

2 Recursivitate

În limbajele de programare, recursivitatea e un concept fundamental care le extinde în mod esențial *puterea de exprimare (expresivitatea)*: recursivitatea permite scrierea unor programe care nu s-ar putea exprima doar cu noțiunile fundamentale de *secvențiere* și *decizie* prezentate până acum. Pentru rezolvarea practică a problemelor, recursivitatea e foarte importantă deoarece permite să descriem soluția unei probleme complexe folosind una sau mai multe probleme de același tip, dar mai simple. Ea e astfel strâns legată de principiul *descompunerii în subprobleme (divide et impera)* în proiectarea soluțiilor.

2.1 Definiție și exemple

O noțiune e *recursivă* dacă e folosită în propria sa definiție. Cunoaștem un exemplu din matematică: *șirurile recurente*, unde un termen al șirului e definit printr-o relație în raport cu termenii anteriori. Exemple sunt:

- progresia aritmetică: $x_0 = a, x_n = x_{n-1} + p$ pentru $n > 0$.
- progresia geometrică: $x_0 = b, x_n = q \cdot x_{n-1}$ pentru $n > 0$.
Acestea sunt *recurențe de ordinul I*, în care termenul definit depinde doar de termenul imediat anterior. Alte recurențe mai complexe sunt:
- șirul lui Fibonacci: $F_0 = F_1 = 1, F_n = F_{n-1} + F_{n-2}$ pentru $n \geq 2$ (un șir recurent de ordinul II)
- coeficienții binomiali: $C_n^0 = C_n^n = 1$ pentru $n \geq 0, C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$ pentru $0 < k < n$. Acest exemplu definește o mărime în funcție de *doi* parametri (indici), însă de asemenea printr-o relație de recurență în raport cu termeni de indici mai mici.

2.2 O funcție recursivă în C

În toate exemplele date de șiruri recurente, definiția conține două părți: *cazul de bază* pentru primul termen (sau primii câțiva), și relația recursivă propriu-zisă pentru *termenul general*. Aceasta sugerează că putem folosi operatorul condițional pentru a exprima o definiție recursivă, similar cu funcțiile definite anterior pe mai multe cazuri.

Considerăm ca exemplu funcția putere (cu bază reală și exponent natural), care poate fi privită recursiv ca progresie geometrică, indicele fiind exponentul: puterea de exponent n (termenul de ordin n) este baza (rația) înmulțită cu puterea de exponent $n-1$. Grupăm în definiție cazul de bază (termenul inițial, 1) și termenul general în felul următor:

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

Având o definiție pe două variante, funcția se poate exprima în C cu operatorul condițional ca și exemplele nerecursive dinainte:

```
float pwr(float x, unsigned n)
{
    return n==0 ? 1
           : x * pwr(x, n-1);
}
```

Pentru exponent am folosit tipul `unsigned` (cuvânt cheie în limbajul C), corespunzând numerelor naturale (nenegative); orice valoare diferită de zero e deci pozitivă și tratată corect pe ramura "altfel".

Pentru scrierea funcției `pwr` nu au fost necesare facilități noi de limbaj. Esențial e doar ca limbajul să permită ca în corpul unei funcții să fie *apelată chiar această funcție* (știm că e permisă apelarea unei funcții care e deja *declarată*). În limbajul C, după ce a fost scris *antetul* funcției ca parte a *definiției* ei complete se cunosc deja numele funcției, tipul și parametrii ei. Antetul reprezintă deci o *declarație* a funcției (chiar înainte de a fi fost scris corpul ei), ceea ce e suficient pentru a permite apelul recursiv.

2.3 Mecanismul apelului de funcție. Apelul recursiv

Deși scrierea acestui prim exemplu recursiv nu a necesitat elemente noi de limbaj, pentru a înțelege corect recursivitatea sunt necesare mai multe detalii despre mecanismul apelului de funcție. Începem cu

un exemplu nerecursiv: funcția `int sqr(int x) { return x * x; }` și expresia `sqr(3 * sqr(2))`.

Expresia e un apel la funcția `sqr`. Înainte de apel, trebuie evaluat argumentul funcției, pentru a cunoaște valoarea a cărei pătrat trebuie calculat. Argumentul e un produs în care unul din factori e el însuși o expresie apel de funcție. Ca atare, din întreaga expresie se evaluează întâi `sqr(2)`, apoi se înmulțește 3 cu rezultatul (4), iar cu valoarea 12 se efectuează al doilea apel la `sqr`, cu rezultatul 144.

