

Recursivitate

Marius Minea

7 martie 2006

Recursivitate

2

Recursivitatea: Noțiuni fundamentale

Recursivitatea e un concept fundamental în matematică și informatică. Un obiect (o 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: $x_0 = b, x_n = a * x_{n-1}$, pentru $n > 0$

ș.a.m.d.: combinații C_n^k , șirul lui Fibonacci, ...

Alte exemple:

– *obiecte* definite recursiv:

un șir e un element, sau un element urmat de un șir

(de exemplu: un număr scris în baza 10 e un șir de cifre zecimale)

– *acțiuni* definite recursiv:

un drum e un pas, sau un drum urmat de încă un pas

(de exemplu, o cale într-un graf)

Programarea calculatoarelor. Curs 2

Marius Minea

Recursivitate

3

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* negativă care descrește pe măsură ce urmărim definiția)

– pentru șiruri: indicele (scade în definiție; e negativ)

– pentru obiecte: dimensiunea (la fel, scade)

Programarea calculatoarelor. Curs 2

Marius Minea

Recursivitate

4

Sunt corecte următoarele definiții ?

– $x_{n+1} = 2 * x_n$

– $x_n = x_{n+1} - 3$

– $a^n = a * a * \dots * a$ (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 definiție recursivă trebuie să fie *bine formată* (v. condițiile 1-3)

– o noțiune nu se poate defini *doar* în funcție de sine însuși ($x = x$)

– se poate folosi *doar* de noțiuni deja definite

– nu poate genera un calcul infinit (trebuie să se oprească)

Programarea calculatoarelor. Curs 2

Marius Minea

Recursivitate

5

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 $P(0)$ și $P(n) \Rightarrow P(n+1)$

– recurența se asigură de oprire printr-o măsură *descrescătoare*

Programarea calculatoarelor. Curs 2

Marius Minea

Recursivitate

6

Recursivitatea în definirea sintaxei limbajului

Multe elemente ale limbajului au o complexitate arbitrară, dar o structură riguros definită \Rightarrow se pretează la definiții recursive:

– înșiruire: un program poate avea oricâte funcții,

o funcție oricâte argumente și instrucțiuni, etc.

– alte structuri complexe: o expresie are oricâți operanzi / operatori

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

antet-funcție ::= *tip identificador* (*parametri*)

parametri ::= void | *lista-parametri*

lista-parametri ::= *tip identificador* | *tip identificador* , *lista-parametri*

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

Definițiile pot fi recursive, cu aceeași noțiune în stânga și dreapta definiției

Programarea calculatoarelor. Curs 2

Marius Minea

Exemplu: funcția 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 funcției e scris, se permite folosirea (apelarea) ei, chiar în propriul corp de funcție (apel recursiv).
- chiar dacă scriem `putere(-2, 3)`, `-2` va fi convertit la real, întrucât se cunoaște tipul necesar pentru fiecare parametru

Exemplu: cel mai mare divizor comun

```
#include <stdio.h>
unsigned cmmdc(unsigned a, unsigned b)
{
    return a == b ? a :
        a > b ? cmmdc(a-b, b) : cmmdc(a, b-a);
}
cmmdc(a,b) =
{
    a = b
    cmmdc(a-b,b) a > b
    cmmdc(a,b-a) a < b
}
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` și `b` 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 sfârșit, 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 parțial 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 $1/1 + 1/2 + \dots + 1/n$

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

Serii cu număr necunoscut de termeni, și precizie dată

Calculăm $e^x = x^0/0! + x^1/1! + x^2/2! + \dots$ 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 aproximații 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 funcției:

```
#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;
}
```