5 Asocieri (dicționare) în ML

Putem modela relațiile cu ajutorul modulului *asociere* (`Map`), prin care *unora* din valorile unui tip numit convențional *cheie* (*key*) le sunt asociate valori dintr-un tip numit *domeniu* (sau tip valoare). Tipul asociere se mai numește *dicționar*, deoarece pentru fiecare cheie conține definiția (valoarea) sa. `Map` definește deci o *funcție parțială* de pe tipul cheie pe tipul valoare. Putem reprezenta relații arbitrare între mulțimile A și B dacă alegem ca și tip valoare *mulțimi* cu elemente de tip B.

Ca și pentru mulțimi, e necesar să avem un modul care stabilește o relație de ordine pe tipul cheie (`Map` nu impune constrângeri tipului valoare). De exemplu, creăm un modul `Map` cu chei de tip șir:

```
module M = Map.Make(String)
```

Putem scrie apoi

```
let m = M.add "x" 5 M.empty
```

Funcția `add` ia o cheie k, o valoare v și o asociere m și creează o *altă* asociere cu același conținut ca și cea dată m, dar care asociază lui k valoarea v. Orice asociere anterioară a lui k (dacă exista) nu se păstrează în rezultat.

De subliniat că `add` *nu modifică* asocierea inițială (conform cu modul de programare funcțional), ci creează o *altă* asociere. Valoarea `M.empty` reprezintă asocierea vidă (cu cheie *string*, dar fără vreun tip valoare precizat). Modulul declarat `M` poate fi folosit în același program pentru a crea mai multe asocieri cu domenii distincte, însă fiecare asociere poate avea un singur domeniu (tip valoare). De exemplu, la valoarea `m` declarată mai sus putem adăuga mai departe doar chei șir cu valori întregi.

Pentru a regăsi o valoare într-o asociere, folosim funcția `find`.

`M.find "x" m` va returna valoarea 5.

Asocierile sunt obiecte compuse (abstracte) care nu sunt vizualizate implicit de interpretor. Putem transforma un dicționar într-o listă de asociere (listă de perechi cheie-valoare) cu funcția *bindings*: `M.bindings m` returnează lista `[("x", 5)]`. Putem tipări o asociere prin parcurgere cu funcția `M.iter`.

5.1 Lucrul cu excepții în ML

Dacă încercăm să căutăm o cheie inexistentă într-o asociere, se generează o excepție:

```
M.find "y" (M.singleton "x" 5)
Exception: Not_found.
```

`M.singleton` creează o asociere cu o singură pereche cheie-valoare, deci nu există asociere pentru `"y"`.

Excepțiile sunt un mecanism software prin care se semnalează (după cum sugerează numele) *condiții excepționale* care trebuie *tratate* (argument invalid, împărțire la zero, eroare la citire, etc.). Distingem două aspecte: *generarea* excepțiilor și *tratarea* lor.

Am văzut deja funcții standard care generează excepții, cum ar fi `List.hd` și `List.tl` cu lista vidă. În funcții scrise de noi putem genera o excepție apelând funcția `raise` *excepție*. În ML, excepțiile sunt și ele valori fac parte dintr-un tip special `exn`. Sunt predefinite câteva excepții: `Not_found` (folosită de funcțiile de căutare), `Failure` cu un șir (mesaj de eroare) pentru funcții care eșuează, nefiind definite pentru argumentele date – de exemplu `List.nth [1;2;3] 5` și `Invalid_argument` tot cu un șir pentru a semnala argumente nepotrivite (care nu au sens), cum ar fi `List.nth [1;2;3] (-1)` . Pentru ultimele două există funcțiile predefinite `failwith "mesaj"` echivalentă cu `raise (Failure "mesaj")` și `invalid_arg "altmesaj"` echivalentă cu `raise (Invalid_argument "altmesaj")`.

Când o funcție generează o excepție, execuția ei se încheie *fără a returna o valoare*. Fluxul de control (execuția normală) al programului se modifică și excepția *se propagă* în sus la funcția apelantă, de acolo la funcția care a apelat-o pe aceasta, etc. Dacă undeva în codul care a apelat funcția care a provocat excepția e prevăzut cod pentru *tratarea excepției* respective, execuția continuă cu acel fragment de cod. Altfel, execuția întregului program e abandonată (ca în exemplul cu `M.find`).

