Logică și structuri discrete

Recursivitate

Marius Minea
marius@cs.upt.ro

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

2 octombrie 2017

Recapitulare

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

Am definit funcții într-un *limbaj de programare funcțional*.

Domeniul și *codomeniul* sunt *tipuri* în limbajele de programare.

Tipurile ne spun pe ce fel de valori poate fi folosită o funcție

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

`f` are tipul `'c -> 'a` și `g` are tipul `'a -> 'b`

⇒ domeniul de valori al lui `f` e domeniul de definiție al lui `g`

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

Funcțiile pot avea ca *argumente* și/sau *rezultat* alte *funcții*

Compunând funcții ($g \circ f$) *rezolvăm probleme* mai complexe:

`f` produce un rezultat, `g` îl folosește mai departe

Ce putem face până acum

Putem defini funcții simple:

```
let max x y = if x > y then x else y
```

e de fapt predefinită, nu e nevoie s-o definim încă o dată

Putem compune de un număr dat de ori (număr fix de argumente)

```
let max x1 x2 x3 = max x1 (max x2 x3)
let max x1 x2 x3 x4 = max x1 (max x2 (max x3 x4))
```

Nu putem încă:

exprima că vrem să lucrăm cu N valori (listă, mulțime, tablou)

defini un calcul pentru un număr arbitrar de valori

Recursivitate

Recursivitatea e fundamentală în informatică:
dacă o problemă are soluție, se *poate rezolva recursiv*
reducând problema la un caz mai simplu al *aceleiași probleme*

$\Rightarrow$ înțelegând recursivitatea, putem rezolva orice problemă
(dacă e fezabilă)

Def.:

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

Calculul expresilor aritmetice

O *expresie* (puțin mai) complicată:
$(2+3)*(4+2*3)-5*6/(7-2)+(4+3-2)/(7-3)$

Pentru a calcula, trebuie să înțelegem *structura* expresiei
(nu se vede tot timpul ușor într-un șir de caractere)

E *suma* a două subexpresii ($+$ e calculat ultimul):

$$\begin{array}{ll} & (2+3)*(4+2*3)-5*6/(7-2) \\ + & (4+3-2)/(7-3) \end{array}$$

Apoi calculăm *expresiile mai simple*
$(2+3)*(4+2*3)-5*6/(7-2)=44$
$(4+3-2)\ /\ (7-3)=1$
$44+1=45$

Calculul celor două subexpresii: după *aceleași reguli*

Pași în rezolvarea problemei

Ce ne-a permis să calculăm expresia complicată?

- *Identificarea structurii recursive*
  expresia e suma a două *expresii* mai simple
  vom folosi *tipuri de date* definite *recursiv*
- Exprimăm *pașii de calcul* elementari (cei mai simpli)
  putem aduna, împărți, etc. două *numere*
- Identificăm *condiția de oprire*
  când expresia e un simplu număr, nu mai trebuie făcut nimic

Expresia ca noțiune recursivă

Ce e o expresie numerică?

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

Se poate mai simplu? Da: int   (5 e caz particular de expresie)

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

Putem scrie un număr finit de reguli ?

Expresia, definită recursiv

O *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* (niște reguli de scriere): așa se descriu limbajele de programare

detalii despre gramatici într-un alt curs
urmăriți *diagramele de sintaxă* la cursul de programare

selection-statement:
  `if` **`(`** *expression* **`)`** *statement*
  `if` **`(`** *expression* **`)`** *statement* **`else`** *statement*
  `switch` **`(`** *expression* **`)`** *statement*

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:
$3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$
$11 \rightarrow 34 \rightarrow 17 \rightarrow 52 \rightarrow 26 \rightarrow 13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

Câți pași până la oprire?

Definim funcția $p : \mathbb{N}^* \to \mathbb{N}$ care numără pașii până la oprire:
pentru $3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$ avem 7 pași

Nu avem o formulă cu care să definim $p(n)$ direct.

Dar dacă șirul n, $f(n)$, $f(f(n)), \ldots$ ajunge la 1,
atunci numărul de pași parcurși de la n (mai sus: 3)
e cu unul mai mare decât continuând de la $f(n)$ (aici: 10)

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

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
```

Revedem: `if c then e1 else e2` e o *expresie condițională*
dacă `c` e adevărată, are valoarea lui `e1`, altfel valoarea lui `e2`

```
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

Fără `rec`, fie `p` din dreapta ar fi necunoscut (eroare), fie s-ar folosi o eventuală definiție anterioară (deci nu ar fi recursivă).

Mecanismul apelului recursiv

În interpretor, vizualizăm apelurile și revenirea cu directiva
`#trace` *numefuncție*
revenim la normal cu `#untrace` *numefuncție*

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
(ultimul apelul revine primul, apoi revine penultimul apel, etc.)

