cdot 3) \cdot 2) \cdot 1$
$n! = n \cdot (n-1)! \Rightarrow$ trebuie înmulţit cu `n`, chiar dacă nu ştim cât e $(n-1)!$
⇒ acumulăm / actualizăm un rezultat parţial, transmis ca *argument* pentru următorul apel; în cazul de bază, rezultatul e complet şi returnat
valori `res`: 1, 5 (`5*1`), 20 (`4*5`), 60 (`3*20`), 120 (`2*60`), 120 (`1*120`)

Am rezolvat recursiv o problemă mai generală: calculăm *res·n!* Vrem *res*=1 ⇒ definim `unsigned fact(unsigned n) { return fact2(n, 1); }`

Factorialul: secvenţa de apeluri

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

– apelul: în calculul rezultatului
– înmulţirea: după revenirea din apel
calcul: $3 \cdot (2 \cdot (1 \cdot 1))$

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

– calculul: înainte de apel (rezultatul parţial acumulat transmis ca parametru)
– la revenire: nici un calcul; valoarea returnată nemodificat
calcul: $(((1 \cdot 3) \cdot 2) \cdot 1)$

Calculul sumei unei serii

Forma: $s_0 = t_0$, $s_n = s_{n-1} + t_n$, pentru $n > 0$ (t_n = termenul general)
Exemplu pentru seria armonică ($t_n = 1/n$): $1/1 + 1/2 + \ldots + 1/n$

```
#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;
}
```

Am transcris direct definiţia recursivă: $s_0 = t_0$, $s_n = s_{n-1} + 1/n$ ($n > 0$)

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

`1.0 / n` : operaţie între real şi întreg : întregul convertit la real
ATENŢIE: `1/n` dă valoarea `0` când `n > 1` (împărţire intreagă)

Suma unei serii – variantă cu rezultat acumulat

În $s_n = s_{n-1} + 1/n$, trebuie adunat $1/n$, dar nu ştim încă s_{n-1}
⇒ folosim un rezultat parţial la care adunăm $1/n$ (adunăm termenii pornind de la $1/n$ spre $1/1$) ⇒ când facem apelul `suma_inv(n, rez)`, `rez` e suma deja calculată a termenilor din dreapta celui curent (t_n)

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

– dacă $n = 0$, totul e adunat deja în `rez`, care e returnat ca rezultat
– altfel, rezultatul e suma primilor $n - 1$ termeni (apel recursiv) pornind de la rezultatul parţial `rez` plus termenul curent $1/n$

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

`rez` e un detaliu de implementare, nu face parte din enunţul problemei
⇒ definim o funcţie 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 cere = val. funcției *ce se dă* (parametru)
– 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: $|a_{n+1} - a_n| < \epsilon \Rightarrow |a_n - x/a_n| < 2 \cdot \epsilon$

```
#include <math.h>    // pt. 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 suficient de mică.

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 $n > 0$, $t_0 = 1$.
$\Rightarrow$ 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_x(double x, unsigned n, double s_n_2, double t_n_1) {
  return fabs(t_n_1)<1e-6 ? s_n_2
                          : e_x(x, n+1, s_n_2+t_n_1, t_n_1 * x/n);
}
int main(void) {
  printf("e^-1 = %f\n", e_x(-1, 1, 0.0, 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:
e adevărat $P(0)$ 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 dimensiunea (măsura) noțiunii definite scade la zero).

Recursivitatea în sintaxa limbajelor de programare

Multe elemente de limbaj pot fi oricât de complexe, dar au structură riguros definită $\Rightarrow$ 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 (*gramatica*) limbajului se reprezintă uzual printr-o notație standard numită BNF (Backus-Naur Form). Exemplu:
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