Linguaggi di Programmazione (III Anno), Ottobre 1998 - OOP

Si puo' usare una sintassi sia C++ che Java-like (ma si consiglia l'uso di Java).

Esercizio OOP.1 (5 punti) Si assuma di avere a disposizione una classe `TwoDPoint` che implementa, in modo non precisato, punti nel piano bidimensionale.

```
class TwoDPoint {
  \\ ... PARTE PRIVATE
  public TwoDPoint (int x, int y) { ... }  // costruisce un punto con coordinate x,y
  public TwoDPoint (TwoDPoint p) { ... } // costruisce un punto uguale a p
  public boolean opposite (TwoDPoint p) { ... } // testa se p e' opposto rispetto all'origine
}
```

Si completi la seguente classe `3DPoint` che implementa punti nello spazio tridimensionale.

```
class ThreeDPoint extends TwoDPoint {
  private int z;
  public ThreeDPoint (int x, int y, int z) { ... } // costruisce un punto con coordinate x,y,z
  public ThreeDPoint (ThreeDPoint p) { ... } // costruisce un punto uguale a p
  public boolean opposite (ThreeDPoint p) { ... } // testa se p e' opposto rispetto all'origine
}
```

Esercizio OOP.2 (10 punti) Si implementino le seguenti sottoclassi della classe astratta *succ*, i cui oggetti rappresentano successione infinite $(x_n | n \in N)$ di interi.

```
abstract class succ {
abstract public int head (); valore del primo elemento
abstract public succ tail (); resto della successione
}
class succ_arit extends succ {
... PARTE PRIVATE
public succ_arit(int x)       DA IMPLEMENTARE (COSTRUTTORE)
METODI DI succ DA IMPLEMENTARE
}
class succ_geom extends succ {
... PARTE PRIVATE
public succ_geom(int x)       DA IMPLEMENTARE (COSTRUTTORE)
METODI DI succ DA IMPLEMENTARE
}
class succ_even extends succ {
... PARTE PRIVATE
public succ_even(succ s)       DA IMPLEMENTARE (COSTRUTTORE)
METODI DI succ DA IMPLEMENTARE
}
class succ_sum extends succ {
... PARTE PRIVATE
public succ_sum(succ s1, succ s2)       DA IMPLEMENTARE (COSTRUTTORE)
METODI DI succ DA IMPLEMENTARE
}
```

I costruttori delle classi devono corrispondere alle seguenti successioni:

$succ_arit(x)_n = n*x \quad succ_geom(x)_n = x^n \quad succ_even(s)_n = s_{2*n} \quad succ_sum(s1, s2)_n = s1_n + s2_n$

dove $s_0 = s.head()$, $s_1 = s.tail().head()$ e cosi' via.

Linguaggi di Programmazione (III Anno), Ottobre 1998 - FP

Esercizio FP.1 (5 punti) Dare il tipo principale delle seguenti funzioni ML

```
fun f x = (x,x);
fun g (x,y) = (f x,f y);
```

Esercizio FP.2 (10 punti) Si consideri il tipo di dato delle *liste lazy*

```
type 'a D = unit -> 'a;
datatype 'a List = Cons of 'a * ('a List) D;
```

che permette di rappresentare successioni potenzialmente infinite.

- Si definiscano le seguenti funzioni higher-order, che devono comportarsi in modo analogo a quello delle funzioni con lo stesso nome che operano su liste finite:

  ```
  val Map : ('a -> 'b) -> 'a List -> 'b List;
  val Fold : ('a * 'b D -> 'b) -> 'a List -> 'b.
  ```

  Si faccia attenzione alla possibilita' di divergenza presente nelle liste lazy, come illustrato dal seguente esempio di sessione interattiva:

  ```
  - fun undef () = undef ();
  val undef = ... : 'a D
  - val l1 = Cons(true,undef);
  val l1 = Cons (true,...) : bool List
  - val l2 = Map not l1;
  val l2 = Cons (false,...) : bool List
  ```

- Utilizzando $Fold$ con parametro f opportuno, definire la funzione $ExistsTrue : bool\ List \rightarrow bool$, che restituisce $true$ le la lista contiene true, altrimenti diverge.

