Linguaggi e Programmazione Orientata agli Oggetti

Soluzioni della prova scritta

a.a. 2011/2012

6 febbraio 2012

1. Il linguaggio $\mathcal{L}$ contiene tutte le successioni finite di uno o più identificatori separati da `'.'`, dove ogni identificatore è una successione finita e non vuota di caratteri alfa-numerici che inizia con una lettera.

   (a) La seguente grammatica regolare destra genera il linguaggio $\mathcal{L}$ a partire da `Path`.

   ```
   Path ::= a Epth | ... | z Epth | A Epth | ... | Z Epth
   Epth ::= ϵ | a Epth | ... | z Epth | A Epth | ... | Z Epth | 0 Epth | ... | 9 Epth | . Path
   ```

   (b)
   ```
   java.util.regex.Pattern.compile("[a-zA-Z][a-zA-Z0-9]*(\\.[a-zA-Z][a-zA-Z0-9]*)*");
   ```

   Oppure, usando le abbreviazioni POSIX:

   ```
   java.util.regex.Pattern.compile("\\p{Alpha}\\p{Alnum}*(\\.\\p{Alpha}\\p{Alnum}*)*");
   ```

2. (a)
   ```
   let rec ordered = function x::y::l -> x<y && ordered (y::l) | _ -> true;;
   ```

   (b)
   ```
   let rec all p = function x::y::l -> p x y && all p (y::l) | _ -> true;;
   ```

   (c)
   ```
   let ordered = all (function x -> function y -> x < y);;

   let interval = all (function x -> function y -> y = x + 1);;
   ```

3.
```
package scritto2012_02_06;

public interface Function<X, Y> {
        Y apply(X x);
}
```

```
package scritto2012_02_06;

import java.util.Set;
public class SetUtil {
        public static <X> boolean all(Function<X, Boolean> p, Set<X> s) {
                for (X e : s)
                        if (!p.apply(e))
                                return false;
                return true;
        }
        public static <X> boolean exists(Function<X, Boolean> p, Set<X> s) {
                for (X e : s)
                        if (p.apply(e))
                                return true;
                return false;
        }
        public static <X, Y> void map(Function<X, Y> f, Set<X> inSet, Set<Y> outSet) {
                for (X e : inSet)
                        outSet.add(f.apply(e));
        }
        public static <X, Y> X iterate(Function<X, Function<Y, X>> f, X initVal,
                        Set<Y> s) {
                X res = initVal;
                for (Y e : s)
                        res = f.apply(res).apply(e);
                return res;
        }
        public static <X> void union(Set<X> inSet1, Set<X> inSet2, Set<X> outSet) {
                outSet.addAll(inSet1);
                outSet.addAll(inSet2);

        }
        public static <X> void intersect(Set<X> inSet1, Set<X> inSet2, Set<X> outSet) {
                for (X e : inSet1)
                        if (inSet2.contains(e))
                                outSet.add(e);
        }

}
```

```
package scritto2012_02_06;

public class Test {

	static final class Sum implements Function<Integer, Function<Integer, Integer>> {
		@Override
		public Function<Integer, Integer> apply(final Integer x) {
			return new Function<Integer, Integer>() {
				@Override
				public Integer apply(Integer y) {
					return x + y;
				}
			};
		}
	}

	static Integer sumAll(Set<Integer> s) {
		return SetUtil.iterate(new Sum(), 0, s);
	}
}
```

4. (a) Il codice viene compilato correttamente: `ac` ha tipo statico `AC`, l'unico metodo potenzialmente applicabile è `m(A a)`, l'altro metodo è package private, quindi non è accessibile dal package `c`; il tipo statico dell'argomento è `AC`, quindi il metodo è applicabile per sottotipo (fase 1) visto che `AC` implementa `A`.

   Il tipo dinamico dell'oggetto contenuto in `ac` è `AC`, quindi viene eseguito il metodo `m(A a)` in `AC`. Viene stampato `"AC m(A)"`.

   (b) Il codice viene compilato correttamente: `(A) ac` è corretto staticamente dato che `AC` implementa `A` (widening reference conversion) e il tipo statico dell'oggetto target è `A`. L'unico metodo potenzialmente applicabile è `m(A a)` che è anche applicabile (vedi punto 4a).