Deși apelul exterior la `sqr` conține o subexpresie cu un apel la aceeași funcție, expresia nu are caracter recursiv, întrucât valoarea funcției `sqr` e calculată direct din valoarea parametrului transmis. Valoarea lui `sqr(2)` e necesară ca *argument* pentru apel, nu în corpul funcției ca în definițiile recursive.

Apelul de funcție Evaluarea unui apel de funcție e declanșată atunci când valoarea funcției e necesară în evaluarea unei expresii (inclusiv în execuția unei instrucțiuni de forma *expresie ;*). Ea se efectuează în următorii pași:

- se evaluează toate expresiile care constituie argumentele funcției. Deci orice apeluri de funcții care apar în argumente se efectuează *înainte* de apelul funcției considerate.
- valorile argumentelor se atribuie parametrilor formali din antetul funcției, cu conversiile necesare de tip (de exemplu întreg–real). Compatibilitatea de tip a expresiilor argument e verificată deja la compilare, dacă se cunosc tipurile parametrilor formali – un motiv în plus pentru care se cere ca o funcție să fie *declarată* înainte de a fi *folosită*.
- se execută corpul funcției, cu parametrii formali având valori inițiale ca mai sus. La întâlnirea instrucțiunii `return`, execuția funcției se încheie cu valoarea obținută prin evaluarea expresiei date.
- execuția programului revine la locul de apel, unde valoarea returnată de funcție e folosită.

Transmiterea parametrilor În limbajul C transmiterea parametrilor la funcții se face *prin valoare*: În momentul apelului, parametrii formali iau *valoarea* argumentelor (care au fost evaluate); ei *nu* sunt substituiți cu *expresiile* argumentelor. În consecință, pentru apelul discutat mai sus, în instrucțiunea `return` se va evalua expresia `12 * 12` și nu `(3 * sqr(2)) * (3 * sqr(2))`. Altfel spus, o funcție lucrează cu *valori* (numerice), nu cu expresii simbolice. Expresia `3 * sqr(2)` se evaluează o singură dată, înainte de apelul sintactic exterior la `sqr`, și deci apelul la `sqr(2)` e deja încheiat în momentul celui de-al doilea apel, `sqr(12)`. Acest fapt poate fi vizualizat augmentând funcția `sqr` cu o instrucțiune de tipărire, o practică utilă în urmărirea execuției programelor.

```
#include <stdio.h>
int sqr(int x)
{
    printf("calculam patratul lui %d\n", x);
    return x * x;
}
int main(void)
{
    printf("sqr(3 * sqr(2)) = %d\n", sqr(3*sqr(2)));
    return 0;
}

calculam patratul lui 2
calculam patratul lui 12
sqr(3 * sqr(2)) = 144
```

Rezultatul rulării programului (prezentat sub textul sursă) evidențiază și pentru apelul `printf` din `main` evaluarea argumentelor înainte de începerea execuției funcției. Evaluarea argumentului al doilea produce cele două linii tipărite în apelurile la `sqr`. Aceste linii apar înainte de a scrie chiar și porțiunea de text obișnuit din primul argument (formatul), scriere în care constă tocmai execuția funcției `printf`.

Returnarea valorilor La întâlnirea instrucțiunii `return`, execuția funcției se încheie; orice alte instrucțiuni care urmează în corpul funcției nu se mai execută. Dacă execuția unei funcții se termină prin atingerea ultimei acolade `}` fără a executa o instrucțiune `return`, iar programul utilizează valoarea funcției, efectul e *nedefinit* (programul se comportă imprevizibil). O funcție care nu prevede în orice situație o valoare returnată e scrisă eronat sub aspect logic.

