PROGRAMMAZIONE II (A,B) - a.a. 2017-18
Primo Appello – 15 gennaio 2018

Esercizio 1

Si consideri il tipo `Agenda` che rappresenta un insieme di appuntamenti (detti anche eventi). Ogni evento è un oggetto della classe `Evento`, che si assume già fornita allo sviluppatore e che contiene informazioni sulle date di inizio e fine utilizzando oggetti di tipo `Date`, presente nella libreria Java (e che implementa `Comparable`). La classe `Agenda` ha un metodo di inserimento, che aggiunge un evento se questo non esiste. Se l'evento è già presente in agenda, viene sollevata l'eccezione `EventDuplicateException`. Si ha un conflitto tra eventi quando si inserisce un evento sovrapposto temporalmente con un altro evento in agenda. In tal caso, l'evento comunque inserito in agenda, ma viene segnalata l'eccezione `EventConflictException`. Supponiamo che la definizione del tipo di dato `Agenda` contenga tra gli altri i metodi

```
interface Agenda {
   /* aggiunge l'evento all'agenda; solleva un'eccezione se nell'agenda
      si trova un evento che si sovrappone temporalmente con e */
   void addEvent(Evento e) throws EventConflictException, EventDuplicateException

   /* restituisce gli eventi attivi in date, se esistono, null altrimenti */
   List<Event> readEvents(Date date);

   /* elimina dall'agenda l'evento e */
   void deleteOldEvent(Evento e);

   /* restituisce gli eventi che inizieranno dopo date, in ordine temporale */
   List<Event> readFutureEvents(Date date);

   /* restituisce true se e solo se l'agenda e' vuota */
   boolean emptyAgenda();
}
```

1. Si descrivano le ipotesi di comportamento che vengono richieste alla classe `Evento` per il buon funzionamento del tipo di dato `Agenda`.

   La classe deve contenere ad esempio metodi per ottenere la data di inizio e di fine di un evento e per indicare se due eventi coincidono (condizione che può non corrispondere alla sola completa sovrapposizione temporale, ma dipendere dal luogo dell'appuntamento etc.).

2. Assumendo di adottare una strategia di programmazione difensiva, si completi la specifica dei metodi, definendo le clausole REQUIRES, MODIFIES e EFFECTS, indicando le eccezioni eventualmente lanciate e se sono checked o unchecked.

   Si veda il file `Agenda.java`.

3. Si consideri la seguente struttura di implementazione per la classe `Agenda`

   ```
   private Evento[] elems;
   private int numEvents;
   ```

   Intuitivamente, `elems` contiene gli eventi presenti in agenda, in un ordine qualunque, mentre `numEvents` rappresenta il numero degli eventi memorizzati in agenda.

   Si definisca l'invariante di rappresentazione per l'implementazione `MiaAgenda` di `Agenda`.

4. Si fornisca l'implementazione del metodo `addEvent` e si dimostri che preserva l'invariante di rappresentazione.

   Si veda il file `MiaAgenda.java`.

5. Si consideri una classe `MiaAgendaSenzaConflitti` che estende `MiaAgenda` modificando il metodo `addEvent(Evento e)` in modo che, in caso di conflitto con altri eventi dell'agenda, il metodo non inserisca l'evento `e` pur lanciando l'eccezione `EventConflictException`.

   Giustificando la risposta, si dica se l'estensione proposta verifica il principio di sostituzione.

   Si veda il file `MiaAgendaSC.java`. Non vale il principio di sostituzione perché la regola dei metodi non è soddisfatta.

Esercizio 2

Si estenda il linguaggio didattico funzionale con il costrutto `DynFun of ide * exp` per la dichiarazione di funzioni non ricorsive che utilizzano regole di scoping *dinamico* nella risoluzione dei riferimenti non locali.

