2 Recursivitate

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

2.1 Definiţie şi exemple

O noţiune e *recursivă* dacă e folosită în propria sa definiţie. Cunoaştem un exemplu din matematică: *şirurile recurente*, unde un termen al şirului e definit printr-o relaţie 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 *recurenţe de ordinul I*, în care termenul definit depinde doar de termenul imediat anterior. Alte recurenţe mai complexe sunt:
- şirul lui Fibonacci: $F_0 = F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ pentru $n \geq 2$ (un şir recurent de ordinul II)
- coeficienţii 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 defineşte o mărime în funcţie de *doi* parametri (indici), însă de asemenea printr-o relaţie de recurenţă în raport cu termeni de indici mai mici.

2.2 O functie recursivă în C

În toate exemplele date de şiruri recurente, definiţia conţine două părţi: *cazul de bază* pentru primul termen (sau primii câţiva), şi relaţia recursivă propriu-zisă pentru *termenul general*. Aceasta sugerează că putem folosi operatorul condiţional pentru a exprima o definiţie recursivă, similar cu funcţiile definite anterior pe mai multe cazuri.

Considerăm ca exemplu funcţia putere (cu bază reală şi exponent natural), care poate fi privită recursiv ca progresie geometrică, indicele fiind exponentul: puterea de exponent n (termenul de ordin n) este baza (raţia) înmulţită cu puterea de exponent $n-1$. Grupăm în definiţie cazul de bază (termenul iniţial, 1) şi 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 definiţie pe două variante, funcţia se poate exprima în C cu operatorul condiţional ca şi 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 şi tratată corect pe ramura "altfel".

Pentru scrierea funcţiei `pwr` nu au fost necesare facilităţi noi de limbaj. Esenţial e doar ca limbajul să permită ca în corpul unei funcţii să fie *apelată chiar această funcţie* (ştim că e permisă apelarea unei funcţii care e deja *declarată*). În limbajul C, după ce a fost scris *antetul* funcţiei ca parte a *definiţiei* ei complete se cunosc deja numele functiei, tipul şi parametrii ei. Antetul reprezintă deci o *declaraţie* a funcţiei (chiar înainte de a fi fost scris corpul ei), ceea ce e suficient pentru a permite apelul recursiv.

2.3 Mecanismul apelului de funcţie. Apelul recursiv

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

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

Expresia e un apel la funcţia `sqr`. Înainte de apel, trebuie evaluat argumentul funcţiei, pentru a cunoaşte valoarea a cărei pătrat trebuie calculat. Argumentul e un produs în care unul din factori e el însuşi o expresie apel de funcţie. Ca atare, din întreaga expresie sa evaluează întâi `sqr(2)`, apoi se înmulţeşte 3 cu rezultatul (4), iar cu *valoarea* 12 se efectuează al doilea apel la `sqr`, cu rezultatul 144.

Deşi apelul exterior la `sqr` conţine o subexpresie cu un apel la aceeaşi funcţie, expresia nu are caracter recursiv, întrucât valoarea funcţiei sqr e calculată direct din valoarea parametrului transmis. Valoarea lui `sqr(2)` e necesară ca *argument* pentru apel, nu în corpul funcţiei ca în definiţiile recursive.

Apelul de funcţie Evaluarea unui apel de funcţie e declanşată atunci când valoarea funcţiei e necesară în evaluarea unei expresii (inclusiv în execuţia unei instrucţiuni de forma *expresie* `;`). Ea se efectuează în următorii paşi:

- se evaluează toate expresiile care constituie argumentele funcţiei. Deci orice apeluri de funcţii care apar în argumente se efectuează *înainte* de apelul funcţiei considerate.
- valorile argumentelor se atribuie parametrilor formali din antetul funcţiei, 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 funcţie să fie *declarată* înainte de a fi *folosită*.
- se execută corpul funcţiei, cu parametrii formali având valori iniţiale ca mai sus. La întâlnirea instrucţiunii `return`, execuţia funcţiei se încheie cu valoarea obţinută prin evaluarea expresiei date.
- execuţia programului revine la locul de apel, unde valoarea returnată de funcţie e folosită.

