Programmazione Funzionale

Esonero 1 – Giovedi 30 novembre 2023

Esercizio 1 (Liste Palindrome)**.** Un palindrome e una parola che si legge uguale sia dalla sinistra alla destra che dalla destra alla sinistra. Questa nozione puo essere definita anche per le liste. Ad esempio le liste $[1;1]$ e $[1;2;3;3;2;1]$ sono palindrome, invece $[1;2]$ e $[2;3;3]$ non sono palindrome.

1. Definire la funzione `invert : a' list-> a' list` che prende una lista $[a_1;\ldots;a_n]$ e ritorna la lista $[a_n;\ldots;a_1]$.
2. Definire la funzione `pali : a' list -> bool` che ritorna `true` se la lista e un palindrome e `false` altrimenti.

La funzione precedente risolve il problema, pero quando viene applicata a una lista di lunghezza n necessita l'uso dell'ugualianza per le liste di lunghezza m.

Un altro metodo puo essere usato per le liste di lunghezza $m > 2$ che usa l'ugualianza per liste di lunghezza $m/2$, cerchiamo di implementare una tale funzione:

1. Definire la funzione `length` che prende una lista e restituisce la sua lunghezza.
2. Definire una funzione `invertWithControl` di tipo `a' list * a' list * int` che prende una tripla (l_1, l_2, n) e restituisce la coppia (l_1', l_2') dove l_1' corrisponde a l_1 da cui sono stati tolti i primi n elementi, e l_2' corrisponde concatenazione $l_p \cdot l_2$ di l_p la lista dei n primi elementi di l invertita, con l_2.
3. Definire la funzione `divide` che prende una lista $[a_1;\ldots;a_{2n}]$ di lunghezza pari e ritorna la coppia di liste $[a_1;\ldots;a_n]$ e $[a_{n+1};\ldots;a_{2n}]$. Se la lunghezza e dispari la funzione sollevera un eccezione `Dispari`.
4. Mediante le funzione `invert`, `length`, `divide` definire una funzione `pali1` che verifica se una lista e un palindrome. Cioè, *per ogni* liste `l`, `pali(l)` e `pali1(l)` sono uguali.

Esempi di comportamento delle funzione;
- `invert [2;15;6]` restituisce `[6;15;2]`.
- `length [3;3;2]` restituisce 3. `length [88]` restituisce 1.
- `invertWithControl ([3;45;7;1] , [11;5] , 2)` restituisce `([7;1], [45;3;11;5])`.
- `divide [1;2;5]` soleva l'eccezione `Dispari`. Invece `divide [2;3;15;7]` restituisce `([2;3],[15;7])`.

Esercizio 2 (Clausole)**.** Una *clausola* e un proposizione della forma seguente

$$C = p_1 \wedge \cdots \wedge p_n \Rightarrow q_1 \vee \cdots \vee q_k$$

Dove $(p_i)_{1\leq i\leq n}$ e $(q_i)_{1\leq i\leq k}$ sono liste di booleani. Una clausola C e vera se e solo se tutti p_i sono veri e almeno uno dei q_j e vero oppure quando un p_i e falso.

1. Definire la funzione `andList : bool list -> bool` che prende una lista di booleani `[b1;. . . ;bn]` e ritorna `true` se tutti i booleani `bi` sono veri. Se la lista e vuota restituisce `true`.
2. Definire la funzione `orList : bool list -> bool` che prende una lista di booleani `[b1;. . . ;bn]` e ritorna `true` se un booleano `bi` e vero. Se la lista e vuota restituisce `false`.
3. Definire la funzione `clausola : bool list * bool list -> bool` che dato una coppia di liste

   `([p1;. . . ;pk],[q1;. . . ;qk])` restituisce `true` se la clausola $p_1 \wedge \cdots \wedge p_n \Rightarrow q_1 \vee \cdots \vee q_k$ e vera e `false` altrimenti.

Esempi di comportamento delle funzione;
- `andList [true ; false; true]` restituisce `false`, mentre `andList [true ; true; true]` restituisce `true`.
- `orList [true ; false; true]` restituisce `true`, invece `orList[false;false]` restituisce `false`.
- `clausola([true;true;false],[false;false])` restituisce `true`.
- `clausola([true;true;true],[false;false])` restituisce `false`.
- `clausola([true;true;true],[true;false])` restituisce `true`.

Esercizio 3 (Massimo Comune Divisore)**.** Un intero n *divide* un intero m se esiste un intero k tale che $m = n \times k$. Il *massimo comune divisore* di due interi n e m e il piu grande intero che divide sia n che m.

Ad esempio, 3 divide 12 perche $12 = 3 \times 4$, pero 5 non divide 12. Il massimo comune divisore di 12 e 18 e 6.

1. Definire una funzione `divide : int * int-> int` che prende una coppia di interi `(n,m)` e restituisce `true` se n divide m e `false` altrimenti.
2. La funzione `divisors : int-> intlist` che prende un intero n e restituisce la lista dei divisori di n.

   Ad esempio, possiamo usare una funzione ausiliaria e ricorsiva di coda `divTailAux` di tipo `int*int* (int list)`.
3. Definire la funzione `mcd : int * int-> int` che prende due interi n e m e restituisce il loro massimo comune divisore.

   Per esempio possiamo definire due funzione ausiliaria `max: int list -> int` che restituisce il piu grande elemento di una lista di interi e `divisors2:int *int -> int list` che restituisce la lista dei interi che dividono i due interi di una coppia (n, m).

Esempi di comportamento delle funzione;
- `divide(3, 6) = true`, `divide(2, 17) = false` e `divide(6, 9) = false`.
- `divisors(18) = [1; 2; 3; 6; 9; 18]`, `mcd(7,5) = 1` e `mcd(12,30) = 6`.

