

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 problemelor, recursivitatea e foarte importantă deoarece permite exprimarea rezolvării unei probleme complexe în funcție de 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. Discutăm câteva categorii de exemple.

Șiruri recurente Din matematică, cunoaștem noțiunea de *șiruri recurente*, în care un termen al șirului e definit printr-o relație matematică în raport cu termenii anteriori. Exemple simple 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 exemple 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: $C_0^n = 1, C_n^n = C_n^0 = 1$ pentru $n > 0, C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$ pentru $0 < k < n$.

Înșiruirea văzută recursiv Și noțiuni 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 (o secvență, o 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 element urmat de un șir.

Pe lângă faptul că am ales un nou punct de pornire, în a doua variantă am schimbat poziția noțiunii recursive în secvența de 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. Cele două variante definesc aceeași noțiune de șir, dar folosirea lor ca tipare în scrierea de programe duce la moduri de prelucrare diferite, pe care le vom studia și compara ulterior.

Recursivitatea în sintaxa limbajelor Recursivitatea apare natural în definirea precisă a sintaxei limbajelor de programare. Multe elemente de limbaj au în componentă *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-parametri
lista-parametri ::= tip identificador | tip identificador , lista-parametri

```

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 discutate până acum:

```

expresie      ::= constantă | identificador | ( expresie ) | identificador ( argumente )
              | - expresie | expresie operator-binar 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 `functie()`, fără argumente între paranteze.

2.2 Elementele unei definiții recursive

Într-o definiție recursivă corectă se pot identifica următoarele elemente 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 , ca șirul lui Fibonacci sau combinările); pentru liste, cea vidă, sau cea cu un element; pentru expresii, constantele și identificatorii (nu și apelul de funcție, care are expresii ca argumente).
- *Relația de recurență*: partea propriu-zis recursivă a definiției, în care noțiunea definită apare și în corpul definiției. Exemple: ecuațiile (formulele) de recurență pentru șiruri; ramura de definiție “o listă e un element urmat de o listă”; iar pentru expresii, variantele de definiție cu paranteze, apel de funcții (cu argumente expresii), și cele cu diverși operatori aplicați expresiilor.
- *Terminarea recursivității*. Cât timp definiția urmează o ramură care conține din nou noțiunea de definit, pentru aceasta din urmă 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 șiruri recurente putem evidenția aceasta scriind de exemplu:

$$x_n = \begin{cases} a & n = 0 \\ x_{n-1} + p & n > 0 \end{cases} \quad \text{ceea ce ne ajută și la transcrierea în program.}$$

Pentru terminarea recursivității, adesea cel mai simplu argument e dat de o măsură (cantitate) care descrește la fiecare aplicare a definiției, până atinge o valoare pentru care definiția e dată direct. Pentru șiruri recurente, această cantitate e chiar *indicele* n al termenului general x_n . Pentru construcțiile de limbaj definite prin reguli sintactice, definiția devine riguroasă precizând că o noțiune trebuie definită prin aplicarea regulilor de un număr *finit* de ori. Astfel, obținem că $-(2 + 3)$ e o expresie aplicând pe rând regulile *expresie* ::= $-$ *expresie*, *expresie* ::= (*expresie*), *expresie* ::= *expresie* + *expresie*, și de două ori regula *expresie* ::= *constantă*.

2.3 O funcție recursivă în C

Funcția putere Definim recursiv funcția putere (cu bază reală și exponent natural) în felul următor:

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

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

Funcția are o definiție simplă, pe două variante, și se poate exprima cu operatorul condițional ca și exemplele nerecursive dinainte. Pentru exponent am folosit tipul `unsigned`, 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 *antetului 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.

2.4 Mecanismul apelului de funcție și al apelului recursiv

Am descris inițial lucrul cu funcții în C prin analogie cu matematica. Pentru a înțelege corect apelul de funcție și recursivitatea sunt însă necesare mai multe detalii despre mecanismul de apel. 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 sa 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 un apel, nu în *corpul* funcției ca în definițiile recursive.

