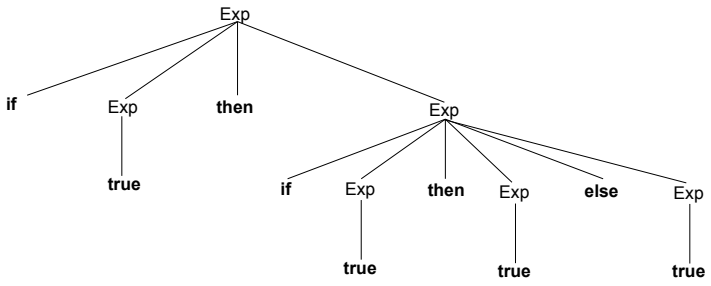
Linguaggi e Programmazione Orientata agli Oggetti

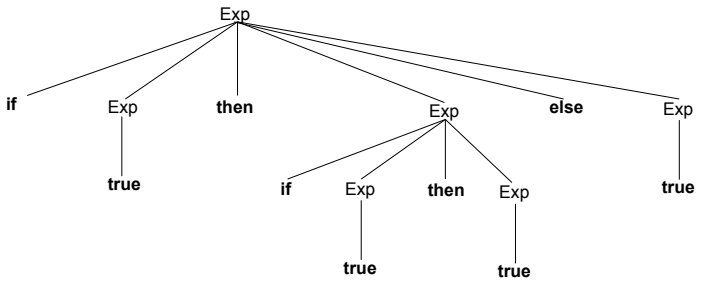
Soluzioni della prova scritta

a.a. 2011/2012

16 gennaio 2012

1. Esistono due diversi alberi di derivazione per la stringa **`if true then if true then true else true`**.





La seguente grammatica genera lo stesso linguaggio, ma non è ambigua: in questo caso il ramo **`else`** viene associato all'**`if`** più interno (come in Java), ossia l'**`if`** con il ramo **`else`** ha la precedenza su quello senza.

```
Exp ::= if Exp then Exp | BExp
BExp ::= if Exp then BExp else Exp | ( Exp ) | true | false
```

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

diff : 'a list -> 'a list -> 'a list
let diff = itlist delete;;
```

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

let rec diff2 = function
    h1::t1 as l1 -> (function
        h2::t2 as l2 -> if h1 < h2 then
          h1::diff2 t1 l2 else if h2 < h1 then diff2 l1 t2
        else diff2 t1 t2
      | _ -> l1)
  | _ -> function _ -> [];;
```

3.
```java
package scritto2012_01_16;

public interface Stack<E> extends Iterable<E>{
	public E push(E item);
	public E pop();
	public E peek();
	public boolean empty();
}
```

```java
package scritto2012_01_16;

import java.util.Iterator;
import java.util.EmptyStackException;
import java.util.NoSuchElementException;

public class LinkedStack<E> implements Stack<E> {
	private final Node<E> dummyNode;

	private static class Node<E> {

		private E elem;
		private Node<E> next;

		private Node(E elem, Node<E> next) {
			this.elem = elem;
			this.next = next;
		}

	}

	public LinkedStack() {
		dummyNode = new Node<E>(null, null);
		dummyNode.next = dummyNode;
	}

	@Override
	public String toString() {
		String res = "[";
		for (E e : this)
			res += " " + e;
		return res + " ]";
	}

	@Override
	public Iterator<E> iterator() {
		return new Iterator<E>() {
			private Node<E> prevNode = dummyNode;

			@Override
			public boolean hasNext() {
				return prevNode.next != dummyNode;
			}

			@Override
			public E next() {
				if (!hasNext())
					throw new NoSuchElementException();
				prevNode = prevNode.next;
				return prevNode.elem;
			}

			@Override
			public void remove() {
				throw new UnsupportedOperationException();
			}

		};
	}

	@Override
	public E push(E item) {
		dummyNode.next = new Node<E>(item, dummyNode.next);
		return item;
	}

	@Override
	public E pop() {
		E res = peek();
		dummyNode.next = dummyNode.next.next;
		return res;
	}

	@Override
	public E peek() {
		if (empty())
			throw new EmptyStackException();
		return dummyNode.next.elem;
	}

	@Override
	public boolean empty() {
		return dummyNode.next == dummyNode;
	}
```

```
        @Override
        public int hashCode() {
                int hashCode = 17;
                for (E e : this)
                        hashCode = 31 * hashCode + ((e == null) ? 0 : e.hashCode());
                return hashCode;
        }

        @Override
        public boolean equals(Object obj) {
                if (this == obj)
                        return true;
                if (!(obj instanceof Stack))
                        return false;
                Iterator<E> thisIt = iterator();
                @SuppressWarnings("unchecked")
                Iterator<E> objIt = ((Iterable<E>) obj).iterator();
                while (thisIt.hasNext() && objIt.hasNext()) {
                        E e1 = thisIt.next();
                        E e2 = objIt.next();
                        if (!(e1 == null ? e2 == null : e1.equals(e2)))
                                return false;
                }
                return !(thisIt.hasNext() || objIt.hasNext());
        }
}
```

