Logică și structuri discrete

Recursivitate

Marius Minea
marius@cs.upt.ro

http://www.cs.upt.ro/~marius/curs/lsd/

30 septembrie 2013

Recapitulare

Am revăzut: funcții (injective, surjective, bijective, inversabile)

Am definit funcții într-un limbaj de programare funcțional

Funcția e dată prin formulă, dar și și prin domeniu și codomeniu:
tipuri în limbajele de programare
Tipurile ne spun pe ce fel de valori poate fi folosită o funcție

Recapitulare

Am revăzut: funcții (injective, surjective, bijective, inversabile)

Am definit funcții într-un limbaj de programare funcțional

Funcția e dată prin formulă, dar și și prin domeniu și codomeniu:
tipuri în limbajele de programare
Tipurile ne spun pe ce fel de valori poate fi folosită o funcție

```
# let comp f g x = f (g x);;
val comp: ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

`f` are tipul `'a -> 'b` și `g` are tipul `'c -> 'a`,
compunerea are tipul `'c -> 'b` `'a`, `'b`, `'c` pot fi orice tip

Prin compunerea funcțiilor rezolvăm probleme mai complexe:
funcția `f` produce un rezultat intermediar
funcția `g` prelucrează rezultatul mai departe
calculul complet: $g \circ f$ (aplică `f`, apoi `g`)

Recursivitate

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

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

$\Rightarrow$ o unealtă simplă și puternică în rezolvarea problemelor

O problemă nerezolvată: “problema $3 \cdot n + 1$”

Fie un număr pozitiv n:
dacă e par, îl împărțim la 2: $n/2$
dacă e impar, îl înmulțim cu 3 și adunăm 1: $3 \cdot n + 1$

$$f(n) = \begin{cases} n/2 & \text{dacă } n \equiv 0 \mod 2 \\ 3 \cdot n + 1 & \text{altfel (dacă } n \equiv 1 \mod 2) \end{cases}$$

Se ajunge la 1 pornind de la orice număr pozitiv ?

= Conjectura lui Collatz (1937), cunoscută sub multe alte nume

Exemple:
3, 10, 5, 16, 8, 4, 2, 1
11, 34, 17, 52, 26, 13, 40, 10, 5, 16, 8, 4, 2, 1

Câți pași până la oprire?

Vrem să definim funcția $p : \mathbb{N}^* \to N$ care exprimă numărul de pași până la oprire.

Nu avem o formulă cu care să definim $p(n)$ direct.
Dar dacă șirul n, $f(n)$, $f(f(n)), \ldots$ ajunge la 1, numărul de pași pornind de la n e cu unul mai mare decât de la $f(n)$:

$$p(n) = \begin{cases} 0 & \text{dacă } n = 1 \\ 1 + p(f(n)) & \text{altfel (dacă } n > 1) \end{cases}$$

Funcția p a fost definită *recursiv*: e folosită în propria definiție

Problema $3 \cdot n + 1$ în ML

```
let f n = if n mod 2 = 0 then n / 2 else 3 * n + 1
let rec p = function
  | 1 -> 0
  | n -> 1 + p (f n)
```

sau

```
let rec p n = if n = 1 then 0 else 1 + p (f n)
```

Cuvintele cheie `let rec` introduc o *definiție recursivă*:
funcția p e folosită (apelată) în propria definiție

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

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

Definițiile de mai sus nu calculează x_n *direct* (deși se poate) ci *din aproape în aproape*, folosind x_{n-1}.

șirul x_n e *folosit în propria definiție* $\Rightarrow$ recursivitate / recurență

Să exprimăm progresiile în program

întâi o progresie aritmetică cu baza și rația fixate:

$$x_0 = 3,\ x_n = x_{n-1} + 2 \text{ (pentru } n > 0)$$

Noțiunea recursivă (șirul) devine o *funcție*

Valorile de care depinde (indicele) devin *argumentele* funcției

