Fundamente de informatică

Recursivitate

Marius Minea
marius@cs.upt.ro

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

30 septembrie 2011

În cursul de azi

Funcții recursive

Un limbaj funcțional: ML

funcții ca obiecte fundamentale (parametri, rezultate)

Complexitatea calculului - un prim exemplu

Structuri de date recursive: liste

Recursivitate: definiție, exemple

Din matematică cunoaștem *șiruri recurente*:

progresie aritmetică: $\begin{cases} x_0 = b & \text{(adică: } x_n = b \text{ pentru } n = 0) \\ x_n = x_{n-1} + r & \text{pentru } n > 0 \end{cases}$

progresie geometrică: $\begin{cases} x_0 = b & \text{(adică: } x_n = b \text{ pentru } n = 0) \\ x_n = x_{n-1} \cdot r & \text{pentru } n > 0 \end{cases}$

$\Rightarrow$ nu calculează x_n *direct*, ci *din aproape în aproape*, folosind x_{n-1}.

Recursivitate: definiție, exemple

Din matematică cunoaștem *șiruri recurente*:

progresie aritmetică: $\begin{cases} x_0 = b & \text{(adică: } x_n = b \text{ pentru } n = 0) \\ x_n = x_{n-1} + r & \text{pentru } n > 0 \end{cases}$

progresie geometrică: $\begin{cases} x_0 = b & \text{(adică: } x_n = b \text{ pentru } n = 0) \\ x_n = x_{n-1} \cdot r & \text{pentru } n > 0 \end{cases}$

$\Rightarrow$ nu calculează x_n *direct*, ci *din aproape în aproape*, folosind x_{n-1}.

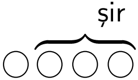
Un obiect (noțiune) e recursiv(ă) dacă e *folosit în propria sa definiție*.

Alte exemple: combinări C_n^k, șirul lui Fibonacci, ... (scrieți relațiile!)

Recursivitate: definiție, exemple

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

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



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

Recursivitate: definiție, 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

Recursivitate: definiție, 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

O *expresie*: număr (7), sau identificator (`x`), sau *expresie* + *expresie*, sau *expresie* - *expresie*, sau (*expresie*), ...

Limbajul ML

Un limbaj *funcțional* = funcția e elementul de bază
funcțiile pot fi parametri, rezultate, stocate
combinând funcții simple $\rightarrow$ programe complexe

Compilatorul deduce automat majoritatea *tipurilor* din declarații
verificări stricte de tip $\Rightarrow$ mai puține erori în program
tipuri și funcții *polimorfice* (ex. liste de orice tip)

Poate fi *compilat* sau *interpretat*
(execută pe rând fragmentele de program introduse)

ML are diverse variante de limbaj; folosim *Caml* și sistemul *Ocaml*
`http://caml.inria.fr/`

ML în exemple simple

```
3 + 4 ;;               (* calculează valoarea unei expresii *)
- : int = 7            (* interpretorul afișează valoarea și tipul ei *)
```

Un *tip* de date e o mulțime de *valori*, împreună cu un set de *operații* posibile pe aceste valori.
Tipuri de bază în ML: `bool`, `char`, `float`, `int`, `string`
Exemplu: în ML, `+` e operator pe întregi, dar `+.` e operator pentru reali

```
let x = 3 ;;                                        (* declaratie *)
val x : int = 3      (* raspuns: interpretorul deduce că x e intreg *)
```

declară identificatorul (variabila) `x` și îl *leagă* de expresia 3 (engl. *binding*)
NU este o atribuire (`x` nu poate fi modificat ulterior

ML în exemple simple (cont.)

`let f x = x + 1 ;;` (* definește funcția `f` de argument `x` *)

Interpretorul deduce *tipul* lui `f`: functie de la întregi la întregi

```
val f: int -> int = <fun>
```

Rezultatul se obține prin *evaluarea expresiei* `x + 1`. `f 4 ;;` (* apelul functiei f cu argumentul 4 *)

```
- : int = 5
```

```
let abs x =
  if x < 0 then -x else x
```

În ML, `if` e *operatorul condițional*, rezultatul e o *valoare*:
se evaluează expresia booleană;
  dacă e adevărată, se evaluează *expresia* de pe ramura `then` ca rezultat
  dacă e falsă, rezultatul se obține evaluând *expresia* de pe ramura `else`

Exemplu: funcția putere

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

```
let rec pow x n =
  if n = 0 then 1 else x * (pow x (n-1))
```

```
:- val pow : int -> int -> int = <fun>
```

`let rec` introduce o definiție *recursivă*
(identificatorul definit, `pow` poate fi *folosit* în interiorul definiției)

Remarcăm că funcția e definită cu bază *întreagă*.
Pentru bază reală, înmulțirea e `*.` iar cazul de bază e `1.` (sau `1.0`)

```
let rec powr x n =
  if n = 0 then 1. else x *. (powr x (n-1))
```

Funcția putere în C

```
#include <stdio.h>
double pwr(double x, unsigned n) {
 return n==0 ? 1 : x * pwr(x, n-1);
}
int main(void) {
  printf("-2 la 3 = %f\n", pwr(-2.0, 3));
  return 0;
}
```

Tipul `unsigned` reprezintă întregi fără semn (numere naturale)

Antetul funcției `pwr` reprezintă o *declarație* a ei
deci putem mai târziu folosi funcția în propriul corp (apelul recursiv)

Chiar dacă scriem `pwr(-2, 3)`, *întregul* −2 va fi *convertit la real*,
(se cunoaște tipul necesar pentru fiecare parametru)

Mecanismul apelului recursiv

Funcția `pwr` face două calcule:
– un *test* (`n == 0` ? a ajuns la *cazul de bază* ?) dacă da, returnează `1`
– dacă nu, o *înmulțire*; pt. operandul drept trebuie un *nou apel, recursiv*

```
pwr(5, 3)
      apel↓↑125
          5 * pwr(5, 2)
                apel↓↑25
                    5 * pwr(5, 1)
                          apel↓↑5
                              5 * pwr(5, 0)
                                    apel↓↑1
                                        1
```

Mecanismul apelului recursiv (cont.)

