

Sommario: 22 Maggio, 2019

- Programmazione Funzionale: Principi e Proprietà
 - > Modello di Calcolo
 - > Trasparenza Referenziale
- Strutture Fondamentali:
 - > Espressione, Applicazione, Strategia di Valutazione
 - > Sistema di Tipi e Polimorfismo Generico
 - > API-ADT: Modulo Segnature, Moduli Implementazione
- Higher Order:
 - > Principio di Astrazione e Riutilizzo di Codice
 - > Funzioni Higher Order: `List.fold_left`
 - > `List.fold_right`
 - > `List.map`
 - > `List.filter`
 - > Definizioni Tail Recursive

- **Modello di Calcolo.** Lambda-Calcolo è il modello ispirante questi linguaggi. Ne consegue:
 - Senza Stato ¹: No Valori Modificabili
 - Trasparenza Referenziale:²

Definition

Sia P un programma ed E_i un suo termine (espressione) all'occorrenza i . Se la computazione di P riduce E_i al valore V_{E_i} , allora il programma $P[E_i \leftarrow V_{E_i}]$ è equivalente a P e può rimpiazzarlo in ogni computazione.

Ad esempio. $f(x)+f(x) \equiv 2 * f(x)$, con $+$ somma, e $*$ prodotto.

- Correttezza e Verifica: Trasparenza Referenziale permette tecniche di prova piuttosto semplici.

¹inteso come Memoria Modificabile

² $P[E \leftarrow V_E]$ indica il rimpiazzamento di ogni occorrenza di E in P con V_E

- **Modello di Calcolo.** Lambda-Calcolo è il modello ispirante questi linguaggi. Ne consegue:
 - Senza Stato
 - Trasparenza Referenziale
 - Correttezza e Verifica
 - Funzioni sono valori:
 - calcolabili da un'applicazione di funzione
 - passabili come argomenti a un'applicazione di funzione
 - Programmazione (astratta e) Higher Order
 - Astrazioni sui dati: Modellare valori con funzioni ³
 - Astrazioni sul controllo: Modellare strutture di controllo con funzioni ⁴
 - Computazione per Riduzione: Come per il Lambda-Calcolo (vedi Esercizio su SOS per Lambda-Calcolo)

³Es: Interi di Barendregt, $[n+1] = \lambda x y.x([n]xy)$, env e store in Lab.LPL

⁴Es: iteratori `fold_left` e `fold_right` in OCaml  3/26

- **Programmi** sono:
 - Una struttura (di Moduli) con definizioni di Tipi, Funzioni e Dati, che forma un ambiente (di bindings)
 - Una espressione da valutare in tale ambiente
- **Funzioni:**
 - Sono definite da espressioni:
Ad es.: (OCaml) `fun x → let...in exp`
 - Possono contenere blocchi-espressione, `let_in_`
 - Sono Programming Units: Il corpo ha la stessa struttura del programma:
- **Applicazione:** Il Meccanismo di calcolo fondamentale
- **Trasmissione e Strategia di Valutazione:**
 - OCaml: Trasmissione per valore
 - Haskell: Valutazione esterna e Lazy costruttori

Strutture Fondamentali: Tipi

- Sistema dei Tipi (Pure OCaml).
 - **Tipi Basici Scalari:** int, float, bool, char, e (char)string
 - **T. Polimorfi e Variabili di tipo.** Ad es.: 'a
 - **Tipi Basici Strutturati:** Tuples, List polimorfe (generiche).
Ad es.: ('a * int) list
 - **Tipi funzione:** \rightarrow .
Ad es.: 'a list \rightarrow ('a \rightarrow 'a \rightarrow bool) \rightarrow 'a list
 - **Tipi Concreti:** Record e Variant Types.
 - **Definiti** mediante costrutto:
type typeName = typeExpression
 - **Record.** Ad es.:
type ratio = {num:int; denum:int}
 - **Variant o Algebrici.** Ad es.:
type ('a,'b) myType = Either of 'a | Or of 'b;;
 - **Variant Ricorsivi.** Ad es.:
type 'a myList = Nil | Cons of 'a * 'a myList;;
 - **Tipi Astratti**

Tipi Algebrici o Concreti

Definition (Tipo di Dato Algebrico)

