

# febbraio 2025

4. Considerare le seguenti interface

```
public interface FormulaAST {<T> T accept(Visitor<T> v);}
public interface Visitor<T> {
    T visitBoolLiteral(boolean value);
    T visitNegationOp(FormulaAST exp);
    T visitImplicationOp(FormulaAST left, FormulaAST right);
}
```

che definiscono i tipi degli alberi immutabili della sintassi astratta di formule booleane e dei loro corrispondenti oggetti visitor.

L'interfaccia FormulaAST è implementata dalle classi BoolLiteral, NegationOp e ImplicationOp che corrispondono rispettivamente alle costanti booleane e agli operatori di negazione e implicazione logica.

```
// costanti booleane
public class BoolLiteral implements FormulaAST { /* da completare */ }
```

```
// operatore di negazione
public class NegationOp implements FormulaAST { /* da completare */ }
```

```
// operatore di implicazione
public class ImplicationOp implements FormulaAST { /* da completare */ }
```

Per esempio, le variabili impl e neg nel seguente codice

```
var trueLit = new BoolLiteral(true);
var falseLit = new BoolLiteral(false);
var impl = new ImplicationOp(trueLit, falseLit);
var neg = new NegationOp(impl);
```

fanno riferimento a oggetti che rappresentano gli alberi della sintassi astratta rispettivamente delle formule  $true \Rightarrow false$  e  $\neg(\neg(true) \Rightarrow false)$ .

- (a) completare la classe BoolLiteral.
- (b) completare la classe NegationOp.
- (c) completare la classe ImplicationOp.
- (d) completare la classe

```
public class Eval implements Visitor<Boolean> { /* ... */ }
```

che definisce visitor che permettono la valutazione di una formula booleana secondo la semantica convenzionale.

Esempio:

```
assert !impl.accept(new Eval());
assert neg.accept(new Eval());
```

dove impl e neg sono definiti come sopra.

**Importante:** quando necessari, i costruttori definiti nelle classi devono garantire che l'invariante di classe sia soddisfatto.

```
public class Eval implements Visitor<Boolean> {
    @Override
    public Boolean visitBoolLiteral(boolean value) {
        return value;
    }
    @Override
    public Boolean visitNegationOp(FormulaAST exp) {
        return !exp.accept(this);
    }
    @Override
    public Boolean visitImplicationOp(FormulaAST left, FormulaAST right) {
        return !left.accept(this) || right.accept(this);
    }
}
```

```
public class BoolLiteral implements FormulaAST {
    private final boolean value;
    public BoolLiteral(boolean value) {
        this.value = value;
    }
    @Override
    public <T> accept(Visitor<T> v) {
        return v.visitBoolLiteral(value);
    }
}
```

```
public class NegationOp implements FormulaAST {
    private final FormulaAST exp;
    public NegationOp(FormulaAST exp) {
        this.exp = requireNonNull(exp);
    }
    @Override
    public <T> accept(Visitor<T> v) {
        return v.visitNegationOp(exp);
    }
}
```

```
public class ImplicationOp implements FormulaAST {
    private final FormulaAST left;
    private final FormulaAST right;
    public ImplicationOp(FormulaAST left, FormulaAST right) {
        this.left = requireNonNull(left);
        this.right = requireNonNull(right);
    }
    @Override
    public <T> accept(Visitor<T> v) {
        return v.visitImplicationOp(left, right);
    }
}
```

# gennaio 2025

4. Considerare le seguenti interfacce

```
public interface BinTree {<T> T accept(Visitor<T> v);}
public interface Visitor<T> { // oggetti visitor di alberi di tipo BinTree
    T visitEmpty();
    T visitNonEmpty(int label, BinTree left, BinTree right);
}

che definiscono i tipi degli alberi binari immutabili etichettati con valori interi e dei loro corrispondenti oggetti visitor.

L'interfaccia BinTree è implementata dalle classi EmptyTree e NonEmptyTree degli alberi rispettivamente vuoti e non.
```

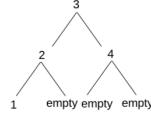
```
// alberi vuoti
public class EmptyTree implements BinTree { /* da completare */ }

// alberi con radice etichettata da un intero,
// sotto-albero destro e sinistro diversi da null
public class NonEmptyTree implements BinTree { /* da completare */ }
```

Per esempio, la variabile t3 nel seguente codice

```
var emptyT = new EmptyTree();
var t1 = new NonEmptyTree(1, emptyT, emptyT);
var t2 = new NonEmptyTree(2, t1, emptyT);
var t4 = new NonEmptyTree(4, emptyT, emptyT);
var t3 = new NonEmptyTree(3, t2, t4);
```

fa riferimento a un oggetto che rappresenta l'albero



- (a) completare la classe EmptyTree.
- (b) completare la classe NonEmptyTree.
- (c) completare la classe

```
public class InOrder implements Visitor<java.util.List<Integer>> { /* ... */ }
che definisce visitor che permettono la visita in-order (prima le etichette del sotto-albero di sinistra, poi l'etichetta della radice e infine le etichette del sotto-albero di destra) degli alberi binari.
```

Esempio: `assert t3.accept(new InOrder()).toString().equals("[1, 2, 3, 4]");`  
dove t3 è definita come sopra.

**Suggerimento:** usare la classe `java.util.LinkedList<E>` che implementa `java.util.List<E>` e i metodi `boolean add(E)` e, optionalmente, `boolean addAll(Collection<? extends E>)`:  
`1.add(e)` aggiunge l'elemento e in fondo alla lista 1, `1.addAll(12)` aggiunge in fondo alla lista 11 tutti gli elementi della lista 12 rispettandone l'ordine.

**Importante:** quando necessari, i costruttori definiti nelle classi devono garantire che l'invariante di classe sia soddisfatto.

# gennaio 2024

4. Considerare l'interfaccia ListExp implementata dalle classi EmptyList (il costruttore di lista vuota) e ListCons (il costruttore di lista non vuota).

```
public interface Exp { <T> T accept(Visitor<T> v); }
public interface ListExp extends Exp { }
```

Nel seguente esempio vengono assegnati a 11, 12 e 13, oggetti che rappresentano, rispettivamente, la lista vuota, la lista che contiene un solo elemento, che è la lista vuota, e la lista che contiene due elementi, entrambi corrispondenti alla lista vuota.

```
var l1 = new EmptyList(); // la lista []
var l2 = new ListCons(11, l1); // la lista [11]
var l3 = new ListCons(l1, 12); // la lista [11, 12]
```

È possibile visitare valori di tipo ListExp mediante oggetti della classe Length che implementa il visitor pattern. La visita restituisce come risultato la lunghezza della lista visitata.

Per esempio, se le variabili 11, 12 e 13 sono state inizializzate come sopra, allora le asserzioni nel codice sotto sono valide.

```
Length length = new Length();
assert l1.accept(length) == 0;
assert l2.accept(length) == 1;
assert l3.accept(length) == 2;
```

- (a) completare il metodo accept della classe EmptyList.

```
public class EmptyList implements ListExp {
    public <T> T accept(Visitor<T> v) {
        // completare
    }
}
```

- (b) completare il costruttore e il metodo accept della classe ListCons.

```
import static java.util.Objects.requireNonNull;

public class ListCons implements ListExp {
    private final Exp head; // invariant head != null
    private final ListExp tail; // invariant tail != null
    public ListCons(Exp head, ListExp tail) {
        // completare
    }
    public <T> T accept(Visitor<T> v) {
        // completare
    }
}
```

- (c) completare i metodi visitEmptyList e visitListCons della classe Length.

```
public class Length implements Visitor<Integer> {
    public Integer visitListCons(Exp head, ListExp tail) {
        // completare
    }
    public Integer visitEmptyList() {
        // completare
    }
}
```

```
public class EmptyTree implements BinTree {
```

**@Override**

```
public <T> accept(Visitor<T> v) {
    return v.visitEmpty();
}
```

}

```
} public class NonEmptyTree implements BinTree {
```

**private final int label;**

**private final BinTree left, right;**

```
public NonEmptyTree(int label, BinTree left, BinTree right) {
```

**this.label = label;**

**this.left = requireNonNull(left);**

**this.right = requireNonNull(right);**

}

**@Override**

```
public <T> T accept(Visitor<T> v) {
    return v.visitNonEmpty(label, left, right);
}
```

}

```
public class InOrder implements Visitor<List<Integer>> {
```

**@Override**

```
public List<Integer> visitEmpty() {
    return new LinkedList<>();
```

}

**@Override**

```
public List<Integer> visitNonEmpty(int label, BinTree left, BinTree right) {
    var res = left.accept(this);
    res.add(label);
    res.addAll(right.accept(this));
    return res;
}
```

}

a) **public class EmptyList implements ListExp {**

**@Override**

```
public <T> T accept(Visitor<T> v) {
    return v.visitEmptyList();
}
```

}

b) **public class ListCons implements ListExp {**

**private final Exp head;**  
**private final ListExp tail;**

```
public ListCons(Exp head, ListExp tail) {
    this.head = requireNonNull(head);
    this.tail = requireNonNull(tail);
}
```

**@Override**

```
public <T> T accept(Visitor<T> v) {
    return v.visitListCons(head, tail);
}
```

}

c) **public class Length implements Visitor<Integer> {**

**@Override**

```
public Integer visitListCons(Exp head, ListExp tail) {
    return 1 + tail.accept(this);
}
```

}

**@Override**

```
public Integer visitEmptyList() {
    return 0;
}
```

}

# Settembre 2023

4. Considerare l'interfaccia JavaEntity implementata dalle classi InstanceEntity (gli oggetti di una certa classe) e ClassEntity (tutte le classi).

```
public interface JavaEntity { <T> T accept(Visitor<T> v); }
public class InstanceEntity implements JavaEntity {
    public <T> T accept(Visitor<T> v) { return v.visitInstanceEntity(); }
}
```

Ogni classe c in ClassEntity (vedere sotto) ha un nome name diverso da null e un array entities di tipo JavaEntity[] che è diverso da null e contiene tutti gli oggetti e le sotto-classi dirette di c.

La gerarchia delle classi e dei loro oggetti è navigabile tramite il visitor pattern e i visitor di tipo SuperClassOf permettono di stabilire se un'entità Java è super-classe diretta o indiretta di una data classe.

Nel seguente esempio class1 si chiama C1, non ha sotto-classi e ha l'oggetto instance1, class2 si chiama C2, non ha sotto-classi e ha l'oggetto instance2, class3 si chiama C3, non ha oggetti e ha due sotto-classi class1 e class2.

```
var instance1 = new InstanceEntity();
var instance2 = new InstanceEntity();
var class1 = new ClassEntity("C1", instance1);
var class2 = new ClassEntity("C2", instance2);
var class3 = new ClassEntity("C3", class1, class2);
assert class3.accept(new SuperClassOf(class3)); // class3 e' super-classe di se stessa
assert class3.accept(new SuperClassOf(class1)); // class3 e' super-classe di class1
assert class3.accept(new SuperClassOf(class2)); // class3 e' super-classe di class2
assert !class1.accept(new SuperClassOf(class3)); // class1 non e' super-classe di class3
assert !class1.accept(new SuperClassOf(class2)); // class1 non e' super-classe di class2
// un oggetto non puo' essere una super-classe
assert !instance1.accept(new SuperClassOf(class3));
```

(a) completare il costruttore e il metodo accept della classe ClassEntity.

```
public class ClassEntity implements JavaEntity {
    private final String name;
    private final JavaEntity[] entities; // instances or subclasses

    public ClassEntity(String name, JavaEntity... entities) { // shallow copy
        // completare
    }

    public <T> T accept(Visitor<T> v) { /* completare */ }
    public String getName() { return name; }
}
```

(b) completare il costruttore e il metodo visitInstanceEntity della classe SuperClassOf.