În ML, tratarea excepțiilor se face cu construcția

`try` *expresie în care poate apărea o excepție*
`with` *tipar pentru tratarea excepțiilor*

Orice expresie (fragment de program) în care poate apărea o excepție trebuie inclusă într-un `try ... with ...`, altfel, la apariția excepției, netratate, întreg programul va fi abandonat.
Potrivirea de tipare după `with` are sintaxa deja întâlnită *tipar1* `->` *expr1* `|` *tipar2* `->` *expr2* etc.
unde *tipar1*, *tipar2* etc. se potrivesc cu *excepții*. Pentru a scrie tratarea excepțiilor, trebuie să știm deci

ce excepții pot fi generate de funcțiile folosite. Căutarea într-o asociere generează excepția `Not_found`. Putem scrie deci o funcție:

```
let find_default k m =
  try M.find k m
  with Not_found -> 0
```

care caută o cheie într-un dicționar *cu valori întregi*, și returnează 0 dacă cheia nu e găsită. (Dicționarul trebuie să aibă valori întregi, deoarece o funcție trebuie să returneze întotdeauna același tip).

Excepțiile sunt produse de operații sau funcții standard, sau le putem genera noi, cu funcția `raise`. Excepțiile sunt variante ale tipului special `exn` și numele de excepții sunt constructori (cu sau fără argumente), deci sunt scrise cu majusculă. Excepția `Exit` e predefinită pentru a fi folosită de utilizator.

De exemplu, putem scrie funcția de test de membru într-o listă folosind o parcurgere standard, pe care o întrerupem când elementul a fost găsit:

```
let mem x lst =
  try List.iter (fun e -> if x = e then raise Exit) lst; false
  with Exit -> true
```

Pe fiecare element al listei e aplicată funcția din paranteză, care generează o excepție dacă elementul curent e cel căutat. Dacă elementul nu există în listă, nu se generează nicio excepție, `List.iter` se termină normal, și valoarea din `try` e cea a ultimei expresii, `false`. Altfel, se generează excepția `Exit`, care e tratată în partea de `with`, cu rezultatul `true`.

Putem defini excepții proprii care au și argumente și putem transmite astfel informație utilă (chiar rezultate) spre locul de tratare a excepției. De exemplu, vrem să calculăm suma tuturor numerelor pozitive dintr-o listă până la apariția primului număr negativ sau nul. Putem scrie

```
exception ExcIntreg of int
let pospart lst =
  try List.fold_left (fun r e -> if e > 0 then r + e else raise (ExcIntreg r)) 0 lst
  with ExcIntreg r -> r
```

Apelând `pospart [1;2;3;-4;5]` obținem valoarea 6. Aceasta ne dă posibilitatea de a continua să folosim parcurgerile standard (mai ales pentru liste și dicționare care nu identifică un "prim" element anume) și să întrerupem parcurgerea în momentul în care avem rezultatul.

5.2 Crearea unui dicționar dintr-o listă de asociere

Să construim acum un dicționar pornind de la o listă de asociere (listă de perechi). Ne fixăm din nou șiruri ca tip de cheie.

```
module M = Map.Make(String)
let map_of_assoc lst =
  List.fold_left (fun m (a, b) -> M.add a b m) M.empty lst
let m = map_of_assoc [("x", 3); ("num", 17); ("y", 7); ("x", 5)]
```

Putem examina apoi `M.bindings m`, cu rezultatul `[("num", 17); ("x", 5); ("y", 7)]`
Funcția folosită cu `List.fold_left` preia la fiecare pas un rezultat parțial care e un dicționar și o pereche (cheie, valoare) din listă, și creează un nou dicționar care conține perechea ca nouă asociere. O nouă asociere la aceeași cheie o suprascrie pe prima – în final, cheia `"x"` are valoarea 5 și nu 3.

5.3 Relații ca dicționare cu valori de tip mulțime

Să implementăm acum o relație care poate asocia unei chei mai multe valori – vom stoca deci în dicționar un tip *mulțime*. Când adăugăm o nouă pereche (cheie, valoare) trebuie să obținem întâi vechea mulțime de valori asociată cheii. Dacă dicționarul nu conține cheia, mulțimea de valori asociată acesteia e vidă. Scriem o funcție care tratează și cazul de excepție:

```
let get m k =
  try M.find k m
  with Not_found -> S.empty
```

Am presupus că avem un modul `M` pentru dicționar și altul `S` pentru mulțimi cu tipul dorit de valori. Scriem acum o funcție care creează un dicționar dintr-o listă de asocieri (perechi) unde fiecare cheie ar putea să apară de mai multe ori. Parcurgem lista cu `List.fold_left`, acumulând la fiecare pas noul dicționar. Funcția de actualizare (pe care o numim `addpair`) obține întâi cu `get` valoarea (aici: o mulțime de valori) asociată cheii `a` (primul element din pereche), adaugă la ea al doilea element din pereche (`b`) și asociază noua mulțime cu `a`. Adăugarea începe de la asocierea vidă `M.empty`.

```
let setmap_of_assoc lst =
  let addpair m (a, b) = M.add a (S.add b (get m a)) m
  in List.fold_left addpair M.empty lst
```