I Tipi di Dato Algebrico sono collezioni di valori (anche strutturati), detti "termini", definiti per iniezione a partire da un insieme finito $G = \{g_i : T_{i_1} \times \dots \times T_{i_k} \rightarrow T_G\}$ di funzioni iniettive, dette "costruttori", di nome g_i , e segnatura $T_{i_1} \times \dots \times T_{i_k} \rightarrow T_G$ dati, dove $k \geq 0$ e T_G sia il tipo dei valori definiti con G . Allora l'insieme di tali termini è

$$\mathcal{D}^{T_G} \equiv \{g_i(v_{i_1}, \dots, v_{i_k}) \mid v_{i_j} \in \mathcal{D}^{T_{i_j}}\},$$

dove abbiamo indicato con \mathcal{D}^T l'insieme dei valori di tipo T .

Proposition (Presentazione e Identità Sintattica)

Presentazione. *Un valore algebrico, o termine, ha sempre forma $g(v_1, \dots, v_k)$, dove g sia un costruttore di arità $k \geq 0$ e v_1, \dots, v_k valori di tipo atteso dalla segnatura di g .*

Identità Sintattica \equiv . *Siano $g(v_1, \dots, v_n)$ e $g'(v'_1, \dots, v'_m)$ due valori di uno stesso tipo di dato algebrico. Allora,*

$$g(v_1, \dots, v_n) = g'(v'_1, \dots, v'_m) \quad \text{sse } g \equiv g' \wedge n = m \wedge v_1 = v'_1 \wedge \dots \wedge v_n = v'_m$$

- In Ocaml, i tipi algebrici sono introdotti con dichiarazioni di tipo della forma:

```
type TG = g1 of (Texp1,1 * ... * Texp1,k1)  
          | ...  
          | gn of (Texpn,1 * ... * Texpn,kn)
```

Dove: - ...

Ocaml: Tipi Algebrici o Concreti

- In Ocaml, i tipi algebrici sono introdotti con dichiarazioni di tipo della forma:

$$\begin{array}{l} \text{type } T_G = g_1 \text{ of } (T_{\text{exp}_{1,1}} \star \dots \star T_{\text{exp}_{1,k_1}}) \\ \quad \quad \quad | \dots \\ \quad \quad \quad | g_n \text{ of } (T_{\text{exp}_{n,1}} \star \dots \star T_{\text{exp}_{n,k_n}}) \end{array}$$

- **Dove:**

- T_G è il nome del tipo e deve iniziare con un carattere **minuscolo**
- g_1, \dots, g_n sono i nomi dei costruttori e hanno primo carattere **maiuscolo**
- $\text{of } (T_{\text{exp}_{i,1}} \star \dots \star T_{\text{exp}_{i,k_i}})$ è omissso quando $k_i = 0$, i.e. g_i ha arità 0
- $\text{of } (T_{\text{exp}_{i,1}} \star \dots \star T_{\text{exp}_{i,k_i}})$ definisce la tupla dei tipi degli argomenti
- $T_{\text{exp}_{i,j}}$ è una qualunque espressione di tipo Ocaml
- l'ordine dei costruttori è inessenziale
- Nel caso di tipi algebrici polimorfi:
 - T_G ha prefisso la lista $'a_1, \dots, 'a_r$ delle variabili generiche
 - $T_{\text{exp}_{i,j}}$ può contenere variabili in $\{'a_1, \dots, 'a_r\}$ come libere

- **Ricorsiva** T_G può occorrere come tipo in $T_{\text{exp}_{i,k_i}}$

- **Allocazione Dinamica** Ogni valutazione di $g_i(v_{i,1}, \dots, v_{i,k_i})$ alloca dinamicamente una struttura di costruttore g_i avente k_i componenti di valore $v_{i,1}, \dots, v_{i,k_i}$ in tale ordine.

- Programmare con Tipi Algebrici richiede non solo saper:

- **Definire tipi algebrici.**

Esempio, alberi ('a, 'b) gTree in Ocaml

```
type('a, 'b)gTree = E | L of 'a | R of ('b * ('a, 'b)gTree list)
```

- **Esprimere valori** di un dato tipo algebrico.

Esempio, l'albero tree:

```
let tree = R("root1", [L 1; L 2; R("root2", [])]; E)
```

- anche saper:

- **Visitare la Struttura**

Esempio: l'albero tree è una foglia? Ha più di 1 sottoalbero?

