Linguaggi di Programmazione con Laboratorio
facsimile di Prova di Esame AA. 2015

(*Parti da omettere in caso di 1 ora di tempo)

Esercizio1.
==========
Un multinsieme è una collezione di valori, possibilmente ripetuti, di uno stesso tipo. Si fornisca in OCaml:
(a) un API, MULTIS, per tipi astratti multinsieme di tipo generico, NON modificabili, equipaggiati, in aggiunta agli opportuni costruttori, delle seguenti operazioni:
+ mul che fornisce la molteplicità, ovvero il numero di ripetizioni, con cui un valore occorre nel multinsieme.
+ add che aggiunge un valore (già occorrente o no) al multinsieme
+ rem che rimuove un'occorrenza di un valore, sollevando eccezione se tale valore non occorre.
(b) un ADT, MultiSADT, per tale API (*costruttori, mul*, rem*)
(c) Un programma che utilizza tale ADT per creare il multinsieme M = {2,4,2} (* e rimuovere 4 e calcolare la molteplicità di 4.)

Esercizio2.
==========
Si consideri il multinsieme di esercizio1. Si fornisca in Java:
(a) un API, MultiS, come in (a) di esercizio1, ma per multinsiemi MODIFICABILI.
(b) Si assuma data la classe ImmMultiS.java che implementa un ADT per MultiS. Si estenda tale ADT in un un nuovo ADT che preveda anche la seguente operazione:
+ size che restituisce il numero di valori diversi correntemente presenti.
(c*) Un programma che utilizza tale ADT per creare il multinsieme M = {2,4,2} e rimuovere 4 e calcolare la molteplicità di 4.

Esercizio3 (Laboratorio)
==========
Si mostri la semantica (frammento della funzione "csem", in appendice) per il caso CCondAssign:

```
let csem c ev st =
  match c with
    ...
  | CCondAssign (i, cond, e1, e2) -> ...
```

la cui semantica informale è di valutare l'espressione "cond" usando la funzione "esem", verificare che questa sia un valore booleano, e nel caso assegnare alla variabile "i" il valore dell'espressione "e1" se "cond" valuta a "True", oppure quello dell'espressione "e2" in caso contrario.

Esercizio4 * (Laboratorio)
==========
Si mostri la semantica (frammento della funzione "csem", in appendice) per il caso CFor:

```
let csem c ev st =
  match c with
    ...
  | Cfor (i, e1, e2, body) -> ...
```

La semantica informale è quella del costrutto "for", corrispondente a "for i = e1 to e2 do body" in OCaml.

Si richiede dunque di valutare "e1" ed "e2", e nel caso in cui entrambi siano numeri naturali, con il valore di "e1" minore o uguale a quello di "e2", eseguire il comando "body" per ogni numero "n" compreso fra il valore di "e1" e quello di "e2". Per ogni esecuzione di "body", nell'ambiente, all'identificatore "i" deve essere assegnato il valore di "n", e lo stato risultante deve essere passato da una iterazione all'altra e restituito dalla funzione csem.

APPENDICE
==========
Si faccia riferimento a un linguaggio imperativo, i cui domini sintattici sono i seguenti:

```
type ide = string

type boolean = True | False

type exp =
  Eint of int
| Eplus of (exp * exp)
| Eminus of (exp * exp)
| Eide of ide
| Ebool of boolean
| Eeql of (exp * exp)
| Eleq of (exp * exp)
| Enot of exp
| Eand of (exp * exp)
| Eor of (exp * exp)
| Eifthenelse of (exp * exp * exp)

type com =
  Cassign of ide * exp
| CCondAssign of ide * exp * exp * exp
| Cvar of ide * exp
| Cconst of ide * exp
| Cifthenelse of exp * com * com
| Cwhile of exp * com
| Cfor of ide * exp * exp * com

type prog = Pseq of com * prog | Pend of exp
```

Si usino i domini semantici:

```
type eval = Int of int | Bool of bool   (* Valori esprimibili *)
type loc = int
type mval = eval  (* Valori memorizzabili *)

type store = int * (loc -> mval) (* il primo elemento della coppia è la minima
                    locazione non definita *)
let empty_store = (0,fun l -> memory_error ())
```

```
    let apply_store st l = (snd st) l
    let allocate : store -> loc * store =
      fun st ->
        let l = fst st in
        let l1 = l + 1 in
        let st1 = (l1,snd st) in
        (l,st1)
    let update : store -> loc -> mval -> store = fun st l mv ->
      match st with
        (maxloc,fn) ->
          let fn1 l1 = if l = l1 then mv else fn l1 in
          (maxloc,fn1)

    type dval = E of eval | L of loc  (* Valori denotabili *)

    type env = ide -> dval
    let empty_env = fun v -> unbound_identifier_error v
    let bind e v r = fun v1 -> if v1 = v then r else e v1
    let apply_env e v = e v
```

Infine, si considerino i seguenti tipi per le funzioni di valutazione semantica

```
let rec esem : exp -> env -> store -> eval = ...
let rec csem : com -> env -> store -> (env * store) = ...
let rec psem : prog -> env -> store -> eval = ...
```

N.B.: si usa ocaml come linguaggio di specifica; non verranno presi in considerazione piccoli errori di sintassi ocaml.