Apelul de funcție În general, evaluarea unui apel de funcție se petrece în următorii pași:

- evaluarea e declanșată de faptul că valoarea funcției e necesară pentru calculul altei expresii (în care apelul e o subexpresie), sau în execuția unei instrucțiuni de forma *expresie* ;
- se evaluează toate expresiile care constituie argumentele funcției. În speță, aceasta înseamnă că orice apeluri de funcții care apar în argumente se efectuează *înainte* de apelul funcției considerate.
- valorile argumentelor se atribuie parametrilor formali ai funcției, cu eventuale conversii de tip (cum ar fi cele întreg–real deja discutate, și altele prezentate ulterior). Compatibilitatea dintre tipul expresiilor argument și ai parametrilor formali a fost deja verificată la compilare, dacă se cunoaște *prototipul* funcției (adică tipul returnat și tipul parametrilor, precizate în declarație).
- se execută corpul funcției (cu parametrii având valorile inițiale atribuite mai sus). La întâlnirea instrucțiunii `return`, execuția funcției se încheie, cu valoarea obținută evaluând expresia indicată.
- 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))`. 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 alături de sursă) evidențiază și pentru `printf` evaluarea argumentelor *înainte* de începerea execuției funcției. Evaluarea argumentului al doilea produce cele două linii tipărite în apelurile la `sqr`, care apar înainte de a scrie chiar și porțiunea de text obișnuit din format (primul argument). Analiza și tipărirea formatului este tocmai sarcina funcției `printf` și e executată de codul acesteia, nefiind efectul evaluării argumentelor. Primul argument (formatul) e în acest exemplu o *constantă șir de caractere* și e transmis ca atare, fără vreo altă operație de evaluare.

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 poate comporta imprevizibil). Scrierea unei funcții fără a prevedea în orice situație o valoare returnată e o eroare în logica programului.

Apelul recursiv Discutăm mecanismul apelului recursiv folosind exemplul funcției putere pentru calculul lui 5^3 . În apelul `pwr(5, 3)`, expresia condițională conduce la evaluarea lui `5 * pwr(5, 2)`. Aceasta necesită un nou apel la `pwr` și 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. Rezultatul, 5, e returnat ca valoare de `pwr(5, 1)` și folosit 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

```

Din analiza secvenței de calcul, reprezentată schematic și în figură, 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 aceleiași funcții, cu *propriile valori de parametri*, cele primite în momentul apelului.

În exemplul dat, recursivitatea are o structură liniară: fiecare apel recursiv generează altul, 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 pentru orice apel de funcție, informația se transmite înspre 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 la rândul lui înapoi spre locul de apel. Rezultatul final e astfel efectul unui calcul în care câte un pas e efectuat în 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 fi necesară formularea *directă* a unei soluții complexe.

2.5 Două tipare de calcul recursiv

În cazul funcției `pwr`, rezultatul a fost construit pas cu pas la *revenirea* din apelurile recursive, fără a efectua vreun calcul parțial la *înaintarea* în recursivitate, și deci fără ca vreo instanță apelată să folosească un rezultat parțial calculat anterior, în afară, bineînțeles, de parametrii bază și exponent. Există însă situații în care parte din prelucrare se efectuează *înainte* de fiecare apel recursiv, și e necesar să transmitem în jos la înaintarea în recursivitate valori ce vor fi folosite ulterior în calcule.

Fie ca exemplu o funcție care transformă un întreg fără semn într-un întreg cu aceleași cifre zecimale în ordine inversă. Scriem soluția pornind de la un exemplu: numărul 1472. Ultima cifră, 2, devine prima cifră a rezultatului. Luând ultima cifră rămasă din 147, o scriem după 2, obținând $27 = 2 \cdot 10 + 7$. Din valoarea rămasă, 14, plasăm ultima cifră după 27, rezultând $274 = 27 \cdot 10 + 4 = 274$, 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 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 e formatul pentru 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.