Apelul recursiv Discutăm mecanismul apelului recursiv luând ca exemplu calculul lui 5^3 cu funcția putere. În apelul `pwr(5, 3)` ($x = 5, n = 3$), expresia condițională conduce la evaluarea lui $5 * \text{pwr}(5, 2)$. Aceasta necesită un nou apel la `pwr`, de data aceasta cu parametrii $x = 5, n = 2$. Procesul se repetă cu `pwr(5, 1)` până la apelul `pwr(5, 0)` pentru care valoarea se calculează direct: 1. Din acest apel se revine în locul unde a fost făcut: la evaluarea lui $5 * \text{pwr}(5, 0)$ care poate fi efectuată acum. Apelul `pwr(5, 1)` returnează valoarea 5 folosită în calculul $5 * \text{pwr}(5, 1)$ pentru valoarea lui `pwr(5, 2)`. În final, această valoare, 25, e folosită în expresia $5 * \text{pwr}(5, 2)$ pentru a calcula valoarea 125 a lui `pwr(5, 3)`.

```

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

```

Urmărind secvența de apel, rezultă că la un moment dat pot fi în execuție *mai multe apeluri diferite* la aceeași funcție. Fiecare apel reprezintă o *instanță* (copie) distinctă a funcției, cu *propriile valori de parametri*, cele primite în momentul apelului. La fel se întâmplă și în calculul pe hârtie, prin “desfășurarea” formulei de recurență: pentru a calcula pe 5^3 , înlocuim pe x cu 5 și n cu 3. Trebuie calculat 5^2 : aplicând din nou formula înlocuim pe n cu 2; aceasta nu afectează însă instanța inițială a problemei (5^3), unde n este în continuare 3.

În exemplul dat, recursivitatea are o structură liniară: fiecare apel recursiv generează un singur nou apel, până la oprirea pentru cazul de bază. În acel moment, sunt active toate apelurile, în număr de 4. Revenirea se face *în ordine inversă* față de cea de apel: din fiecare apel se revine în instanța care a efectuat apelul. Aici, execuția se reia în contextul dinainte de apel: adică din locul în care a fost făcut apelul (unde e folosită valoarea returnată), și cu acele valori ale parametrilor corespunzând instanței respective.

Ca la orice apel de funcție, informația se transmite spre funcția apelată prin parametri, și înapoi spre locul de apel (funcția *apelantă*) prin rezultat. În exemplul dat se creează un lanț în care la revenire, valoarea returnată de fiecare apel e folosită în instanța apelantă pentru calculul propriului rezultat, care e transmis mai departe înapoi spre locul de apel. Rezultatul final e astfel efectul unui calcul în care câte un pas e efectuat de fiecare din instanțele apelate. Aceasta e esența recursivității și în același timp puterea ei în rezolvarea de probleme: ea permite exprimarea *indirectă* a soluției prin pași simpli, din aproape în aproape, fără a necesita formularea directă a unei soluții complexe.

2.4 Elementele unei definiții recursive

Într-o definiție recursivă corectă se pot identifica următoarele componente:

- *Cazul de bază.* Tratează situațiile (cele mai simple) în care noțiunea recursivă e definită *direct*. Exemple: pentru șirurile recurente, primul termen (sau mai mulți, la recurențele de ordin > 1) pentru liste, cea vidă, sau cea cu un element; pentru expresii, constantele și identificatorii
- *Relația de recurență:* partea propriu-zis recursivă a definiției, în care noțiunea definită apare și în corpul definiției. Exemple: formulele de recurență pentru șiruri; ramura de definiție ”o listă e un element urmat de o listă”; pentru expresii, variantele de definiție cu expresie între paranteze, apel de funcții (cu parametri expresii) și cele cu expresii compuse cu operatori
- *Terminarea recursivității.* Dacă definiția urmează o ramură care conține din nou noțiunea definită, atunci definiția trebuie aplicată din nou. O noțiune e corect definită recursiv dacă acest proces se oprește întotdeauna. Pentru a fi riguroasă, o definiție recursivă trebuie însoțită de o demonstrație că aplicarea definiției se oprește după un număr finit de pași.

Rezultă că o definiție recursivă nu poate fi corectă fără un caz de bază, pentru că nu se ajunge niciodată la un punct unde noțiunea poate fi definită direct. Cazul de bază și relația recursivă sunt de fapt *alternative* ale aceleiași definiții. Acest lucru e explicit în regulile sintactice care definesc

construcții de limbaj cum ar fi expresiile. Pentru un șir recurent putem evidenția aceasta folosind acolade: $x_n = \begin{cases} a & n = 0 \\ x_{n-1} + p & n > 0 \end{cases}$ și transcrie apoi ușor în program.