```
let rec aritpr_3_2 = function
  | 0 -> 3.
  | n -> 2. +. aritpr_3_2 (n-1)
```

Cum parametrizăm funcția cu bază și rație ?

Definiții locale în ML

Până acum: definiții *globale*: `let` *noțiune* = *expresie*

`let identificator =` *expresie* sau

`let functie param1 ... paramN =` *expresie*

Putem scrie definiții *locale*, folosite doar *într-o expresie*

`let` *noțiune* = *expresie1* `in` *expresie2*

rezultatul e *expresie2* în care *noțiune* e înlocuită cu *expresie1*

```
let aritpr base step =
  let rec ap = function
    | 0 -> base
    | n -> step + ap (n-1)
  in ap
```

La fel ca `let aritpr base step = ap` cu `ap` definit doar local

Funcția `ap` (de întreg) vede parametrii `base` și `step` ai lui `aritpr`

Putem defini apoi funcții care corespund unor progresii individuale:

```
let aritpr_3_2 = aritpr 3 2 (* progr. baza 3, ratia 2 *)
# aritpr_3_2 4
- : int = 11                (* termenul 4 al progresiei *)
```

Mecanismul apelului recursiv

În calculul recursiv

Fiecare apel face *"în cascadă" un nou apel*, până la cazul de bază

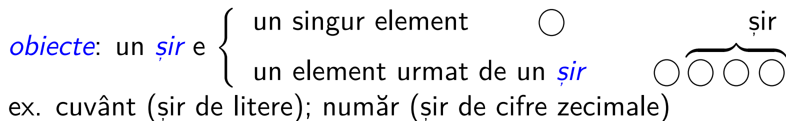
Fiecare apel execută *același cod*, dar cu *alte date*
(valori proprii pentru parametri)

Ajunși la cazul de bază, toate apelurile făcute sunt încă *neterminate*
(fiecare mai are de făcut adunarea cu rezultatul apelului efectuat)

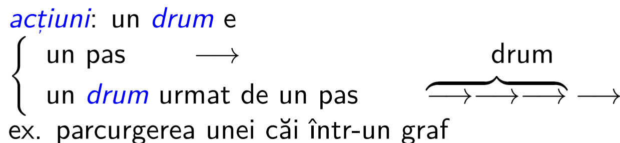
Revenirea se face *în ordine inversă* apelării
(apelul cu indice 0 revine primul, apoi cel cu indice 1, etc.)

Recursivitate: exemple

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



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



ex. parcurgerea unei căi într-un graf

O noțiune recursivă: expresia

Ce e o expresie numerică?

int + int   5 + 2
int – int   2 – 3
int * int   -1 * 4
int / int   7 / 3

și mai simplu ? Da: int   (5 e un caz particular de expresie)

și mai complicat? Da:
int * (int + int)
(int - int) / int
...

Putem scrie un număr finit de reguli ?

Expresia, definită recursiv

$$\text{O } \textit{expresie}: \begin{cases} \text{întreg} \\ \textit{expresie} + \textit{expresie} \\ \textit{expresie} - \textit{expresie} \\ \textit{expresie} * \textit{expresie} \\ \textit{expresie} \ / \ \textit{expresie} \end{cases}$$

Am descris expresia printr-o *gramatică* (detalii în alt curs)

Ne interesează modul în care sunt structurate calculele, nu sintaxa concretă, deci nu tratăm paranteze, spații, etc.

Tipuri recursive

Definim un *tip recursiv* care să reprezinte structura unei expresii (împreună cu eventualul operator pentru calcul)

```
type expr = I of int
          | A of expr * expr | S of expr * expr
          | M of expr * expr | D of expr * expr
```

Am definit un tip cu mai multe *variante*. Fiecare din ele trebuie scrisă cu un *constructor de tip* (etichetă), ales de noi: `I`, `A`, etc. (orice identificator cu literă mare)

Tipul `expr` e *recursiv* (o valoare de tip expresie poate conține la rândul ei componente de tip expresie)

Prelucrarea unei astfel de valori se face prin *potrivire de tipare* (engl. pattern matching), pentru fiecare variantă din tip

Evaluarea recursivă a unei expresii