Transmiterea parametrilor În limbajul C transmiterea parametrilor la funcţii se face *prin valoare*: În momentul apelului, parametrii formali iau *valoarea* argumentelor (care au fost evaluate); ei *nu* sunt substituiţi cu *expresiile* argumentelor. În consecinţă, pentru apelul discutat mai sus, în instrucţiunea `return` se va evalua expresia `12 * 12` şi nu `(3 * sqr(2)) * (3 * sqr(2))`. Altfel spus, o funcţie lucrează cu *valori* (numerice), nu cu expresii simbolice. Expresia `3 * sqr(2)` se evaluează o singură dată, înainte de apelul sintactic exterior la `sqr`, şi deci apelul la `sqr(2)` e deja încheiat în momentul celui de-al doilea apel, `sqr(12)`. Acest fapt poate fi vizualizat augmentând funcţia `sqr` cu o instrucţiune de tipărire, o practică utilă în urmărirea execuţiei 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) evidenţiază şi pentru apelul `printf` din `main` evaluarea argumentelor înainte de începerea execuţiei functiei. Evaluarea argumentului al doilea produce cele două linii tipărite în apelurile la `sqr`. Aceste linii apar înainte de a scrie chiar şi porţiunea de text obişnuit din primul argument (formatul), scriere în care constă tocmai execuţia funcţiei `printf`.

Returnarea valorilor La întâlnirea instrucţiunii `return`, execuţia funcţiei se încheie; orice alte instrucţiuni care urmează în corpul funcţiei nu se mai execută. Dacă execuţia unei funcţii se termină prin atingerea ultimei acolade `}` fără a executa o instrucţiune `return`, iar programul utilizează valoarea funcţiei, efectul e *nedefinit* (programul se comportă imprevizibil). O funcţie care nu prevede în orice situaţie o valoare returnată e scrisă eronat sub aspect logic.

Apelul recursiv Discutăm mecanismul apelului recursiv luând ca exemplu calculul lui 5^3 cu funcţia putere. În apelul `pwr(5, 3)` ($x = 5, n = 3$), expresia conditională conduce la evaluarea lui `5 * 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 * pwr(5, 0)` care poate fi efectuată acum. Apelul `pwr(5, 1)` returnează valoarea `5` folosită în calculul `5 * pwr(5, 1)` pentru valoarea lui `pwr(5, 2)`. În final, această valoare, `25`, e folosită în expresia `5 * 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 secvenţa de apel, rezultă că la un moment dat pot fi în execuţie *mai multe apeluri diferite* la aceeaşi funcţie. Fiecare apel reprezintă o *instanţă* (copie) distinctă a funcţiei, cu *propriile valori de parametri*, cele primite în momentul apelului. La fel se întâmplă şi în calculul pe hârtie, prin "desfăşurarea" formulei de recurenţă: pentru a calcula pe 5^3, înlocuim pe x cu 5 şi n cu 3. Trebuie calculat 5^2: aplicând din nou formula înlocuim pe n cu 2; aceasta nu afectează însă instanţa iniţială 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* faţă de cea de apel: din fiecare apel se revine în instanţa care a efectuat apelul. Aici, execuţia se reia în contextul dinainte de apel: adică din locul în care a fost făcut apelul (unde e folosită valoarea returnată), şi cu acele valori ale parametrilor corespunzând instanţei respective.

Ca la orice apel de funcţie, informaţia se transmite spre funcţia apelată prin parametri, şi înapoi spre locul de apel (funcţia *apelantă*) prin rezultat. În exemplul dat se creează un lanţ în care la revenire, valoarea returnată de fiecare apel e folosită în instanţa 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 instanţele apelate. Aceasta e esenta recursivităţii şi în acelaşi timp puterea ei în rezolvarea de probleme: ea permite exprimarea *indirectă* a soluţiei prin paşi simpli, din aproape în aproape, fără a necesita formularea directă a unei soluţii complexe.