Pentru terminarea recursivității, cel mai uzual argument folosește o măsură (cantitate) care descrește la fiecare aplicare a definiției, până atinge o valoare pentru care definiția e dată direct. La șiruri recurente, această cantitate e chiar indicele n al termenului general x_n .

2.5 Alte exemple de recursivitate

Înșiruirea văzută recursiv Și notiuni din afara matematicii, uneori foarte simple, se pretează la definiții recursive. Un tipar de definiții des întâlnit se bazează pe faptul că iterația (repetiția) poate fi definită prin recursivitate. Putem defini astfel:

Un șir (secvență, listă) e fie un element, fie un șir urmat de un element.

Uneori e util să includem în definiție șirul vid, care apare natural în diverse operații (ca element neutru la concatenare; la inițializarea sau după ștergerea tuturor elementelor unei liste, etc.):

Un șir e fie un șir vid, fie un element urmat de un șir.

Cele două variante diferă atât prin cazul de bază (un element sau zero), cât și prin poziția noțiunii recursive în definiție: prima variantă e *recursivă la stânga* (noțiunea definită recursiv "șir" e *prima* în rescrierea "șir urmat de un element"), iar a doua e *recursivă la dreapta*, deoarece în expandarea "element urmat de un șir" noțiunea definită "șir" e pe ultima poziție. Tiparele de recursivitate la stânga și la dreapta conduc la prelucrări diferite în program, pe care le studiem și comparăm în continuare.

Recursivitatea în sintaxa limbajelor Recursivitatea apare natural în definirea precisă a sintaxei limbajelor de programare. Multe elemente de limbaj au în componență *repetiția*, care poate fi exprimată recursiv. Astfel, antetul unei funcții poate fi definit (în limita celor prezentate până acum) ca:

```
antet-funcție ::= tip identificador ( parametri )
parametri   ::= void | lista-param
lista-param  ::= tip identificador | tip identificador , lista-param
```

Am folosit convențional simbolurile ::= pentru *definiție* și | pentru *alternativă*. Acest mod de a descrie regulile sintactice, adică *gramatica* unui limbaj se numește formă Backus-Naur (BNF).

Putem defini recursiv și altă noțiune fundamentală, expresia. Din cele prezentate până acum:

```
expresie     ::= constantă | identificador | identificador ( argumente )
              | - expresie | expresie operator-binar expresie
              | expresie ? expresie : expresie | ( expresie )
argumente    ::= ε | lista-argumente
lista-argumente ::= expresie | expresie , lista-argumente
```

unde ϵ denotă convențional alternativa cu conținut *vid* (fără simboluri de limbaj), aici pentru apeluri de forma `funcție()`, fără argumente între paranteze.

Pentru o definiție riguroasă, trebuie să precizăm că orice construcție de limbaj trebuie definită printr-un număr *finit* de aplicări ale regulilor. Astfel, $-(2 + 3)$ e o expresie: se pot aplica pe rând regulile $expresie ::= - expresie$, $expresie ::= (expresie)$, $expresie ::= expresie + expresie$, și de două ori regula $expresie ::= constantă$.

2.6 Două tipare de calcul recursiv

Să examinăm un alt exemplu tipic de calcul recursiv, factorialul.

Avem: $n! = \begin{cases} 1 & n = 0 \\ (n-1)! \cdot n & \text{altfel } (n > 0) \end{cases}$ și putem transcrie direct în C:

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

În evaluarea lui `fact` pentru $n > 0$, ultima operație efectuată e înmulțirea cu n , restul calculelor fiind efectuate anterior în apelul `fact(n-1)` (și celelalte apeluri recursive care rezultă din acesta). Ordinea calculelor e indicată de paranteze în expresia $n! = (((1 \cdot 1) \cdot 2) \cdot \dots) \cdot (n - 1) \cdot n$.

Expandând definiția pentru $n \geq 2$, obținem $n! = ((n - 2)! \cdot (n - 1)) \cdot n$. Înainte de evaluarea lui `fact(n-2)` știm că în produs apar atât n cât și $n-1$, dar așa cum e scrisă funcția, nu se efectuează direct înmulțirea lor: se apelează întâi `fact(n-2)`, rezultatul e înmulțit cu $n-1$ în cadrul apelului `fact(n-1)`, și doar în final se face înmulțirea cu n , pentru rezultatul lui `fact(n)`.

