PROGRAMARE FUNCȚIONALĂ

Lucrarea 5

Introducere

Istoric

ML (Meta-Language) este un set de limbaje de programare avansată, făcand parte din familia limbajelor de programare funcţionala. Limbajul este o combinaţiie a proprietăţilor LISP şi Algol, şi este primul limbaj care include tipizarea polimorfică. ML a fost proiectată la universitatea Edinburgh in 1973.

La fel ca la LISP, şi ML are mai multe dialecte (Standard ML, Lazy ML, CAML, etc.). Dialectul CAML (original Categorical Abstract Machine Language) este o versiune franceză de ML şi suportă printre altele şi programarea orientată pe obiecte (varianta OCAML), precum şi programarea iterativă.

Original ML a fost un limbaj construit pentru a scrie programe care manipulează alte programe ca:

-compilatoare şi interpretoare, dar intre timp a devenit un limbaj de sine stătător, utilizat mai frecvent in verificarea programelor şi in demonstrarea automată a teoremelor matematice, dar are aplicatibilitate mare şi in implementarea rapidă a aplicaţiilor WEB, protocoalelor de comunicaţie sau calcule distriuite.

Proprietăţi CAML

- Limbaj funcţional

Funcţiile sunt tratate ca restul tipurilor de date, ele putand fi parametrul sau rezultatul altor funcţii. Programarea funcţională permite programarea fară efecte secundare.

- Tipuri de date bine definite.

Limbajul pune la dispoziţia programatorului un set de tipuri de date şi operatori necesare manevrării acestor tipuri. Este posibilă definirea de noi tipuri de variabile.

- Limbaj puternic tipizat

Declaraţia tipului unei variabile nu este necesară, tipul fiind determinat automat. Chiar în timpul compilării, şi nu în timpul rulării, se verifică corectitudinea parametrilor funcţiilor.

- Tipuri polimorfice

Este posibilă declararea funcţiilor cu parametri a cărui tip nu este precizat.

- Potrivirea de şabloane

ML permite scrierea de programe bazate pe reguli.

TIPURI DE DATE IN ML

Expresie simplă în ML

```
# 2+3;;
- : int = 5
```

O expresie se poate scrie pe mai multe linii; expresia se termina cu ;; ,care determină evaluarea expresiei şi afişarea rezultatului pe randul urmator.

Tipuri primitive

Numere intregi şi reale

În CAML există două tipuri de numere: numere intregi şi numere reale. Cele două sunt tipuri diferite, fiecare avand un set operaţii diferite:

Intregi:

+ adunare
- scădere
* inmulţire
/ impărţire intreagă
mod restul impărţirii

Reale:
+. adunare
-. scădere
*. inmulţire
/. impărţire intreagă
** ridicare la putere

Operaţii – exemple:
2 + 3;;
- : int = 5
4.0 +. 5.5;;
- : float = 9.5

Tipurile caracter şi şir de caractere

Caractere sunt coduri ASCI intre 0 şi 255. Ele sunt specificate intre caracterele apostrof. Stringurile sunt şiruri de caractere de lungime maximă 2^{24}-6. Operatorul ^ concatenează două şiruri.

Caractere şi stringuri
'a';;
- : char = 'a'
"rezultatul" ^ " este " ^ "un string";;
- : string = "rezultatul este un string"

Tipul boolean

Tipul boolean are valoarea true sau false.

Operatori logici:

&& sau & - şi logic
|| sau or -sau logic
not - negare logica

Exemple:
true && false;;
- : bool = false
false or not (2=3);;
: bool = true

Operatori relationali:

Operatori =, <, <=, >, >=, <> sunt polimorfi, ei pot să compare numere, caractere sau şiruri.

Exemple:
5 < 6;;
- : bool = true
"beta" > "alfa";;
- : bool = true

Lista

Listele sunt structuri de date cu n elemente, vide sau formate din elementul din capul listei şi lista cu restul elementelor. Toate elementele unei liste sunt de acelaşi tip.

Exemple:

```
# [];;
- : 'a list = []
# [1;2;3];;
- : int list = [1; 2; 3]
```

Observaţie: int list semnifică că rezultatul este o listă de intregi, iar ‘a list semnifică că rezultatul este o listă de tip nespecificat. Tipul elementului putand fi orice.