2.4 Elementele unei definiţii recursive

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

- *Cazul de bază.* Tratează situaţiile (cele mai simple) în care noţiunea recursivă e definită *direct*. Exemple: pentru şirurile recurente, primul termen (sau mai mulţi, la recurenţele de ordin > 1) pentru liste, cea vidă, sau cea cu un element; pentru expresii, constantele şi identificatorii
- *Relaţia de recurenţă*: partea propriu-zis recursivă a definiţiei, în care noţiunea definită apare şi în corpul definiţiei. Exemple: formulele de recurenţă pentru şiruri; ramura de definiţie "o listă e un element urmat de o listă"; pentru expresii, variantele de definiţie cu expresie între paranteze, apel de funcţii (cu parametri expresii) şi cele cu expresii compuse cu operatori
- *Terminarea recursivităţii.* Dacă definiţia urmează o ramură care conţine din nou noţiunea definită, atunci definiţia trebuie aplicată din nou. O noţiune e corect definită recursiv dacă acest proces se opreşte întotdeauna. Pentru a fi riguroasă, o definiţie recursivă trebuie însoţită de o demonstratie că aplicarea definiţiei se opreşte după un număr finit de paşi.

Rezultă că o definiţie recursivă nu poate fi corectă fără un caz de bază, pentru că nu se ajunge niciodată la un punct unde noţiunea poate fi definită direct. Cazul de bază şi relaţia recursivă sunt de fapt *alternative* ale aceleiaşi definiţii. Acest lucru e explicit în regulile sintactice care definesc

construcţii de limbaj cum ar fi expresiile. Pentru un şir recurent putem evidenţia aceasta folosind acolade: $x_n = \begin{cases} a & n = 0 \\ x_{n-1} + p & n > 0 \end{cases}$ şi transcrie apoi uşor în program.

Pentru terminarea recursivităţii, cel mai uzual argument foloseşte o măsură (cantitate) care descreşte la fiecare aplicare a definiţiei, până atinge o valoare pentru care definiţia e dată direct. La şiruri recurente, această cantitate e chiar indicele n al termenului general x_n .

2.5 Alte exemple de recursivitate

Înşiruirea văzută recursiv Şi notiuni din afara matematicii, uneori foarte simple, se pretează la definiţii recursive. Un tipar de definiţii des întâlnit se bazează pe faptul că iteraţia (repetiţia) poate fi definită prin recursivitate. Putem defini astfel:

Un şir (secvenţă, listă) e fie un element, fie un şir urmat de un element.

Uneori e util să includem în definiţie şirul vid, care apare natural în diverse operaţii (ca element neutru la concatenare; la iniţializarea sau după ştergerea tuturor elementelor unei liste, etc.):

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

Cele două variante diferă atât prin cazul de bază (un element sau zero), cât şi prin poziţia noţiunii recursive în definiţie: prima variantă e *recursivă la stânga* (noţiunea definită recursiv "şir" e *prima* în rescrierea "şir urmat de un element"), iar a doua e *recursivă la dreapta*, deoarece în expandarea "element urmat de un şir" noţiunea definită "şir" e pe ultima poziţie. Tiparele de recursivitate la stânga şi la dreapta conduc la prelucrări diferite în program, pe care le studiem şi comparăm în continuare.

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

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

Am folosit convenţional simbolurile ::= pentru *definiţie* şi | pentru *alternativă* . Acest mod de a descrie regulile sintactice, adică *gramatica* unui limbaj se numeşte formă Backus-Naur (BNF).

Putem defini recursiv şi altă noţiune fundamentală, expresia. Din cele prezentate până acum:

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

unde ϵ denotă convenţional alternativa cu conţinut *vid* (fără simboluri de limbaj), aici pentru apeluri de forma `functie()`, fără argumente între paranteze.

Pentru o definiţie riguroasă, trebuie să precizăm că orice construcţie 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*, şi de două ori regula *expresie* ::= *constantă* .

2.6 Două tipare de calcul recursiv

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

Avem: $n! = \begin{cases} 1 & n = 0 \\ (n-1)! \cdot n & \text{altfel } (n > 0) \end{cases}$ şi 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 operaţie efectuată e înmulţirea cu `n`, restul calculelor fiind efectuate anterior în apelul `fact(n-1)` (şi celelalte apeluri recursive care rezultă din acesta). Ordinea calculelelor e indicată de paranteze în expresia $n! = ((((1 \cdot 1) \cdot 2) \cdot \ldots) \cdot (n-1)) \cdot n$.