- **Accedere Componenti.**

Esempio: se tree ha sottoalberi, qual'è il suo primo sottoalbero?

- **Modificare Componenti**

Esempio: Ocaml permette tipi algebrici a componenti modificabili

- Operazioni per Visita, Accesso e Modifica (quando ammesso) ottenuti:

- Meccanismo di Pattern-Matching (Ocaml, e L funzionali)
 - Meccanismi ad hoc

Pattern-Matching

Pattern-Matching è un meccanismo per *visitare*, *accedere*, "*modificare*" valori e componenti di tipi algebrici o concreti.

- È realizzato mediante:
 - Patterns per dato tipo algebrico
 - Operazione Match per coppie pattern-termine
- Sia $G = \{g_i : T_{i_1} \times \dots \times T_{i_k} \rightarrow T_G\}$ l'insieme dei costruttori di un Tipo Algebrico T_G , e indichiamo con X^{T_G} un insieme delle variabili di tipo T_G
 - **Termini.** $\mathcal{D}^{T_G} \equiv \{g_i(v_{i_1}, \dots, v_{i_k}) \mid v_{i_j} \in \mathcal{D}^{T_{i_j}}\}$
Ad esempio: $R(\text{"root1"}, [L\ 1; L\ 2; R(\text{"root2"}, []); E])$
è un termine di $(\text{'a'}, \text{'b'})\text{gTree}$, in Ocaml.
 - **Patterns** $\mathcal{P}^{T_G} \equiv \{g_i(v_{i_1}, \dots, v_{i_k}) \mid v_{i_j} \in \mathcal{P}^{T_{i_j}}\} \cup X^{T_G}$
Ad esempio: $R(x, [L\ 1; L\ 2; y; z])$
è un pattern con x variabile di tipo `string`, y e z di tipo $(\text{'a'}, \text{'b'})\text{gTree}$.
 - Tutte le variabili che occorrono in un pattern sono **variabili libere**
 - L'operazione match lega
- Operazione Match : $\mathcal{D}^{T_G} \times \mathcal{P}^{T_G} \rightarrow$ Sostituzione,
 - Match(t, p) definisce un ambiente i cui binding sono i legami tra le variabili (libere) del pattern p e i corrispondenti sotto-termini del termine t .

Pattern-Matching in Ocaml

Pattern-Matching è un meccanismo per *visitare*, *accedere*, *"modificare"* valori e componenti di tipi algebrici o concreti

- Operazione Match: Dato T_G ,
 - $\text{Match} : \mathcal{D}^{T_G} \times \mathcal{P}^{T_G} \rightarrow \text{Sostituzione}$
 - Sostituzione è l'insieme dei legami di variabili-valori che rendono il pattern uguale al termine dato, oppure *fail* se tale insieme non esiste.
Esempio: $\text{Match}(\text{R}(\text{"root1"}, [\text{L } 1; \text{L } 2; \text{R}(\text{"root2"}, [])]; \text{E}]), \text{R}(x, [\text{L } 1; \text{L } 2; y; z])$
calcola la sostituzione $\{x = \text{"root1"}, y = \text{R}(\text{"root2"}, []), z = \text{E}\}$
- Ocaml incorpora l'operazione Match in un costrutto `match_with`.
- Costrutto `match_with` è un condizionale multi-way avente la seguente forma:

```
match term with
| pattern1 -> exp1
| ...
| patternn -> expn
```

dove `term` è un valore di un tipo algebrico T , e `pattern1, ..., patternn` sono n patterns per lo stesso tipo T che devono fornire una copertura per T (ovvero, per ogni termine di T deve esistere un pattern nella copertura e una sostituzione per esso che rende il pattern identico al termine)

calcola `expi` tale che i è il più piccolo indice per cui:

$\text{Match}(\text{term}, \text{pattern}_i) \neq \text{fail}$

Pattern-Matching in Ocaml: Esempi

Pattern-Matching è un meccanismo per *visitare*, *accedere*, "*modificare*" valori e componenti di tipi algebrici o concreti

- Costrutto `match with` è un condizionale multi-way avente la seguente forma:

```
match term with
  | pattern1 -> exp1
  | ...
  | patternn -> expn
```

dove `term` è un valore di un tipo algebrico `T`, e `pattern1, ..., patternn` sono ...
calcola `expi` tale che `i` è il più piccolo indice per cui:

`Match(term, patteri) ≠ fail`

Esempio (Calcolo della sostituzione)

Definiamo un tipo per la presentazioni delle sostituzioni:

```
type 'a subst = Sub of 'a | Fail;;
```

ed usiamolo nel match di termini di tipo ('a, 'b)gTree definito prima.

```
let tree = R("root1", [L 1; L 2; R("root2", []); E]);;
```

```
match tree with R(x, [L 1; L 2; y; z]) -> Sub(x, y, z) | w -> Fail;;
```

Cosa otteniamo in Ocaml?