Operatori:

Operatorul **::** adaugă un nou element la inceputul unei liste.

Operatorul **@** concatenează două liste.

Exemple:

```
# 1::2::3::[];;
- : int list = [1; 2; 3]
# ['a';'b'] @ ['c';'d';'e'];;
: char list = ['a'; 'b'; 'c'; 'd'; 'e']
```

Functii predefinite:

Funcţiile hd şi tl returnează primul element a listei respectiv coada listei fără primul element.

Exemple:

```
# List.hd [1;2;3;4];;
- : int = 1
# List.tl [1;2;3;4];;
- : int list = [2; 3; 4]
```

STRUCTURI CONDIŢIONALE

La fel ca şi în alte limbaje şi în ML există structuri condiţionale. Spre deosebire de limbajele iterative evaluarea unei expresii condiţionale, în ML rezultă o valoare.

Sintaxa:

if expr1 then expr2 else expr3

Evaluare expresiei va rezulta expr2 în caz că expr1 este evaluată la adevărat şi expr3 în caz că expr1 este falsă.

IF ... THEN ... ELSE …

```
# if 2=3 then 2 else 3;;
: int = 3
```

VARIABILE GLOBALE

Folosirea primitivei **let** determină legarea valorii rezultate a unei expresii la o variabilă.

LET

```
# let a = 5 * 2;;
val a : int = 10
```

```
# a ;;
- : int = 10
```

Declarări paralele pot fi făcute folosind sintaxa:

```
let var1 = expr1
and var2 = expr2
...
and varn = exprn ;;
```

Exemple:

LET paralel

```
# let a = 1;;
val a : int = 1
# let a = 2
and b = a + 1;;
val a : int = 2
val b : int = 2
# a + b ;;
- : int = 4
```

Este posibilă şi declararea secvenţială a variabilelor, ele având acces la toate declaraţiile anterioare. În acest caz scriem mai multe instrucţiuni let una după cealalalta. Secvenţa de declaraţie va fi terminată de ;;.

LET secvenţial

```
# let z = 1
let u = z+2;;
val z : int = 1
val u : int = 3
```

Declaraţii locale

Declaraţiile locale au rolul de a limita domeniul de existenţă a variabilelor. Declaraţiile locale se fac folosind primitiva let cu sintaxa mai sus prezentată, expresia asignată fiind urmată de **in** şi de expresia în care vrem să fie vizibilă declaraţia.

Definiţie locală

```
# let x = 1 and y = 2 in x+y;;
: int = 3
```

Funcţii

O funcţie se defineşte cu sintaxa:

```
function p -> expr
```

Această definiţie este foarte asemănătoare cu lambda din LISP. Expresia specificată cu function rezultă o funcţie apelabilă:

```
# function x -> x + 1 ;;
- : int -> int = <fun>
# (function x -> x + 1) 2;;
- : int = 3
```

Expresia la randul ei poate să fie o altă funcţie.

În exemplul următor se vor da exemple cu declaraţia funcţiilor cu mai mulţi parametri.

```
# function x -> (function y -> x + y + 1) ;;
- : int -> int -> int = <fun>
# function x -> function y -> x + y + 1 ;; (* declaratie echivanenta*)
```

```
- : int -> int -> int = <fun>
# (function x -> function y -> x + y + 1) 2;; (* rezulta functia y -> 2 + y + 1 *)
- : int -> int = <fun>
# (function x -> function y -> x + y + 1) 2 5;;
- : int = 8
```

POTRIVIREA DE ŞABLOANE

Potrivirea de şabloane este un mecanism de selecţie flexibilă şi foarte puternică prin care dependent de valoarea unei expresii se selectează valoarea rezultatului. Mecanismul de bază este similar instrucţiunii case din Pascal sau switch din C, dar oferă facilităţi mai puternice. Pentru potrivirea de şabloane există mai multe mecanisme ML.

MATCH

Sintaxă:

```
match expr with
    | p1 -> expr1
     ...
    | pn -> exprn
```

Instucţiunea **match** evaluează expresia expr şi pe rand o compară cu probele p1...pn. În caz de potrivirea rezultatului cu pi , match va returna expri.