Pornind de la această observație, rescriem factorialul folosind asociativitatea înmulțirii, pentru a efectua cât mai multe înmulțiri îndată ce factorii devin disponibili: $n! = 1 \cdot (2 \cdot (\dots ((n - 1) \cdot n)))$

Transcriind în C, am dori să efectuăm înmulțirea $(n-1) \cdot n$ în cadrul apelului funcției pentru $n-1$, înainte de a calcula recursiv factorialul pentru $n-2$. În apelul pentru $n-2$ s-ar putea înmulți apoi rezultatul lui $(n-1) \cdot n$ cu $n-2$, etc. Pentru aceasta, avem nevoie să transmitem la fiecare apel recursiv pe lângă valoarea lui n și rezultatul înmulțirilor deja efectuate. Obținem astfel:

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

Parametrul `r` reprezintă rezultatul parțial calculat. La fiecare apel recursiv, el e înmulțit cu valoarea curentă a lui n și rezultatul e transmis mai departe la apelul pentru $n-1$. Când $n==0$, toate înmulțirile au fost deja efectuate; rezultatul se găsește acumulat în `r` și poate fi returnat. Pe ramura recursivă, la revenire nu se mai efectuează nici un calcul: valoarea provenită din apelul recursiv e deja rezultatul complet și e returnată direct mai departe la apelant.

Din primul apel pentru n , rezultatul parțial pe care dorim să-l transmitem spre apelul pentru $n-1$ e tot valoarea n . Deci, inițial, `r` ar trebui să fie 1, și pentru a calcula pe $n!$ vom apela `fact_r(n, 1)`. De fapt, am definit o funcție mai generală: `fact_r(n, r)` calculează pe $r \cdot n!$.

Pentru a nu complica utilizatorul cu parametrul suplimentar datorat modului de calcul, definim funcția `fact2` cu un singur parametru. Aceasta doar “împachetează” apelul inițial `fact_r(n, 1)`:

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

Pentru factorial, putem alege între cele două variante de scriere. În prima, calculul se face la *revenirea* din apelurile recursive, iar acestea nu au nevoie de vreun rezultat parțial calculat anterior. În a doua, rezultatul parțial e transmis în adâncime ca parametru, și actualizat *înainte* de fiecare apel recursiv.

Există însă situații în care parte din prelucrare se efectuează înainte de fiecare apel recursiv, fiind *necesar* să transmitem “în jos” la înaintarea în recursivitate valori ce vor fi folosite ulterior în calcule.

Inversarea cifrelor unui număr Scriem o funcție care ia un întreg fără semn și-l transformă în numărul cu aceleași cifre zecimale dar în ordine inversă. Scriem soluția pornind de la un exemplu: 1472. Ultima cifră, 2, devine prima cifră a rezultatului. Punem ultima cifră rămasă din 147 după 2, obținând $27 = 2 \cdot 10 + 7$. Din numărul rămas, 14, plasăm ultima cifră după 27, obținând $27 \cdot 10 + 4 = 274$, etc.

În cuvinte, pasul recursiv de prelucrare poate fi exprimat: rezultatul inversării, dacă a mai rămas de inversat n , iar din inversarea ultimelor cifre s-a obținut deja v , e același cu rezultatul inversării lui $n/10$, cu valoarea intermediară $10 \cdot v + n \bmod 10$. Deși enunțul problemei are un singur parametru, soluția recursivă obținută manipulează două cantități, deci scriem o funcție recursivă cu doi parametri. Prelucrarea se oprește când n e 0 (nu mai sunt cifre de inversat), iar inițial, v e valoarea fără nici o cifră, deci tot 0; astfel, pentru prima cifră c , expresia $10 \cdot 0 + c$ dă valoarea dorită c .

```
#include <stdio.h>
unsigned revnum_r(unsigned n, unsigned r)
{
    return n == 0 ? r : revnum_r(n / 10, 10 * r + n % 10);
}

unsigned revnum(unsigned n) { return revnum_r(n, 0); }
```

```
int main(void)
{
    printf("%u\n", revnum(1472));    // %u pt. tiparire unsigned
    return 0;
}
```

Dorim ca soluție o funcție cu un singur parametru, ca în enunț, pentru a nu complica utilizatorul cu un parametru suplimentar pentru valoarea intermediară. Am scris astfel funcția `revnum` care apelează funcția recursivă `revnum_r(n, 0)` cu valoarea inițială necesară pentru al doilea parametru.