În calculul recursiv al funcției putere:

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 *începute* sunt încă *neterminate*
(fiecare mai are de făcut înmulțirea cu rezultatul apelului efectuat)

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

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 (nenegativ; 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   NU: $x = f(x)$

se pot utiliza doar noțiuni deja definite

nu se poate genera un calcul infinit (trebuie să se oprească)

Atenție la numărul de apeluri recursive!

Șirul lui Fibonacci: $\begin{cases} F_0 = F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad \text{pentru } n \geq 2 \end{cases}$

Transcriind direct definiția:

```
let rec fib n =
  if n < 2 then 1 else fib (n-1) + fib (n-2);;
```

Calculăm numărul A_n de apeluri necesare:
$A_0 = A_1 = 1$
$A_n = 1 + A_{n-1} + A_{n-2} \quad n \geq 2$ (apelul inițial + cele recursive)

Prin inducție putem arăta: $A_n = 2 \cdot F_n - 1$, deci numărul de apeluri crește exponențial ! (sunt recalculate ineficient aceleași valori)

Exercițiu: scrieți o funcție eficienta, tot recursivă, pentru F_n .

Factorialul, calculat recursiv

```
let rec fact1 n =                (* in ML *)
  if n = 0 then 1 else n * fact1 (n-1)

unsigned fact1(unsigned n)       // in C
{
  return n == 0 ? 1 : n * fact1(n-1);
}
```

Corespunde scrierii: $5! = 5 \cdot (4 \cdot (3 \cdot (2 \cdot (1 \cdot 1))))$

Calcule succesive: `1*1` (1), `2*1` (2), `3*2` (6), `4*6` (24), `5*24` (120), etc.

Calculul (∗) făcut la sfârșitul funcției, *după* revenirea din apelul recursiv

E nevoie de loc în *stiva calculatorului* pentru toate apelurile în curs (parametri, eventuale variabile locale, adresă de revenire, regiștri salvați)
$\Rightarrow$ ineficient, necesită mult loc pe stivă

Factorialul: recursivitate cu revenire (tail recursion)

Reordonăm înmulțirile: $5! = ((((1 \cdot 5) \cdot 4) \cdot 3) \cdot 2) \cdot 1$

apel recursiv cu un *rezultat parțial* (acumulator) ca *argument*

în cazul de bază ($n = 0$), *rezultatul e complet*, îl returnăm

valori `res`: 1, 5 (`1*5`), 20 (`5*4`), 60 (`20*3`), 120 (`60*2`), 120 (`120*1`)

```
let rec fact2 res n =                    (* in ML *)
  if n = 0 then res else fact2 (res * n) (n-1);;
```

```
unsigned fact2(unsigned res, unsigned n)       // in C
{ return n == 0 ? res : fact2(res*n, n-1); }
```

Funcția auxiliară scrisă calculează *res*$\cdot$*n!* $\Rightarrow$ luăm `res=1`

```
unsigned fact(unsigned n) { return fact2(1, n); }
let fact = fact2 1         (* are un parametru 1; mai cere unul *)
```

Factorialul: secvența de apeluri

```
fact1(3)
    apel↓ ↑6
        3 * fact1(2)
            apel↓ ↑2
                2 * fact1(1)
                    apel↓ ↑1
                        1 * fact1(0)
                            apel↓ ↑1
                                1
```

calcul: $3 \cdot (2 \cdot (1 \cdot 1))$

Apelul: în calculul rezultatului;

înmulțirea: după revenire

```
fact2(1, 3)
apel↓   ↑6
    fact2(3, 2)
      apel↓   ↑6
        fact2(6, 1)
          apel↓   ↑6
            fact2(6, 0)
              apel↓   ↑6
                  6
```

calcul: $(((1 \cdot 3) \cdot 2) \cdot 1)$

Calculul: înainte de apel, actualizând rezultatul parțial transmis ca argument

La revenire: returnează direct valoarea *tail recursion*, recursivitate cu revenire

În cazul 2 (*tail-recursive*): nici un calcul la revenire

$\Rightarrow$ nu e nevoie de înregistrarea de apel (adresă, parametri) pe stivă

$\Rightarrow$ compilatorul poate *transforma recursivitatea în iterație (eficient)*

Liste

O listă e o înșiruire ordonată de elemente de același tip

Definiție recursivă:
lista vidă (niciun element) sau
un element urmat de *o listă*

Definiție recursivă ⇒ prelucrările de liste sunt natural recursive

Liste în ML

Tipul listă e predefinit în ML (parametrizat cu tip arbitrar de elemente)
ex. tipul unei liste de întregi e `int list`

Lista vidă (de orice tip): `[]`
Operatorul `::` construiește o listă, dintr-un cap (element) și altă listă

cap `::` coadă

Valori de tip listă: între `[]` cu separator `;` `[2; 7; -4]`

Listele se pot prelucra cu *tipare* (pattern matching) pentru cele 2 cazuri:

`match` *listă* `with`
`[] ->` *expr1*
`|` *cap* `::` *coadă* `->` *expr2*

Construcția de mai sus e o *expresie*, cu rezultatul *expr1* dacă *listă* e vidă; altfel, identificatorii *cap* și *coadă* sunt *legați* la cele două părți ale listei, și pot fi folosiți în *expr2*, a cărei evaluare dă rezultatul