(c) completare il metodo visitClassEntity della classe SuperClassOf.

```
public class SuperClassOf implements Visitor<Boolean> {
    private final ClassEntity classEntity;

    public SuperClassOf(ClassEntity classEntity) { /* completare */ }
    public Boolean visitClassEntity(String name, JavaEntity... entities) {
        // completare
    }

    public Boolean visitInstanceEntity() { /* completare */ }
}
```

# Gennaio 2023

4. Considerare il codice che implementa il visitor pattern per binary decision tree (BDT), costituito dai seguenti elementi:

- interfaccia BDT corrispondente ai nodi del binary decision tree, che possono essere di tipo Leaf, con campo value di tipo boolean (nodi foglia etichettati con un valore booleano) oppure InnerNode, con campi left e right di tipo BDT (nodi interni con due figli, rispettivamente sinistro e destro).
- interfaccia Path implementata dalla classe PathClass, corrispondente ai cammini dei binary decision tree dalla radice a una foglia.
- interfaccia Visitor<T> implementata da GetValue i cui oggetti visitor restituiscono il valore booleano della foglia individuata da un determinato cammino nell'albero. Un cammino è una sequenza di stringhe che possono essere solo "L" (per left) o "R" (per right); il visitor solleva l'eccezione IllegalStateException se il cammino non contiene le stringhe corrette o non ha la lunghezza giusta.

Esempio:

```
BDT t1 = new InnerNode(new Leaf(true), new Leaf(false));
BDT t2 = new InnerNode(new Leaf(false), new Leaf(true));
BDT t = new InnerNode(t1, t2);
var getValue = new GetValue(new PathClass("L", "R"));
assert !t.accept(getValue);
getValue = new GetValue(new PathClass("L", "L"));
assert t.accept(getValue);
```

Esempi di cammini per cui il visitor solleva l'eccezione IllegalStateException per l'albero t definito sopra:

```
new PathClass("L", "C") // stringa "C" non valida
new PathClass("L") // cammino troppo corto
new PathClass("L", "R", "L") // cammino troppo lungo
```

- Completare il costruttore e il metodo della classe Leaf.
- Completare il costruttore e il metodo della classe InnerNode.
- Completare il costruttore e i metodi della classe GetValue.

Codice da completare:

```
import static java.util.Objects.requireNonNull;

public class Leaf implements BDT {
    private final boolean value;
    public Leaf(boolean value) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}
```

```
public class InnerNode implements BDT {
    private final BDT left, right; // invariant: left,right != null
    public InnerNode(BDT left, BDT right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}
```

```
public interface Visitor<T> {
    T visitLeaf(boolean value);
    T visitInnerNode(BDT left, BDT right);
}
```

```
public interface Path {
    int size(); // lunghezza del cammino
    String get(int index); // elemento del cammino a livello index
}
```

```
public class GetValue implements Visitor<Boolean> {
    private final Path path; // invariant: path != null
    private int level = 0; // la visita inizia dal livello 0, ossia dalla radice
    public GetValue(Path path) { /* completare */ }
    public Boolean visitLeaf(boolean value) { /* completare */ }
    public Boolean visitInnerNode(BDT left, BDT right) { /* completare */ }
}
```

```
public class ClassEntity implements JavaEntity {
```

```
    public final String name;
```

```
    public final JavaEntity[] entities;
```

```
    public ClassEntity(String name, JavaEntity... entities) {
```

```
        this.name = requireNonNull(name);
```

```
        this.entities = requireNonNull(entities);
```

```
}
```

## @Override

```
public <T> T accept(Visitor<T> v) {
    return v.visitClassEntity(name, entities);
}
```

```
public String getName() { return name; }
```

```
}
```

```
public class SuperClassOf implements Visitor<Boolean> {
```

```
    private final ClassEntity classEntity;
```

```
    public SuperClassOf(ClassEntity classEntity) {
        this.classEntity = requireNonNull(classEntity);
    }
```

## @Override

```
public Boolean visitClassEntity(String name, JavaEntity... entities) {
    if (name.equals(classEntity.getName()))
        return true;
```

```
for (var e : entities) {
    if (e.accept(this))
        return true;
}
return false;
}
```

## @Override

```
public Boolean visitInstanceEntity() {
    return false;
}
```

```
}
```

```
public class Leaf implements BDT {
```

```
    private final boolean value;
```

```
    public Leaf(boolean value) {
        this.value = value;
    }
```

## @Override

```
public <T> T accept(Visitor<T> v) {
    return v.visitLeaf(value);
}
```

```
public class InnerNode implements BDT {
```

```
    private final BDT left, right;
```

```
    public InnerNode(BDT left, BDT right) {
        this.left = requireNonNull(left);
        this.right = requireNonNull(right);
    }
```

## @Override

```
public <T> T accept(Visitor<T> v) {
    return v.visitInnerNode(left, right);
}
```

```
public class GetValue implements Visitor<Boolean> {
```

```
    private final Path path;
```

```
    private int level = 0;
```

```
    public GetValue(Path path) {
        this.path = requireNonNull(path);
    }
```

## @Override

```
public Boolean visitLeaf(boolean value) {
    if (level < path.size())
        throw new IllegalStateException();
    return value;
}
```

## @Override

```
public Boolean visitInnerNode(BDT left, BDT right) {
    if (level >= path.size())
        throw new IllegalStateException();
    BDT tree;
```

```
    if (path.get(level).equals("L"))
        tree = left;
    else if (path.get(level).equals("R"))
        tree = right;
    else
        throw new IllegalStateException();
    level++;
    return tree.accept(this);
}
```

# Settembre 2022

4. Considerare il codice che implementa le seguenti funzionalità:

- oggetti che implementano l'interfaccia `FileSystemNode` e che sono nodi della struttura ad albero di un file system che possono essere di tipo `File`, con attributo `permission`, oppure `Directory`, con attributo `children`; i permessi associati ai file possono essere solamente uno tra i seguenti: `READ`, `WRITE`, `READ_WRITE`, `EXECUTE`.
- visitor `FindWritableFiles` che contano i file che hanno permesso `WRITE` o `READ_WRITE` a partire da un certo nodo, inclusi tutti i discendenti (se il nodo è di tipo `Directory`).

Esempio:

```
var subdir = new Directory(new File(READ), new File(READ_WRITE)); // directory con 2 file
var exec_file = new File(EXECUTE); // file eseguibile
var write_file = new File(WRITE); // file scrivibile
var dir = // directory con 3 file e una sotto-directory
new Directory(exec_file, subdir, write_file, new File(WRITE));
var findWritable = new FindWritableFiles();
assert dir.accept(findWritable) == 3; // dir e subdir contengono 3 file scrivibili
assert subdir.accept(findWritable) == 1; // subdir contiene 1 file scrivibile
assert write_file.accept(findWritable) == 1; // write_file e' scrivibile
assert exec_file.accept(findWritable) == 0; // exec_file non e' scrivibile
```

- Completare il costruttore e il metodo della classe `File`.
- Completare il costruttore e il metodo della classe `Directory`; il costruttore inizializza il campo `this.children` con un nuovo array che contiene tutti gli elementi specificati dal parametro `children`.
- Completare i metodi della classe `FindWritableFiles`.

Codice da completare:

```
package exam2022_09_07.visitor;

public enum Permission {READ, WRITE, READ_WRITE, EXECUTE; }

import static java.util.Objects.requireNonNull;

public class File implements FileSystemNode {
    private Permission permission; // invariant permission!=null
    public File(Permission permission) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class Directory implements FileSystemNode {
    private FileSystemNode[] children;
    public Directory(FileSystemNode... children) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public interface Visitor<T> {
    T visitFile(Permission permission);
    T visitDirectory(FileSystemNode... children);
}

import static exam2022_09_07.visitor.Permission.*;

public class FindWritableFiles implements Visitor<Integer> {
    public Integer visitFile(Permission permission) { /* completare */ }
    public Integer visitDirectory(FileSystemNode... children) { /* completare */ }
}
```

public class FindWritableFiles implements Visitor<Integer> {

```
    @Override
    public Integer visitFile(Permission permission) {
        return permission == WRITE || permission == READ_WRITE ? 1 : 0;
    }
```

@Override

public Integer visitDirectory(FileSystemNode... children) {

```
        var res = 0;
        for (var node : children)
            res += node.accept(this);
        return res;
    }
```

}

```
public class File implements FileSystemNode {
    private Permission permission; // invariant permission!=null
}

public File(Permission permission) {
    this.permission = requireNonNull(permission);
}

@Override
public <T> T accept(Visitor<T> v) {
    return v.visitFile(permission);
}

}

public class Directory implements FileSystemNode {
    private FileSystemNode[] children;
}

public Directory(FileSystemNode... children) {
    var size = children.length;
    this.children = new FileSystemNode[size];
    for (var i = 0; i < size; i++) {
        this.children[i] = children[i];
    }
}

@Override
public <T> T accept(Visitor<T> v) {
    return v.visitDirectory(children);
}
```

# gennaio 2022

3. Completare il seguente codice che implementa

- gli alberi della sintassi astratta AST composti da nodi corrispondenti ai literal di tipo booleano (BoolLit) e all'operatore di congiunzione (And);
- il visitor Eval che valuta l'espressione nel modo convenzionale restituendo un booleano come risultato.

Esempio:

```
var exp = new And(new BoolLit(true), new BoolLit(true));
var v = new Eval();
assert exp.accept(v); // true && true si valuta in true
exp = new And(new BoolLit(true), new BoolLit(false));
assert !exp.accept(v); // true && false si valuta in false
```

Codice da completare:

```
import static java.util.Objects.requireNonNull;
public interface AST { <T> T accept(Visitor<T> v); }

public interface Visitor<T> {
    T visitBoolLit(boolean b);
    T visitAnd(AST left, AST right);
}

public class BoolLit implements AST {
    private final boolean b;
    public BoolLit(boolean b) {
        // completare
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        // completare
    }
}

public class And implements AST {
    // invariant left != null && right !=null
    private final AST left;
    private final AST right;
    public And(AST left, AST right) {
        // completare
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        // completare
    }
}

public class Eval implements Visitor<Boolean> {
    @Override
    public Boolean visitBoolLit(boolean b) {
        // completare
    }
    @Override
    public Boolean visitAnd(AST left, AST right) {
        // completare
    }
}
```

# settembre 2021

3. Completare il seguente codice che implementa

- gli alberi FSTree che rappresentano la struttura gerarchica di un file system con nodi di tipo File e attributo size (la dimensione del file) e nodi di tipo Folder e attributo children (la lista dei file e dei sotto-folder contenuti nel folder);
- il visitor FilesGreater che conta i file di dimensione maggiore dell'attributo minSize che corrispondono al nodo visitato (se di tipo File) o a tutti i suoi discendenti (se di tipo Folder).