Esempio (Predicati e Selettori di gTree)

Definiamo i predicati isE, isL, isR per riconoscere i valori delle 3 diverse strutture :

```
let isE tree = match tree with E -> true | _ -> false;;
```

```
let isL tree = match tree with L _ -> true | _ -> false;;
```

```
let isR tree = match tree with R _ -> true | _ -> false;;
```

Pattern-Matching in Ocaml: Errori sulle Variabili Libere

Esempio (Errori sull'uso delle variabili nei pattern)

Definiamo in OCaml la seguente funzione:

```
let equal x y =  
  match x with  
  | y -> true  
  | _ -> false  
;;
```

Cosa otteniamo in Ocaml?

In particolare, si dica:

- (a) Perché il Top Level Evaluator segnala: "warning 11: this match is unused."?
- (b) Perché il TLE introduce un binding:
"val equal : 'a -> 'b -> bool = <fun>"
attribuendo al valore di g tale tipo?
- (c) Cosa calcola il TLE , e perchè, allorchè, scriviamo l'espressione: "equal 5 7;"?

Tipi Astratti: API o Segnatura in OCaml

- Modulo Signature: ha un nome,
- Contiene la signature dei pubblici
- La signature è racchiusa tra **sig...end**
- Elenca i tipi esportati (e implementati dall'ADT)
- Elenca la segnatura di ogni operazione esportata
- La corrispondenza tra API e ADT è controllata dal sistema dei tipi.

```
module type RELAZIONE =  
  sig  
    type ('a,'b) relazione  
    val relazioneC: unit -> ('a,'b) relazione  
    val addPair: ('a,'b) relazione -> 'a -> 'b -> ('a,'b) relazione  
    val removePair: ('a,'b) relazione -> 'a -> 'b -> ('a,'b) relazione  
    val getUno: ('a,'b) relazione -> 'b -> 'a list  
    val getDue: ('a,'b) relazione -> 'a -> 'b list  
  end;;
```

Tipi Astratti: ADT e Implementazione

- Modulo ADT: Ne possono esistere più d'uno.
- Ogni ADT ha un nome,
- Contiene le implementazioni
- L'implementazione è racchiusa tra **struct...end:NomeSign.**
- Implementazione tipi esportati
- Implementazione operazioni esportate

```
module Relazione =  
  (struct  
    type ('a,'b) relazione = ('a * 'b) list  
    let relazioneC = fun () -> []  
    let removePair r x y =  
      let g = fun u -> if u=(x,y) then [u] else []  
      in List.fold_right(List.append)(List.map g r) []  
    let isnl r v = List.fold_right (||) (List.map ((=)v) r) false  
    let addPair r x y = if (isnl r (x,y)) then r else (x,y)::r  
    let isDue(r,y) = List.fold_right (||) (List.map(fun u -> snd(u)=y) r) false  
    let getUno r y = let g = fun u -> if(snd(u)=y) then [fst(u)] else []  
      in List.fold_right(List.append)(List.map g r) []  
    let getDue r x = let g = fun u -> if(fst(u)=x) then [snd(u)] else []  
      in List.fold_right(List.append)(List.map g r) []  
  end:RELAZIONE);;
```

Tipi Astratti: Quali Additional?

