2023-10-16 - Currying, Map, Filter, Reduce, Fold, Exists, For all, Zip, Group by, Pairwise, Enumerate, Numero variabile di parametri

Currying

Il **currying** è una tecnica che trasforma una funzione con **molti argomenti** in una **catena** di funzioni ognuna con un **solo argomento** (attraverso *applicazioni parziali*).

$f(x,y) = \frac{y}{x} \Rightarrow 2 \Rightarrow f(2) = \frac{y}{2} \Rightarrow 3 \Rightarrow f(2)(3) = \frac{3}{2}$

In OCaML, il currying è **gestito automaticamente**, attraverso le *applicazioni parziali* di funzioni. È possibile anche utilizzare **parametri con nome** per non passare solo il *primo parametro*.

```
let f x y z = x +. y *. z;;
val f : float -> float -> float -> float = <fun>

f 5. 3. 7.;;
- : float = 26

let f = f 5.;;
val f : float -> float -> float = <fun>

let f = f 3.;;
val f : float -> float = <fun>

f 7.;;
- : float = 26
```

Pattern di programmazione funzionale ricorrenti

Map

La funzione `map` applica una **funzione** (`f`) a **tutti gli elementi** in una lista, generando una nuova lista con tutti i risultati.

```
(* possibile implementazione: *)
let rec map f list =
    match list with
        | h::l -> (f h)::(map f l)
        | _ -> [];;
(* map : ('a -> 'b) -> 'a list -> 'b list *)

let l = [1;2;3;7;25;4];;
map (fun x -> (x mod 2) = 0) l;;
(* [false; true; false; false; false; true] *)
```

Filter

La funzione `filter` applica un **filtro** (una funzione booleana, un predicato `p`) ad **ogni elemento**, generando una nuova lista con **solo gli elementi** che hanno restituito `true`.

```
(* possibile implementazione: *)
let rec filter p list =
    match list with
        | [] -> []
        | h::l -> if p h
            then h :: (filter p l)
            else filter p l;;
(* filter : ('a -> bool) -> 'a list -> 'a list *)

let l = [1;2;3;7;25;4];;
```

```
filter (fun x -> (x mod 2) == 0) l;;
(* [2;4] *)
```

Reduce

La funzione `reduce` **riduce** una lista ad un **singolo valore** (**accumulatore** `acc`), grazie ad una **funzione accumulatrice** (operazione `op`).

```
(* possibile implementazione: *)
let rec reduce acc op list =
    match list with
        | [] -> acc
        | h::l -> reduce (op acc h) op l;;
(* reduce : 'a -> ('a -> 'b -> 'a) -> 'b list -> 'a *)

let l = [1;2;3;7;25;4];;
reduce 0 (+) l;; (* 42 *)
reduce 100 (+) l;; (* 142 *)
reduce 1 ( * ) l;; (* 4200 *)
```

Fold

L'operazione di fold è la **costruzione di un valore** *(riduzione della lista ad un valore)* attraverso una **funzione binaria** (tra due elementi) scorrendo **ricorsivamente** gli elementi di un dataset. L'operazione `reduce` è un caso particolare del folding.

```
((((((0+1)+2)+3)+7)+25)+4) left fold
(0+(1+(2+(3+(7+(25+4)))))) right fold
```

Exists

La funzione `exists` restituisce `true` se **almeno un elemento** rispetta una **funzione booleana** passata (predicato `p`), `false` altrimenti. È possibile implementarla sfruttando le funzioni `reduce` e `map`.

```
(* possibile implementazione: *)
let exists p l = reduce false (||) (map p l);;
(* exists : ('a -> bool) -> 'a list -> bool *)

let l = [1;2;3;7;25;4];;
exists (fun x -> x=100) l;; (* false *)
exists (fun x -> x=2) l;; (* true *)
exists (fun x -> (x mod 2)==0) l;; (* true *)
```

For all

La funzione `forall` restituisce `true` se **tutti gli elementi** rispettano una **funzione booleana** passata (predicato `p`), `false` altrimenti. È possibile implementarla sfruttando le funzioni `reduce` e `map`.

```
(* possibile implementazione: *)
let forall p l = reduce true (&&) (map p l);;
(* forall : ('a -> bool) -> 'a list -> bool *)

let l = [1;2;3;7;25;4];;
forall (fun x -> x=100) l;; (* false *)
forall (fun x -> x=2) l;; (* false *)
forall (fun x -> (x mod 2)==0) l;; (* false *)
forall (fun x -> (x mod 1)==0) l;; (* true *)
```

Zip

La funzione `zip` **accoppia gli elementi** allo stesso indice di due liste, restituendo una **lista di coppie** *(tuple di due elementi)*. Se le due liste sono di lunghezza diversa, gli elementi "extra" saranno **persi**.

```
(* possibile implementazione *)
let rec zip l1 l2 =
    match (l1, l2) with
        | ([], []) | (_, []) | ([], _) -> []
        | (h1::ll1, h2::ll2) -> (h1, h2) :: (zip ll1 ll2);;
(* zip : 'a list -> 'b list -> ('a * 'b) list *)

let l0 = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;
let l1 = ['a';'b';'c';'d'];;
zip l0 l1;; (* [(1,'a');(2,'b');(3,'c');(4,'d')] *)
```

