PROGRAMMAZIONE II - a.a. 2017-18
Esercitazione — 13 dicembre 2018

Esercizio 1 Si consideri il seguente programma OCaML

```
let n = 5;;
let h = fun x -> n + x;;
let rec f p n =
   let g = fun y -> n * y in
      if n = 0 then p 1
      else if n > 1 then f g (n-1)
           else f p (n-1);;
f h 2;;
```

1. Si indichi il tipo inferito dall'interprete OCaML per la funzione ricorsiva `f`.

```
val n : int = 5
val h: int -> int = <fun>
val f: (int -> int) -> int -> int = <fun>
```

2. Si simuli la valutazione del programma mostrando la struttura della pila dei record di attivazione.

   Si segua lo svolgimento alla lavagna!

3. Si indichi il valore restituito dal programma.

Esercizio 2

Si estenda il linguaggio didattico funzionale con il costrutto `CodaLimitata` per la definizione di code con lunghezza massima prefissata. In aggiunta, il linguaggio è esteso con le operazioni primitive `insert` e `remove`, che rispettano la politica FIFO, e `peek`, che restituisce l'elemento in cima alla coda.

1. Si mostri come deve essere modificato l'interprete del linguaggio didattico funzionale.

   Una soluzione minimale è riportata di seguito. Una soluzione di stile più funzionale al posto della `peek` *avrebbe introdotto dei costruttori per gestire un risultato più complesso dell'operazione di rimozione, che intuitivamente dovrebbe restituire una coppia.*

```
type exp = ...
   | CodaLimitata of exp * queue
   | Insert of exp * exp
   | Remove of exp
   | Peek of exp
and queue = Empty | Item of exp * queue

type evT = ...
   | CodaLimVal of evT * (evT list)

let rec eval (e : exp) (r : evT env) : evT = match e with
   ...
   | CodaLimitata(i, q) ->
        let iv = (eval i r) in
           let ql = (evalQ q r) in
              (match iv with
                 Int ivv -> if ((List.length ql) <= ivv)
                               then CodaLimVal(iv, ql)
                               else failwith("wrong limit") |
                 _ -> failwith("not a limit"))
   ...
   | Insert(e1, e2) ->
         let q = (eval e1 r) in
            (match q with
                CodaLimVal(lim, ql) -> if ((List.length ql) < lim)
                                          then CodaLimVal(lim, ql@[(eval e2 r)])
                                          else failwith("out of bound") |
                _ -> failwith("not a queue"))
   | Remove(e1) ->
         let q = (eval e1 r) in
            (match q with
                CodaLimVal(lim, v::ql) -> CodaLimVal(lim, ql) |
                CodaLimVal(_, []) -> failwith("empty queue") |
                _ -> failwith("not a queue"))
   | Peek(e1) ->
         let q = (eval e1 r) in
            (match q with
                CodaLimVal(_, v::ql) -> v |
                CodaLimVal(_, []) -> failwith("empty queue") |
                _ -> failwith("not a queue"))
   ...
and let rec evalQ (q : queue) (r : evT env) : evT list = match q with
     Empty -> []
   | Item (e, q1) -> (eval e r)::(evalQ q1 r)
```

Esercizio 3

Si estenda il linguaggio didattico funzionale introducendo il tipo di dato `IntSet` che permette di dichiarare insiemi di interi di cardinalità finita. In aggiunta, il linguaggio è esteso con le operazioni primitive `insert myset elem`, `remove myset elem` che permettono di operare su insiemi finiti di interi.

1. Si mostri come deve essere modificato l'interprete del linguaggio didattico funzionale. `insert` aggiunge sempre un intero alla lista, mentre `remove` ne rimuove tutte le occorrenze.

```
type exp = ...
   | IntSet of seq | insert of exp * exp | remove of exp * exp
   ...
and seq = Empty | Item of exp * seq

type evT = ...
   | IntSetVal of int list

let rec eval (e : exp) (r : evT env) : evT = match e with
   ...
   | IntSet e1 -> IntSetVal (evalSeq e1 r)
   | insert e1 e2 = (match (eval e2 r, eval e1 r) with
        IntSetVal s, Int i -> IntSetVal (i :: s)
      | _, _ -> failwith("IntSet error") )
   | remove e1 e2 = (match (eval e2 r, eval e1 r) with
        IntSetVal s, Int i -> IntSetVal (removeSeq s i)
      | _, _ -> failwith("IntSet error") )
         ...
and let rec evalSeq (s : seq) (r : evT env) : int list = match s with
     Empty -> []
   | Item (e1, s1) -> (eval e1 r)::(evalSeq s1 r)
   | _ -> failwith("IntSet error")
and let rec removeSeq (s : int list) (i: int) : int list = match s with
     [] -> []
   | i1 :: s1 -> if (i = i1) then (removeSeq s1 i) else i1 :: (removeSeq s1 i)
```

Esercizio 4

Si consideri il seguente programma OCaML

```
let n = 10;;
let m x y =
  let z = x + y + n in
    fun w -> w + z;;
let rec shift f l n =
  match l with
      [] -> []
    | h :: t -> f(h + n) :: shift f t n;;
let n = 40;;
let g = m 10 n;;
shift g [5; 10] n;;
```

1. Si determini il tipo inferito dall'interprete OCaML per gli identificatori di funzione (`m`, `shift` e `g`) che compaiono nel programma scelto.

```
val m : int -> int -> int -> int = <fun>
val shift : (int -> 'a) -> 'int list -> int -> 'a list = <fun>
val g : int -> int = <fun>
```

2. Si simuli la valutazione del programma mostrando la struttura della pila dei record di attivazione.

Esercizio 5

Si consideri il seguente programma scritto con una sintassa Java-like.

```
class A {
  private int x;
  static public int y;

  public void foo() {
    :
  }

  public void bar() {
    :
  }

    :
}
```

1. Supponiamo di considerare due oggetti `a1, a2` istanza della classe `A`, nell'ipotesi che `A.y = 1`, `a1..x = 9`, e `a2.x = 40`, si simuli la struttura della memoria (stack e heap).

Esercizio 6

Si consideri il seguente programma scritto con una sintassa Java-like.

```
class A {
  protected int x = 0;
}

class B extends A {
  public int x = 1;
  public void f() {
    System.out.println(x);
  }
}
```

1. Supponiamo di considerare il frammento di codice `B b = new B(); b.f();` si simuli la struttura della memoria (stack e heap).