Linguaggi e Programmazione Orientata agli Oggetti

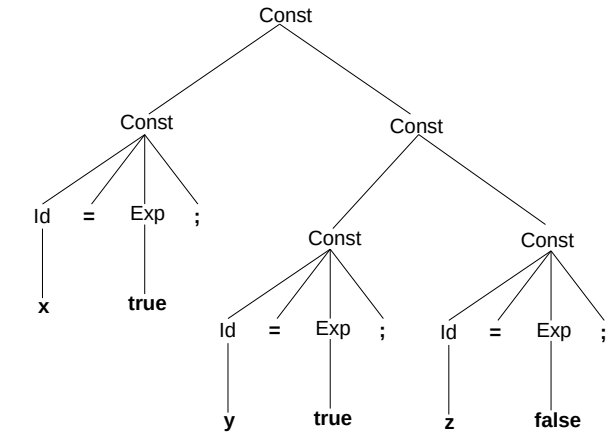
Soluzione della prova scritta

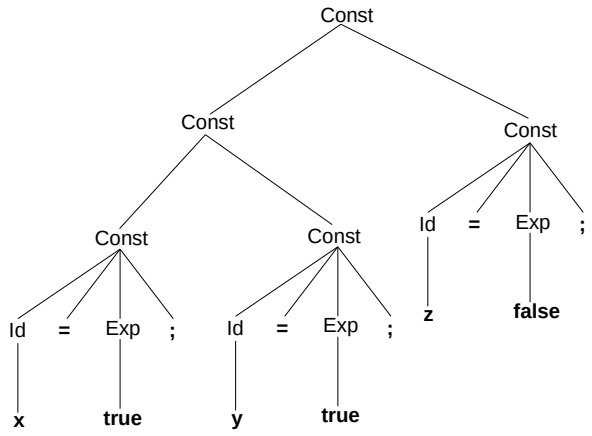
a.a. 2013/2014

10 settembre 2014

1. (a) l'espressione regolare definisce una semplice sintassi per literal numerici floating-point (senza notazione scientifica), quindi solo le prime due asserzioni falliscono.
   i. `assert p.matcher("42.0.42").matches();` fallisce
   ii. `assert p.matcher(".").matches();` fallisce
   iii. `assert p.matcher("420.042").matches();` ha successo;
   iv. `assert p.matcher("0042").matches();` ha successo;
   v. `assert p.matcher("42.").matches();` ha successo;
   vi. `assert p.matcher(".42").matches();` ha successo.

   (b) Esistono due diversi alberi di derivazione per la stringa **`x=true; y=true; z=false;`**.





La seguente grammatica genera lo stesso linguaggio, ma non è ambigua: in questo caso la concatenazione di definizioni di costanti associa da sinistra.

```
Const ::= Id = Exp ; | Const Id = Exp ;
Id ::= x | y | z
Exp ::= false | true | Exp && false | Exp && true
```

2. (a) `let rec compose x = function f::l -> f (compose x l) | _ -> x;;`

(b)
```
let compose_acc x l =
  let rec aux a =
    function f::t -> aux (f a) t
      |  _ -> a
  in (aux x (List.rev l));;
```

(c)
```
let compose_it x l = it_list (fun a f -> f a) x (List.rev l);;
```

3. (a) Il codice viene compilato correttamente: `x` ha tipo statico `X`, l'unico metodo potenzialmente applicabile è quello con il parametro di tipo `Integer` (l'altro metodo `m` di `X` è **`private`**); il metodo è applicabile per method invocation conversion (fase 2) (ma non per sottotipo): l'argomento di tipo statico **`int`** viene convertito in `Integer` (boxing conversion). Il tipo dinamico dell'oggetto contenuto in `x` è `X`, quindi viene eseguito il metodo in `X` con parametro di tipo `Integer`; la chiamata all'interno del metodo sull'oggetto **`this`** viene risolta con il metodo **`private`** di tipo `Number`: il metodo è applicabile per method invocation conversion (fase 2), il tipo statico **`double`** dell'argomento viene convertito a `Double` (boxing conversion) e quindi `Double` viene convertito in `Number` (widening reference conversion opzionale). Viene stampato `"Bar Foo"`.

