Fundamente de informatică

Prelucrări recursive de liste și expresii

Marius Minea
marius@cs.upt.ro

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

7 octombrie 2011

Prelucrări iterative pe liste

Pe liste se pot defini funcții generice de prelucrare.
⇒ putem itera prelucrări fără a scrie repetat același cadru

Modulul `List` din ML are astfel de funcții:

`iter : ('a -> unit) -> 'a list -> unit`
`iter f [a1; a2; ...; an]` apelează `f a1; f a2; ... f an; ()`

`map : ('a -> 'b) -> 'a list -> 'b list`
`map [a1; a2; ...; an]` e lista `[f a1; f a2; ... f an]`

`fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
`fold_left f a [b1; b2; ... bn] = f (...f (f a b1) b2...) bn`

`fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`
`fold_right f [a1; a2;...; an] b = f a1 (f a2 (...(f an b)...))`

`filter f [a1; a2; ...; an]` : elementele pentru care `f` e adevărată

Implementarea iteratorilor

```
let rec iter f = function
    | [] -> ()
    | h :: t -> (f h; iter f t)

let rec map f = function
    | [] -> []
    | h :: t -> f h :: map f t

let rec fold_left f a = function
    | [] -> a
    | h :: t -> fold_left f (f a h) t

let rec fold_right f lst b = match lst with
      [] -> b
    | h :: t -> f h (fold_right f t b)

let filter f = function
    | [] -> []
    | h :: t -> if f h then h :: filter f t else filter f t
```

Exemple de folosire a iteratorilor

`List.iter print_int [1;2;3]` tipărește 123

`List.map ((+) 2) [3; 7; 4]` are ca rezultat `[5; 9; 6]`

Putem implementa inversarea `rev` cu `fold_left` :

```
let rev = List.fold\_left (fun t h -> h :: t) []
```

Putem implementa minimul unei liste:

```
let list_min = function
    | [] -> invalid_arg "empty list" (* exceptie *)
    | h :: t -> List.fold_left min h t
```

Recursivitatea în sintaxa limbajelor de programare

Programele pot fi oricât de complexe, dar au structură riguros definită

$\Rightarrow$ se pretează la definiții recursive

– înșiruiri liniare: un program are oricâte funcții,
o funcție are oricâte argumente și instrucțiuni, etc.

– structuri mai complexe, ex. expresie formată din operator și 2 expresii

Structura (sintaxa, *gramatica*) limbajului se reprezintă uzual
într-o notație standard numită BNF (Backus-Naur Form). Ex.

antet-funcție ::= *tip identificator* (*parametri*)
parametri ::= `void` | *lista-parametri*
lista-parametri ::= *tip identificator* | *tip identificator* , *lista-parametri*

unde ::= denotă *definiție* iar | *alternativă* (alegere)
Termenii definiți prin reguli se numesc *neterminali* (engl. nonterminal)

Cazuri particulare: recursivitate *la stânga* și *la dreapta*,
după locul în care apare noțiunea recursivă în corpul definiției

Calculul recursiv al expresiilor

expr ::= întreg | (expr)
| expr + expr | expr − expr | expr ∗ expr | expr / expr

Definiția e *ambiguă* pentru că o expresie poate avea mai multe *derivări* (interpretări), dacă nu stabilim reguli de precedență.
Pentru $9 + 5 * 4$ am putea interpreta: $9 + 5 = expr$, deci $(9 + 5) * 4$, sau $5 * 4 =$ expr, deci $9 + (5 * 4)$. Rescriem gramatica:

expr ::= term | expr + term | expr − term
term ::= factor | term ∗ factor | term / factor
factor ::= întreg | (expr)

expr și term au definiții *direct recursive*.
Toate trei sunt *mutual (circular) recursive*: expr → term → factor → expr
Implementăm câte o funcție pentru fiecare neterminal.
Pentru expr ::= expr_1 + term dăm ca parametru expr_1 din dreapta (valoare deja calculată) și apelăm recursiv cu $\text{expr}_1 \pm$ term dacă apare $\pm$; altfel returnăm valoarea primită (vezi cod în C și ML).