Esercizio 4 (Iterazione e Interi di Church)**.** Dato un intero n l'iterazione di una funzione f si scrive f^n e applicata a un intero k restituisce $f(f(\dots(f(k))\dots)$ dove f occore n volte. Piu formalmente:

$$f^0(k) = k \qquad \text{e} \qquad f^n(k) = f(f^{n-1}(k)).$$

1. Definire la funzione `cur` che prende una funzione `f : A*B -> C` e ritorna la sua versione curryficata cioé di tipo `A -> (B -> C)`.
2. Definire una funzione `iter` che prende una funzione f e un intero n e restituisce la **funzione** f^n. Si puo per esempio, usare una funzione ausiliaria che prende tre argomenti (f, n, k) e ritorna $f^n(k)$ e poi usare una forma curryficata.

   Ad esempio, dato `f = function x -> x+2` allora `iter(f,3) = function x -> (((x+2)+2)+2)`.

Gli interi di church sono una rappresentazione degli interi basata sul costruttore `function` e l'applicazione iterata delle funzioni;

$$n \mapsto \lambda x.\lambda f.f^n(x) \quad \text{cioè} \quad n \mapsto \texttt{function x -> function f -> f}^{\texttt{n}}\texttt{(x)}.$$

3. Definire la funzione `church` che prende un intero n e restituisce la sua rappresentazione di Church.
4. Dato un intero n indicare il tipo della sua rappresentazione di Church `function x -> function f -> f`$^{\texttt{n}}$`(x)`.
5. Semplicemente usando il punto precedente, indicare il tipo della funzione `church`.

Esercizio 5 (Espressione ricorsive e stringhe)**.** Consideriamo di aver dichiarato il tipo ricorsivo seguente:

```
type espr = Name of String | Space of espr | Concat of espr*espr
```

Un elemento di tipo `espr` e della forma `Name s`, `Space e` o `Concat (e1,e2)` dove s e una stringha, e, e_1, e_2 sono espressione. La sostituzione in un espressione `E` di un nome `Name s` con un espressione `E'` si scrive `E [ Name s <- E']`, e corrisponde a l'espression `E` dove tutte le occorrenze di `Name s` vengono sostituite con l'espressione `E'`.

1. Definire la funzione `substitution : espr * string * espr -> espr` che implementa la sostituzione, cioè il valore di `substitution(e,s,e')` corrisponde a `E [ Name s <- E']`.

   Ad esempio,
   - `substitution(Name "a", "a" , Space(Name "c"))` restituisce `Space(Name c)`.
   - `substitution(Name "b", "a" , Space(Name "c"))` restituisce `Name "b"`.
   - `substitution(Concat( Space(Name "a") , Name "cd"), "a" , Space(Name "c"))` restituisce `Concat( Space(Space((Name "c"))) , Name "cd")`.
2. Definire la funzione `eval: espr -> string` che prende un espressione e e ritorna una stringa interpretando `Concat` come la concatenazione e `Space` come la concatenazione con `" "`.

   Ad esempio,
   - `eval Name "frase"` restituisce `"frase"`.
   - `eval Space(Name "frase")` restituisce `"frase "`.
   - `eval Concat (Space(Name "una"), Name "frase")` restituisce `"una frase"`.

Esercizio 6 (Ordinamento di liste)**.** Una lista di interi `[a1; . . . ; an]` e *crescente* se $[a_1 \le \cdots \le a_n]$. Ordinare una lista corrisponde a trasformare una lista qualsiasi in una lista crescente. Ad esempio `[2;1;7;8;4]` diventa `[1;2;4;7;8]`.

L'obiettivo del esercizio e di implementare un modo per ordinare le liste di interi.

1. Definire un funzione `transition` che prende due liste (l_1, l_2) e se l'elemento in testa di l_2 e piu piccolo dell'elemento in testa h_1 di l_1 restituisce $h :: l_2$, altrimenti restituisce l_2. Se $l_1 = []$ allora restituisce l_2. Se `l1` non e vuota e $l_2 = []$ restituisce `[x]`.
2. Definire una funzione `suborder` che prende una lista `l` una lista che corrisponde alla lista `l` ottenuta togliendo i elementi non ordinati in `l`. Si puo usare la funzione `invert` e la funzione `transition`.
3. Definire una funzione `orderSplit` che prende una lista l e restituisce la coppia di liste `(l1,l2)` dove `l1` corrisponde a `suborder(l)` e `l2` corrisponde ai elementi di `l` che non occorono in `l1`.
4. definire una funzione `orderAdd` che prende una lista di interi `[a1;. . . ;an]` e un intero `n` restituisce la lista `[a1;. . . ;ai;n;`a_{i+1}`;. . . ;an]` tale che per tutti $1 \le k \le i$ l'intero `n` maggiora `ak`.
5. Definire la funzione `order` che prende una lista e restituisce la sua versione ordinata.

   Si puo per esempio, usare le funzione `orderSplit` e `orderAdd` con un altra funzione che gestisce l'iterazione.

Esempi di comportamento delle funzione;
- `transition([3;4;1] , [6;2])` restituisce `[6;2]`, `transition( [6;2] , [3;4;1] )` restituisce `[6;3;4;1]`.
- `transition([],[2;2])` restituisce `[2;2]`.
- `suborder [2;1;7;8;4]` restituisce `[2;7;8]`.
- `orderSplit [2;1;7;8;4]` restituisce `([2;7;8],[1;4])`.
- `orderAdd ([2;3;6;4] ,4)` restituisce `[2;3;4;6;4]`. Oppure `orderAdd ([1;3;5;6] ,4)` restituisce `[1;3;4;5;6]`.
- `order [2;1;7;8;4]` restituisce `[1;2;4;7;8]`.