Esempio:

```
var dir = new Folder(new File(10), new Folder(new File(2), new File(21)), new File(5), new File(42));
assert dir.accept(new FilesGreater(0)) == 5; // dir e il suo sotto-folder contengono 5 file con size>0
assert dir.accept(new FilesGreater(20)) == 2; // dir e il suo sotto-folder contengono 2 file con size>20
assert dir.accept(new FilesGreater(42)) == 0; // dir e il suo sotto-folder contengono 0 file con size>42
var f = new File(35);
assert f.accept(new FilesGreater(30)) == 1; // f ha size>30
assert f.accept(new FilesGreater(40)) == 0; // f non ha size>40
```

Completa costruttori e metodi delle classi File, Folder e FilesGreater:

```
import java.util.List;

public interface Visitor<T> {
    T visitFile(int size);
    T visitFolder(List<FSTree> children);
}

public interface FSTree {
    <T> T accept(Visitor<T> v);
}

public class File implements FSTree {
    private int size; // invariant size >= 0
    public File(int size) { /* completare */ }
    @Override public <T> T accept(Visitor<T> v) { /* completare */ }
}

import java.util.List;
import java.util.LinkedList;

public class Folder implements FSTree {
    private final List<FSTree> children = new LinkedList<>();
    public Folder(FSTree... children) { /* completare */ }
    @Override public <T> T accept(Visitor<T> v) { /* completare */ }
}

import java.util.List;

public class FilesGreater implements Visitor<Integer> {
    private final int minSize;
    public FilesGreater(int minSize) { /* completare */ }
    @Override public Integer visitFile(int size) {
        if(size > minSize) return 1;
        else return 0;
    }
    @Override public Integer visitFolder(List<FSTree> children) {
        int res = 0;
        for(FSTree child : children) res += child.accept(this);
        return res;
    }
}
```

```
public class BoolLit implements AST {
    private boolean b;
    public BoolLit(boolean b) {
        this.b = b;
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitBoolLit(b);
    }
}
```

```
public class And implements AST {
    // invariant left != null && right !=null
    private final AST left;
    private final AST right;
    public And(AST left, AST right) {
        this.left = requireNonNull(left);
        this.right = requireNonNull(right);
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitAnd(left, right);
    }
}
```

```
public class Eval implements Visitor<Boolean> {
    @Override
    public Boolean visitBoolLit(boolean b) {
        return b;
    }
    @Override
    public Boolean visitAnd(AST left, AST right) {
        return left.accept(this) && right.accept(this);
    }
}
```

```
public class Folder implements FSTree {
    private final List<FSTree> children = new LinkedList<>();
}
```

```
public Folder(FSTree... children) {
    for(var node : children)
        this.children.add(requireNonNull(node));
}
```

```
@Override
public <T> T accept(Visitor<T> v) {
    return v.visitFolder(children);
}
```

```
// conta tutti i file con size > minSize
public class FilesGreater implements Visitor<Integer> {
    private final int minSize;
    public FilesGreater(int minSize) { /* completare */ }
    @Override public Integer visitFile(int size) {
        if(size > minSize) return 1;
        else return 0;
    }
}
```

```
public FilesGreater(int minSize) {
    this.minSize = minSize;
}
```

```
@Override
public Integer visitFolder(List<FSTree> children) {
    int res = 0;
    for(FSTree child : children)
        res += child.accept(this);
    return res;
}
```

```
@Override
public Integer visitFile(int size) {
    if(size > minSize) return 1;
    else return 0;
}
```

# Luglio 2021

3. Completare il seguente codice che implementa gli alberi di ricerca binari con nodi etichettati da numeri interi (con la relazione d'ordine usuale) e la ricerca di un elemento nell'albero tramite visitor pattern.

Esempio:

```
var t = new Node(10,new Leaf(5),new Node(42,new Leaf(31),null));
assert t.accept(new Search(31)); // 31 è un'etichetta contenuta nell'albero t
assert !t.accept(new Search(43)); // 43 non è un'etichetta contenuta nell'albero t
assert !t.accept(new Search(30)); // 30 non è un'etichetta contenuta nell'albero t
```

**Definizione di albero binario di ricerca:** Un albero binario di ricerca con nodi etichettati da numeri interi soddisfa le seguenti proprietà: ogni nodo  $v$  è etichettato da un numero intero  $i$ , le etichette dei nodi del sottoalbero sinistro di  $v$  sono  $\leq i$ , le etichette dei nodi del sottoalbero destro di  $v$  sono  $\geq i$ .

Completare costruttori e metodi delle classi Leaf, Node e Search:

```
public interface Visitor<T> {
    T visitLeaf(int value);
    T visitNode(int value, BSTree left, BSTree right);
}

public abstract class BSTree {
    protected final int value;
    protected BSTree(int value) { this.value = value; }
    public abstract <T> T accept(Visitor<T> v);
}

public class Leaf extends BSTree { // nodi foglia
    public Leaf(int value) { /* completare */ }
    @Override public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class Node extends BSTree { // nodi interni
    private final BSTree left, right; // left, right possono contenere null, ma non entrambi
    public Node(int value, BSTree left, BSTree right) { /* completare */ }
    @Override public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class Search implements Visitor<Boolean> { // ricerca l'elemento searchedValue nell'albero
    private final int searchedValue; // elemento da cercare
    public Search(int searchedValue) { /* completare */ }
    @Override public Boolean visitLeaf(int value) { /* completare */ }
    @Override public Boolean visitNode(int value, BSTree left, BSTree right) { /* completare */ }
}
```

```
public class Node extends BSTree {
    private final BSTree left, right;

    public Node(int value, BSTree left, BSTree right) {
        super(value);
        if(left==null && right==null) throw new NullPointerException();
        this.left = left;
        this.right = right;
    }

    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitNode(value, left, right);
    }
}
```

```
public class Search implements Visitor<Boolean> {
    private final int searchedValue;

    public Search(int searchedValue) {
        this.searchedValue = searchedValue;
    }

    @Override
    public Boolean visitLeaf(int value) {
        return value == searchedValue;
    }

    @Override
    public Boolean visitNode(int value, BSTree left, BSTree right) {
        if (searchedValue < value)
            return left != null && left.accept(this);
        else
            return value == searchedValue || right != null && right.accept(this);
    }
}
```

```
public class Leaf extends BSTree {
    public Leaf(int value) {
        super(value);
    }

    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitLeaf(value);
    }
}
```

# Settembre 2019

3. Sia dato il seguente programma per la valutazione di espressioni formate da literal di tipo intero e tipo stringa, e dall'operatore binario di prodotto tra una stringa e un intero:

```

public interface Exp { <T> T accept(Visitor<T> visitor); }
public interface Visitor<T> { ... }
public interface Value { ... }
public abstract class PrimVal<T> implements Value { ... }
public class IntVal extends PrimVal<Integer> { ... }
public class StringVal extends PrimVal<String> { ... }
public class Eval implements Visitor<Value> { ... }

(a) Completare le seguenti classi che implementano i valori di tipo intero e stringa.

public interface Value {
    public default String asString() { throw new RuntimeException("Expecting a string"); }
    public default int asInt() { throw new RuntimeException("Expecting an integer"); }
}
// valori primitivi generici
public abstract class PrimVal<T> implements Value {
    protected T val;
    // invarianti di classe: val!=null
    protected PrimVal(T val) { /* completare */ }
}
public class IntVal extends PrimVal<Integer> {
    protected IntVal(Integer val) { /* completare */ }
    @Override public int asInt() { /* completare */ }
}
public class StringVal extends PrimVal<String> {
    protected StringVal(String val) { /* completare */ }
    @Override public String asString() { /* completare */ }
}

```

- (b) Completare la classe `Eval` per la valutazione delle espressioni. Nel metodo `visitTimes(Exp left, Exp right)` (prodotto tra stringhe e interi) l'operando `left` si deve valutare in una stringa `s` e `right` in un intero `i`; il risultato calcolato è la concatenazione di `s` ripetuta `i` volte. Viene sollevata `RuntimeException` se i valori degli operandi non sono del tipo giusto e `IllegalArgumentException` se `i < 0`.

```

public class Eval implements Visitor<Value> {
    @Override public Value visitIntLit(int val) { /* completare */ }
    @Override public Value visitStringLit(String val) { /* completare */ }
    @Override public Value visitTimes(Exp left, Exp right) { /* completare */ }
}

Suggerimento: per il calcolo della concatenazione, utilizzare il seguente metodo della classe predefinita String:
```

```

// Returns a string whose value is the concatenation of this string repeated count times.
public String repeat(int count)

```

# Luglio 2019

3. (a) Completare le seguenti classi che implementano i nodi di un AST per espressioni con literal interi positivi e operazione di elevamento a potenza.

```

public interface Visitor<T> {
    T visitNatLit(int val);
    T visitPow(Exp left, Exp right);
}
public interface Exp { <T> T accept(Visitor<T> visitor); }

// nodi AST per literal interi positivi
public class PosLit implements Exp {
    private final int val;
    // precondizione: val > 0
    public PosLit(int val) { /* completare */ }
    public <T> T accept(Visitor<T> visitor) { /* completare */ }
}

// nodi AST per elevamento a potenza
public class Pow implements Exp {
    private final Exp left; // base
    private final Exp right; // potenza
    // precondizione: left!=null, right!=null
    public Pow(Exp left, Exp right) { /* completare */ }
    public <T> T accept(Visitor<T> visitor) { /* completare */ }
}

```

- (b) Completare la classe `Eval` per la valutazione degli AST; la classe `Test` contiene una prova esplicativa.

```

public class Eval implements Visitor<Integer> {
    public Integer visitNatLit(int val) { /* completare */ }
    public Integer visitPow(Exp left, Exp right) { /* completare */ }
}

public class Test {
    public static void main(String[] args) {
        Exp e = new Pow(new PosLit(3), new PosLit(4)); // AST per 3 elevato 4
        assert e.accept(new Eval()) == 81;
    }
}

public class Eval implements Visitor<Integer> {

    @Override
    public Integer visitNatLit(int val) {
        return val;
    }

    @Override
    public Integer visitPow(Exp left, Exp right) {
        int base = left.accept(this);
        int exp = right.accept(this);
        int res;
        for (res = 1; exp > 0; exp--)
            res *= base;
        return res;
    }
}

```

```

    // valori primitivi generici
    public abstract class PrimVal<T> implements Value {
        protected T val;
        // invarianti di classe: val!=null
        protected PrimVal(T val) {
            this.val = requireNonNull(val);
        }
    }
    public class IntVal extends PrimVal<Integer> {
        protected IntVal(Integer val) {
            super(val);
        }
        @Override
        public int asInt() {
            return val;
        }
    }
    public class StringVal extends PrimVal<String> {
        protected StringVal(String val) {
            super(val);
        }
        @Override
        public String asString() {
            return val;
        }
    }
    public class Eval implements Visitor<Value> {
        @Override
        public Value visitIntLit(int val) {
            return new IntVal(val);
        }
        @Override
        public Value visitStringLit(String val) {
            return new StringVal(val);
        }
        @Override
        public Value visitTimes(Exp left, Exp right) {
            // left si deve valutare in una stringa
            // right si deve valutare in un intero
            String s = left.accept(this).asString();
            int i = right.accept(this).asInt();
            return new StringVal(s.repeat(i));
        }
    }
}

```

```
// nodi AST per elevamento a potenza
```

```
public class Pow implements Exp {
    private final Exp left; // base
    private final Exp right; // potenza
    // precondizione: left!=null, right!=null
    public Pow(Exp left, Exp right) {
        this.left = requireNonNull(left);
        this.right = requireNonNull(right);
    }
    @Override
    public <T> T accept(Visitor<T> visitor) {
        return visitor.visitPow(left, right);
    }
}
```

```
// nodi AST per literal interi positivi
```

```
public class PosLit implements Exp {
    private final int val;
    // precondizione: val > 0
    public PosLit(int val) {
        if (val <= 0)
            throw new IllegalArgumentException("Literal must be positive");
        this.val = val;
    }
    @Override
    public <T> T accept(Visitor<T> visitor) {
        return visitor.visitNatLit(val);
    }
}
```