Cel mai mare divizor comun Algoritmul lui Euclid pentru calculul celui mai mare divizor comun a doi întregi pozitivi e un exemplu clasic de algoritm exprimat recursiv, în care o problemă e rezolvată prin reducerea la o instanță mai simplă a aceleiași probleme. Exprimat informal, algoritmul e:

- dacă numerele sunt egale, rezultatul e chiar valoarea lor comună
- altfel, se scade cel mai mic număr din cel mai mare, și se repetă procedura cu noile numere.

Exprimarea din urmă (“se repetă procedura”) indică abordarea recursivă: soluția se obține rezolvând aceeași problemă pentru valori noi ale numerelor (mai mici, deci după un număr finit de pași se ajunge la cazul de bază).

Putem scrie deci pe cazuri:

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

și transcrie direct în C:

```
unsigned cmmdc(unsigned a, unsigned b)
{
    return a == b ? a
        : a > b ? cmmdc(a - b, b)
        : cmmdc(a, b - a);
}
```

Deși am transpus direct din cuvinte în formula recursivă și apoi în cod, a rămas netratat un aspect: enunțul inițial e dat pentru numere *pozitive*, iar tipul `unsigned` permite și valoarea 0. Apelarea (chiar și accidentală) a funcției scrise mai sus cu un parametru nul va duce la o secvență infinită de apeluri recursive, deoarece scăzând 0 celălalt număr nu se modifică și reluăm *același* apel (până când, în funcție de mediul de rulare, programul se va termina probabil forțat epuizând resursele de memorie).

Este important ca funcțiile pe care le scriem să fie *robuste* și să nu producă erori neprevăzute și catastrofe. Ca atare, rescriem funcția ținând cont că 0 e divizibil cu orice număr, și deci $cmmdc(a, 0) = cmmdc(0, a) = a$:

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

În această scriere, cazul $a = b \neq 0$ va intra pe ultima ramură, având ca efect apelul $cmmdc(a, 0)$ care va returna a .

2.7 Calculul recursiv al seriilor

Calculul sumei parțiale a unei serii se pretează natural la o exprimare recursivă. Notând cu t_n termenul general, și $s_n = \sum_{k=0}^n t_k$, obținem imediat pentru termenul general: $s_n = s_{n-1} + t_n$ (pentru $n \geq 1$), și deci:

$$s_n = \begin{cases} t_0 & n = 0 \\ s_{n-1} + t_n & \text{altfel } (n > 0) \end{cases}$$

Având o formulă de calcul direct pentru termenul t_n al seriei (exprimat deci ca funcție de n), putem transforma direct formula de mai sus într-o funcție recursivă s , care apelează pentru calcul funcția t . Pentru exemplul simplu $t_n = 1/n$ (pentru $n > 1$, iar $t_0 = 0$), putem scrie următorul program:

```
#include <math.h>
#include <stdio.h>

double t(unsigned n)
{
    return n == 0 ? 0 : 1.0/n;
}

double s(unsigned n)
{
    return n == 0 ? t(0) : s(n-1) + t(n);
}

int main(void)
{
    printf("Constanta lui Euler e aprox. %f\n", s(1000)-log(1000));
    return 0;
}
```

Din matematică, știm că seria $\sum_n 1/n$ are sumă infinită, și crește aproximativ ca logaritmul natural al lui n . Mai precis,

$$\lim_{n \rightarrow \infty} \left(\sum_{k=1}^n 1/k - \ln n \right) = \gamma \simeq 0.5772\dots$$

Aproximația calculată de program are primele 3 zecimale exacte.

Tipul `double` folosit în program reprezintă, ca și `float`, numere reale, dar cu precizie mai bună (de aici și numele), și e recomandabil în calcule pentru a micșora acumularea erorilor de rotunjire. Este tipul standard folosit de funcțiile matematice de bibliotecă declarate în `math.h` (cum e și funcția `log` pentru logaritmul natural) și tipul implicit pentru constantele reale. Scrierea `1.0/n` pentru termenul matematic $1/n$ e necesară pentru a obține *împărțire reală*: operandul `1.0` fiind real, va fi convertit implicit și întregul `n`. Altfel, $1/n$ ar fi însemnat împărțire întreagă, cu rest, și valoarea 0 pentru $n > 1$.