- **Anatomia di ADT OCaml**

Nessuna differenza con quella di ADT Java

- **Stato Concreto c:** Struttura Implementazione dei Va⁵
 - `type ('a,'b) relazione = ('a*'b) list`
 - In generale, una tupla (o record) di tipi, o algebrici
 - `type tree = {label: labelType; sons: tree list}`
 - `type tree = Tree of labelType * tree list`
- **AF e I:** Funzione di Astrazione e Invariante
 - $AF(c) = \{(x, y) \mid (\exists n \in [0..length(c) - 1]) hd(tl^n c) == (x, y)\}$ ⁶
- **Additional:** Ma per valori senza stato
 - `toString:` Stringa di presentazione del Va
 - `toString: ('a,'b)relazione → String`
 - `elements:` Lista degli elementi contenuti nel Va
 - `elements('a,'b)relazione → ('a * 'b)list`

⁵Va indica un valore astratto

⁶ g^n indica n composizioni di g

Riuso e Principio di Astrazione

- Una prima definizione:

```
let rec addAll acc = function
  | [ ] -> acc
  | x::xs -> addAll (acc+x) xs;;
```
- Una seconda definizione:

```
let rec rev acc = function
  | [ ] -> acc
  | x::xs -> rev (x::acc) xs;;
```
- Hanno identica forma (pattern)

Definition (Principio di Astrazione)

Evitare programmi che richiedono di dichiarare più volte una "stesso pattern di controllo". Le similarità devono essere individuate e astratte creando opportune funzioni che le implementino.

- Un'unica dichiarazione:

```
let rec fold_left g acc = function
  | [ ] -> acc
  | x::xs -> fold_left g (g acc x) xs;;
```

- Tanti usi diversi:

```
let addAll = fold_left (+)
let rev = fold_left (fun a x -> x::a)
let addAll0 = fold_left (+) 0
let revT = fold_left (fun a x -> x::a) [ ]
#addAll [1;2;3;4;5];;
- : int = 15
#revT [1;2;3;4;5];;
- : int list = [5; 4; 3; 2; 1]
```

Funzioni Higher Order

- Una funzione Higher Order ha funzioni come parametro: `g`

```
let rec fold_left g acc = function
  | [ ] -> acc
  | x::xs -> fold_left g (g acc x) xs;;
```

- oppure, calcola una funzione: `addAll, rev, addAll0, revT`

```
let addAll = fold_left (+)
let rev = fold_left (fun a x -> x::a)
let addAll0 = fold_left (+) 0
let revT = fold_left (fun a x -> x::a) [ ]
```

- oppure, entrambe le cose:

```
fold_left g
```

- Ancora pattern simili:

```
let rec squareEach = function
  | [] -> []
  | x::xs -> (x * x)::(squareEach xs);;
let rec show = function
  | [] -> []
  | x::xs -> (x ^ ", "):(show xs);;
```

- Un'unica funzione:

```
let rec map g = function
  | [] -> []
  | x::xs -> (g x)::(map g xs);;
```

- tanti usi diversi:

```
let squareEach = map (fun x -> x*x);;
let show = map (fun x -> x ^ ", ");;
```

- Ancora pattern simili:

```
let rec pos = function
  | [] -> []
  | x::xs -> if x >= 0 then x::(pos xs) else pos xs;;
let rec even = function
  | [] -> []
  | x::xs -> if x mod 2 == 0 then x::(even xs) else even xs;;
```

- Un'unica funzione:

```
let rec filter g = function
  | [] -> []
  | x::xs -> if g x then x::(filter g xs) else filter g xs;;
```

- tanti usi diversi:

```
let pos = filter (fun x -> x >= 0);;
let even = filter (fun x -> x mod 2 == 0);;
```

- Ancora pattern simili:

```
let rec sub fin = function
  | [] -> fin
  | x::xs -> x - (sub xs);;
let rec toString fin = function
  | [] -> fin
  | x::xs -> x ^ (toString xs);;
```

- Un'unica funzione:

```
let fold_right g xx fin = match xx with
  | [] -> fin
  | x::xs -> g x (fold_right g xs fin);;
```

- tanti usi diversi:

```
let sub fin = fun xx -> fold_righth (-) xx fin;;
let toString fin = fun xx -> fold_right (^) xx fin;;
```

