Limbaje de programare

Introducere. Recursivitate

Marius Minea

3 octombrie 2011

Programarea necesită

Atenție

Precizie – limbajul are regulile lui

Înțelegere – *întrebați dacă e neclar!*

Gândire – rezolvăm probleme (noi)

Exercițiu – *acasă* și în laborator

Despre limbajul C

dezvoltat în 1972 la *AT&T Bell Laboratories* de Dennis Ritchie
asociat cu sistemul de operare UNIX și utilitarele acestuia
(C dezvoltat inițial sub UNIX, apoi UNIX a fost rescris în C)
Brian Kernighan, Dennis Ritchie: The C Programming Language (1978)

Limbaj vechi, dar în evoluție
standardul ANSI C, 1988 (American National Standards Institute)
versiunea curentă: C99 (standard ISO 9899)

De ce folosim C?
versatil: acces direct la reprezentarea binară a datelor, libertate în lucrul cu memoria, bună interfață cu hardware
matur, bază mare de cod (biblioteci pentru multe scopuri)
eficient: compilatoare bune, generează cod compact, rapid

ATENȚIE: foarte ușor de făcut *erori*!

Calcule, funcții și programe

Un program
citește date de intrare
le *prelucrează* prin *calcule* (matematice)
produce (scrie) niște *rezultate*

În matematică, exprimăm calculele prin *funcții*:
cunoaștem funcții predefinite (sin, cos, etc.)
definim funcții noi (pentru problema dată)
combinăm funcțiile existente și definite de noi
le *folosim* într-o anumită ordine

La fel folosim funcțiile în programare.

Funcții în matematică și în C

Ridicarea la pătrat pentru întregi:

$sqr : \mathbb{Z} \to \mathbb{Z}$

$sqr(x) = x \cdot x$

tipul funcției	numele funcției	tipul și numele parametrului

```
int sqr(int x)
{
  return x * x;
}
```

Definiția unei funcții conține:

antetul funcției, care specifică: domeniul de valori (întregi), numele funcției (`sqr`) și parametrii acesteia (întregul `x`)

corpul funcției: aici, o singură *instrucțiune* (`return`), cu o *expresie* care dă valoarea funcției (pornind de la parametri)

Limbajul are *reguli* precise de scriere (*sintaxa*):

elementele se scriu într-o anumită *ordine*;

se folosesc *separatori* pentru a le delimita precis: `( )` `;` `{ }`

O a doua funcție

Ridicarea la pătrat pentru numere *reale*

$sqrf : IR \to IR$

$sqrf(x) = x \cdot x$

```
float sqrf(float x)
{
  return x * x;
}
```

Alt domeniu de definiție și de valori (reali) $\Rightarrow$ altă funcție
(chiar și operația `*` e alta, fiind definită pe altă mulțime)

Pentru a o deosebi în program de `sqr` trebuie să-i dăm alt nume.

Cuvintele `int`, `float` denotă *tipuri*.

Un *tip* e o *mulțime de valori* împreună cu un *set de operații* permise pentru aceste valori.

Pentru reali, e preferabil tipul `double` (dublă precizie)
(folosit și de funcțiile standard: `sin`, `cos`, `exp`, etc.)

Întregi și reali

Tipurile numerice diferă între C și matematică.

În matematică, $\mathbb{Z} \subset I\!R$, ambele sunt infinite, $I\!R$ e densă.

În C, `int`, `float`, `double` sunt tipuri finite; realii au precizie finită.

Constantele numerice au tip determinat de modul de scriere:

- 2 e un întreg, `2.0` e un real
- putem scrie un real în notație științifică: `1.0e-3` în loc de `0.001`
- sunt echivalente scrierile `1.0` și `1.` respectiv `0.1` și `.1`

Operații matematice

`+   -   *   /`

Înmulțirea trebuie scrisă explicit!
nu putem scrie $2x$, ci `2 * x` (sau `x * 2`)

Unele operații sunt diferite pentru întregi și reali:

Împărțirea întreagă e *împărțire cu rest* !!!