   (b) Il codice viene compilato correttamente: `y` ha tipo statico `Y`, l'unico metodo potenzialmente applicabile è quello dichiarato in `Y`: i due metodi in `X` non vengono ereditati, uno perché **`private`**, l'altro perché ridefinito in `Y`. Il metodo è applicabile per la stessa ragione del punto (3a). Il tipo dinamico dell'oggetto contenuto in `y` è `Y`, quindi viene eseguito il metodo in `Y`. Per quanto riguarda l'invocazione con **`super`**, l'unico metodo di `X` potenzialmente applicabile è quello **`protected`** (l'altro non è accessibile in quanto **`private`**); il metodo è ovviamente applicabile per sottotipo (fase 1). La stringa restituita dal metodo in `X` è la stessa specificata al punto (3a), quindi viene stampato `"Bar Baz Foo"`.

   (c) Il codice non compila correttamente: `x` ha tipo statico `X`, l'unico metodo potenzialmente applicabile è quello con il parametro di tipo `Integer` (l'altro metodo `m` di `X` è **`private`**); il tipo statico dell'argomento è **`double`**, che non può essere convertito a `Integer` (`Double` non è sottotipo di `Integer`).

   (d) Il codice viene compilato correttamente: il cast è corretto, visto che si tratta di widening reference conversion (da `Y` a `X`). Il tipo statico dell'oggetto su cui viene invocato il metodo è `X`, quindi l'espressione è corretta per gli stessi motivi del punto (3a) e il metodo selezionato è lo stesso. Il tipo dinamico dell'oggetto su cui viene invocato il metodo è `Y`, quindi viene eseguito il metodo di `Y` che ridefinisce quello in `X` **`protected`**. Viene stampata la stessa stringa del punto (3b), ossia `"Bar Baz Foo"`.

   (e) Il codice viene compilato correttamente: `z` ha tipo statico `Z` quindi ci sono due metodi potenzialmente applicabili, quello in `Y` e quello in `Z`; visto che l'argomento ha tipo statico **`double`**, nessuno dei due è applicabile né per sottotipo (fase 1), né per method invocation conversion: quello in `Y` non è applicabile perché `Double` non è sottotipo di `Integer`, quello in `Z` perchè ha arità variabile. Il metodo in `Z` è però applicabile per method invocation conversion con arità variabile (fase 3): il tipo **`double`** viene convertito a `Double` (box conversion) e `Double` a `Number` (widening reference conversion opzionale). Il tipo dinamico dell'oggetto contenuto in `z` è `Z`, quindi viene eseguito il metodo in `Z` e stampato `"Goo"`.

   (f) Il codice viene compilato correttamente: `z` ha tipo statico `Z` quindi ci sono due metodi potenzialmente applicabili, quello in `Y` e quello in `Z`; nessuno dei due è applicabile per sottotipo (fase 1), ma il metodo in `Y` è applicabile per method invocation conversion (fase 2) (caso analogo al punto (3a)), mentre il metodo in `Z` no, quindi viene selezionato il metodo in `Y`. Il tipo dinamico dell'oggetto contenuto in `z` è `Z`, quindi il metodo con parametro di tipo `Integer` viene cercato a partire da `Z` e, di conseguenza, viene eseguito il metodo in `Y`. Viene stampata la stessa stringa del punto (3b), ossia `"Bar Baz Foo"`.

4. (a)
```
public class BinTreeFactory {
    private BinTreeFactory() {
    }

    public static <T> BinTree<T> leaf(final T e) {
        return new BinTree<T>() {
            public <R> R accept(Visitor<T, R> v) {
                return v.visitLeaf(e);
            }
        };
    }

    public static <T> BinTree<T> branch(final T e,final BinTree<T> l, final BinTree<T> r) {
        return new BinTree<T>() {
            public <R> R accept(Visitor<T, R> v) {
                return v.visitBranch(e, l, r);
            }
        };
    }
}
```

```
public class Depth<E> implements Visitor<E, Integer> {

    @Override
    public Integer visitLeaf(E elt) {
        return 0;
    }

    @Override
    public Integer visitBranch(E e, BinTree<E> left, BinTree<E> right) {
        return Math.max(left.accept(this), right.accept(this)) + 1;
    }

}
```

(b)
```
public class InOrder<E> implements Visitor<E, StringBuilder> {
    final private StringBuilder stb = new StringBuilder();

    @Override
    public StringBuilder visitLeaf(E e) {
        return stb.append(e.toString());
    }

    @Override
    public StringBuilder visitBranch(E e, BinTree<E> left, BinTree<E> right) {
        left.accept(this);
        stb.append(" ").append(e.toString()).append(" ");
        return right.accept(this);
    }

}
```