Alte exemple: cel mai mare divizor comun

$$cmmdc(a, b) = \begin{cases} a & a = b \\ cmmdc(a-b, b) & a > b \\ cmmdc(a, b-a) & a < b \end{cases}$$

```
unsigned cmmdc(unsigned a, unsigned b) {
  return a == b ? a
                : a > b ? cmmdc(a-b, b)
                        : cmmdc(a, b-a);
}
int main(void) {
  printf("cmmdc(20, 8) e %u\n",  // %u = unsigned
                         cmmdc(20, 8));
  return 0;
}
```

Calculul e corect doar cu a și b nenule. Pentru a trata și cazul zero:

```
return a == 0 ? b
              : b == 0 ? a
                       : a > b ? cmmdc(a-b, b) : cmmdc(a, b-a);
```

Calculul sumei unei serii

Forma: $s_0 = t_0, \qquad s_n = s_{n-1} + t_n$, pentru $n > 0$

$(t_n =$ termenul general, pentru care avem o formulă$)$

Exemplu pentru seria armonică $(t_n = 1/n)$

$s_n = 1/1 + 1/2 + \ldots + 1/n$

recursiv: $s_0 = 0, \quad s_n = s_{n-1} + 1/n$ pentru $n > 0$

În cuvinte: știm să răspundem direct cât e s_0 : 0.
Nu putem calcula direct s_n (pentru $n > 0$),
dar dacă aflăm cât e s_{n-1} mai trebuie să adunăm $1/n$.
$\Rightarrow$
Funcția care calculează pe $s(n)$ răspunde 0 dacă $n = 0$
iar altfel, calculează $s(n-1)$, adună $1/n$ și returnează rezultatul.

Calculul sumei unei serii

```
#include <stdio.h>
double suma_rec(unsigned n) {
  return n == 0 ? 0 : suma_rec(n-1) + 1.0/n;
}
int main(void) {
  printf("suma pana la 1/100: %f\n", suma_rec(100));
  return 0;
}
```

Termenii se adună începând de la $1/1$ la $1/100$, la revenirea din apel

`1.0 / n` : operație între real și întreg : întregul convertit la real
ATENȚIE: `1/n` dă valoarea 0 când `n > 1` (împărțire intreagă)

Suma unei serii – variantă cu rezultat acumulat

În $s_n = s_{n-1} + 1/n$, trebuie adunat $1/n$, dar nu știm încă s_{n-1}
⇒ folosim un rezultat parțial `rez` la care adunăm $1/n$
⇒ apelăm recursiv cu valoarea `rez + 1.0/n`

```
double suma_inv(unsigned n, double rez) {
  return n == 0 ? rez : suma_inv(n - 1, rez + 1.0/n);
}
```

Când $n = 0$, totul e adunat deja în `rez`, care e returnat ca rezultat

În apelul inițial, rezultatul acumulat e zero: `suma_inv(100, 0.0)`

`rez` e un detaliu de implementare, nu face parte din enunțul problemei
⇒ definim o funcție cu un singur parametru, care apelează `suma_inv`

```
double serie_armonica(unsigned n) { return suma_inv(n, 0.0); }
```

Calculul cu aproximații: rădăcina pătrată

Din matematică: $a_0 = 1$, $a_{n+1} = \frac{1}{2}(a_n + \frac{x}{a_n})$

Formulăm recursiv:

Calculul *aproximației dorite* (ex. cu $\epsilon = 10^{-3}$) de la o *aproximație dată*:

ce se dă (parametri): x și aproximația curentă

ce se cere = val. funcției (o aproximație suficient de bună)

