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

Şiruri recurente Din matematică, cunoaştem noţiunea de *şiruri recurente*, în care un termen al şirului e definit printr-o relaţie 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 *recurenţe de ordinul I*, în care termenul definit depinde doar de termenul imediat anterior. Alte exemple 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: $C_0^0 = 1$, $C_n^0 = C_n^n = 1$ pentru $n > 0$, $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$ pentru $0 < k < n$.

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

Un şir (o secvenţă, o 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-parametri*
lista-parametri ::= *tip identificator* | *tip identificator* `,` *lista-parametri*

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* | `(` *expresie* `)` | *identificator* `(` *argumente* `)`
`-` *expresie* | *expresie operator-binar 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.

2.2 Elementele unei definiţii recursive

Într-o definiţie recursivă corectă se pot identifica următoarele elemente 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*. Cât timp definiţia urmează o ramură care conţine din nou noţiunea definită, pentru aceasta 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 şiruri recurente putem evidenţia aceasta scriind de exemplu: $x_n = \begin{cases} a & n = 0 \\ x_{n-1} + p & n > 0 \end{cases}$ ceea ce ne ajută şi la transcrierea în program.

Pentru terminarea recursivităţii, adesea cel mai simplu argument e dat de o măsură (cantitate) care descreşte la fiecare aplicare a definiţiei, până atinge o valoare pentru care definiţia e dată direct. Pentru şiruri recurente, această cantitate e chiar indicele n al termenului general x_n . Pentru construcţiile de limbaj definite prin reguli sintactice, definiţia devine riguroasă precizând că o noţiune trebuie definită prin aplicarea regulilor de un număr finit de ori. Astfel, obţinem că `- (2 + 3)` e o expresie aplicând pe rând regulile *expresie* ::= `-` *expresie*, *expresie* ::= `(` expresie `)`, *expresie* ::= *expresie* + *expresie*, şi de două ori regula *expresie* ::= *constantă* .

2.3 O functie recursivă în C

Definim recursiv funcţia putere (cu bază reală şi exponent natural) în felul următor:

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

iar în C:

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

Funcţia are o definiţie simplă, pe două variante, şi se poate exprima cu operatorul condiţional ca şi exemplele nerecursive dinainte. Pentru exponent am folosit tipul `unsigned`, 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.

2.4 Mecanismul apelului de funcţie şi al apelului recursiv

Am descris iniţial lucrul cu funcţii în C prin analogie cu matematica. Pentru a înţelege corect apelul de funcţie şi recursivitatea sunt necesare mai multe detalii despre mecanismul de apel. 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 În general, evaluarea unui apel de funcţie se petrece în următorii paşi:

- evaluarea e declanşată când valoarea funcţiei e necesară în evaluarea unei expresii sau în execuţia unei instrucţiuni de forma *expresie* ;
- 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 ai funcţiei, cu conversiile necesare de tip (de exemplu întreg–real). Compatibilitatea dintre tipul expresiilor argument şi ai parametrilor formali e verificată deja la compilare, dacă se cunosc tipurile din declaraţia completă a funcţiei.
- 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))`. 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 alături de 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`, care 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 poate comporta imprevizibil). Scrierea unei funcţii fără a prevedea în orice situaţie o valoare returnată e o eroare în logica programului.

Apelul recursiv Discutăm mecanismul apelului recursiv folosind exemplul funcţiei putere pentru calculul lui 5^3 . În apelul `pwr(5, 3)`, expresia conditională conduce la evaluarea lui `5 * pwr(5, 2)`. Aceasta necesită un nou apel la `pwr` şi 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)` şi 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 secvenţei de calcul, reprezentată schematic şi în figură, 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 aceleiaşi funcţii, 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* 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 pentru orice apel de funcţie, informaţia se transmite înspre 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 la rândul lui 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 fi necesara formularea directă a unei soluţii complexe.

2.5 Două tipare de calcul recursiv

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

Fie ca exemplu o funcţie care ia un întreg fără semn şi-l transformă întregul cu aceleaşi cifre zecimale dar în ordine inversă. Scriem soluţia 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, obtinând $27 = 2 \cdot 10 + 7$. Din valoarea rămasă, 14, plasăm ultima cifră dup 27, rezultând $27 \cdot 10 + 4 = 274$, 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 e formatul pentru 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.