Pentru o soluție cât mai generală, scriem un modul (functor) parametrizat cu module pentru tipul cheie și tipul element de mulțime,

```
module SetMap(Key: Map.OrderedType)(Val: Set.OrderedType) =
struct
  module M = Map.Make(Key)
  include M    (* permite folosirea directa a functiilor din M *)
  module S = Set.Make(Val)
  let get m k = try M.find k m with Not_found -> S.empty
  let setmap_of_assoc lst =
    let addpair m (x, y) = M.add x (S.add y (get m x)) m
    in List.fold_left addpair M.empty lst
end
```

Instanțiem acum un astfel de modul, după ce definim obișnuitul modul pentru întregi:

```
module Int = struct     type t = int     let compare = compare     end
module SM = SetMap(String)(Int)
```

Sintagma `include M` din modulul `SetMap` ne permite să folosim direct funcții cu numele `SM.add`, `SM.find`, etc., în loc de `SM.M.add`, `SM.M.find`, etc. ele fiind de fapt funcții din modulul `M` declarat în `SetMap`. Spre comparație, pentru funcțiile de mulțimi trebuie să scriem `SM.S.elements`, etc.
Putem crea atunci dicționarul dorit apelând

```
let m = (SM.setmap_of_assoc [("x", 3); ("y", 6); ("z", 1); ("y", 5)])
```

Pentru a vizualiza în interpretor dicționarul, putem obține lista de asocieri folosind `SM.bindings`. Al doilea element al fiecărei perechi e o mulțime; putem să o transformăm în listă, care poate fi afișată:

```
List.map (fun (a, b) -> (a, SM.S.elements b)) (SM.bindings m)
# - : (SM.key * SM.S.elt list) list = [("x", [3]); ("y", [5; 6]); ("z", [1])]
```

Remarcăm că `"y"` are asociate două valori, 5 și 6.

Sau, putem scrie direct funcții de tipărire, întâi pentru mulțimi și apoi pentru dicționar:

```
let print_set s =
  print_char '{';
    if s <> SM.S.empty then (let e1 = SM.S.min_elt s
    in print_int e1; SM.S.iter (Printf.printf ", %d") (SM.S.remove e1 s))
  print_char '}'
let print_map m = SM.iter (fun k v -> Printf.printf " %s -> " k; print_set v) m
```

Parcurgerea cu `SM.iter` are nevoie de o funcție de doi parametri, cheie și valoare. Se va afișa `x -> {3} y -> {5, 6} z -> {1}`. Aceste funcții se puteau scrie și în modulul `SetMap`, devenind parte din el și fiind disponibile pentru fiecare instanțiere a modulului.

5.4 Închiderea tranzitivă a unei relații

Pentru o relație binară pe o mulțime, putem calcula închiderea tranzitivă a relației. De exemplu, pentru mulțimea $A = \{a, b, c, d\}$ și relația $R = \{(a, b), (b, c), (c, d), (b, a)\}$, obținem:
$R^2 = \{(a, a), (a, c), (b, b), (b, d)\}$ și $R \cup R^2 = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c), (b, d), (c, d)\}$
$R^3 = \{(a, b), (a, d), (b, a), (b, c)\}$, $R \cup R^2 \cup R^3 = \{(a, a), (a, b), (a, c), (a, d), (b, a), (b, b)(b, c), (b, d), (c, d)\}$
și apoi puterile superioare nu mai adaugă alte perechi noi.

Particularizăm modulul `SetMap` cu *același* tip pentru cheie și elementele mulțimii-valoare:

```
module BinRel(Ord: Map.OrderedType) = struct
  module SM = SetMap(Ord)(Ord) (* tip cheie = tip valoare *)
  include SM
  (* aici vom adauga functiile noastre scrise mai jos *)
end
```

Vom privi relația ca funcție cu valori în mulțimea părților, $f_R(x) = \{y \mid R(x,y)\}$. În exemplul nostru, $f_R = \{a \mapsto \{b\}, b \mapsto \{a,c\}, c \mapsto \{d\}, d \mapsto \{\}\}$, reprezentând-o explicit prin perechi. Calculăm apoi relația $R_{12} = R \cup R^2$. Putem scrie $f_{R_{12}}(x) = f_R(x) \cup_{e \in f_R(x)} f_R(e)$: de la x parcurgem relația o dată cu $f_R(x)$ sau de două ori – încă o dată pentru fiecare element din mulțimea $f_R(x)$. Obținem deci $f_{R_{12}}(x) = \{a \mapsto \{a,b,c\}, b \mapsto \{a,b,c,d\}, c \mapsto \{d\}, d \mapsto \{\}\}$.