Dacă precizia e bună $|a_{n+1} - a_n| < \epsilon$ returnăm *aproximația curentă* a_n
Altfel, returnăm valoarea *calculată recursiv* cu *noua aproximație* a_{n+1}

Dezvoltăm: $|a_{n+1} - a_n| < \epsilon \Rightarrow |a_n - x/a_n| < 2 \cdot \epsilon$

Calculul cu aproximații: rădăcina pătrată

```
#include <math.h>
// pentru declaraţia double fabs(double x);  (val. abs. nr. real)

double rad(double x, double a_n) { // rad.lui x, se da aprox.a_n
  return fabs(a_n - x/a_n) < 2e-3 ? a_n : rad(x, (a_n + x/a_n)/2);
}

double radacina(double x) { return x < 0 ? -1 : rad(x, 1.0); }
```

Soluția dorită e funcția `radacina`: apelează `rad` cu aprox. inițială 1

Pentru argument negativ, returnează −1 (îl interpretăm ca eroare)

Calculul sumei unei serii cu precizie dată

Calculăm $s_n = s_{n-1} + t_n$ ($n \geq 0$), cu $s_0 = 0$
până când valoarea absolută a termenului $t_n = x^n/n!$ e neglijabilă.

Formulăm recursiv: calculul *sumei dorite*, dată fiind *suma curentă* s_{n-1}:
– dacă termenul curent t_n e suficient de mic, returnăm suma curentă
– altfel, returnăm suma calculată *recursiv*, de la *noua sumă* $s_{n-1} + t_n$

Exemplu: seria $1 + 1/2^2 + 1/3^2 + \ldots = \pi^2/6$ $\qquad$ $t_n = 1/n^2$ ($n > 0$):

```
double sum_2(unsigned n, double s_n_1) {
  return 1./n/n < 1e-6 ? s_n_1 : sum_2(n+1, s_n_1 + 1./n/n);
}              // 1. == 1.0 = 1 real, forteaza impartire reala
```

și folosim apelul inițial `sum_2(1, 0)` (pornind de la $n = 1$, $s_0 = 0$)

Exemplu: seria Taylor pentru e^x

$e^x = x^0/0! + x^1/1! + x^2/2! + \dots$ cu $t_n = x^n/n!$ $(n \geq 0)$
Pentru a nu recalcula inutil în t_n pe x^{n-1} și $(n-1)!$
exprimăm recursiv $t_n = t_{n-1} \cdot x/n$, pentru $n > 0$, $t_0 = 1$.
$\Rightarrow$ la pasul curent, avem s_{n-2} și t_{n-1}, calculăm t_n și $s_{n-1} = s_{n-2} + t_{n-1}$

```
#include <math.h>
#include <stdio.h>
double e_xr(double x, unsigned n, double s_n_2, double t_n_1) {
  return fabs(t_n_1)<1e-6 ? s_n_2
                          : e_xr(x, n+1, s_n_2+t_n_1, t_n_1 * x/n);
} // apelam cu valori initiale potrivite in e_x cu 1 parametru, x
double e_x(double x) { return e_xr(x, 1, 0.0, 1.0); }
int main(void) {
  printf("e^-1 = %f\n", e_x(-1.0));
  return 0;
}
```

Recursivitate și inducție

Recursivitatea e strâns legată de inducția matematică; ambele:
– au un *caz de bază*
– leagă o *noțiune* de *ea însăși* (relatia de recurent / pasul inductiv)

Diferă al treilea element, *sensul* în care se face raționamentul:

– *crescător* la principiul inducției matematice:
O afirmație $P(n)$ e valabilă pentru orice n (*crescând* spre infinit) dacă:

$P(0)$ e adevărat și
$P(n) \Rightarrow P(n+1)$ dacă $P(n)$ adevărat atunci $P(n+1)$ adevărat

– *descrescător* la recurență: definim ceva *mai mare* prin ceva *mai mic*
se oprește când ajungem la cazul de bază (suficient de simplu)