Sintaxă: Potrivirea de tipare

Putem scrie funcția și așa:

```
let rec p = function
  | 1 -> 0
  | n -> 1 + p (f n)
```

Săgeata sugerează funcția definită ca diagramă. Citim astfel:

`p` e o funcție:
– dacă argumentul e 1, valoarea funcției e 0
– dacă argumentul are orice altă valoare (o notăm `n`), valoarea funcției e `1 + p (f n)`

Cuvintele cheie `fun` și `function` se folosesc diferit!
cu `fun x1 x2 ... ->` *expresie* putem scrie *orice funcție*
cu oricâți parametri x1, x2 ...

cu `function` definim o funcție prin *potrivire de tipare*,
cu *un* parametru *implicit* (NU scriem ~~let p n = ...~~)

Fiecare ramură definește în stânga lui `->` un *tipar* și în dreapta rezultatul (putem folosi numele introduse în tiparul din stânga)

Potrivirea de tipare (cont.)

Argumentul *implicit* care e potrivit cu tiparul poate fi:

- o *constantă* (aici, 1)
- o *valoare structurată* (pereche, listă cu cap/coadă, etc.)
  - perechile se notează `(x, y)` ca în matematică
  - triplete: `(a, b, c)`, etc
- un *identificator* (nume) care indică tot argumentul (oricare ar fi)

Nu putem avea ca tipar (doar) o condiție ~~`x > 5`~~

Potrivirile se încearcă în ordinea indicată, până la prima reușită.

Identificatorul special _ (linie de subliniere) se potrivește cu orice
Îl folosim dacă nu avem nevoie de valoarea respectivă.

```
let pozitie = function
  | (0, 0) -> print_string "origine"
  | (_, 0) -> print_string "pe axa x"
  | (0, y) -> Printf.printf "pe axa y la %d" y
  | (_, _) -> print_string "nu e pe axe"
```

Potrivirea de tipare: exemple

Ex.: o funcție care ia triplete de întregi și dă suma componentelor până la primul zero.

```
let sumto0 = function
  | (0, _, _) -> 0
  | (x, 0, _) -> x
  | (x, y, z) -> x + y + z
```

dacă prima componentă e 0, rezultatul e 0, indiferent de celelalte
altfel, dacă a doua componentă e 0, adunăm doar prima (nu și a treia)
altfel, primele două sunt nenule, și le sumăm pe toate trei

Dacă am uitat un tipar posibil, compilatorul ne avertizează.

Rescriere echivalentă cu if-then-else:

```
let sumto0 (x, y, z) =
      if x = 0 then 0 else if y = 0 then x else x + y + z
```

Sintaxă: Definiții locale

Până acum: definiții *globale*: `let identificator =` *expresie*
`let fct arg1 ... argN =` *expresie*

Uneori sunt utile definiții auxiliare. Am vrea să scriem:
Definim funcția arie(a, b, c) astfel: (a, b, c = laturile triunghiului)
întâi definim $p = (a + b + c)/2$
cu această notație, aria e $\sqrt{p(p-a)(p-b)(p-c)}$

```
let arie a b c =       (* traducem in ML *)
  let p = (a +. b +. c) /. 2. in
  sqrt (p *. (p -. a) *. (p -. b) *. (p -. c))
```

Definiția e tot de forma `let` *funcție arg1 ... argN* = **expresie**
dar **expresie** are noua formă: `let` *id_aux* = *expr_aux* `in` **expr_val**
Expresia are valoarea lui **expr_val**, $\sqrt{p(p-a)(p-b)(p-c)}$, unde *id_aux* (p) din stânga lui = are sensul din dreapta, $p = (a + b + c)/2$.

Astfel dăm un nume, p, pentru o expresie folosită de mai multe ori, $(a + b + c)/2$, scriind mai concis și evitând recalcularea.

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

Programăm: progresia aritmetică

Scriem 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*

Valoarea de care depinde (indicele) devine *argumentul* funcției

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

sau, echivalent

```
let rec aritpr_3_2 n =
       if n = 0 then 3 else 2 + aritpr_3_2 (n-1)
```

Cum parametrizăm funcția cu bază și rație ?

Exemplu: generalizăm progresia aritmetică

Putem scrie o funcție care are baza și rația ca parametri:

```
let rec aritpr baza ratie = function (* mai ia un indice *)
 | 0 -> baza
 | n -> ratie + aritpr baza ratie (n-1)
```

sau echivalent

```
let rec aritpr baza ratie n =
  if n = 0 then baza else ratie + aritpr baza ratie (n-1)
```

Putem defini apoi funcții care corespund unor progresii individuale:

```
let aritpr_3_2 = aritpr 3 2 (* baza 3, ratia 2 *)
# aritpr_3_2 4
- : int = 11                (* termenul de indice 4 *)
```

Rescriem cu definiții locale

```
let rec aritpr baza ratie = function
 | 0 -> baza
 | n -> ratie + aritpr baza ratie (n-1)
```

Apare de două ori expresia `aritpr baza ratie`, o funcție de un argument (indicele), în care `baza` și `ratie` sunt deja fixate.