Dată fiind o funcție $f : A \to \mathcal{P}(A)$, vrem o funcție care pentru o mulțime s să calculeze $s \cup_{e \in s} f(e)$. Facem aceasta prin parcurgere cu `S.fold`: pentru fiecare element `e`, reunim `f e` cu mulțimea deja acumulată, pornind de la `s`.

```
let add_image f s = S.fold (fun e r -> S.union (f e) r) s s
```

sau, folosind compunerea `let comp f g x = f (g x)`:

```
let add_image f s = S.fold (comp S.union f) s s
```

În implementarea noastră, relația R e dicționarul `m`, și funcția f_R e `get m`, care din cheia `x` ne dă mulțimea `s = get m x`, adică $f_R(x)$. Apoi, `add_image (get m) s` ne dă $s \cup_{e \in s} f_R(e)$, adică $f_{R_{12}}(x)$. Deci, obținem $f_{R_{12}}(x)$ în doi pași: o mapare `s = get m x` folosind dicționarul `m`, apoi aplicând la `s` funcția `add_image (get m)`. Am reușit să exprimăm astfel R_{12} tot ca dicționar. El poate fi obținut din `m` cu funcția standard `SM.map` care transformă un dicționar în alt dicționar cu aceleași chei, dată fiind funcția care transformă prima valoare în a doua – aici, `add_image (get m)`.

```
let add_sqr m = SM.map (add_image (get m)) m
```

Închiderea tranzitivă se poate calcula aplicând `add_sqr` până la punct fix, pentru că la fiecare pas, numărul maxim de aplicări al relației se dublează. Rămâne să scriem funcția de punct fix.

Ținem cont că dicționarele nu se pot compara cu `=`, fiindcă asocieri egale pot avea reprezentări structural diferite. Parametrizăm deci funcția de punct fix cu o funcție de comparație `eq`. În cazul de față, folosim funcția `SM.equal`, unde primul parametru e funcția de comparație pentru valori. Acestea sunt mulțimi, deci și pentru ele folosim funcția `equal`, cu modulele noastre, `SM.S.equal`:

```
let fix eq f =
  let rec fix1 x =
    let y = f x in if eq x y then x else fix1 y
  in fix1
let transclose m = fix (SM.equal SM.S.equal) (add_sqr m) m
```

Construim `m = M.setmap_of_assoc [('a', 'b'); ('b', 'c'); ('c', 'd'); ('b', 'a')]`, după ce am creat `module M = BinRel(Char)`, și apoi tipărim `M.transclose m` (de exemplu cu funcția `print_map` anterior definită). Obținem rezultatul calculat manual la începutul secțiunii.

5.5 Citirea de la intrare cu `scanf`

Când scriem programe care citesc de la intrare, e preferabil să le rulăm din interpretorul lansat în terminal, sau și mai bine, de sine stătător, din terminal, compilând întâi cu `ocamlc program.ml` și apoi rulând `./a.out` (sau `./a` în Windows). Dacă rulăm din Emacs, caracterele `;;` pe care le introducem la sfârșit pot fi interpretate nedorit ca făcând parte din textul dat la intrare.

Funcția cea mai versatilă pentru citire în OCaml e `scanf`. Dacă o folosim în mod repetat, e util să deschidem modulul respectiv: `open Scanf`. Funcția `scanf` ia doi parametri: un *șir de format*, și *o funcție* care va fi apelată pe valorile citite; rezultatul eidevine și rezultatul apelului la `scanf`.

Șirul de format are un rol similar celui din C: specificatorii de format care încep cu `%` precizează ce se dorește citit: `%s` (șir), `%c` (caracter), `%d` (întreg), `%f` (real). Un caracter obișnuit în șirul de format trebuie să se regăsească în intrare pentru ca citirea să aibe succes. Spre deosebire de C, la citirea formatelor șir și numerice nu se consumă spațiile albe inițiale. Pentru aceasta, precedați `%d`, `%s`, etc. cu un spațiu în șirul de format; aceasta produce citirea și ignorarea oricâtor caractere de tip spațiu alb (spațiu, tab, linie nouă, etc.).