# gennaio 2019

3. (a) Completare le classi `BoolLit` e `And` che rappresentano i nodi di un albero della sintassi astratta corrispondenti, rispettivamente, a literal booleani e all'and logico.

```
public interface AST { <T> T accept(Visitor<T> v); }

public interface Visitor<T> {
    T visitBoolLit(boolean b);
    T visitAnd(AST left, AST right);
}

public class BoolLit implements AST {
    private final boolean value;
    public BoolLit(boolean value) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class And implements AST {
    private final AST left, right;
    public And(AST left, AST right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}
```

- (b) Completare le classi `Eval` e `ToString` che implementano visitor su oggetti di tipo `AST`.

```
/* Un visitor Eval restituisce il valore dell'espressione,
calcolato secondo le regole convenzionali;
la valutazione dell'and logico e' con short-circuit */
public class Eval implements Visitor<Boolean> {
    public Boolean visitBoolLit(boolean b) { /* completare */ }
    public Boolean visitAnd(AST left, AST right) { /* completare */ }
}

/* Un visitor ToString restituisce la rappresentazione in
notazione polacca postfissa dell'espressione;
usare String.valueOf per la conversione da boolean a String */
public class ToString implements Visitor<String> {
    public String visitBoolLit(boolean b) { /* completare */ }
    public String visitAnd(AST left, AST right) { /* completare */ }
}

// Classe di prova
public class Test {
    public static void main(String[] args) {
        AST b1 = new BoolLit(true), b2 = new BoolLit(true), b3 = new BoolLit(false);
        AST b1_b2_and_b3_and = new And(new And(b1, b2), b3);
        assert !b1_b2_and_b3_and.accept(new Eval());
        assert b1_b2_and_b3_and.accept(new ToString()).equals("true true && false &&");
    }
}
```

# settembre 2018

3. (a) Completare le classi `IntLit` e `Add` che rappresentano i nodi di un albero della sintassi astratta corrispondenti, rispettivamente, a literal interi e all'operazione aritmetica di addizione.

```
public interface AST { <T> T accept(Visitor<T> v); }

public interface Visitor<T> {
    T visitIntLit(int i);
    T visitAdd(AST left, AST right);
}

public class IntLit implements AST {
    private final int value;
    public IntLit(int value) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class Add implements AST {
    private final AST left, right;
    public Add(AST left, AST right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}
```

- (b) Completare le classi `Eval` e `ToString` che implementano visitor su oggetti di tipo `AST`.

```
/* Un visitor Eval restituisce il valore dell'espressione visitata,
calcolato secondo le regole convenzionali */
public class Eval implements Visitor<Integer> {
    public Integer visitIntLit(int i) { /* completare */ }
    public Integer visitAdd(AST left, AST right) { /* completare */ }
}

/* Un visitor ToString restituisce la stringa che rappresenta l'espressione visitata
secondo la sintassi convenzionale senza parentesi */
public class ToString implements Visitor<String> {
    public String visitIntLit(int i) { /* completare */ }
    public String visitAdd(AST left, AST right) { /* completare */ }
}

// Classe di prova
public class Test {
    public static void main(String[] args) {
        AST i1 = new IntLit(1), i2 = new IntLit(2), i3 = new IntLit(3);
        AST i1_plus_i2_plus_i3 = new Add(new Add(i1, i2), i3);
        assert i1_plus_i2_plus_i3.accept(new Eval()) == 6;
        assert i1_plus_i2_plus_i3.accept(new ToString()).equals("1+2+3");
    }
}
```

```
public class BoolLit implements AST {
    private final boolean value;
    public BoolLit(boolean value) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class And implements AST {
    private final AST left, right;
    public And(AST left, AST right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class ToString implements Visitor<String> {
    public String visitBoolLit(boolean b) { /* completare */ }
    public String visitAnd(AST left, AST right) { /* completare */ }
}

public class Eval implements Visitor<Boolean> {
    public Boolean visitBoolLit(boolean b) { /* completare */ }
    public Boolean visitAnd(AST left, AST right) { /* completare */ }
}

public class IntLit implements AST {
    private final int value;
    public IntLit(int value) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class Add implements AST {
    private final AST left, right;
    public Add(AST left, AST right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class ToString implements Visitor<String> {
    public String visitIntLit(int i) { /* completare */ }
    public String visitAdd(AST left, AST right) { /* completare */ }
}

public class Eval implements Visitor<Integer> {
    public Integer visitIntLit(int i) { /* completare */ }
    public Integer visitAdd(AST left, AST right) { /* completare */ }
}
```

# luglio 2017

3. Considerare la seguente implementazione degli alberi della sintassi astratta (AST) di un semplice linguaggio di espressioni formate a partire da literal di tipo stringa e dagli operatori di addizione intera e di calcolo della lunghezza di una stringa:

```
public interface Visitor<T> {
    T visitStringLit(String value);
    T visitLength(Exp exp);
    T visitAdd(Exp left, Exp right);
}

public class StringLitExp implements Exp {
    private final String value;
    public StringLitExp(String value) { /* da completare */ }
    public <T> T accept(Visitor<T> v) { /* da completare */ }
}

public class LengthExp implements Exp {
    private final Exp exp;
    public LengthExp(Exp exp) { /* da completare */ }
    public <T> T accept(Visitor<T> v) { /* da completare */ }
}

public class AddExp implements Exp {
    private final Exp left, right;
    public AddExp(Exp left, Exp right) { /* da completare */ }
    public <T> T accept(Visitor<T> v) { /* da completare */ }
}
```

- (a) Completare le definizioni dei costruttori di tutte le classi.
- (b) Completare le definizioni dei metodi `accept` delle classi `StringLitExp`, `LengthExp` e `AddExp`.
- (c) Completare la classe `Typecheck`, i cui oggetti permettono di effettuare il typechecking (secondo la semantica statica convenzionale) dell'espressione rappresentata dall'AST visitato.

Esempio:

```
Exp exp = new AddExp(new LengthExp(new StringLitExp("abc")), new LengthExp(new StringLitExp("de")));
assert exp.accept(new Typecheck()) == INT;
```

Definizioni:

```
public enum Type { INT, STRING }
```

```
public class Typecheck implements Visitor<Type> {
    private static Type check(Type expected, Type found) {
        if (expected != found)
            throw new RuntimeException("Expected " + expected + ", found " + found);
        return expected;
    }

    public Type visitStringLit(String value) { /* da completare */ }
    public Type visitLength(Exp exp) { /* da completare */ }
    public Type visitAdd(Exp left, Exp right) { /* da completare */ }
}
```

- (d) Completare la classe `Eval`, i cui oggetti permettono di valutare l'espressione rappresentata dall'AST visitato secondo la semantica dinamica convenzionale.

Esempio:

```
Exp exp = new AddExp(new LengthExp(new StringLitExp("abc")), new LengthExp(new StringLitExp("de")));
assert exp.accept(new Eval()).equals(new IntValue(5));
```

Definizioni:

```
public interface Value {
    default int asInt() { throw new ClassCastException(); }
    default String asString() { throw new ClassCastException(); }
}

import static java.util.Objects.requireNonNull;
public abstract class PrimValue<T> implements Value {
    final protected T value;
    protected PrimValue(T value) { this.value = requireNonNull(value); }
    public int asInt() { return (int) value; }
    public String asString() { return (String) value; }
    public int hashCode() { return value.hashCode(); }
}

public class IntValue extends PrimValue<Integer> {
    protected IntValue(int value) { super(value); }
    public boolean equals(Object obj) {
        return this == obj || obj instanceof IntValue && value.equals(((IntValue) obj).value);
    }
}

public class StringValue extends PrimValue<String> {
    protected StringValue(String value) { super(value); }

    public boolean equals(Object obj) {
        return this == obj || obj instanceof StringValue && value.equals(((StringValue) obj).value);
    }
}

public class Eval implements Visitor<Value> {
    public Value visitStringLit(String value) { /* da completare */ }
    public Value visitLength(Exp exp) { /* da completare */ }
    public Value visitAdd(Exp left, Exp right) { /* da completare */ }
}
```

```
public class StringLitExp implements Exp {
    private final String value;
    public StringLitExp(String value) {
        this.value = requireNonNull(value);
    }

    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitStringLit(value);
    }
}

public class LengthExp implements Exp {
    private final Exp exp;
    public LengthExp(Exp exp) {
        this.exp = requireNonNull(exp);
    }

    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitLength(exp);
    }
}

public class AddExp implements Exp {
    private final Exp left, right;
    public AddExp(Exp left, Exp right) {
        this.left = requireNonNull(left);
        this.right = requireNonNull(right);
    }

    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitAdd(left, right);
    }
}

public class Typecheck implements Visitor<Type> {
    private static Type check(Type expected, Type found) {
        if (expected != found)
            throw new RuntimeException("Expected " + expected + ", found " + found);
        return expected;
    }

    @Override
    public Type visitStringLit(String value) {
        return STRING;
    }

    @Override
    public Type visitLength(Exp exp) {
        check(STRING, exp.accept(this));
        return INT;
    }

    @Override
    public Type visitAdd(Exp left, Exp right) {
        check(INT, left.accept(this));
        return check(INT, right.accept(this));
    }
}

public class Eval implements Visitor<Value> {
    @Override
    public Value visitStringLit(String value) {
        return new StringValue(value);
    }

    @Override
    public Value visitLength(Exp exp) {
        return new IntValue(exp.accept(this).asString().length());
    }

    @Override
    public Value visitAdd(Exp left, Exp right) {
        return new IntValue(left.accept(this).asInt() + right.accept(this).asInt());
    }
}
```

# giugno 2017

3. Considerare la seguente implementazione degli alberi della sintassi astratta (AST) di un semplice linguaggio di espressioni booleane formate a partire dagli operatori standard (and, or e not), dai literal booleani e dalle variabili.

```
public interface Exp { <T> T accept(Visitor<T> visitor); }

public interface Visitor<T> {
    T visitLit(boolean value);
    T visitVar(String name);
    T visitNot(Exp exp);
    T visitAnd(Exp left, Exp right);
    T visitOr(Exp left, Exp right);
}

public abstract class BinOp implements Exp {
    final protected Exp left, right;
    protected BinOp(Exp left, Exp right) { /* completare */ }
}

public class LitExp implements Exp {
    private final boolean value;
    public LitExp(boolean value) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class VarExp implements Exp {
    private final String name;
    public VarExp(String name) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class NotExp implements Exp {
    private final Exp exp;
    public NotExp(Exp exp) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class OrExp extends BinOp {
    public OrExp(Exp left, Exp right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class AndExp extends BinOp {
    public AndExp(Exp left, Exp right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}
```

(a) Completare le definizioni dei costruttori di tutte le classi.

(b) Completare le definizioni dei metodi accept delle classi LitExp, VarExp, NotExp, OrExp e AndExp.

(c) Completare la classe Display, i cui visitor restituiscono la stringa corrispondente alla sintassi concreta dell'espressione rappresentata dall'AST visitato, usando le convenzioni usuali: operatori binari infissi `&&` e `||` con parentesi tonde per evitare problemi di precedenza tra operatori, operatore unario `:` prefisso, nessuno spazio tra i vari lessimi.

Esempio:

```
Exp e = new OrExp(new AndExp(new LitExp(true), new VarExp("x")),
                 new NotExp(new AndExp(new VarExp("y"), new VarExp("z"))));
System.out.println(e.accept(new Display())); // stampa ((true&&x)||!(y&z))

public class Display implements Visitor<String> {
    public String visitLit(boolean value) { /* completare */ }
    public String visitVar(String name) { /* completare */ }
    public String visitNot(Exp exp) { /* completare */ }
    public String visitAnd(Exp left, Exp right) { /* completare */ }
    public String visitOr(Exp left, Exp right) { /* completare */ }
}
```

(d) Completare la classe Subst, i cui visitor costruiscono un nuovo AST ottenuto da quello visitato rimpiazzando le occorrenze dei nodi variabile identificati da name con il nodo literal che rappresenta il valore value.