Desigur că în acest program s-ar fi putut scrie mai simplu, direct

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

Varianta prezentată evidențiază însă forma generală a funcției s pentru suma exprimată recursiv, și programul poate fi adaptat la altă serie prin simpla înlocuire a funcției t .

E valabilă aceeași observație generală făcută întâi pe exemplul factorialului despre cele două variante de definire a calculului recursiv. Funcția s așa cum a fost scrisă mai sus efectuează calculul *la revenirea* din recursivitate, ultima adunare fiind $s_{n-1} + t_n$, corespunzând unei grupări a calculelor de forma: $s_n = (((t_0 + t_1) + t_2) + \dots) + t_n$.

Cealaltă alternativă o constituie transmiterea ca parametru suplimentar a unui rezultat parțial deja calculat (inițial zero), la care se acumulează în fiecare pas termenul curent. Astfel, termenii sunt adunați efectiv în ordine inversă: $s_n = t_0 + (t_1 + (\dots + (t_{n-1} + t_n)))$. Funcția se scrie:

```
double s2(unsigned n, double res)
{
    return n == 0 ? res + t(0) : s2(n-1, res + t(n));
}
```

Pentru a păstra aceeași formă naturală cu un singur parametru a funcției folosite direct de programator, redefinim funcția `s`, apelând pe `s2` cu valoarea 0 ca sumă parțială deja calculată:

```
double s(unsigned n) { return s2(n, 0); }
```

Acest cod poate înlocui funcția `s` din programul anterior, păstrând funcția `t`.

2.8 Calculul de aproximări cu o precizie dată

Majoritatea exemplelor de calcul numeric recursiv date până acum au aceeași structură: se calculează termenul unui șir definit printr-o relație de recurență, iar numărul de apeluri recursive necesar pentru calcul e determinat de ordinul (indicele) termenului, dat la primul apel.

În matematică și practică întâlnim însă adesea cazuri se dorește un calcul efectuat cu o anumită precizie: se generează o secvență de aproximări, iar când aproximarea curentă a atins precizia dorită, calculul se oprește.

Dăm ca exemplu o metodă de aproximație pentru calculul rădăcinii pătrate \sqrt{x} . O cunoscută formulă matematică (poate fi derivată din metoda lui Newton, dar în acest caz particular e mult mai veche) ne dă secvența de aproximări: $a_{n+1} = (a_n + x/a_n)/2$, cu o aproximare inițială arbitrară (de exemplu $a_0 = 1$).

Elementul cheie al soluției e formularea problemei pentru a-i identifica și scoate în evidență caracterul recursiv: mai precis, *parametrii* și *cazul de bază* (oprirea din secvența de apeluri recursive). Condiția de oprire nu e dată de aproximarea inițială, ci, după cum rezultă din enunț, de *precizia* atinsă. Pentru a o calcula, avem nevoie de aproximarea curentă, care devine astfel *parametru al problemei* (cu valoarea inițială dată de a_0).

Putem atunci enunța soluția în cuvinte în felul următor: funcția căutată returnează o aproximație de precizie dată a lui \sqrt{x} , știind o aproximație curentă a_n dată de asemenea ca parametru. Dacă aproximația curentă e suficient de bună, poate fi returnată (cazul de bază). Dacă nu, rezultatul va fi dat de aceeași funcție de calcul, apelată tot pentru x , dar cu o aproximație curentă mai bună, dată de a_{n+1} .

```
#include <math.h>
#include <stdio.h>

double rad(double x, double a)
{
    return fabs(a - x/a) < 1e-6 ? a : rad(x, .5*(a + x/a));
}

int main(void)
{
    printf("%f\n", rad(2, 1));
    return 0;
}
```

Funcția standard `fabs`, declarată în `math.h` returnează valoarea absolută a unui număr real (`double`), spre deosebire de funcția `abs` (din `stdlib.h`) aplicabilă doar la numere întregi. Logica comparației este următoarea: a și x/a se află întotdeauna de o parte și de alta a valorii exacte \sqrt{x} . Deci, dacă intervalul dintre ele e mai mic decât precizia dorită (în exemplul dat, 10^{-6}), oricare din ele va fi o aproximație de precizie suficientă.