- Si definisca la seguente funzione higher-order

  ```
  val generate : ('a -> 'b * 'a) -> 'a -> 'b List;
  ```

  che genera una lista lazy a partire da un generatore $g : a \rightarrow b * a$ ed un seme $s : a$, piu' precisamente se $g(s) = (x, s')$ allora x e' il primo elemento della lista e s' e' il seme per generare il resto della lista (continuando ad usare g come generatore).

- Utilizzando $generate$ con parametri g e s opportuni, definire le liste lazy che corrispondono alle seguenti successioni di interi:
  - $(0, 0, \ldots, 0, 0, \ldots)$
  - $(0, 1, \ldots, n, n+1, \ldots)$
  - $(1, x, \ldots, x^n, x^{n+1}, \ldots)$

```
(* OOP.1 *)
class TwoDPoint {
  private int x,y;
  public TwoDPoint (int x, int y) { this.x = x; this.y =y; }
  public TwoDPoint (TwoDPoint p) { this(p.x,p.y);}
  public boolean opposite (TwoDPoint p) { return x ==-p.x && y == -p.y; }
}
class ThreeDPoint extends TwoDPoint {
  private int z;
  public ThreeDPoint (int x, int y, int z) { super(x,y); this.z = z; }
  public ThreeDPoint (ThreeDPoint p) { super(p); z=p.z; }
  public boolean opposite (ThreeDPoint p) {
 return super.opposite (p) && z == -p.z; }
}

(* OOP.2 *)
abstract class succ {
  abstract public int head ();
  abstract public succ tail ();
}

class succ_arit extends succ {
  private int x;
  private int first;
  private succ_arit (int x, int first) {
   this.x = x;
   this.first = first;
  }
  public succ_arit (int x) { this(x,0);}
  public int head () { return first;}
  public succ tail () { return new succ_arit(x,first+x);}
}

class succ_geom extends succ {
  private int x;
  private int first;
  private succ_geom (int x, int first) {
   this.x = x;
   this.first = first;
  }
  public succ_geom (int x) { this(x,1);}
  public int head () { return first;}
  public succ tail () { return new succ_geom(x,first*x);}
}

class succ_even extends succ {
  private succ s;
  public succ_even (succ s) {
   this.s = s;
  }
  public int head () { return s.head();}
  public succ tail () { return new succ_even (s.tail().tail());}
}
```

```
class succ_sum extends succ {
  private succ s1,s2;
  public succ_sum (succ s1, succ s2) {
   this.s1 = s1;
   this.s2 = s2;
  }
  public int head () { return s1.head() + s2.head();}
  public succ tail () { return new succ_sum (s1.tail(),s2.tail());}
}

(* FP.1 *)
- fun f x = (x,x);
 val f = fn : 'a -> 'a * 'a
- fun g (x,y) = (f x,f y);
val g = fn : 'a * 'b -> ('a * 'a) * ('b * 'b)

(* FP.2 *)
type 'a D = unit -> 'a;
datatype 'a List = Cons of 'a * ('a List) D;

local
fun Dmap f d = fn ()=> f (d ());
in
fun Map f (Cons(x,l)) = Cons(f x,Dmap (Map f) l);
fun Fold f (Cons(x,l)) = f(x,Dmap (Fold f) l);
end;

local
fun f(b,d) = if b then true else d();
in
val ExistsTrue = Fold f;
end;

fun generate g s = let val (x,s') = g s in Cons(x,fn _=>generate g s') end;

generate (fn () => (0,())) ();
generate (fn n => (n,n+1)) 0;
generate (fn (x:int,xn) => (xn,(x,xn*x))) (x,1);
```