`7 / 2` dă valoarea 3, dar `7.0 / 2.0` dă valoarea `3.5`

`-7 / 2` dă valoarea −3, la fel cu `-(7 / 2)`

Operatorul *modulo* (scris %) e definit doar pentru întregi.

`9 / 5` = `1`	`9 % 5` = `4`	`9 / -5` = `-1`	`9 % -5` = `4`
`-9 / 5` = `-1`	`-9 % 5` = `-4`	`-9 / -5` = `1`	`-9 % -5` = `-4`

Semnul restului e același cu semnul deîmpărțitului.

Ecuația împărțirii cu rest: `a` = `a / b * b + a % b`

Puțină terminologie

Cuvinte cheie: au un înțeles predefinit (nu poate fi schimbat)
Exemple: instrucțiuni (`return`), tipuri (`int`, `float`, `double`), etc.

Identificatori (de ex. `sqr`, `x`) aleși de programator pentru a denumi funcții, parametri, variabile, etc.

Un identificator e o secvență de caractere formată din litere (mari și mici), liniuța de subliniere _ și cifre, care nu începe cu o cifră și nu este un cuvânt cheie.
Exemple: `x3, a12_34, _exit, main, printf, int16_t`

Constante (întreg: `-2`; real: `3.14`; caracter: `'a'`, șir: `"a"`)

Semne de punctuație, cu diverse semnificații:

- `*` e un operator
- `;` delimitează sfârșitul unei instrucțiuni
- parantezele `( )` grupează parametrii funcției sau o subexpresie
- acoladele `{ }` grupează instrucțiuni sau declarații

Funcții cu mai mulți parametri

Exemplu: discriminantul ecuației de gradul II: $a \cdot x^2 + b \cdot x + c = 0$

```
float discrim(float a, float b, float c)
{
  return b * b - 4 * a * c;
}
```

Între parantezele rotunde `( )` din antetul funcției putem specifica oricâți parametri, fiecare cu tipul propriu, separați prin virgulă.

Apelul de funcție

Până acum, am *definit* funcții, fără să le folosim.
Valoarea unei funcții poate fi *folosită* într-o expresie.
Sintaxa: ca în matematică: *funcție(param, param, $\cdots$, param)*
Exemplu: în discriminant, puteam folosi funcția `sqrf`:

```
    return sqrf(b) - 4 * a * c;
```

Sau, folosind funcțîa `sqr` dinainte putem defini:

```
int cube(int x)
{
  return x * sqr(x);
}
```

IMPORTANT: înainte de a folosi orice identificator (nume) în C,
el trebuie să fie *declarat* (trebuie să știm ce reprezintă)
$\Rightarrow$ Exemplele sunt corecte dacă `sqrf` respectiv `sqr` sunt definite
înainte de `discrim`, respectiv `cube` în program.

Un prim program C

```
int main(void)
{
  return 0;
}
```

Cel mai mic program: nu face nimic !

Orice program conține functia *main* și e executat prin apelarea ei (programul poate conține și alte funcții)

În acest caz: funcția nu are parametri (`void`)

`void` e un cuvânt cheie pentru tipul vid (fară nici un element)

În standard: `main` returnează sistemului de operare un cod întreg (convenție: 0 pt. terminare cu succes, $\neq 0$ pt. cod de eroare)

Un program comentat

```
/* Acesta este un comentariu */
int main(void)  // comentariu pana la capat de linie
{
  /* Acesta e un comentariu pe mai multe linii
    obisnuit, aici vine codul programului */
  return 0;
}
```

Programele pot conține comentarii, înscrise între `/*` și `*/`
sau începând cu `//` și terminându-se la capătul liniei

Orice conținut între aceste caractere nu are nici un efect asupra generării codului și execuției programului

Programele *trebuie* comentate
- pentru ca un cititor să le înțeleagă (alții, sau noi, mai târziu)
- ca documentație și specificație: funcționalitate, restricții, etc.
- ce reprezintă parametrii funcțiilor, rezultatul, variabilele,
- ce condiții trebuie îndeplinite, cum se comportă la eroare

Tipărirea (scrierea)

```
#include <stdio.h>
int main(void)
{
  printf("hello, world!\n");    // tipareste un text
  return 0;
}
```