Exemple:

```
mynot
# let mynot p = match p with true -> false | false -> true;;
val mynot : bool -> bool = <fun>
```

Este indicat ca probele să acopere întregul domeniu de valori a expresiei de testat. CAML verifică acest lucru şi avertizează în cazul incompletitudinii lui match.

Exemple:

Sau exclusiv 1

```
# let xor p = match p with
     (true,true)   -> false
   | (true,false)  -> true
   | (false,true)  -> true
   | (false,false) -> false;;
val xor : bool * bool -> bool = <fun>
# xor (true, false);;
- : bool = true
```

Expresiile pot conţine şi variabile. Se numeşte şablon liniar un şablon care conţine variabile care apar maxim odată în şablon.

Sau exclusiv 2

Este permis:

```
# let xor p = match p with
     (true, x)  -> not x
   | (false, x) -> x;;
val xor : bool * bool -> bool = <fun>
```

însă următoarea expresie nu mai este permisă:

```
# let xor p = match p with
     (x, x) -> false
   | (false, true) -> true
   | (true, false) -> true;;
     Characters 33-34:
     (x, x) -> false
     ^
     This variable is bound several times in this matching
```

Compactarea cazurilor

Semnul **_** este şablonul oarecare, valoarea lui, coincide cu orice expresie.

Sau exclusiv 3

```
# let xor p = match p with
     (false, true) -> true
   | (true, false) -> true
   | _ -> false;;
val xor : bool * bool -> bool = <fun>
```

Prin utilizarea semnului | (pipe) este posibil combinarea a mai multor şabloane a căror rezultat este identic.

Test dacă un întreg este o cifră :

```
# let cifra n = match n with
  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 -> true
| _ -> false;;
val cifra : int -> bool = <fun>
```

Potrivirea de şabloane la parametri funcţiilor

Folosirea potrivirii de şabloane în definirea funcţiilor cu mai multe cazuri este esenţială în ML. Sintaxa mai precisă a lui function, prezentată deja este:

```
function | p1 -> expr1
         | p2 -> expr2
         ...
         | pn -> exprn
```

Acestă sintaxă fiind echivalentă de fapt cu

```
function expr -> match expr with
           p1 -> expr1
         | p2 -> expr2
         | ...
         | pn -> exprn
```

Test pentru 0 şi 1

```
# let test01 = function | 0 -> true
     | 1 -> true
     | _ -> false;;
val test01 : int -> bool = <fun>
# test01 1;;
- : bool = true
# test01 2;;
- : bool = false
```

Potrivirea de șabloane la liste:

Exemplu:

Calcularea lungimii unei liste

```
# let rec length p = match p with
    [] -> 0
  | head::tail -> (1+length tail);;
```

Declaraţii de tipuri:

Noi tipuri pot fi declarate folosind cuvântul cheie type.

Sintaxa:

```
type  tip1 = typedef1
  and tip2 = typedef2
  ...
  and tipn = typedefn ;;
```

Unde tipul tipi va fi echivalent cu tipul typedefi.

Exemplu:

```
# type t1 = (int*int)
  and t2 = (int*char);;
type t1 = int * int
type t2 = int * char
```

Este permisa declarearea tipurilor parametrizate:

```
type 'a tip = typedef ;;
type ('a1 ...'an) tip = typedef ;;
```

Articolul

La fel ca în Pascal sau C există şi în ML posibilitatea utilizării tipului de articol, articolul fiind un conglomerat de tipuri, fiecare element (câmp) a articolului având propriul nume. Declaraţia unui articol se face cu sintaxa:

```
type articol = { camp1 : tip1; ...; campn : tipn } ;;
```

Număr raţional

```
# type numarrat = { num: float ; den : float};;
type numarrat = { num : float; den : float; }
```

Atribuirea valorii campurilor articolului se face asignând valori câmpurilor articolului într-o ordine arbitrară:

```
{ campi1 = expr1; ...; campin = exprn } ;;
```

Exemplu:

```
# let numar = {den =2.;num = 3.};;
val numar : numarrat = {num = 3.; den = 2.}
# numar = {num = 1.+.2.; den = 2.};;
- : bool = true
```

Accesarea valorii unei câmp se poate face prin notaţia obişnuită cu punct. O alternativă este potrivirea şabloanelor conform sintaxei:

```
{ namei = pi ; ...; namej = pj }
```

unde pi sunt şabloane formate de obicei din variabile. Nu este necesar enumerarea tuturor câmpurilor articolului.