Expandând definiţia pentru $n \geq 2$, obţinem $n! = ((n-2)! \cdot (n-1)) \cdot n$. Înainte de evaluarea lui `fact(n-2)` ştim că în produs apar atât `n` cât şi `n-1`, dar aşa cum e scrisă funcţia, nu se efectuează direct înmulţirea lor: se apelează întâi `fact(n-2)`, rezultatul e înmulţit cu `n-1` în cadrul apelului `fact(n-1)`, şi doar în final se face înmulţirea cu `n`, pentru rezultatul lui `fact(n)` .

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

Transcriind în C, am dori să efectuăm înmulţirea `(n-1)*n` în cadrul apelului funcţiei pentru `n-1`, înainte de a calcula recursiv factorialul pentru `n-2`. În apelul pentru `n-2` s-ar putea înmulţi apoi rezultatul lui `(n-1)*n` cu `n-2`, etc. Pentru aceasta, avem nevoie să transmitem la fiecare apel recursiv pe lângă valoarea lui `n` şi rezultatul înmulţirilor deja efectuate. Obţinem astfel:

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

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

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

Pentru a nu complica utilizatorul cu parametrul suplimentar datorat modului de calcul, definim funcţia `fact2` cu un singur parametru. Aceasta doar "împachetează" apelul iniţial `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 parţial calculat anterior. În a doua, rezultatul parţial e transmis în adâncime ca parametru, şi actualizat *înainte* de fiecare apel recursiv.

Există însă situaţii 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 funcţie care ia un întreg fără semn şi-l transformă în numărul cu aceleaşi cifre zecimale dar în ordine inversă. Scriem soluţia pornind de la un exemplu: 1472. Ultima cifră, 2, devine prima cifră a rezultatului. Punem ultima cifră rămasă din 147 după 2, obtinând $27 = 2 \cdot 10 + 7$. Din numărul rămas, 14, plasăm ultima cifră după 27, obţinând $27 \cdot 10 + 4 = 274$, 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 obtinut deja v, e acelaşi cu rezultatul inversării lui $n/10$, cu valoarea intermediară $10 \cdot v + n \bmod 10$. Deşi enunţul problemei are un singur parametru, soluţia recursivă obţinută manipulează două cantităţi, deci scriem o functie recursivă cu doi parametri. Prelucrarea se opreşte când n e 0 (nu mai sunt cifre de inversat), iar iniţial, 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 soluţie o funcţie cu un singur parametru, ca în enunţ, pentru a nu complica utilizatorul cu un parametru suplimentar pentru valoarea intermediară. Am scris astfel functia `revnum` care apelează funcţia recursivă `revnum_r(n, 0)` cu valoarea iniţială 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 instanţă mai simplă a aceleiaşi 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, şi se repetă procedura cu noile numere.

Exprimarea din urmă ("se repetă procedura") indică abordarea recursivă: soluţia se obţine rezolvând aceeaşi problemă pentru valori noi ale numerelor (mai mici, deci după un număr finit de paşi se ajunge la cazul de bază).

Putem scrie deci pe cazuri:

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

şi transcrie direct în C:

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

Deşi am transpus direct din cuvinte în formula recursivă şi apoi în cod, a rămas netratat un aspect: enunţul iniţial e dat pentru numere *pozitive*, iar tipul `unsigned` permite şi valoarea 0. Apelarea (chiar şi accidentală) a funcţiei scrise mai sus cu un parametru nul va duce la o secvenţă infinită de apeluri recursive, deoarece scăzând 0 celălalt număr nu se modifică şi reluăm *acelaşi* apel (până când, în funcţie de mediul de rulare, programul se va termina probabil forţat epuizând resursele de memorie).