Esempio:

```
Exp e = new OrExp(new AndExp(new LitExp(true), new VarExp("x")),
                 new NotExp(new AndExp(new VarExp("y"), new VarExp("z"))));
e = e.accept(new Subst("x", false));
System.out.println(e.accept(new Display())); // stampa ((true&&false)||!(y&false))

public class Subst implements Visitor<Exp> {
    private final String name;
    private final boolean value;
    public Subst(String name, boolean value) { /* completare */ }
    public Exp visitLit(boolean value) { /* completare */ }
    public Exp visitVar(String name) { /* completare */ }
    public Exp visitNot(Exp exp) { /* completare */ }
    public Exp visitAnd(Exp left, Exp right) { /* completare */ }
    public Exp visitOr(Exp left, Exp right) { /* completare */ }
}
```

```
public class Subst implements Visitor<Exp> {
    private final String name;
    private final boolean value;

    public Subst(String name, boolean value) {
        this.name = requireNonNull(name);
        this.value = value;
    }

    @Override
    public Exp visitLit(boolean value) {
        return new LitExp(value);
    }

    @Override
    public Exp visitVar(String name) {
        if (this.name.equals(name))
            return new LitExp(value);
        return new VarExp(name);
    }

    @Override
    public Exp visitNot(Exp exp) {
        return new NotExp(exp.accept(this));
    }

    @Override
    public Exp visitAnd(Exp left, Exp right) {
        return new AndExp(left.accept(this), right.accept(this));
    }

    @Override
    public Exp visitOr(Exp left, Exp right) {
        return new OrExp(left.accept(this), right.accept(this));
    }
}
```

```
public class LitExp implements Exp {
    private final boolean value;
}

public abstract class BinOp implements Exp {
    final protected Exp left, right;
}

protected BinOp(Exp left, Exp right) {
    this.left = requireNonNull(left);
    this.right = requireNonNull(right);
}

public class VarExp implements Exp {
    private final String name;
}

public VarExp(String name) {
    if (name.length() == 0)
        throw new IllegalArgumentException();
    this.name = name;
}

@Override
public <T> T accept(Visitor<T> v) {
    return v.visitVar(name);
}

public class NotExp implements Exp {
    private final Exp exp;
}

public NotExp(Exp exp) {
    this.exp = requireNonNull(exp);
}

@Override
public <T> T accept(Visitor<T> v) {
    return v.visitNot(exp);
}

public class OrExp extends BinOp {
    public OrExp(Exp left, Exp right) {
        super(left, right);
    }

    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitOr(left, right);
    }
}

public class AndExp extends BinOp {
    public AndExp(Exp left, Exp right) {
        super(left, right);
    }

    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitAnd(left, right);
    }
}

public class Display implements Visitor<String> {

    @Override
    public String visitLit(boolean value) {
        return String.valueOf(value);
    }

    @Override
    public String visitVar(String name) {
        return name;
    }

    @Override
    public String visitNot(Exp exp) {
        return "!" + exp.accept(this);
    }

    private final String visitBin(Exp left, Exp right, String op) {
        return "(" + left.accept(this) + op + right.accept(this) + ")";
    }

    @Override
    public String visitAnd(Exp left, Exp right) {
        return visitBin(left, right, "&&");
    }

    @Override
    public String visitOr(Exp left, Exp right) {
        return visitBin(left, right, "||");
    }
}
```

# Settembre 2016

3. Considerare la seguente implementazione della sintassi astratta di un semplice linguaggio di espressioni che denotano insiemi di interi e sono formate dall'operatore binario di unione, dall'operatore unario di complementazione, dai literal di tipo insieme e dagli identificatori di variabile.

```

public interface Exp { <T> T accept(Visitor<T> visitor); }

public interface Visitor<T> {
    T visitComplement(Exp exp);
    T visitUnion(Exp left, Exp right);
    T visitVarId(String name);
    T visitSetLit(java.util.Set<Integer> value);
}

public abstract class UnaryOp implements Exp {
    protected final Exp exp;
    protected UnaryOp(Exp exp) { /* completare */ }
}

public abstract class BinaryOp implements Exp {
    protected final Exp left;
    protected final Exp right;
    protected BinaryOp(Exp left, Exp right) { /* completare */ }
}

public abstract class AbstractLit<V> implements Exp {
    protected final V value;
    protected AbstractLit(V value) { /* completare */ }
    public int hashCode() { return value.hashCode(); }
}

public class Union extends BinaryOp {
    public Union(Exp left, Exp right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class Complement extends UnaryOp {
    public Complement(Exp exp) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class SetLit extends AbstractLit<java.util.Set<Integer>> {
    public SetLit(java.util.Set<Integer> value) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof SetLit))
            return false;
        return value.equals(((SetLit) obj).value);
    }
}

public class VarId implements Exp {
    private final String name;
    public VarId(String name) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

```

(a) Completare le definizioni dei costruttori di tutte le classi.

(b) Completare le definizioni dei metodi `accept` delle classi `Union`, `Complement`, `SetLit`, `e VarId`.

(c) Completare la classe `Eval` che valuta un'espressione restituendo un valore booleano secondo le seguenti regole:

- l'operatore di complementazione corrisponde alla negazione logica;
- l'operatore di unione corrisponde alla disgiunzione logica;
- i literal vengono valutati in `true` se e solo se l'insieme rappresentato non è vuoto;
- le variabili si valutano sempre in `true`.

Per esempio, la seguente asserzione ha successo:

```

Exp exp = new Complement(new Union(new SetLit(new java.util.HashSet()), new VarId("x")));
assert !exp.accept(new Eval());
}

public class Eval implements Visitor<Boolean> {
    public Boolean visitComplement(Exp exp) { /* completare */ }
    public Boolean visitUnion(Exp left, Exp right) { /* completare */ }
    public Boolean visitVarId(String name) { /* completare */ }
    public Boolean visitSetLit(java.util.Set<Integer> value) { /* completare */ }
}

```

(d) Completare la classe `ReplaceVar` che costruisce una nuova espressione ottenuta da quella visitata rimpiazzando le variabili con il literal che rappresenta l'insieme vuoto. Per esempio, la seguente asserzione ha successo:

```

Exp exp = new Complement(new Union(new SetLit(new java.util.HashSet()), new VarId("x")));
assert exp.accept(new ReplaceVar()).accept(new Eval());
}

public class ReplaceVar implements Visitor<Exp> {
    private static final Exp emptyLit = new SetLit(new java.util.HashSet());
    public Exp visitComplement(Exp exp) { /* completare */ }
    public Exp visitUnion(Exp left, Exp right) { /* completare */ }
    public Exp visitVarId(String name) { /* completare */ }
    public Exp visitSetLit(java.util.Set<Integer> value) { /* completare */ }
}

```

```

public class Eval implements Visitor<Boolean> {

    @Override
    public Boolean visitComplement(Exp exp) {
        return !exp.accept(this);
    }

    @Override
    public Boolean visitUnion(Exp left, Exp right) {
        return left.accept(this) || right.accept(this);
    }

    @Override
    public Boolean visitVarId(String name) {
        return true;
    }

    @Override
    public Boolean visitSetLit(Set<Integer> value) {
        return !value.isEmpty();
    }
}

```

```

public abstract class UnaryOp implements Exp {
    protected final Exp exp;
    protected UnaryOp(Exp exp) {
        this.exp = requireNonNull(exp);
    }
}

public abstract class AbstractLit<V> implements Exp {
    protected final V value;
    protected AbstractLit(V value) {
        this.value = requireNonNull(value);
    }
    @Override
    public int hashCode() {
        return value.hashCode();
    }
}

public class Union extends BinaryOp {
    public Union(Exp left, Exp right) {
        super(left, right);
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitUnion(left, right);
    }
}

public class Complement extends UnaryOp {
    public Complement(Exp exp) {
        super(exp);
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitComplement(exp);
    }
}

public class SetLit extends AbstractLit<Set<Integer>> {
    public SetLit(Set<Integer> value) {
        super(value);
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitSetLit(value);
    }
    @Override
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof SetLit))
            return false;
        return value.equals(((SetLit) obj).value);
    }
}

public class VarId implements Exp {
    private final String name;
    public VarId(String name) {
        if (name.length() == 0)
            throw new IllegalArgumentException();
        this.name = name;
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitVarId(name);
    }
}

public class ReplaceVar implements Visitor<Exp> {
    private static final Exp emptyLit = new SetLit(new HashSet());
    @Override
    public Exp visitComplement(Exp exp) {
        return new Complement(exp.accept(this));
    }
    @Override
    public Exp visitUnion(Exp left, Exp right) {
        return new Union(left.accept(this), right.accept(this));
    }
    @Override
    public Exp visitVarId(String name) {
        return emptyLit;
    }
    @Override
    public Exp visitSetLit(Set<Integer> value) {
        return new SetLit(value);
    }
}

```

# Luglio 2016

3. Considerare la seguente implementazione della sintassi astratta di un semplice linguaggio di espressioni su stringhe formate dall'operatore binario di concatenazione, dall'operatore unario *reverse*, dai literal di tipo stringa e dagli identificatori di variabile.

```

public interface Exp { <T> T accept(Visitor<T> visitor); }

public interface Visitor<T> {
    T visitReverse(Exp exp);
    T visitConcat(Exp left, Exp right);
    T visitVarId(String name);
    T visitStringLit(String value);
}

public abstract class UnaryOp implements Exp {
    protected final Exp exp;
    protected UnaryOp(Exp exp) { /* completare */ }
    public Exp getLeft() { return exp; }
}

public abstract class BinaryOp implements Exp {
    protected final Exp left;
    protected final Exp right;
    protected BinaryOp(Exp left, Exp right) { /* completare */ }
    public Exp getLeft() { return left; }
    public Exp getRight() { return right; }
}

public abstract class AbstractLit<V> implements Exp {
    protected final V value;
    protected AbstractLit(V value) { /* completare */ }
    public V getValue() { return value; }
    public int hashCode() { return value.hashCode(); }
}

public class Concat extends BinaryOp {
    public Concat(Exp left, Exp right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class Reverse extends UnaryOp {
    public Reverse(Exp exp) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
}

public class StringLit extends AbstractLit<String> {
    public StringLit(String value) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof StringLit))
            return false;
        return value == ((StringLit) obj).value;
    }
}

public class VarId implements Exp {
    private final String name;
    public VarId(String name) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public String getName() { return name; }
}

```

(a) Completare le definizioni dei costruttori di tutte le classi.

(b) Completare le definizioni dei metodi `accept` delle classi `Concat`, `Reverse`, `StringLit`, e `VarId`.