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

  Come già visto a lezione, la soluzione è data da

```
type exp = ...
   | DynFun of ide * exp

type evT = ...
   | DynFunVal of ide * exp

let rec eval (e : exp) (r : evT env) : evT = match e with
   ...
   | DynFun(i, a) -> DynFunVal(i, a)
   ...
   | FunCall(f, eArg) ->
         let fClosure = (eval f r) in
            (match fClosure with
                ...
                DynFunVal(arg, fBody) ->
                      let v = (eval eArg r) in eval fBody (bind r arg v) |
                _ -> failwith("non functional value")) |
   ...
```

- Si indichino le eventuali altre modifiche necessarie per poter gestire funzioni ricorsive.

  Non è necessaria nessuna esplicita estensione: con scoping dinamico la sintassi precedente permette di definire senza problemi funzioni ricorsive.

Esercizio 3

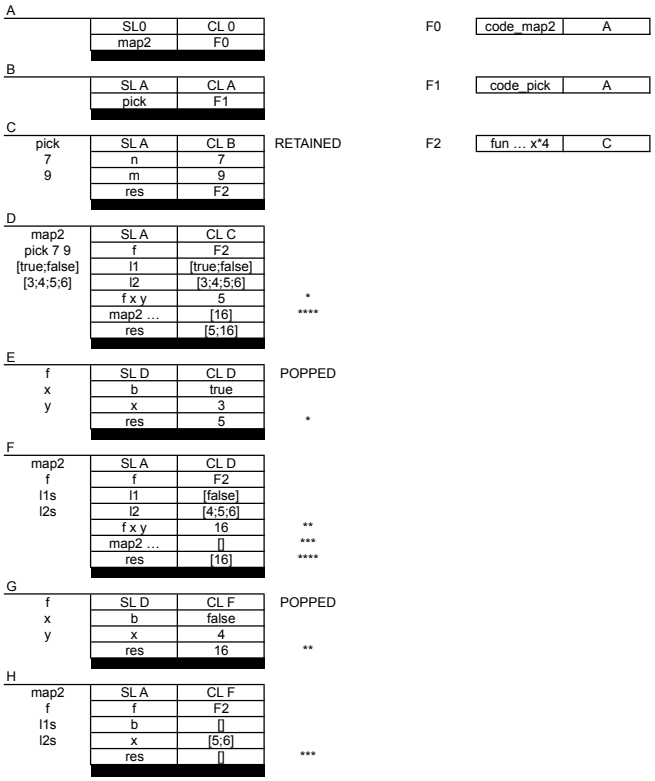
Si consideri il seguente programma OCaML

```
let rec map2 f l1 l2 =
   match (l1,l2) with
        ([], _) -> []
      | (_, []) -> []
      | (x::l1s,y::l2s) -> (f x y) :: map2 f l1s l2s;;

let pick n m =
   if n > m then (fun b x -> if b then x+1 else x*2)
            else (fun b x -> if b then x+2 else x*4);;

map2 (pick 7 9) [true;false] [3;4;5;6];;
```

1. Si determini il tipo inferito dall'interprete OCaML per gli identificatori di funzione (`map2` e `pick`) che compaiono nel programma scelto.

```
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
val pick : 'a -> 'a -> bool -> int -> int = <fun>
```

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



3. Si determini il valore calcolato dal programma.

```
- : int list = [5; 16]
```

Esercizio 4

Si consideri il seguente frammento di codice Java

```
class A { static B b1; B b2; }
class B { }
public class Tester {
   public static void main(String[] args) {
      B ob1 = new B(); B ob2 = new B();
      A oa1 = new A(); A oa2 = new A();

      oa1.b1 = ob1; oa1.b2 = ob1; oa2.b2 = ob2;

      oa1 = null; ob1 = null; ob2 = null;

      // (gc)
      // altro codice
   }
}
```

Si indichi, motivando la risposta, quali sono gli oggetti eleggibili per essere restituiti allo heap dal garbage collection nel punto *gc*.
Nel punto gc *solo l'oggetto inizialmente associato a* `oa1` *è eleggibile per essere restituito.*