   Poiché si tratta di widening reference conversion, il cast `(A) ac` non corrisponde ad alcuna azione a runtime; il tipo dinamico dell'oggetto contenuto in `ac` è `AC`, quindi analogamente a quanto accade per il punto 4a, viene stampato `"AC m(A)"`.

   (c) Il codice viene compilato correttamente: `bc` ha tipo statico `BC`, i metodi potenzialmente applicabili sono `m(A a)` (ereditato da `AC`) e `m(B b)`; l'altro metodo dichiarato in `BC` è package private, quindi non è accessibile dal package `c`, mentre il metodo package private `m(AC ac)` in `AC` non viene ereditato da `BC` poiché non accessibile dal package `b`. Il tipo statico dell'argomento è `AC` che è sottotipo di `A` ma non di `B`, quindi l'unico metodo applicabile per sottotipo (fase 1) è `m(A a)`.

   Il tipo dinamico dell'oggetto contenuto in `bc` è `BC`, quindi la ricerca del metodo parte da `BC`; poiché `m(A a)` non è definito in `BC`, la ricerca del metodo continua nella superclasse diretta `AC` dove il metodo viene trovato. Viene quindi stampato `"AC m(A)"`.

   (d) Il codice **non** viene compilato correttamente: `bc` ha tipo statico `BC`, quindi per gli stessi motivi del punto 4c i metodi potenzialmente applicabili sono `m(A a)` e `m(B b)`. Il tipo statico dell'argomento è `BC` che è sottotipo sia di `A`, sia di `B`, quindi entrambi i metodi `m(A a)` e `m(B b)` sono applicabili per sottotipo (fase 1). Poiché nessuno dei due tipi `A` e `B` è sottotipo dell'altro, nessuno dei due metodi è più specifico e, quindi, la selezione del metodo fallisce.

   (e) Il codice viene compilato correttamente: `c` ha tipo statico `C` quindi i metodi potenzialmente applicabili sono tutti quelli dichiarati in `C` (gli unici due metodi ereditati da `BC` e da `AC` sono ridefiniti in `C`). Il tipo statico dell'argomento è `AC` che è sottotipo sia di `A`, sia di `AC`, ma non di `B` e di `BC`; quindi, i soli metodi applicabili per sottotipo (fase 1) sono `m(A a)` e `m(AC ac)`; tra i due viene selezionato il più specifico che è `m(AC ac)`, visto che `AC` è sottotipo di `A`.

   Il tipo dinamico dell'oggetto contenuto in `c` è `C`, quindi il metodo `m(AC ac)` viene trovato nella classe `C` e viene stampato `"C m(AC)"`.

   (f) Il codice viene compilato correttamente: `c` ha tipo statico `C` quindi i metodi potenzialmente applicabili sono tutti quelli dichiarati in `C`, come accade nel punto 4e. Il tipo statico dell'argomento è `BC` che è sottotipo di `A`, di `AC`, di `B` e di `BC`; quindi, tutti e quattro i metodi sono applicabili per sottotipo (fase 1); il metodo selezionato è quello più specifico, ossia `m(BC bc)`, dato che `BC` è sottotipo di `A`, di `AC` e di `B`.

   Il tipo dinamico dell'oggetto contenuto in `c` è `C`, quindi il metodo `m(BC bc)` viene trovato nella classe `C`. Il metodo contiene l'invocazione di metodo `bc.m((B) bc)` dove `bc` ha tipo statico `BC`; il cast è staticamente corretto perché `BC` è sottotipo di `B` (widening reference conversion, nessuna azione a runtime), i metodi potenzialmente applicabili sono `m(B b)`, `m(BC ac)` (`C` e `BC` sono nello stesso package) e `m(A a)` (ereditato da `AC`; l'altro metodo di `AC` non viene ereditato perché è package private). Il tipo statico dell'argomento è `B` e, poiché `B` è sottotipo di `B` e di `A`, ma non di `BC`, solo i metodi `m(B b)` e `m(A a)` sono applicabili per sottotipo (fase 1). Tra questi viene selezionato il più specifico, ossia `m(B b)`, visto che `B` è sottotipo di `A`. Poiché `bc` contiene un oggetto di tipo dinamico `BC`, il metodo `m(B b)` viene cercato a partire da `BC` e, quindi, `bc.m((B) bc)` restituisce la stringa `"BC m(B)"` che viene concatenata con `" C m(BC)"`, ossia, viene stampata la stringa `"BC m(B) C m(BC)"`.