`printf` (de la "print formatted"): o funcție standard

- nu este *instrucțiune* sau *cuvânt cheie*
- e apelată aici cu un parametru șir de caractere
- constantele șir de caractere: incluse între ghilimele `" "`

`\n` este notația pentru caracterul de linie nouă

Prima linie e o *directivă de preprocesare*, include fișierul `stdio.h` cu *declarațiile* funcțiilor standard de intrare / ieșire

Declarația = tip, nume, parametri: necesare pentru folosire

Implementarea (codul obiect, compilat): într-o bibliotecă din care compilatorul ia cele necesare la generarea programului executabil

Tipărirea de numere

```
#include <math.h>
#include <stdio.h>
int main(void)
{
  printf("cos(0) = ");
  printf("%f", cos(0));
  return 0;
}
```

```
#include <stdio.h>
int sqr (int x) { return x * x; }
int main(void)
{
  printf("2 ori -3 la patrat e ");
  printf("%d", 2 * sqr(-3));
  return 0;
}
```

Pentru a tipări valoarea unei expresii, `printf` ia două argumente:
– un șir de caractere (specificator de format):
  `%d` (întreg, *decimal*), `%f` (real, *floating point*)
– expresia, al cărei tip trebuie să fie compatibil cu cel indicat
(verificarea cade în sarcina programatorului !!!)

Secvențierea: instrucțiunile unei funcții se execută *una după alta*
Excepții: instrucțiunea `return` încheie execuția funcției
  (după ea nu se mai execută nimic)

Funcții definite pe cazuri

$$abs : \mathbb{Z} \rightarrow \mathbb{Z} \qquad abs(x) = \begin{cases} x & x \geq 0 \\ -x & \text{altfel} \end{cases}$$

Cu cele învățate pâna acum, nu putem defini această funcție în C.

Valoarea funcției nu e dată de o singură expresie, ci de una din două expresii diferite (x sau -x), depinzând de o condiție ($x \geq 0$)

$\Rightarrow$ În limbaj, trebuie să putem *decide* valoarea luată de expresie în funcție de valoarea unei *condiții* (adevărat/fals)

Operatorul condițional ? : în C

Expresia conditională în C are sintaxa: *condiție* `?` *expr1* `:` *expr2*
– dacă condiția e adevărată, se evaluează doar *expr1*, și întreaga expresie ia valoarea acesteia
– dacă e falsă, se evaluează doar *expr2* și întreaga expresie ia valoarea acesteia

```
int abs(int x)
{
  return x >= 0 ? x : -x;       // operator minus unar
}
```

Operatorii de comparație: == (egalitate), `!=` (diferit), <, <=, >, >=

IMPORTANT! Testul de egalitate în C e == și nu = simplu !!!

Funcția `abs` există ca funcție standard, declarată în `stdlib.h`

Funcții definite pe mai mult de două cazuri

$$sgn : \mathbb{Z} \to \{-1, 0, 1\} \qquad sgn(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

Nu putem transcrie funcția direct în C
(operatorul condițional permite doar decizia cu *două* ramuri
(adevărat/fals), nu cu mai multe condiții / ramuri)
$\Rightarrow$ trebuie să descompunem decizia asupra valorii lui x
$\Rightarrow$ *descompunerea în subprobleme* mai mici:
principiu *foarte important* în rezolvarea de probleme
Rescriem funcția cu o singură decizie în fiecare punct:

$$sgn(x) = \begin{cases} \text{dacă } x < 0 & -1 \\ \text{altfel } (x \geq 0) & \begin{cases} \text{dacă } x = 0 & 0 \\ \text{altfel } (x > 0) & 1 \end{cases} \end{cases}$$

Scrierea unei funcții pe mai multe cazuri în C

$$sgn(x) = \begin{cases} \text{dacă } x < 0 & -1 \\ \text{altfel } (x \geq 0) & \begin{cases} \text{dacă } x = 0 & 0 \\ \text{altfel } (x > 0) & 1 \end{cases} \end{cases}$$

```
int sgn (int x)
{
  return x < 0 ? -1
               : x == 0 ? 0 : 1;
}
```

