Recursivitate

Marius Minea

7 martie 2006

Recursivitatea: Noţiuni fundamentale

Recursivitatea e un concept fundamental în matematică şi informatică.
Un obiect (o noţiune) e recursiv(ă) dacă e folosit în propria sa definiţie.

Exemplu din matematică: şiruri recurente
– progresie aritmetică: $x_0 = a$, $x_n = x_{n-1} + p$, pentru $n > 0$
– progresie geometrică: $x_0 = b$, $x_n = a * x_{n-1}$, pentru $n > 0$
ş.a.m.d.: combinări C_n^k, şirul lui Fibonacci, ...

Alte exemple:
– *obiecte* definite recursiv:
un şir e un element, sau un element urmat de un şir
(de exemplu: un număr scris in baza 10 e un şir de cifre zecimale)
– *acţiuni* definite recursiv:
un drum e un pas, sau un drum urmat de încă un pas
(de exemplu, o cale într-un graf)

Elementele unei definiții recursive

1. cazul de bază (condiția de oprire)
– cel mai simplu caz pentru definiția respectivă:
  termenul inițial într-un șir recurent;
  cel mai mic obiect (un element, în cazul șirului);
  cea mai simplă acțiune (un pas în cazul drumului)
– uneori, e util un caz de bază vid (ex. o listă fără elemente)

2. relația de recurență
– definește noțiunea, printr-o relație cu un caz mai simplu al aceleiași noțiuni

3. o demonstrație/argument pt. oprirea definiției în număr finit de pași (ex.: o *măsură* nenegativă care descrește pe măsură ce urmărim definiția)
– pentru șiruri: indicele (scade în definiție; e nenegativ)
– pentru obiecte: dimensiunea (la fel, scade)

Sunt corecte următoarele definiţii ?

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

O definiţie recursivă trebuie să fie *bine formată* (v. condiţiile 1-3)
– o noţiune nu se poate defini *doar* în funcţie de sine însuși ($x = x$)
– se poate folosi *doar* de noţiuni deja definite
– nu poate genera un calcul infinit (trebuie să se oprească)

Recursivitate și inducție

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

Diferă al treilea element, sensul în care se face raționamentul:
– principiul inducției matematice garantează valabilitatea pentru *orice* n (*crescând* spre infinit) dacă $P(0)$ si $P(n) \Rightarrow P(n+1)$
– recurența se asigură de oprire printr-o măsură *descrescătoare*

Recursivitatea în definirea sintaxei limbajului

Multe elemente ale limbajului au o complexitate arbitrară, dar o structură riguros definită ⇒ se pretează la definiţii recursive:
– înşiruiri: un program poate avea oricâte funcţii,
o funcţie oricâte argumente şi instrucţiuni, etc.
– alte structuri complexe: o expresie are oricâţi operanzi / operatori

Structura (*gramatica*) limbajului se reprezintă uzual printr-o notaţie standard numită (BNF, Backus-Naur Form). Exemplu:

antet-funcţie ::= *tip identificator* (parametri)
parametri ::= `void` | *lista-parametri*
lista-parametri ::= *tip identificator* | *tip identificator* , *lista-parametri*

unde ::= denotă definiţie, iar | alternativă (alegere)
Definiţiile pot fi recursive, cu aceeaşi noţiune în stânga şi dreapta definiţiei

Exemplu: funcţia putere

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

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

– tipul `unsigned` reprezintă întregi fără semn (nr. naturale)
– după ce antetul funcţiei e scris, se permite folosirea (apelarea) ei, chiar în propriul corp de funcţie (apel recursiv).
– chiar dacă scriem `putere(-2, 3)`, `-2` va fi convertit la real, întrucât se cunoaşte tipul necesar pentru fiecare parametru

Exemplu: cel mai mare divizor comun

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

```
#include <stdio.h>
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`

Codul e corect doar pentru 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: două variante

```
unsigned fact1(unsigned n)
{
  return n == 0 ? 1
                : n * fact1(n-1);
}
// varianta 2: apelata cu fact2(n, 1)
unsigned fact2(unsigned n, unsigned res)
{
  return n == 0 ? res : fact2(n-1, n*res);
}
```

Varianta 1: calcul se face la sfârşit, după revenirea din apelul recursiv.
Calcul: 1, apoi 2 * 1, apoi 3 * (2 * 1), 4 * (3 * 2 * 1), ...
Varianta 2: calcul înainte de apel, rezultat parţial transmis ca parametru
Calcul: n, apoi n * (n-1), apoi (n*(n-1)) * (n-2), ...

Calculul sumei unei serii

$s_0 = 0$, $s_n = s_{n-1} + t_n$, pentru $n > 0$. Dacă avem o formula pentru t_n, putem calcula recursiv, ex. mai jos pentru seria $1/1 + 1/2 + \ldots + 1/n$

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

– `double`: tip pentru numele reale în dublă precizie (tipul implicit pentru constante reale, folosit și în funcţiile standard, și uzual în calcule)
– tipărit cu formatul `%f` (de fapt `printf` convertește `float` la `double`)
– `1.0 / n` : operaţie între real și întreg $\Rightarrow$ întregul convertit la real

Serii cu număr necunoscut de termeni, și precizie dată

Calculăm $e^x = x^0/0! + x^1/1! + x^2/2! + \ldots$ cu recurența $s_n = s_{n-1} + t_n$ până valoarea absolută a termenului $t_n = x^n/n!$ e suficient de mică. Pentru a nu recalcula x^n și $n!$ se exprimă recursiv și $t_n = t_{n-1} \cdot x/n$. Transmitem la funcție n, s_{n-2} și t_{n-1} pentru a calcula și apela recursiv în continuare cu $n+1$, s_{n-1} și t_n. Inițial: $n = 1$, $s_{-1} = 0$, $t_0 = 1$.

```
#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;
}
```

Calculul rădăcinii pătrate

Calculul de aproximaţii succesive pentru $\sqrt{x}$: $a_0 = 1$, $a_{n+1} = \frac{1}{2}(a_n + \frac{x}{a_n})$
– ne oprim din calcul atunci când s-a atins precizia dorită $|a_{n+1} - a_n| < \epsilon$
– avem nevoie de termenul a_n pentru a calcula a_{n+1},
deci îl transmitem ca parametru funcţiei:

```
#include <stdio.h>
#include <math.h>
double rad(double x, double a_n)
{
  return x < 0 ? 0 :
    fabs(a_n - x/a_n) < 1e-3 ? a_n : rad(x, (a_n + x/a_n)/2);
}
int main(void)
{
  printf("radical din 2 este %f\n", rad(2.0, 1.0));
  return 0;
}
```