Riuso, Efficienza, Definizioni Tail Recursive

- **Astrazione di Pattern.** Invece di tante dichiarazioni g_1, \dots, g_n che istanziano in modo differente uno stesso pattern, nel programma si fa uso di una sola funzione G higher order, avente pattern che astrae ciascuna di tali istanze e si rimpiazza ogni g_i con un'istanza di G , diversa e specifica per g_i .
- **Proprietà.** Proprietà provate sul pattern G possono essere opportunamente riformulate ed utilizzate in ogni sua istanza g_i .
 - **Proviamo.** $\forall (g: 'a \rightarrow 'b \rightarrow 'b, xx: 'a \text{ list}, fin: 'b)$
 $\text{fold_right } g \text{ } xx \text{ } fin == \text{fold_left } (eX \ g) \text{ } fin \text{ } (rev \ xx)$ ⁷
dove: $eX \ h = \text{fun } a \ x \rightarrow h \ x \ a$
- **Efficienza.** Una ri-definizione più efficiente di G conduce a una corrispondente maggiore efficienza di ciascun istanza fornita per g_1, \dots, g_n , fornendo possibilmente pattern di calcolo alternativo e più efficiente per ognuna di tali funzioni g_1, \dots, g_n .
 - **Tail Recursive.** Mostriamo che le funzioni higher order, `fold_righth`, `map`, `filter` hanno (un pattern tail recursive e quindi) una definizione tail recursive (in slide "folder_right è tail recursive" e successive).

⁷ Una dimostrazione è nella slide successiva e `rev` indica la funzione che calcola la reverse di lista (definita anche in slide "Riuso e Principio di Astrazione")

Possiamo esprimere fold_right con fold_left

- **Lemma1** $\forall ((g:'a \rightarrow 'b \rightarrow 'b), (z:'a), (fin:b))$
 $g\ z\ fin == fold_left(eX\ g)\ fin\ [z]$
- **Lemma2** $\forall (g:'a \rightarrow 'b \rightarrow 'b, xx,yy:'a\ list, fin:'b)$
 $fold_left\ g\ fin\ (xx@yy) == fold_left\ g\ (fold_left\ g\ fin\ xx)\ yy$
- **Lemma3** $\forall ((x:'a), (xx:'a\ list))\ [x]@xx == x::xx$
- **Lemma4** $\forall (x:'a), rev[x] == [x]^8$
- **Lemma5** $\forall (xx:'a\ list, yy:'a\ list),$
 $rev(xx@yy) == (rev\ yy)@(rev\ xx)$

- **Proviamo.** $\forall (g:'a \rightarrow 'b \rightarrow 'b, xx:'a\ list, fin:'b)$
 $fold_right\ g\ xx\ fin == fold_left\ (eX\ g)\ fin\ (rev\ xx)$
dove: $eX\ h = fun\ a\ x \rightarrow h\ x\ a$
- **Induzione** sulla dimensione della lista. Banale il caso lista vuota. Assumiamo:
 $\forall (g:'a \rightarrow 'b \rightarrow 'b, zz:'a\ list, fin:'b) \ \&\& List.length\ zz < k,$
 $fold_right\ g\ xx\ fin == fold_left\ (eX\ g)\ fin\ (rev\ xx)$
 - Sia $xx == z::zz.$

⁸

rev indica la funzione che calcola la reverse di lista (definita anche in slide [Riuso e Principio di Astrazione*](#))

La dimostrazione

- **L1.** $g\ z\ \text{fin} == \text{fold_left}(eX\ g)\ \text{fin}\ [z]$
- **L2.** $\text{fold_left}\ g\ \text{fin}\ (\text{xx}@\text{yy}) == \text{fold_left}\ g(\text{fold_left}\ g\ \text{fin}\ \text{xx})\text{yy}$
- **L3.** $[\text{x}]@\text{xx} == \text{x}::\text{xx}$
- **L4.** $\text{rev}[\text{x}] == [\text{x}]$
- **L5.** $\text{rev}(\text{xx}@\text{yy}) == (\text{rev}\ \text{yy})@(\text{rev}\ \text{xx})$

- **Proviamo.** $\forall (g:'a \rightarrow 'b \rightarrow 'b, \text{xx}:'a\ \text{list}, \text{fin}:'b)$
 $\text{fold_right}\ g\ \text{xx}\ \text{fin} == \text{fold_left}\ (eX\ g)\ \text{fin}\ (\text{rev}\ \text{xx})$

dove: $eX\ h = \text{fun}\ a\ x \rightarrow h\ x\ a$

- **Induzione** sulla dimensione della lista. Banale il caso lista vuota. Assumiamo:

$\forall (g:'a \rightarrow 'b \rightarrow 'b, \text{zz}:'a\ \text{list}, \text{fin}:'b) \ \&\& \ \text{List.length}\ \text{zz} < k,$
 $\text{fold_right}\ g\ \text{xx}\ \text{fin} == \text{fold_left}(eX\ g)\ \text{fin}\ (\text{rev}\ \text{xx})$