Group by

La funzione `group_by` **raggruppa** gli elementi di una lista in gruppi di elementi che rispettano la **stessa proprietà** (funzione booleana passata `f`).

```
(* possibile implementazione *)
type 'a group = { mutable g: 'a list };;
let empty_group = function x -> { g = [] };;
let rec group_by l ?(ris:'a group array = (Array.init 10 empty_group)) f =
    match l with
        | [] -> ris
        | h::l1 ->
            ( ris.((f h)).g <- ris.((f h)).g@[h];
            group_by l1 ~ris:ris f );;
(* group_by : 'a list -> ?ris:'a group array -> ('a -> int) -> 'a group array *)

let l = [10; 11; 22; 23; 45; 25; 33; 72; 77; 16; 30; 88; 85; 99; 9; 1];;
group_by l (fun x -> x/10);;
(* [|{g = [9; 1]}; {g = [10; 11; 16]};
    {g = [22; 23; 25]}; {g = [33; 30]};
    {g = [45]}; {g = []};
    {g = []}; {g = [72; 77]};
    {g = [88; 85]}; {g = [99]}|] *)
```

Pairwise

La funzione `pairwise` **accoppia** gli elementi di una lista *(a due a due, in ordine)*, restituendo una lista di **coppie** *(tuple)*.

```
(* possibile implementazione *)
let rec pairwise list =
    match list with
        | h1::h2::l -> (h1,h2)::(pairwise (h2::l))
        | _ -> [];;
(* pairwise : 'a list -> ('a * 'a) list *)

let l = ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h'; 'i'];;
pairwise l;;
(* [('a', 'b'); ('b', 'c'); ('c', 'd'); ('d', 'e');
    ('e', 'f'); ('f', 'g'); ('g', 'h'); ('h', 'i')] *)
```

Enumerate

La funzione `enumerate` **accoppia** ogni elemento di una lista con il **suo indice**, restituendo una lista di **coppie** (`indice * elemento`).

```
(* possibile implementazione *)
let enumerate l =
    let rec enumerate acc n list =
        match list with
```

```
            | h :: ls -> enumerate ((n,h)::acc) (n+1) ls
            | [] -> List.rev acc
    in enumerate [] 0 l;;
(* enumerate : 'a list -> (int * 'a) list *)

let l = ['a'; 'b'; 'c'];;
enumerate l;; (* [(0, 'a'); (1, 'b'); (2, 'c')] *)
```

Pacchetto List

La maggior parte dei pattern elencati sono già **implementati** nel pacchetto `List`:

- **map**: `List.map : ('a -> 'b) -> 'a list -> 'b list`
- **filter**: `List.filter : ('a -> bool) -> 'a list -> 'a list`
- **reduce e fold**: `List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` (oppure `List.fold_right`)
- **exists**: `List.exists : ('a -> bool) -> 'a list -> bool`
- **for all**: `List.for_all : ('a -> bool) -> 'a list -> bool`
- **zip**: *non implementata*
- **group** by: *non implementata*
- **pairwise**: *non implementata*
- **enumerate**: *non implementata*

Funzioni con numero di parametri variabile

È possibile sfruttare il concetto del `reduce` per realizzare una funzione con un **numero variabile di parametri**.
Al posto di avere una *lista di parametri concreta*, utilizziamo una funzione (`arg`) che dopo aver **accumulato il parametro** passato (`x`) restituisce una **nuova funzione** che accetta un **altro argomento** con l'**accumulatore aggiornato** (`rest`). Per **interrompere la catena** basta una funzione **identità sull'accumulatore** (`stop`), che lo restituisce.
Per far funzionare questi elementi deve essere definita un **operazione da effettuare** tra parametro e accumulatore (`op`) e l'inizializzazione dell'**accumulatore** (`init`).

```
(* operazione da eseguire su ogni parametro *)
let op x y = x + y;;
(* inizializzazione dell'accumulatore *)
let init = 0;;

(* singolo parametro *)
let arg x = (fun y rest -> rest (op x y));;
(* terminazione catena parametri *)
let stop x = x;;
(* inizializzazione funzione con numero di parametri variabile *)
let f g = g init;;

f (arg 1) stop;; (* 1 *)
f (arg 10) (arg 20) stop;; (* 30 *)
f (arg 100) (arg 200) (arg 300) stop;; (* 600 *)
```

È possibile implementare una *struttura* che effettui queste operazioni, attraverso l'uso di un **funtore**, che accetta in input un modulo che rispetta l'interfaccia `OpVarADT`:

- `a`, `b`, `c` (i tipi di dati usati da `op`)
- `op` (l'operazione necessaria, presi due dati `a`, `b` restituisce un `c`)
- `init` (l'inizializzazione dell'accumulatore, di tipo `c`)

Sfrutta poi questi *due valori* per implementare le operazioni `arg`, `stop` ed `f` in modo *analogo* a prima.

```
(* interfaccia che rappresenta le operazioni necessarie *)
module type OpVarADT =
    sig
```

```
        type a and b and c
        val op: a -> b -> c
        val init : c
    end;;

(* funtore che rappresenta la struttura vera e propria *)
module VarArgs (OP : OpVarADT) =
    struct
        let arg x = fun y rest -> rest (OP.op x y)
        let stop x = x
        let f g = g OP.init
    end;;
```

Esempi di utilizzo:

```
module Sum = struct
    type a=int and b=int and c=int
    let op = fun x y -> x+y
    let init = 0
end;;

module VASum = VarArgs(Sum);;
VASum.f (VASum.arg 1) (VASum.stop);; (* 1 *)
VASum.f (VASum.arg 1) (VASum.arg 5) (VASum.stop);; (* 6 *)

module StringConcat = struct
    type a=string and b=string list and c=string list
    let op = fun (x: string) y -> y @ [x]
    let init = []
end;;

module VAStr = VarArgs(StringConcat);;
VAStr.f (VAStr.arg "Ciao") (VAStr.stop);; (* ["Ciao"] *)
VAStr.f (VAStr.arg "Hel") (VAStr.arg "lo") (VAStr.stop);; (* ["Hel"; "lo"] *)
```