Programmazione Funzionale

Esercitazione 5 – Tipi Ricorsivi

Esercizio 1. Consideriamo di aver dichiarato i seguenti tipi ricorsivi:

```
type espr = Zero | Succ of espr | Plus of espr * espr;;
```

1. Definire la funzione `eval : espr -> int` che calcola il valore di un elemento di `espr`, interpretando `Plus` come la somma degli interi e `succ` come la funzione successore.
2. Definire la funzione opposta `repr : int -> espr` che associa a un intero la sua representazione nel tipo `intero` come l'iterazione del costruttore `Succ`.
3. Definire la funzione `mult : espr * espr -> espr` che calcola la moltiplicazione di due espressioni mediante repr.
4. Definire la funzione `mult : espr * espr -> espr` che calcola la moltiplicazione di due espressioni senza usare la funzione repr.

Esercizio 2. Consideriamo di aver dichiarato il tipo seguente

```
type espr = Int of int | Plus of espr * espr | Var of string;;
```

Un ambiente di esecuzione *amb* sugli interi e una lista di tipo `(string * int) list` dove il fatto che una coppia `("nome", n)` occorre nell'ambiente *amb* significa che la variabile `"nome"` ha il valore *n*.

1. Definire una funzione `eval` che prende un espressione e un ambiente di esecuzione sugli interi e restituisce il valore dell'espressione. Se una variabile occorre nell'espressione ma non e assegnato a nessun intero dall'ambiente allora `eval` restituira un eccezione.
2. Definire una funzione `dependency : espr -> string list` che applicata a un espressione *e* restituisce la lista dei nomi delle variabile contenute nel espressione *e*.
3. Una stringa *s* occorre in un espressione *e* se `Var s` e una sottoespressione di *e*, diciamo che occorre *linearmente* se `Var s` occorre un unica volta in *e*.
   Un espressione e *lineare* se tutte le stringe che contiene occorrono in modo lineare.
   Definire una funzione `linear : espr -> espr` che prende un espressione *e* e la trasforma in un espressione lineare, cambiando solamente il nome delle stringe che occorrono non linearmente con nuovi nomi.

Esercizio 3. Consideriamo di aver dichiarato il seguente tipo

```
type espr = Int of int | Plus of espr * espr ;;
```

Ricordiamo che e_1 e una sottoespressione di e_2 quando:

- Le due espressioni sono uguali $e_1 = e_2$.
- l'espressione e_2 e delle forma $Plus(e_3, e_4)$ e e_1 e una sottoespressione di e_3 o di e_4.

1. Definire una funzione `isequal: espr * espr -> bool` che restituisce `true` se le due espressioni sono uguali e `false` altrimenti.
2. Definire la funzione `subexpr : espr * espr -> bool` che applicata a (e_1, e_2) restituisce `true` quando e_1 e una sottoespressione di e_2 e `false` altrimenti.

Esercizio 4. Consideriamo di aver dichiarato il tipo seguente

```
type espr = Int of int | Plus of espr * espr | Mult of espr*espr;;
type direction =  Left | Right;;
```

Una *posizione* su un espressione e una sequenza $(a_1, \ldots, a_n)$ dove tutti gli a_i corrispondo a 'left' o 'right'. In un espressione *e*, la sottoespressione in posizione $l = (a_1, \ldots, a_n)$ è la sottoespressione $pos(e, l)$ di *e* ottenuta per induzione;

- $pos(e, []) = e$.
- $pos(OP(e_1, e_2), (left, a_1, \ldots, a_n)) = pos(e_1, (a_1, \ldots, a_n))$ dove OP corrisponde a `Mult` o `Plus`.
- $pos(OP(e_1, e_2), (right, a_1, \ldots, a_n)) = pos(e_2, (a_1, \ldots, a_n))$ dove OP corrisponde a `Mult` o `Plus`.
- In qualsiasi altro caso la funzione non e definita.

1. Definire la funzione di concatenazione di due liste.
2. Definire la funzione `concatHO : 'a * ('a list) list -> ('a list) list` che applicata a un elemento *a* e una lista di liste $l = [l_1; \ldots; l_n]$ concatena *a* con tutte le liste l_i.

3. Una posizione l è valida su un espressione e quando $pos(e, l)$ e definito. Definire la funzione `positions : espr -> direction list list` che prende un espressione e restituisce la lista delle sue posizione valide.
4. Definire la funzione $pos(\cdot, \cdot)$ in OCaml.
5. Definire una funzione `subexprpos : espr * espr -> direction list` che applicata a (e, e') restituisce la lista delle posizione delle sottoespressione di e uguale a e'.

Esercizio 5. Consideriamo di aver dichiarato il seguente tipo

```
type espr = Const of int | Ident of ident
    | Lambda of ident * espr | App of espr * espr ;;
and ident = string;;
```

Un ambiente di sostituzione e una lista di tipo `(ident*espr) list`.

1. Definire una funzione `sost : espr * (ident*espr) list -> espr` che quando applicata a `E,l` se `("nome", E')` occorre nella lista sostituice ogni occorrenza de `"nome"` con l'espressione `E'`.
2. Definire una funzione `break : (espr * (espr -> bool) * string) -> espr * espr` che applicata a (e, f, s) restituisce:
   - (e_0, e') dove e' e una sottoespressione e tale che $f(e')$ vale `true` e e_0 corrisponde a l'espressione e dove la sottoespressione e' e stata sostituita con un `Ident s`.
   - `(e,Ident null)` se e non contiene una sottoespressione e' tale che $f(e')$ vale `true`.
3. Definire una funzione `eval : espr -> espr` che quando applicata a un espressione e la riduce seguendo la regola sequente:
   - `App(Lambda (s, E) , E')` si riduce in `sost (E , [(s,E')])`.
   - Se l'espressione `E` si riduce in `E'` allora `Lambda (s,E)` si riduce in `Lambda (s,E')`, `App (E0,E)` si riduce in `App (E0,E')` e `App (E,E0)` si riduce in `App (E,E0)`.