Putem grupa arbitrar de mulți operatori `?` `:`
expr1 și *expr2* pot fi la rândul lor expresii condiționale
O expresie scrisă corect are un `:` pentru fiecare `?`

Descompunerea în subprobleme mai simple

Minimul a două numere reale se poate scrie simplu:

```
double min2(double x, double y) {
  return x < y ? x : y;
}
```

Pentru minimul a *trei* numere, comparațiile directe se complică:

$$min3(x,y,z) = \begin{cases} \text{dacă } x < y & \begin{cases} \text{dacă } x < z & \mathbf{x} \\ \text{altfel } (x \geq z) & \mathbf{z} \end{cases} \\ \text{altfel } (x \geq y) & \begin{cases} \text{dacă } y < z & \mathbf{y} \\ \text{altfel } (y \geq z) & \mathbf{z} \end{cases} \end{cases}$$

Se repetă structura funcției `min2` ⇒ putem gândi mai simplu:
Rezultatul e minimul între minimul primelor două și al treilea.
⇒ folosim direct funcția dinainte *fără a mai despărți* pe cazuri.

```
double min3(double x, double y, double z) {
  return min2(min2(x, y), z);    // sau min2(x, min2(y,z))
}
```

Să înțelegem: apelul de funcție

Programul dat calculează $x^6 = (x \cdot x^2)^2$

```
#include <stdio.h>
int sqr(int x) {
  printf("Patratul lui %d e %d\n", x, x*x);
  return x * x;
}
int main(void) {
  printf("2 la a 6-a e %d\n", sqr(2 * sqr(2)));
  return 0;
}
```

În ce ordine se scrie pe ecran ?

```
Patratul lui 2 e 4
Patratul lui 8 e 64
2 la a 6-a e 64
```

Să înțelegem: apelul de funcție

În C, transmiterea parametrilor la funcții se face *prin valoare*.

- se *evaluează* (calculează valoarea) toate argumentele funcției
- valorile se atribuie la *parametrii formali* (numele din def. fct.)
- apoi se începe execuția funcției cu aceste valori

Programul începe cu execuția lui `main`, deci apelul la `printf`

Funcția `printf` are nevoie de valoarea argumentelor sale.

- valoarea primului argument se știe (o *constantă șir*)
- pentru al doilea argument trebuie apelat: `sqr(2 * sqr(2))`
  - la rândul lui, `sqr` exterior are nevoie de valoarea argumentului `2 * sqr(2)`, pentru care trebuie apelat întâi `sqr(2)`

⇒ ordinea apelurilor: `sqr(2)`, apoi `sqr(8)`, apoi `printf` din `main`

Să înțelegem CORECT apelul de funcție

Cum NU se face în C (deși ne-am putea închipui ...)

Funcția NU începe execuția fără să aibă argumentele calculate
`printf` ar tipări `2 la puterea 6 e` , apoi îi trebuie valoarea
ar apela `sqr` exterior care scrie `Patratul lui`, apoi îi trebuie `x`
ar apela `sqr(2)` care scrie `Patratul lui 2 e 4`, returnează `4`,
etc.

NU se substituie *expresiile* argument pentru parametrii funcției
din `printf` s-ar apela `sqr` exterior cu *expresia* `2 * sqr(2)`
pentru `(2*sqr(2))*(2*sqr(2))` s-ar apela `sqr(2)` de două ori

⇒ În C, o funcție calculează numai cu *valori*, niciodată cu *expresii*

Recursivitate

Recursivitate: definiție, exemple

Din matematică cunoaștem *șiruri recurente*:

progresie aritmetică:

$$\begin{cases} x_0 = b & \text{(adică: } x_n = b \text{ pentru } n = 0) \\ x_n = x_{n-1} + r & \text{pentru } n > 0 \end{cases}$$

Exemplu: $1, 4, 7, 10, 13, \dots$ ($b = 1$, $r = 3$)

progresie geometrică:

$$\begin{cases} x_0 = b & \text{(adică: } x_n = b \text{ pentru } n = 0) \\ x_n = x_{n-1} \cdot r & \text{pentru } n > 0 \end{cases}$$

Exemplu: $3, 6, 12, 24, 48, \dots$ ($b = 3$, $r = 2$)

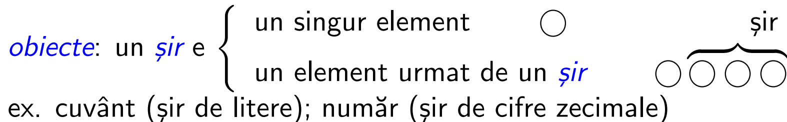
Nu calculează x_n *direct*, ci *din aproape în aproape*, folosind x_{n-1}.