Este important ca funcţiile pe care le scriem să fie *robuste* şi să nu producă erori neprevăzute şi catastrofale. Ca atare, rescriem funcţia ţinând cont că 0 e divizibil cu orice număr, şi 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 parţiale a unei serii se pretează natural la o exprimare recursivă. Notând cu t_n termenul general, şi $s_n = \sum_{k=0}^{n} t_k$, obţinem imediat pentru termenul general: $s_n = s_{n-1} + t_n$ (pentru $n \geq 1$), şi 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 funcţie de n), putem transforma direct formula de mai sus într-o funcţie recursivă `s`, care apelează pentru calcul funcţia `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, ştim că seria $\sum_n 1/n$ are sumă infinită, şi creşte aproximativ ca logaritmul natural al lui n. Mai precis,

$$\lim_{n\to\infty} (\sum_{k=1}^{n} 1/k - \ln n) = \gamma \simeq 0.5772...$$

Aproximaţia calculată de program are primele 3 zecimale exacte.

Tipul `double` folosit în program reprezintă, ca şi `float`, numere reale, dar cu precizie mai bună (de aici şi numele), şi e recomandabil în calcule pentru a micşora acumularea erorilor de rotunjire. Este tipul standard folosit de funcţiile matematice de bibliotecă declarate în `math.h` (cum e şi funcţia `log` pentru logaritmul natural) şi tipul implicit pentru constantele reale. Scrierea `1.0/n` pentru termenul matematic $1/n$ e necesară pentru a obţine *împărţire reală*: operandul `1.0` fiind real, va fi convertit implicit şi întregul `n`. Altfel, `1/n` ar fi însemnat împărţire întreagă, cu rest, şi 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 evidenţiază însă forma generală a funcţiei `s` pentru suma exprimată recursiv, şi programul poate fi adaptat la altă serie prin simpla înlocuire a funcţiei `t`.

E valabilă aceeaşi observaţie generală făcută întâi pe exemplul factorialului despre cele două variante de definire a calculului recursiv. Funcţia `s` aşa 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: $s_n = (((t_0 + t_1) + t_2) + \ldots) + t_n$.

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

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

Pentru a păstra aceeaşi formă naturală cu un singur parametru a funcţiei folosite direct de programator, redefinim funcţia `s`, apelând pe `s2` cu valoarea 0 ca sumă parţială deja calculată:

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

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

2.8 Calculul de aproximări cu o precizie dată

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

În matematică şi practică întâlnim însă adesea cazuri se doreşte un calcul efectuat cu o anumită precizie: se generează o secvenţă de aproximări, iar când aproximarea curentă a atins precizia dorită, calculul se opreşte.

Dăm ca exemplu o metodă de aproximaţie 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 secvenţa de aproximări: $a_{n+1} = (a_n + x/a_n)/2$, cu o aproximare iniţială arbitrară (de exemplu $a_0 = 1$).

Elementul cheie al soluţiei e formularea problemei pentru a-i identifica şi scoate în evidenţă caracterul recursiv: mai precis, *parametrii* şi *cazul de bază* (oprirea din secvenţa de apeluri recursive). Condiţia de oprire nu e dată de aproximarea iniţială, ci, după cum rezultă din enunţ, de *precizia* atinsă. Pentru a o calcula, avem nevoie de aproximarea curentă, care devine astfel *parametru al problemei* (cu valoarea iniţială dată de a_0).

Putem atunci enunţa soluţia în cuvinte în felul următor: funcţia căutată returnează o aproximaţie de precizie dată a lui $\sqrt{x}$, ştiind o aproximaţie curentă a_n dată de asemenea ca parametru. Dacă aproximaţia curentă e suficient de bună, poate fi returnată (cazul de bază). Dacă nu, rezultatul va fi dat de aceeaşi funcţie de calcul, apelată tot pentru x, dar cu o aproximaţie 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;
}
```

Funcţia standard `fabs`, declarată în `math.h` returnează valoarea absolută a unui număr real (`double`), spre deosebire de funcţia `abs` (din `stdlib.h`) aplicabilă doar la numere întregi. Logica comparaţiei este următoarea: a şi x/a se află întotdeauna de o parte şi 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 aproximaţie de precizie suficientă.