(c) Completare la classe `ContainsVarId` che controlla se un'espressione contiene una data variabile. Per esempio, le seguenti asserzioni hanno successo:

```

Exp exp = new Concat(new StringLit("one"), new Reverse(new VarId("x")));
assert exp.accept(new ContainsVarId(new VarId("x")));
assert !exp.accept(new ContainsVarId(new VarId("y")));
}

public class ContainsVarId implements Visitor<Boolean> {
    private final String varName;
    public ContainsVarId(VarId var) { /* completare */ }
    public Boolean visitReverse(Exp exp) { /* completare */ }
    public Boolean visitConcat(Exp left, Exp right) { /* completare */ }
    public Boolean visitVarId(String name) { /* completare */ }
    public Boolean visitStringLit(String value) { /* completare */ }
}

```

(d) Completare la classe `CountStringLit` che conta quanti literal contiene un'espressione. Per esempio, la seguente asserzione ha successo:

```

Exp exp = new Concat(new StringLit("one"), new Concat(new VarId("x"), new StringLit("one")));
assert exp.accept(new CountStringLit()).equals(2);

public class CountStringLit implements Visitor<Integer> {
    public Integer visitReverse(Exp exp) { /* completare */ }
    public Integer visitConcat(Exp left, Exp right) { /* completare */ }
    public Integer visitVarId(String name) { /* completare */ }
    public Integer visitStringLit(String value) { /* completare */ }
}

```

public class ContainsVarId implements Visitor<Boolean> {

private final String varName;

public ContainsVarId(VarId var) {
 this.varName = var.getName();
 }
}

@Override  
public Boolean visitReverse(Exp exp) {
 return exp.accept(this);
}

@Override  
public Boolean visitConcat(Exp left, Exp right) {
 return left.accept(this) || right.accept(this);
}

@Override  
public Boolean visitVarId(String name) {
 return name.equals(varName);
}

@Override  
public Boolean visitStringLit(String value) {
 return false;
}

```

public abstract class UnaryOp implements Exp {
    protected final Exp exp;
    protected UnaryOp(Exp exp) {
        this.exp = requireNonNull(exp);
    }
    public Exp getLeft() {
        return exp;
    }
}

public abstract class BinaryOp implements Exp {
    protected final Exp left;
    protected final Exp right;
    protected BinaryOp(Exp left, Exp right) {
        this.left = requireNonNull(left);
        this.right = requireNonNull(right);
    }
    public Exp getLeft() {
        return left;
    }
    public Exp getRight() {
        return right;
    }
}

public abstract class AbstractLit<V> implements Exp {
    protected final V value;
    protected AbstractLit(V value) {
        this.value = requireNonNull(value);
    }
    public V getValue() {
        return value;
    }
    @Override
    public int hashCode() {
        return value.hashCode();
    }
}

public class Concat extends BinaryOp {
    public Concat(Exp left, Exp right) {
        super(left, right);
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitConcat(left, right);
    }
}

public class Reverse extends UnaryOp {
    public Reverse(Exp exp) {
        super(exp);
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitReverse(exp);
    }
}

public class StringLit extends AbstractLit<String> {
    public StringLit(String value) {
        super(value);
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitStringLit(value);
    }
    @Override
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof StringLit))
            return false;
        return value == ((StringLit) obj).value;
    }
}

public class VarId implements Exp {
    private final String name;
    public VarId(String name) {
        if (name.length() == 0)
            throw new IllegalArgumentException();
        this.name = name;
    }
    public String getName() {
        return name;
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitVarId(name);
    }
}

```

# giugno 2016

3. Considerare la seguente implementazione della sintassi astratta di un semplice linguaggio di espressioni formate dagli operatori binari di congiunzione ( $\wedge$ ) e disgiunzione ( $\vee$ ) logica, dai literal di tipo booleano e dagli identificatori di variabile.

```

public interface Exp { <T> T accept(Visitor<T> visitor); }

public interface Visitor<T> {
    T visitOr(Exp left, Exp right);
    T visitAnd(Exp left, Exp right);
    T visitVarId(String name);
    T visitBoolLit(boolean value);
}

public abstract class BinaryOp implements Exp {
    protected final Exp left;
    protected final Exp right;
    protected BinaryOp(Exp left, Exp right) { /* completare */ }
    public Exp getLeft() { return left; }
    public Exp getRight() { return right; }
}

public class Or extends BinaryOp {
    public Or(Exp left, Exp right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return left.hashCode() + 31 * right.hashCode(); }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof Or))
            return false;
        Or other = (Or) obj;
        return left.equals(other.left) && right.equals(other.right);
    }
}

public class And extends BinaryOp {
    public And(Exp left, Exp right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return left.hashCode() + 37 * right.hashCode(); }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof And))
            return false;
        And other = (And) obj;
        return left.equals(other.left) && right.equals(other.right);
    }
}

public class BoolLit implements Exp {
    protected final boolean value;
    protected BoolLit(boolean value) { /* completare */ }
    public int getValue() { return value; }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return Boolean.hashCode(value); }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof BoolLit))
            return false;
        return value == ((BoolLit) obj).value;
    }
}

public class VarId implements Exp {
    private final String name;
    public VarId(String name) { /* completare */ }
    public String getName() { return name; }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return name.hashCode(); }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof VarId))
            return false;
        return name.equals(((VarId) obj).name);
    }
}

```

(a) Completare le definizioni dei costruttori di tutte le classi.

(b) Completare le definizioni dei metodi `accept` delle classi `Or`, `And`, `BoolLit`, e `VarId`.

(c) Completare la classe `DisplayPrefix` che permette di visualizzare l'espressione in forma polacca pre-fissa. Per esempio, la seguente asserzione ha successo:

```

assert new And(new BoolLit(true), new Or(new VarId("x"), new VarId("y"))).accept(new DisplayPrefix())
    .equals("//\ true \\\ x \\\ y");
}

public class DisplayPrefix implements Visitor<String> {
    public String visitOr(Exp left, Exp right) { /* completare */ }
    public String visitAnd(Exp left, Exp right) { /* completare */ }
    public String visitVarId(String name) { /* completare */ }
    public String visitBoolLit(boolean value) { /* completare */ }
}

```

(d) Completare la classe `Simplify` che permette di semplificare un'espressione applicando le seguenti identità:  $e \text{false} = \text{false}$   $\vee e, e \wedge \text{true} = e$ . Per esempio, la seguente asserzione ha successo:

```

Exp exp1 = new And(new Or(new VarId("x"), new BoolLit(true)), new Or(new BoolLit(true), new BoolLit(false)));
Exp exp2 = new Or(new VarId("x"), new BoolLit(true));
assert exp1.accept(new Simplify()).equals(exp2);

public class Simplify implements Visitor<Exp> {
    public Exp visitOr(Exp left, Exp right) { /* completare */ }
    public Exp visitAnd(Exp left, Exp right) { /* completare */ }
    public Exp visitVarId(String name) { /* completare */ }
    public Exp visitBoolLit(boolean value) { /* completare */ }
}

```

```

public class DisplayPrefix implements Visitor<String> {

    private String visitBinOp(Exp left, Exp right, String opSym) {
        return opSym + " " + left.accept(this) + " " + right.accept(this);
    }

    @Override
    public String visitOr(Exp left, Exp right) {
        return visitBinOp(left, right, "\\/");
    }

    @Override
    public String visitAnd(Exp left, Exp right) {
        return visitBinOp(left, right, "/\\");
    }

    @Override
    public String visitVarId(String name) {
        return name;
    }

    @Override
    public String visitBoolLit(boolean value) {
        return String.valueOf(value);
    }
}

```

```

public abstract class BinaryOp implements Exp {
    protected final Exp left;
    protected final Exp right;
    protected BinaryOp(Exp left, Exp right) {
        this.left = requireNonNull(left);
        this.right = requireNonNull(right);
    }
    public Exp getLeft() {
        return left;
    }
    public Exp getRight() {
        return right;
    }
}

public class And extends BinaryOp {
    public And(Exp left, Exp right) {
        super(left, right);
    }
    @Override
    public int hashCode() {
        return left.hashCode() + 37 * right.hashCode();
    }
    @Override
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof And))
            return false;
        And other = (And) obj;
        return left.equals(other.left) && right.equals(other.right);
    }
}

public class Or extends BinaryOp {
    public Or(Exp left, Exp right) {
        super(left, right);
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitOr(left, right);
    }
    @Override
    public int hashCode() {
        return left.hashCode() + 31 * right.hashCode();
    }
    @Override
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof Or))
            return false;
        Or other = (Or) obj;
        return left.equals(other.left) && right.equals(other.right);
    }
}

public class BoolLit implements Exp {
    private final String name;
    public VarId(String name) {
        if (name.length() == 0)
            throw new IllegalArgumentException();
        this.name = name;
    }
    public String getName() {
        return name;
    }
    @Override
    public int hashCode() {
        return name.hashCode();
    }
    @Override
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof VarId))
            return false;
        return name.equals(((VarId) obj).name);
    }
}

public class Simplify implements Visitor<Exp> {
    private final static BoolLit OrNeutral = new BoolLit(false);
    private final static BoolLit AndNeutral = new BoolLit(true);

    @Override
    public Exp visitOr(Exp left, Exp right) {
        left = left.accept(this);
        right = right.accept(this);
        if (left.equals(OrNeutral))
            return right;
        if (right.equals(OrNeutral))
            return left;
        return new Or(left, right);
    }

    @Override
    public Exp visitAnd(Exp left, Exp right) {
        left = left.accept(this);
        right = right.accept(this);
        if (left.equals(AndNeutral))
            return right;
        if (right.equals(AndNeutral))
            return left;
        return new And(left, right);
    }

    @Override
    public VarId visitVarId(String name) {
        return new VarId(name);
    }

    @Override
    public BoolLit visitBoolLit(boolean value) {
        return new BoolLit(value);
    }
}

```

# febbraio 2016

3. Considerare la seguente implementazione della sintassi astratta di un semplice linguaggio che include il test di ugualianza, le espressioni condizionali e gli identificatori di variabile.

```
public interface AST { <T> T accept(Visitor<T> visitor); }
public interface Variable extends AST {
    boolean equals(Object obj);
    int hashCode();
}
public interface Visitor<T> {
    T visitEq(AST left, AST right);
    T visitSimpleVar(SimpleVar var);
    T visitIfThenElse(AST exp, AST thenStmt, AST elseStmt);
}
public class Eq implements AST {
    private final AST left; // obbligatorio
    private final AST right; // obbligatorio
    public Eq(AST left, AST right) { /* da completare */ }
    public <T> T accept(Visitor<T> v) { return v.visitEq(left, right); }
}
public class IfThenElse implements AST {
    private final AST exp; // obbligatorio
    private final AST thenStmt; // obbligatorio
    private final AST elseStmt; // opzionale
    public IfThenElse(AST exp, AST thenStmt, AST elseStmt) { /* da completare */ }
    public IfThenElse(AST exp, AST thenStmt) { /* da completare */ }
    public <T> T accept(Visitor<T> visitor) { return visitor.visitIfThenElse(exp, thenStmt, elseStmt); }
}
public class SimpleVar implements Variable {
    private final String name; // obbligatorio, non vuoto
    public SimpleVar(String name) { /* da completare */ }
    public <T> T accept(Visitor<T> visitor) { return visitor.visitSimpleVar(this); }
    public final boolean equals(Object obj) { /* da completare */ }
    public int hashCode() { /* da completare */ }
}
```

(a) Completare le definizioni dei costruttori di tutte le classi.

(b) Completare le definizioni dei metodi `equals(Object obj)` e `hashCode()` della classe `SimpleVar`.