- Sia $\text{xx} == \text{z}::\text{zz}.$
- $\text{fold_right}\ g\ (\text{z}::\text{zz})\ \text{fin}$
- $== g\ z\ (\text{fold_right}\ g\ \text{zz}\ \text{fin})$ — fold_right
- $== g\ z\ (\text{fold_left}(eX\ g)\ \text{fin}(\text{rev}\ \text{zz}))$ — ind. ass.
- $== \text{fold_left}(eX\ g)(\text{fold_left}(eX\ g)\ \text{fin}(\text{rev}\ \text{zz}))\ [z]$ — Lemma1
- $== \text{fold_left}(eX\ g)\ \text{fin}((\text{rev}\ \text{zz})@[z])$ — Lemma2
- $== \text{fold_left}(eX\ g)\ \text{fin}((\text{rev}\ \text{zz})@(\text{rev}\ [z]))$ — Lemma4
- $== \text{fold_left}(eX\ g)\ \text{fin}(\text{rev}([z]@\text{zz}))$ — Lemma5
- $== \text{fold_left}(eX\ g)\ \text{fin}(\text{rev}(\text{z}::\text{zz}))$ — Lemma3

- **fold_right**

```
fold_right = fun g xx fin -> fold_left(eX g) fin (rev xx)
```

Usate: `fold_left`, `rev` hanno definizioni Tail Recursive.

- **map**

```
map = fun g xx -> fold_right (fun x a -> (g x)::a) xx [ ]
```

Usate: `fold_right` ha definizione Tail Recursive.

- **filter**

```
filter = fun p xx ->
  let g = fun x a -> if (p x) then x::a else a in
  fold_right g xx [ ]
```

Usate: `fold_right` ha definizione Tail Recursive.

- Perché nella programmazione Funzionale `List.fold_left` è considerato un iteratore? Si spieghi cosa è valutato ad ogni iterazione e come variano gli eventuali indici dell'iterazione.
- (a) Cosa calcola `map (+5) (map (*2) xx)`?
- (b) Cosa si può dire in generale, su un pattern della forma: `map f (map g xx)`?
- (a) Si dia una definizione di `length` che calcola la lunghezza di una lista
- (b) Sapreste dare una definizione Tail Recursive di `length`
- Sia `indList` una funzione che applicata ad una arbitraria lista, calcola la lista di coppie avente come primo componente la posizione della coppia nella lista e come secondo componente il valore in quella posizione. Ad esempio `indList [56;12;-7]` deve calcolare `[(1,56),(2,12),(3,-7)]`.
- (a) Si dia una definizione di `indList`;
- (b) Si dia una definizione iterativa e Tail Recursive di `indList` che utilizzi le sole operazioni: `fst`, `snd`, `List.fold_left`, `List.length`.
- La funzione `zip` è un trasformatore, applicato ad una coppia di liste di stessa lunghezza calcola la lista contenente in ogni posizione, la coppia avente primo e secondo componente uguale al valore contenuto nella stessa posizione nella prima e nella seconda lista.
- (a) Si dia una definizione di `zip`.
- (b) Si dia una definizione tail-recursive
- Anche la funzione `flatten` è un trasformatore: Data una lista di liste di valori 't, `flatten` genera la lista di valori 't formata dalla concatenazione delle liste di 't.
- Si dia una definizione tail recursive di `flatten`.
- La funzione `duplex` rimuove ogni valore duplicato presente in una lista.
- (a) Si dia una definizione ricorsiva di `duplex`
- (b) Si dia una definizione tail recursive di `duplex`
- Il numero di occorrenze di un valore 't in una lista 't può essere calcolato con un pattern simile a quello utilizzato nella definizione di `isIn` di `Queue.ml`, ovvero:
`occ x xx = List.length (filter ((=)x) xx)`.
- Possiamo però, fornire altre definizioni tail recursive che evitano l'uso di `filter` e `length`. Si mostri come.
- Le occorrenze di un valore 't in una lista 't `list` è una int list contenente le posizioni (numerata a partire da 1 per la prima) delle eventuali occorrenze del valore nella lista `data`.
Possiamo fornire una funzione `occList`, definita tail recursive, che calcola la lista delle occorrenze di un valore in una lista?