Rescriem dând un nume `ap1` pentru expresia comună (definiție locală pentru `ap1`)
În exterior definim funcția inițială `aritpr baza rație` egală cu `ap1`

```
let aritpr baza ratie =
  let rec ap1 = function
    | 0 -> baza
    | n -> ratie + ap1 (n-1)
  in ap1
```

Citim: `let` (fie) `aritpr baza ratie` definită astfel:
– definim funcția `ap1` (folosind parametrii lui `aritpr`: `baza`, `ratie`;
  `ap1` ia indicele `n` și dă valoarea termenului al `n`-lea)
– atunci `aritpr baza ratie` e chiar `ap1` (expresia de după `in`)

`ap1` are rol ajutător, nu e vizibil în afara definiției lui `aritpr`

Recursivitate: exemple

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

obiecte: un *șir* e $\begin{cases} \text{un singur element} & \bigcirc \\ \text{un element urmat de un } \textit{șir} & \bigcirc\ \overbrace{\bigcirc\ \bigcirc\ \bigcirc}^{\text{șir}} \end{cases}$

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

acțiuni: un *drum* e

$\begin{cases} \text{un pas} & \longrightarrow \\ \text{un } \textit{drum} \text{ urmat de un pas} & \overbrace{\longrightarrow\longrightarrow\longrightarrow}^{\text{drum}}\ \longrightarrow \end{cases}$

ex. parcurgerea unei căi într-un graf

Tipuri recursive

Pentru a reprezenta structura recursivă a unei probleme, ne trebuie adesea *date* definite recursiv. În ML putem *construi* tipuri recursive.

Un *tip recursiv* pentru expresii (incluzând operatorii de calcul):

```
type expr = I of int
          | Add of expr * expr | Sub of expr * expr
          | Mul of expr * expr | Div 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`, `Add`, etc. (orice identificator cu literă mare)

Notația `expr * expr` reprezintă *produsul cartezian*, deci o pereche de două valori de tipul `expr`

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

Expresia $(2 + 3)$ * 7 se reprezintă: `Mul (Add(I 2, I 3), I 7)`

Evaluarea recursivă a unei expresii

Lucrul cu o valoare de tip recursiv se face prin *potrivire de tipare* (engl. pattern matching), *pentru fiecare variantă* din tip

```
let rec eval = function
  | I i -> i
  | Add (e1, e2) -> eval e1 + eval e2
  | Sub (e1, e2) -> eval e1 - eval e2
  | Mul (e1, e2) -> eval e1 * eval e2
  | Div (e1, e2) -> eval e1 / eval e2
```

Evaluând expresia `eval (Mul (Add(I 2, I 3), I 7))` dă 35.
e nevoie de paranteze, pentru a grupa `Mul` și perechea de după

Pentru *tipuri* definite *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)

De știut

Să recunoaștem și definim *noțiuni recursive*

Să recunoaștem dacă o definiție recursivă e *corectă*
(are caz de bază? se oprește recursivitatea?)

Să rezolvăm probleme scriind *funcții* recursive
cazul de bază + pasul de reducere la o problemă mai simplă

Să definim și folosim *tipuri recursive*

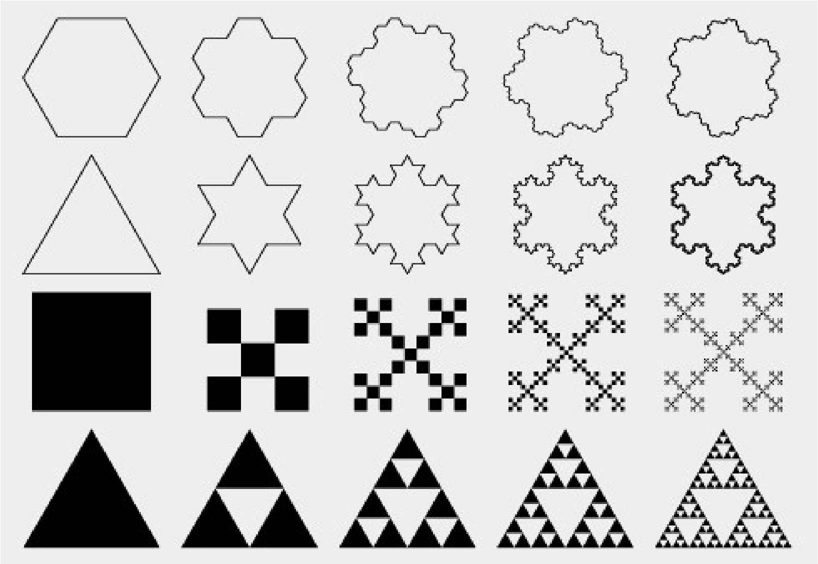
Fractali (opțional)

Figuri geometrice în care o parte a figurii e similară întregului
acesta e aspectul *recursiv*

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)

Punctul cheie: funcția recursivă e *figura* (nu: triunghi, pătrat, etc.)
acestea sunt doar cazurile de bază

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

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

Desene simple în format vectorial

SVG = Scalable Vector Graphics:
- un format de imagine bazat pe XML
- are un cadru standard, specificând că e în format SVG
- și comenzile de desenare propriu-zise

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`)

caz particular:

`h x`	linie orizontală (de lungime `x`)
`v y`	linie verticală (de lungime `y`)

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.printf` (asemănător cu C)
Dacă o folosim des, *deschidem* modulul `Printf` și scriem simplu:

```
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"`