(c) Completare la classe `GetFreeVars` che permette di restituire l'insieme degli identificatori di variabile contenuti in un AST. Per esempio, per l'AST che rappresenta l'espressione `if x==y then z else y`, il risultato della visita è l'insieme `{x,y,z}`.

```
public class GetFreeVars implements Visitor<Set<Variable>> {
    private final Set<Variable> vars = new HashSet<>();
    public Set<Variable> visitEq(AST left, AST right) { /* da completare */ }
    public Set<Variable> visitSimpleVar(SimpleVar var) { /* da completare */ }
    public Set<Variable> visitIfThenElse(AST exp, AST thenStmt, AST elseStmt) { /* da completare */ }
}
```

(d) Completare la classe `ApplySubs` che permette di applicare una sostituzione a un AST. Una sostituzione è una mappa finita da variabili ad AST; per esempio, la sostituzione  $\sigma = \{x \mapsto w, z \mapsto y, w \mapsto s\}$ , sostituisce simultaneamente ogni occorrenza di `x` con `w`, di `z` con `y` e di `w` con `s`, mentre lascia invariate le occorrenze di tutte le variabili diverse da `x`, `z` e `w`. Quindi, l'applicazione di  $\sigma$  all'AST dell'espressione `if x==y then z else y` restituisce l'AST dell'espressione `if w==y then z==y else y`.

```
public interface Subst {
    /* restituisce la sostituzione per var; restituisce var se non esiste sostituzione per var */
    AST apply(Variable var);
    /* aggiorna la sostituzione */
    void update(Variable var, AST value);
}
public class ApplySubs implements Visitor<AST> {
    private final Subst subst; // obbligatorio
    public ApplySubs(Subst subst) {
        this.subst = requireNonNull(subst);
    }
    public AST visitEq(AST left, AST right) { /* da completare */ }
    public AST visitSimpleVar(SimpleVar var) { /* da completare */ }
    public AST visitIfThenElse(AST exp, AST thenStmt, AST elseStmt) { /* da completare */ }
}
```

```
public class GetFreeVars implements Visitor<Set<Variable>> {
    private final Set<Variable> vars = new HashSet<>();
}
```

```
@Override
public Set<Variable> visitEq(AST left, AST right) {
    left.accept(this);
    right.accept(this);
    return vars;
}
```

```
@Override
public Set<Variable> visitSimpleVar(SimpleVar var) {
    vars.add(var);
    return vars;
}
```

```
@Override
public Set<Variable> visitIfThenElse(AST exp, AST thenStmt, AST elseStmt) {
    exp.accept(this);
    thenStmt.accept(this);
    if (elseStmt != null)
        elseStmt.accept(this);
    return vars;
}
```

```
public class ApplySubs implements Visitor<AST> {
    private final Subst subst; // obbligatorio
```

```
        public ApplySubs(Subst subst) {
            this.subst = requireNonNull(subst);
        }
    
```

```
    @Override
    public AST visitEq(AST left, AST right) {
        return new Eq(left.accept(this), right.accept(this));
    }
}
```

```
    @Override
    public AST visitSimpleVar(SimpleVar var) {
        return subst.apply(var);
    }
}
```

```
    @Override
    public AST visitIfThenElse(AST exp, AST thenStmt, AST elseStmt) {
        return new IfThenElse(exp.accept(this), thenStmt.accept(this), elseStmt != null ? elseStmt.accept(this) : null);
    }
}
```

```
public class Eq implements AST {
    private final AST left; // obbligatorio
    private final AST right; // obbligatorio
    public Eq(AST left, AST right) {
        this.left = requireNonNull(left);
        this.right = requireNonNull(right);
    }
}
@Override
public <T> T accept(Visitor<T> v) {
    return v.visitEq(left, right);
}
public class IfThenElse implements AST {
    private final AST exp; // obbligatorio
    private final AST thenStmt; // obbligatorio
    private final AST elseStmt; // opzionale
    public IfThenElse(AST exp, AST thenStmt, AST elseStmt) {
        this.exp = requireNonNull(exp);
        this.thenStmt = requireNonNull(thenStmt);
        this.elseStmt = elseStmt;
    }
}
public IfThenElse(AST exp, AST thenStmt) {
    this(exp, thenStmt, null);
}
@Override
public <T> T accept(Visitor<T> visitor) {
    return visitor.visitIfThenElse(exp, thenStmt, elseStmt);
}
```

```
public class SimpleVar implements Variable {
    private final String name; // obbligatorio, non vuoto
```

```
    public SimpleVar(String name) {
        if (name.length() == 0)
            throw new IllegalArgumentException();
        this.name = name;
    }
}
```

```
@Override
public <T> T accept(Visitor<T> visitor) {
    return visitor.visitSimpleVar(this);
}
```

```
@Override
public final boolean equals(Object obj) {
    if (obj == this)
        return true;
    if (!(obj instanceof SimpleVar))
        return false;
    return name.equals(((SimpleVar) obj).name);
}
```

```
@Override
public int hashCode() {
    return name.hashCode();
}
```

```
@Override
public String toString() {
    return name;
}
```

```
public class SubstClass<T> implements Subst {
    private final Map<Variable, AST> map = new HashMap<>();
}
```

```
@Override
public AST apply(Variable var) {
    AST res = map.get(var);
    return res == null ? var : res;
}
```

```
@Override
public void update(Variable var, AST value) {
    if (value == null)
        map.remove(var);
    else
        map.put(var, value);
}
```

# gennaio 2016

3. (a) Considerare la seguente implementazione della sintassi astratta di un semplice linguaggio di espressioni formate dagli operatori binari di addizione e moltiplicazione, dai literal di tipo intero e dagli identificatori di variabile.

```

public abstract class BinaryOp implements Exp {
    protected final Exp left;
    protected final Exp right;

    protected BinaryOp(Exp left, Exp right) {
        this.left = requireNonNull(left);
        this.right = requireNonNull(right);
    }

    public Exp getLeft() {
        return left;
    }

    public Exp getRight() {
        return right;
    }
}

public class Add extends BinaryOp {
    public Add(Exp left, Exp right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return left.hashCode() + 31 * right.hashCode(); }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof Add))
            return false;
        Add other = (Add) obj;
        return left.equals(other.left) && right.equals(other.right);
    }
}

public class Mul extends BinaryOp {
    public Mul(Exp left, Exp right) { /* completare */ }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return left.hashCode() + 37 * right.hashCode(); }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof Mul))
            return false;
        Mul other = (Mul) obj;
        return left.equals(other.left) && right.equals(other.right);
    }
}

public class IntLiteral implements Exp {
    protected final int value;

    protected IntLiteral(int value) { /* completare */ }
    public int getValue() { return value; }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return value; }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof IntLiteral))
            return false;
        return value == ((IntLiteral) obj).value;
    }
}

public class VarIdent implements Exp {
    private final String name;
    public VarIdent(String name) { /* completare */ }
    public String getName() { return name; }
    public <T> T accept(Visitor<T> v) { /* completare */ }
    public int hashCode() { return name.hashCode(); }
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof VarIdent))
            return false;
        return name.equals(((VarIdent) obj).name);
    }
}

(b) Completare le definizioni dei costruttori di tutte le classi.

(c) Completare le definizioni dei metodi accept delle classi Add, Mul, IntLiteral, e VarIdent.

(d) Completare la classe DisplayPostfix che permette di visualizzare l'espressione in forma polacca post-fissa. Per esempio, la seguente asserzione ha successo:
```

```

assert new Mul(new IntLiteral(42), new Add(new VarIdent("x"), new VarIdent("y"))).
accept(new DisplayPostfix()).equals("42 x y + *").

```

```

public class DisplayPostfix implements Visitor<String> {
    public String visitAdd(Exp left, Exp right) { /* completare */ }
    public String visitMul(Exp left, Exp right) { /* completare */ }
    public String visitVarIdent(String name) { /* completare */ }
    public String visitIntLiteral(int value) { /* completare */ }
}

(c) Completare la classe SimplifyNeutral che permette di semplificare un'espressione applicando le seguenti identità:  $e + 0 = e$ ,  $e * 1 = 1 * e = e$ . Per esempio, la seguente asserzione ha successo:
```

```

Exp exp1 = new Mul(new Add(new VarIdent("x"), new IntLiteral(1)),
new Add(new IntLiteral(1), new IntLiteral(0)));
Exp exp2 = new Add(new VarIdent("x"), new IntLiteral(1));
assert exp1.accept(new SimplifyNeutral()).equals(exp2);

```

```

public class SimplifyNeutral implements Visitor<Exp> {
    public Exp visitAdd(Exp left, Exp right) { /* completare */ }
    public Exp visitMul(Exp left, Exp right) { /* completare */ }
    public Exp visitVarIdent(String name) { /* completare */ }
    public Exp visitIntLiteral(int value) { /* completare */ }
}

public class SimplifyNeutral implements Visitor<Exp> {

    private final static IntLiteral AddNeutral = new IntLiteral(0);
    private final static IntLiteral MulNeutral = new IntLiteral(1);

    @Override
    public Exp visitAdd(Exp left, Exp right) {
        left = left.accept(this);
        right = right.accept(this);
        if (left.equals(AddNeutral))
            return right;
        if (right.equals(AddNeutral))
            return left;
        return new Add(left, right);
    }

    @Override
    public Exp visitMul(Exp left, Exp right) {
        left = left.accept(this);
        right = right.accept(this);
        if (left.equals(MulNeutral))
            return right;
        if (right.equals(MulNeutral))
            return left;
        return new Mul(left, right);
    }

    @Override
    public VarIdent visitVarIdent(String name) {
        return new VarIdent(name);
    }

    @Override
    public IntLiteral visitIntLiteral(int value) {
        return new IntLiteral(value);
    }
}

```

```

public abstract class BinaryOp implements Exp {
    protected final Exp left;
    protected final Exp right;

    protected BinaryOp(Exp left, Exp right) {
        this.left = requireNonNull(left);
        this.right = requireNonNull(right);
    }

    public Exp getLeft() {
        return left;
    }

    public Exp getRight() {
        return right;
    }
}

public class Add extends BinaryOp {
    public Add(Exp left, Exp right) { /* completare */ }
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitAdd(left, right);
    }

    @Override
    public int hashCode() {
        return left.hashCode() + 31 * right.hashCode();
    }

    @Override
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof Add))
            return false;
        Add other = (Add) obj;
        return left.equals(other.left) && right.equals(other.right);
    }
}

public class Mul extends BinaryOp {
    public Mul(Exp left, Exp right) {
        super(left, right);
    }

    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitMul(left, right);
    }

    @Override
    public int hashCode() {
        return left.hashCode() + 37 * right.hashCode();
    }

    @Override
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof Mul))
            return false;
        Mul other = (Mul) obj;
        return left.equals(other.left) && right.equals(other.right);
    }
}

public class IntLiteral implements Exp {
    protected final int value;

    protected IntLiteral(int value) { /* completare */ }
    public int getValue() {
        return value;
    }

    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitIntLiteral(value);
    }

    @Override
    public int hashCode() {
        return value;
    }

    @Override
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof IntLiteral))
            return false;
        return value == ((IntLiteral) obj).value;
    }
}

public class VarIdent implements Exp {
    private final String name;
    public VarIdent(String name) { /* completare */ }
    public String getName() {
        return name;
    }

    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visitVarIdent(name);
    }

    @Override
    public int hashCode() {
        return name.hashCode();
    }

    @Override
    public final boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof VarIdent))
            return false;
        return name.equals(((VarIdent) obj).name);
    }
}

public class DisplayPostfix implements Visitor<String> {
    private String visitBinOp(Exp left, Exp right, String opSym) {
        return left.accept(this) + " " + right.accept(this) + " " + opSym;
    }

    @Override
    public String visitAdd(Exp left, Exp right) {
        return visitBinOp(left, right, "+");
    }

    @Override
    public String visitMul(Exp left, Exp right) {
        return visitBinOp(left, right, "*");
    }

    @Override
    public String visitVarIdent(String name) {
        return name;
    }

    @Override
    public String visitIntLiteral(int value) {
        return String.valueOf(value);
    }
}

```

# febbraio 2014

3. Considerare le dichiarazioni dei seguenti tipi, che modellano AST per le espressioni regolari costruite a partire dai seguenti operatori: stella di Kleene  $*$  (`StarExp`), operatore  $?$  di opzionalità (`OptionalityExp`), operatore  $+$  di ripetizione non vuota (`PlusExp`), operatore di unione  $|$  (`UnionExp`), operatore di concatenazione (`CatExp`), singolo carattere (`SymbolExp`) e stringa vuota (`EmptyStringExp`).

```

public interface Visitor<T> {
    T visit(StarExp e);
    T visit(OptionalityExp e);
    T visit(PlusExp e);
    T visit(UnionExp e);
    T visit(CatExp e);
    T visit(SymbolExp e);
    T visit(EmptyStringExp e);
}

public interface Exp {
    <T> T accept(Visitor<T> v);
    List<Exp> getChildren();
}

public abstract class AbstractExp implements Exp {
    private final List<Exp> children;
    protected AbstractExp(Exp... children) {
        if (children == null)
            throw new IllegalArgumentException();
        this.children = Collections.unmodifiableList(Arrays.asList(children));
    }
    public List<Exp> getChildren() {
        return children;
    }
}

public class PlusExp extends AbstractExp { // plus operator: exp +
    // DA COMPLETARE (1)
}

public class SymbolExp extends AbstractExp { // single string containing just one character
    private final char symbol;
    Public SymbolExp(char symbol) {
        this.symbol = symbol;
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        // DA COMPLETARE (2)
    }
    public char getSymbol() {
        return symbol;
    }
}

public class OptionalityExp extends AbstractExp { // optionality operator: exp ?
    Public OptionalityExp(Exp exp) {
        super(exp);
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        // DA COMPLETARE (3)
    }
}
...

