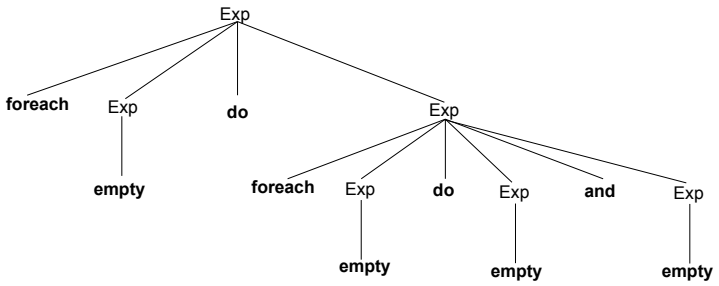
Linguaggi e Programmazione Orientata agli Oggetti

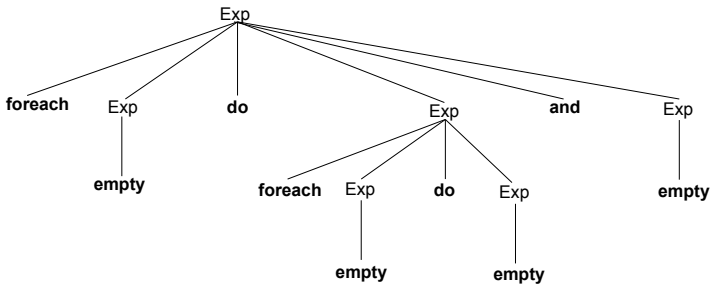
Soluzione della prova scritta

a.a. 2013/2014

23 Luglio 2014

1. Esistono due diversi alberi di derivazione per la stringa **`foreach empty do foreach empty do empty and empty`**.





La seguente grammatica genera lo stesso linguaggio, ma non è ambigua: in questo caso il ramo **`and`** viene associato al **`foreach`** più interno, ossia il **`foreach`** con il ramo **`and`** ha la precedenza su quello senza.

```
Exp ::= foreach Exp do Exp | BExp
BExp ::= foreach Exp do BExp and Exp | ( Exp ) | empty | vector<Exp, Exp>
```

2. (a)
```
erase : 'a -> 'a list -> 'a list
let rec erase e = function
    h::t as l -> if e < h then l else if e = h then t else h::erase e t
  | _ -> [];;
```

   (b)
```
itlist : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b
let rec itlist f a = function x::l -> itlist f (f x a) l | _ -> a;;

first_only : 'a list -> 'a list -> 'a list
let first_only = itlist erase;;
```

   (c)
```
first_only2 : 'a list -> 'a list -> 'a list

let rec first_only2 = function
    h1::t1 as l1 -> (function
        h2::t2 as l2 -> if h1 < h2 then
          h1::first_only2 t1 l2 else if h2 < h1 then first_only2 l1 t2
        else first_only2 t1 t2
      | _ -> l1)
  | _ -> function _ -> [];;
```

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.
```
import java.util.HashSet;

/* ... */

class SetUtil {
  private final SetFactory setFactory; // use this to create empty sets
  public SetUtil(SetFactory setFactory) {
    this.setFactory = setFactory;
  }

  // returns true iff predicate p is true for all elements of s
  public <X> boolean all(Set<X> s, Function<X, Boolean> p) {
    for(X elem : s)
      if (!p.apply(elem))
        return false;
    return true;
  }

  // returns true iff predicate p is true for at least one element of s
  public <X> boolean any(Set<X> s, Function<X, Boolean> p) {
    for(X elem : s)
      if (p.apply(elem))
        return true;
    return false;
  }
```

```
  // returns the results of applying f to all elements of s
  public <X, Y> Set<Y> map(Set<X> s, Function<X, Y> f) {
    Set<Y> result = this.setFactory.newEmptySet();
    for(X elem : s)
      result.add(f.apply(elem));
    return result;
  }

  // f is a curried function of type X -> Y -> X where the accumulator has type X
  // the method returns f (... (f (f initVal e_1) e_2) ...) e_n where s = {e_1, e_2, ..., e_n}
  public <X, Y> X fold(Set<Y> s, Function<X, Function<Y, X>> f, X initVal) {
    X result = initVal;
    for(Y elem : s)
      result = f.apply(result).apply(elem);
    return result;
  }

  // returns the union of inSet1 and inSet2
  public <X> Set<X> union(Set<X> inSet1, Set<X> inSet2) {
    Set<X> result = this.setFactory.newEmptySet();
    result.addAll(inSet1);
    result.addAll(inSet2);
    return result;
  }

  // returns the intersection of inSet1 and inSet2
  public <X> Set<X> intersect(Set<X> inSet1, Set<X> inSet2) {
    Set<X> result = this.setFactory.newEmptySet();
    for(X elem : inSet1)
      if (inSet2.contains(elem))
        result.add(elem);
    return result;
  }
}

class Test {

  // defines the function f such that f x y = x * y
  static final class Prod 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;
        }

      };
    }
  }

  // returns the product of all elements of s
  // by using class Prod and method SetUtil.fold
  static Integer multiplyAll(Set<Integer> s) {
    SetUtil su = new SetUtil(new SetFactory() {
      @Override
      public <X> Set<X> newEmptySet() {
        return new HashSet<X>();
      }
    });
    return su.fold(s, new Prod(), 1);
  }
}
```