4. (a) Il codice viene compilato correttamente: `a` ha tipo statico `A`, l'unico metodo potenzialmente applicabile è quello con il parametro di tipo `Integer` (l'altro metodo `m` di `A` 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 `a` è `A`, quindi viene eseguito il metodo in `A` 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 `"A.Integer A.Number"`.

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

   (c) Il codice viene compilato correttamente: `b` ha tipo statico `B`, l'unico metodo potenzialmente applicabile è quello dichiarato in `B`: i due metodi in `A` non vengono ereditati, uno perché **`private`**, l'altro perché ridefinito in `B`. Il metodo è applicabile per la stessa ragione del punto (4a).

   Il tipo dinamico dell'oggetto contenuto in `b` è `B`, quindi viene eseguito il metodo in `B`. Per quanto riguarda l'invocazione con **`super`**, l'unico metodo di `A` 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 `A` è la stessa specificata al punto (4a), quindi viene stampato `"B.Integer A.Integer A.Number"`.

   (d) Il codice viene compilato correttamente: il cast è corretto, visto che si tratta di widening reference conversion (da `B` ad `A`). Il tipo statico dell'oggetto su cui viene invocato il metodo è `A`, quindi l'espressione è corretta per gli stessi motivi del punto (4a) e il metodo selezionato è lo stesso. Il tipo dinamico dell'oggetto su cui viene invocato il metodo è `B`, quindi viene eseguito il metodo di `B` che ridefinisce quello in `A` **`protected`**. Viene stampata la stessa stringa del punto (4c), ossia `"B.Integer A.Integer A.Number"`.

   (e) Il codice viene compilato correttamente: `c` ha tipo statico `C` quindi ci sono due metodi potenzialmente applicabili, quello in `B` e quello in `C`; nessuno dei due è applicabile per sottotipo (fase 1), ma il metodo in `B` è applicabile per method invocation conversion (fase 2) (caso analogo al punto (4a)), mentre il metodo in `C` no, quindi viene selezionato il metodo in `B`. Il tipo dinamico dell'oggetto contenuto in `c` è `C`, quindi il metodo con parametro di tipo `Integer` viene cercato a partire da `C` e, di conseguenza, viene eseguito il metodi in `B`. Viene stampata la stessa stringa del punto (4c), ossia `"B.Integer A.Integer A.Number"`.

   (f) Il codice viene compilato correttamente: `c` ha tipo statico `C` quindi ci sono due metodi potenzialmente applicabili, quello in `B` e quello in `C`; 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 `B` non è applicabile perché `Double` non è sottotipo di `Integer`, quello in `C` perchè ha arità variabile. Il metodo in `C` 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 `c` è `C`, quindi viene eseguito il metodo in `C` e stampato `"C.Number"`.