O noțiune e *recursivă* dacă e *folosită în propria sa definiție*.

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

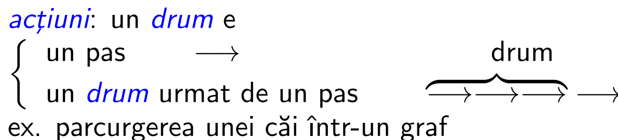
Recursivitate: definiție, exemple

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



obiecte: un *șir* e
- un singur element
- un element urmat de un *șir*

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



acțiuni: un *drum* e
- un pas $\longrightarrow$
- un *drum* urmat de un pas

ex. parcurgerea unei căi într-un graf

O *expresie*:
- număr (7)
- identificator (x)
- *expresie* + *expresie*
- *expresie* – *expresie*
- (*expresie*), etc

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;
}
```

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

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

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

Mecanismul apelului recursiv

Funcția `pwr` face două calcule:

– un *test* (`n == 0` ? *cazul de bază* ?) dacă da, returnează 1

– altfel, o *înmulțire*; operandul drept necesită 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 (≥ 0 dar 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

se pot utiliza doar noțiuni deja definite

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

Factorialul, calculat recursiv

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

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

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

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

E nevoie de loc în *stiva calculatorului* pentru toate apelurile în curs (parametri, variabile locale, adresă de revenire, regiștri salvați)
$\Rightarrow$ ineficient, necesită mult loc pe stivă

Factorialul: recursivitate cu revenire (tail recursion)

Reordonăm înmulțirile: $5! = ((((1 \cdot 5) \cdot 4) \cdot 3) \cdot 2) \cdot 1$

apel recursiv cu un *rezultat parțial* (acumulator) ca *argument*

în cazul de bază ($n = 0$), *rezultatul e complet*, îl returnăm

`res`: 1, 5 (`1*5`), 20 (`5*4`), 60 (`20*3`), 120 (`60*2`), 120 (`120*1`)

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

Funcția auxiliară scrisă calculează *res·n!* ⇒ luăm `res=1`

```
unsigned fact(unsigned n) { return fact2(1, n); }
```

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 revenire

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

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

Calculul: înainte de apel, actualizând rezult. parțial transmis ca argument

Revenire: returnează direct valoarea *tail recursion*, recursivitate cu revenire

În cazul 2 (*tail-recursive*): nici un calcul la revenire

⇒ nu necesităînregistrarea de apel (adresă, parametri) pe stivă

⇒ compilatorul *transformă recursivitatea în iterație (eficient)*

Adunarea numerelor, în variantă recursivă

Calculul sumei întregilor între a și b:
$a + (a + 1) + \ldots + (b - 1) + b$.

Cazul de bază: $a > b$, intervalul $[a, b]$ e vid $\Rightarrow$ suma e 0.

Pasul recursiv: suma e primul număr, plus suma intervalului rămas (sau: ultimul număr, plus suma intervalului dinainte).

$$sum_all(a, b) = a + sum_all(a + 1, b) \quad \text{(dacă } a \geq b\text{)}$$

```
int sum_all(int a, int b)
{
  return a > b ? 0 : a + sum_all(a+1, b);
}
```

Sau, rescriind cu un parametru suplimentar care acumuleaza suma:

```
int sum_acc(int a, int b, int r)
{
  return a > b ? r : a + sum_all(a+1, b, r+a);
}
int sum_all2(int a, int b) { return sum_acc(a, b, 0); }
```

Alte exemple: 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",  // %u = unsigned
                         cmmdc(20, 8));
  return 0;
}
```

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);
```