Exemplu:

Incrementearea cu 1 a unui număr raţional

```
# let increment p = {num = p.num +. p.den; den = p.den};;
val increment : numarrat -> numarrat = <fun>
# let increment p = match p with
  { num = n ; den = d} -> {num = n+.d; den = d};;
val increment : numarrat -> numarrat = <fun>
# increment {num = 4.; den = 2.};;
-       : numarrat = {num = 6.; den = 2.}
```

Tipuri cu variante

Tipurile cu variante sunt asemănatoare tipului union din C, sau a articolului cu variante din Pascal: dependent de un selector tipul are o anumita structură cu diferite câmpuri.

```
type nume = ...
    | Constructori ...
    | Constructorj of tipj ...
    | Constructork of tipk * ...* tipl ...;;
```

Constructorx se numeşte constructor şi este un identificator special cu care se identifică structura curentă a variabilei. Contructorul trebuie totdeauna să înceape cu majusculă.

Exemple:

```
# type calificativ = Admis | Respins;;
type calificativ = Admis | Respins
# type nota = Patru | Cinci | Sase | Sapte | Opt | Noua | Zece;;
type nota = Patru | Cinci | Sase | Sapte | Opt | Noua | Zece
# type examen = Examen of nota*prezentare
      | Colocviu of calificativ;;
# type prezentare = int;;
type prezentare = int
# type examen = Examen of nota * prezentare
      | Colocviu of calificativ
type examen = Examen of nota * prezentare | Colocviu of calificativ
```

Iniţializarea unei variabile se face folosind constructorul specificând şi valorile corespunzătoare variantei.

Instanţierea tipurilor cu variante:

```
# let practica = Colocviu Admis;;
val practica : examen = Colocviu Admis
# let cflp1 = Examen(Zece,1);;
val cflp1 : examen = Examen (Zece, 1)
```

Procesarea variabilelor cu variante se face prin potrivirea şabloanelor.

Conversia tipului examen la string:

```
# let string_of_calificativ = function
    Admis -> "admis."
  | Respins -> "respins.";;
val string_of_calificativ : calificativ -> string = <fun>
# let string_of_nota = function
    Patru -> "patru"
```

```
   | Cinci -> "cinci"
   | Sase  -> "sase"
   | Sapte -> "sapte"
   | Opt   -> "opt"
   | Noua  -> "noua"
   | Zece  -> "zece";;
val string_of_nota : nota -> string = <fun>
# let string_of_examen = function
     Colocviu calif -> "Colocviul cu calificativ " ^
     string_of_calificativ calif
   | Examen (n,p) -> "Prezentarea " ^
     string_of_int p ^ " cu nota " ^ string_of_nota n;;
val string_of_examen : examen -> string = <fun>
# string_of_examen cflp1;;
- : string = "Prezentarea 1 cu nota zece"
```

Tipuri recursive

Folosind tipuri cu variante pot fi declarate structuri recursive. Declaraţia unui nod al unui arbore. Următorul exemplu defineşte un nod al unui arbore binar ordonat.

```
# type 'tipelement nod = Empty
   | Nod of ('tipelement nod*'tipelement*'tipelement nod);;
type 'a nod = Empty | Nod of ('a nod * 'a * 'a nod)
```

Se observă că nodul conţine un element de tip oarecare.

```
# let a= Nod (Empty,1,Empty);;
val a : int nod = Nod (Empty, 1, Empty)
```

PROBLEME

Problema 1

Scrieţi funcţiile **headoflist** şi **tailoflist** folosind mecanismul de potrivirea şabloanelor. Nu se vor folosi List.hd sau List.tl.

Problema 2

Scrieti funcţiile **rotate_left** şi **rotate_right** care rotesc (o singura data) spre stânga respectiv spre dreapta o listă, folosind match La rotate_left nu se vor folosi List.hd , List.tl , iar la rotate-right nu se foloseste functia rotate-left.

Rotate_left: [1;2;3;4] -> [2;3;4;1]
Rotate_right:[1;2;3;4] -> [4;1;2;3]

Problema 3

Scrieţi funcţiile care **inserează** un element într-un arbore, **caută** un element în arbore şi care returnează elementele arborelui în **inordine** şi **preordine**.