```
let rec eval = function
  | I i -> i
  | A (e1, e2) -> eval e1 + eval e2
  | S (e1, e2) -> eval e1 - eval e2
  | M (e1, e2) -> eval e1 * eval e2
  | D (e1, e2) -> eval e1 / eval e2
```

Când un tip de date e definit recursiv
   funcțiile care îl prelucrează vor fi natural recursive
   deobicei cu câte un caz pentru fiecare variantă a tipului respectiv

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)

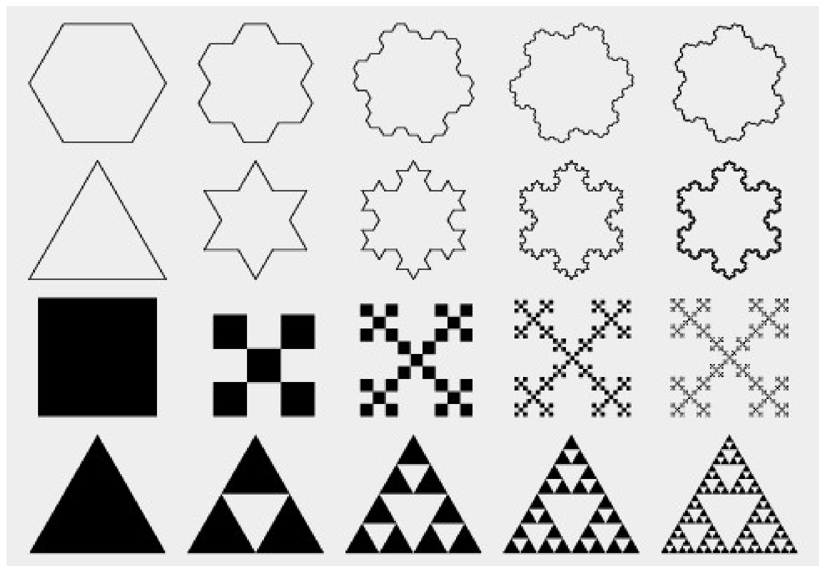
Fractali

Figuri geometrice în care o parte a figurii e similară întregului

Apar în natură, sau pot simula artificial figuri din natură

Analiza lor are aplicații în diverse domenii: geografie/geologie, medicină, prelucrarea semnalelor, electrotehnică (microantene), etc.

Exemple simple de fractali



Imagine: http://mathworld.wolfram.com/Fractal.html

Generarea recursivă a unui fractal

Scriem o *funcție* pentru noțiunea recursivă (figura)

Caracteristicile figurii devin *parametrii* funcției
dimensiunea, poziția (coordonatele), orientarea, etc.

Apelul funcției va *desena* figura
(sau va produce comenzile de desenare)

Să desenăm simplu

SVG = Scalable Vector Graphics:
  un format de imagine bazat pe XML

Comenzi simple:

m x y (moveto): mută punctul curent

l x y (lineto): desenează linie din punctul curent în cel indicat

versiuni cu coordonate absolute (M, L) și relative (m, l)

Structura XML a unui fișier SVG are elemente standard la început/sfârșit

Scrierea/tipărirea în OCaml

Funcții dedicate pentru fiecare tip:
`print_int`, `print_float`, `print_string` etc.

```
print_int 5
print_float 3.4
```

Funcția de tipărire formatată `printf` (din modulul `Printf`) (ca în C)

```
open Printf
printf "un intreg: %d\n" 5
printf "un real %f si inca unul: %f\n" 2.3 4.7
```

Putem defini atunci:
```
let lineto x y = printf "l %.2f %.2f" x y
```
(tipărește coordonatele cu două zecimale)
sau mai simplu: `let lineto = printf "l %.2f %.2f"`