Putem citi un șir dând funcția identitate ca parametrul 2: `let s = scanf "%s" (fun x -> x)`
Pentru a tipări direct șirul citit putem scrie: `scanf "%s" print_string`
Funcția dată ca parametru la `scanf` trebuie să aibe atâția parametri câte elemente sunt citite. Pentru a citi două numere, a le calcula și afișa suma: `print_int (scanf "%d %d" (+))`

Citirea unui șir `%s` produce întotdeauna o valoare; ea este șirul vid dacă la intrare nu urmează un șir (caractere diferite de spații), sau s-a ajuns la sfârșitul intrării.

Putem scrie atunci o funcție care citește și tipărește toate șirurile citite de la intrare: dacă șirul citit e nevid, funcția îl tipărește și se reapelează recursiv, altfel nu face nimic.

```
let rec allstrings () =
  scanf " %s" (fun s -> if s <> "" then (printf "%s " s; allstrings ()))
```

După același principiu, e util să scriem o funcție care citește șiruri de la intrare, și le aplică succesiv o prelucrare a cărei rezultat e acumulat, similar cu `List.fold_left`, doar că lista e implicită, dată de succesiunea șirurilor în intrare:

```
let rec scanfold_s f r =
  scanf " %s" (fun s -> if s = "" then r else scanfold_s f (f r s))
```

Dacă citirea șirului eșuează (șirul vid), se returnează rezultatul `r`. Altfel, se aplică funcția `f` rezultatului acumulat `r` și șirului citit `s`, și funcția e apelată recursiv cu noul rezultat parțial. Cum parametrul `f` nu se schimbă, putem rescrie cu o funcție ajutătoare de un singur parametru:

```
let scanfold_s f =
  let rec scan2 r = scanf " %s" (fun s -> if s = "" then r else scan2 (f r s))
in scan2
```

De exemplu, putem calcula și afișa suma lungimilor tuturor șirurilor din intrare (excluzând spațiile): `print_int (scanfold_s (fun r s -> r + String.length s) 0)` (pornind de la valoarea 0).

Revenind la folosirea dicționarelor, putem calcula numărul de apariții ale fiecărui cuvânt (șir) dintr-un text citit de la intrare. Pentru aceasta, actualizăm la fiecare pas un dicționar cu numărul de apariții. Se caută întâi șirul citit în dicționar, obținând contorul curent (sau 0 în caz de excepție, cuvânt negăsit). Apoi se creează un dicționar nou, cu contorul pentru acel șir incrementat cu 1.

```
module M = Map.Make(String)
let m = scanfold_s (fun m s ->
                let cnt = try M.find s m with Not_found -> 0
                in M.add s (cnt+1) m) M.empty
let _ = M.iter (printf "%s: %d\n") m
```

Spre deosebire de șiruri, pentru valori numerice `scanf` generează excepții dacă citirea eșuează: excepția `Scan_failure` (cu un mesaj ca parametru), sau excepția `End_of_file`. Pentru a scrie o funcție similară cu `scanfold_s` pentru valori numerice, trebuie să tratăm aceste excepții.

```
let scanfold fmt f =
  let rec scanfold2 r =
    (try let r1 = scanf fmt (f r) in fun () -> scanfold2 r1
    with End_of_file | Scan_failure _ -> fun () -> r) ()
  in scanfold2
```

Funcția are ca parametru și formatul `fmt` pentru `scanf`. În primul rând, se încearcă citirea conform formatului; în caz de succes, `scanf` transmite valorile citite (formatul poate specifica mai multe) ca argumente la funcția `f r` (`f` aplicată parțial pe primul argument, care e rezultatul acumulat); `scanf` va returna valoarea produsă de `f`, notată cu `r1`. Ramura de prelucrare normală cât și cea de excepție produc o funcție `fun () -> ...` de argument `unit` ("fără argumente"), care e apelată `()` în final. Pe ramura de excepție, funcția returnează valoarea `r`, calculul fiind încheiat. Altfel, se apelează recursiv `scanfold2` cu noul rezultat. Codul a fost scris în acest fel pentru ca eventuala tratare a excepției să se facă *înaintea* apelului recursiv, care rămâne ultima operație – deci funcția este final recursivă (tail recursive) și va putea trata un șir oricât de lung de date de intrare.

Ca exemplu simplu de folosire, putem scrie `print_int (scanfold " %d" (+) 0)`, care va citi și aduna toate numerele (separate prin spații) citite de la intrare, afișând în final rezultatul.