```

- (a) Completare le definizioni delle classi `PlusExp`, `SymbolExp` e `OptionalityExp`.
- (b) Completare la definizione della seguente classe `MatchEmptyString` che implementa visitor per gli oggetti di tipo `Exp`: la visita di un AST deve restituire il valore `true` se e solo ha successo il match tra la stringa vuota e l'espressione regolare rappresentata dall'AST.

```

public class MatchEmptyString implements Visitor<Boolean> {
    // DA COMPLETARE (4)
}

```

- (c) Completare la definizione della seguente classe `ElimOptPlus` che implementa visitor per gli oggetti di tipo `Exp`: la visita di un AST che rappresenta l'espressione  $e$  deve restituire un nuovo AST corrispondente a un' $e$ -espressione equivalente a  $a$ , dove tutte le occorrenze degli operatori  $?$  e  $+$  sono state opportunamente sostituite usando gli operatori  $*$ ,  $*$  e di concatenazione.

Esempio: la visita dell'AST corrispondente a  $(a|b)+c?$  restituisce l'AST dell'espressione  $(a|b)*(c)$ .

```

public class ElimOptPlus implements Visitor<Exp> {
    @Override
    public Exp visit(StarExp exp) {
        // DA COMPLETARE (5)
    }
    @Override
    public Exp visit(OptionalityExp exp) {
        // DA COMPLETARE (6)
    }
    @Override
    public Exp visit(PlusExp exp) {
        // DA COMPLETARE (7)
    }
    @Override
    public Exp visit(SymbolExp exp) {
        // DA COMPLETARE (8)
    }
}
...

```

```

package functionalVisitor;
public class PlusExp extends AbstractExp {
    // plus operator: exp +
    public PlusExp(Exp exp) {
        super(exp);
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visit(this);
    }
}

package functionalVisitor;
public class SymbolExp extends AbstractExp {
    // single string containing just one character
    private final char symbol;
    public SymbolExp(char symbol) {
        this.symbol = symbol;
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visit(this);
    }
    public char getSymbol() {
        return symbol;
    }
}

package functionalVisitor;
public class OptionalityExp extends AbstractExp {
    // optionality operator: exp ?
    public OptionalityExp(Exp exp) {
        super(exp);
    }
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visit(this);
    }
}

public class MatchEmptyString implements Visitor<Boolean> {
    @Override
    public Boolean visit(StarExp exp) {
        return true;
    }
    @Override
    public Boolean visit(OptionalityExp exp) {
        return true;
    }
    @Override
    public Boolean visit(PlusExp exp) {
        return exp.getChildren().get(0).accept(this);
    }
    @Override
    public Boolean visit(UnionExp exp) {
        List<Exp> children = exp.getChildren();
        return children.get(0).accept(this) || children.get(1).accept(this);
    }
    @Override
    public Boolean visit(CatExp exp) {
        List<Exp> children = exp.getChildren();
        return children.get(0).accept(this) && children.get(1).accept(this);
    }
    @Override
    public Boolean visit(SymbolExp exp) {
        return false;
    }
    @Override
    public Boolean visit(EmptyStringExp exp) {
        return true;
    }
}

public class ElimOptPlus implements Visitor<Exp> {
    @Override
    public Exp visit(StarExp exp) {
        return new StarExp(exp.getChildren().get(0).accept(this));
    }
    @Override
    public Exp visit(OptionalityExp exp) {
        return new UnionExp(new EmptyStringExp(), exp.getChildren().get(0).accept(this));
    }
    @Override
    public Exp visit(PlusExp exp) {
        Exp e = exp.getChildren().get(0).accept(this);
        return new CatExp(e, new StarExp(e));
    }
    @Override
    public Exp visit(UnionExp exp) {
        List<Exp> children = exp.getChildren();
        return new UnionExp(children.get(0).accept(this), children.get(1).accept(this));
    }
    @Override
    public Exp visit(CatExp exp) {
        List<Exp> children = exp.getChildren();
        return new CatExp(children.get(0).accept(this), children.get(1).accept(this));
    }
    @Override
    public Exp visit(SymbolExp exp) {
        return new SymbolExp(exp.getSymbol());
    }
    @Override
    public Exp visit(EmptyStringExp e) {
        return new EmptyStringExp();
    }
}

```

# gennaio 2014

3. Considerare le dichiarazioni dei seguenti tipi, che modellano dei raggruppamenti arbitrari di figure colorate e implementano il design pattern *visitor*. Solo per curiosità, il sistema di coordinate usato è quello "a schermo", dove l'asse delle Y è orientato verso il basso (tutto quello che vi chiederemo di implementare è indipendente dall'orientamento degli assi).

L'interfaccia generica `ShapeVisitor<T>` rappresenta un *visitor* che effettua un'operazione che restituisce un oggetto di tipo T. La classe `Point` rappresenta i punti colorati; un oggetto di tipo `Point` deve poter essere costruito a partire da un colore `Color` color e le due coordinate int x e int y, e deve offrire degli opportuni *getter*.

La classe `Rectangle` rappresenta i rettangoli colorati, dove le coordinate (x1,y1) rappresentano l'angolo in alto a sinistra e (x2, y2) l'angolo in basso a destra.

Infine, la classe `Group` rappresenta un raggruppamento, non vuoto, di figure; il colore di un gruppo corrisponde al colore del suo bordo, mentre le figure raggruppate all'interno mantengono i loro colori. Il costruttore, già implementato, si limita a salvare le figure raggruppate nell'array `this.shapes` (la scelta di usare due argomenti, `firstShape` e `otherShapes`, impedisce a compile-time la costruzione di `Group` vuoti).

```
interface ShapeVisitor<T> {
    T visit(Point p);
    T visit(Rectangle r);
    T visit(Group g);
}

abstract class Shape {
    private final Color color;
    public Shape(Color color) { this.color = color; }
    public Color getColor() { return color; }
    abstract <T> T accept(ShapeVisitor<T> v);
}

class Point extends Shape { /* DA COMPLETARE (1) */ }

class Rectangle extends Shape {
    private final int x1, y1, x2, y2;
    public int getX1() { return x1; }
    public int getY1() { return y1; }
    public int getX2() { return x2; }
    public int getY2() { return y2; }
    public Rectangle(Color color, int x1, int y1, int x2, int y2) {
        super(color);
        if (x1>x2 || y1>y2)
            throw new IllegalArgumentException();
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }
    @Override <T> T accept(ShapeVisitor<T> v) { /* DA COMPLETARE (2) */ }
}

class Group extends Shape {
    private final Shape[] shapes;
    public Group(Color borderColor, Shape firstShape, Shape... otherShapes) {
        super(borderColor);
        final int initialCapacity = otherShapes.length+1;
        List<Shape> l = new ArrayList<>(initialCapacity);
        l.add(firstShape);
        l.addAll(Arrays.asList(otherShapes));
        this.shapes = l.toArray(new Shape[initialCapacity]);
    }
    public Shape[] getShapes() { return shapes; }
    @Override <T> T accept(ShapeVisitor<T> v) { /* DA COMPLETARE (3) */ }
}
```

(a) Completare le definizioni delle classi `Point`, `Rectangle` e `Group`.

(b) Completare la seguente definizione del *visitor* "predicato" `FindColor`. Un oggetto di tipo `FindColor`, costruito a partire da un certo colore c, deve implementare la visita che restituisce un valore di verità che corrisponde al fatto che il colore c sia utilizzato, o meno, all'interno di una figura.

```
class FindColor implements ShapeVisitor<Boolean> {
    private Color color;
    public Boolean visit(Color color) { this.color = color; }
    @Override public Boolean visit(Point p) { /* DA COMPLETARE (4) */ }
    @Override public Boolean visit(Rectangle r) { /* DA COMPLETARE (5) */ }
    @Override public Boolean visit(Group g) { /* DA COMPLETARE (6) */ }
}
```

- (c) Date le seguenti dichiarazioni di classe, completare la definizione del `visitor CalculateBoundingBox`, che deve permettere di calcolare il *bounding-box*, rappresentato da un'istanza della classe `BoundingBox`, di una figura. Il *bounding-box* di una figura è, per definizione, il rettangolo più piccolo che contiene tutti gli elementi della figura. Quindi, per un punto corrisponde al punto stesso (ovvero, `upperLeftX==bottomRightX` e `upperLeftY==bottomRightY`), per un rettangolo corrisponde a un *bounding-box* con le stesse coordinate del rettangolo, e per un gruppo di figure corrisponde all'unione dei *bounding-box*. Notate che l'operazione di unione di *bounding-box* è già implementata nel metodo `union(BoundingBox)`.

```
class BoundingBox {
    private final int upperLeftX, upperLeftY, bottomRightX, bottomRightY;
    public int getUpperLeftX() { return upperLeftX; }
    public int getUpperLeftY() { return upperLeftY; }
    public int getBottomRightX() { return bottomRightX; }
    public int getBottomRightY() { return bottomRightY; }
    public BoundingBox(int upperLeftX, int upperLeftY, int bottomRightX, int bottomRightY) {
        if (upperLeftX>bottomRightX || upperLeftY>bottomRightY)
            throw new IllegalArgumentException();
        this.upperLeftX = upperLeftX;
        this.upperLeftY = upperLeftY;
        this.bottomRightX = bottomRightX;
        this.bottomRightY = bottomRightY;
    }
    BoundingBox union(BoundingBox other) {
        return new BoundingBox(
            min(this.upperLeftX, other.upperLeftX),
            min(this.upperLeftY, other.upperLeftY),
            max(this.bottomRightX, other.bottomRightX),
            max(this.bottomRightY, other.bottomRightY));
    }
    @Override public String toString() {
        return String.format("(%, %d)-(%d, %d)", this.upperLeftX, this.upperLeftY,
                             this.bottomRightX, this.bottomRightY);
    }
}

class CalculateBoundingBox implements ShapeVisitor<BoundingBox> { /* DA COMPLETARE (7) */ }
```

3. (a) class Point extends Shape {
 private final int x, y;
 public Point(Color color, int x, int y) {
 super(color);
 this.x = x;
 this.y = y;
 }
 public int getX() { return x; }
 public int getY() { return y; }
 @Override <T> T accept(ShapeVisitor<T> v) { return v.visit(this); }
}

class Rectangle extends Shape { /\* ... \*/
 @Override <T> T accept(ShapeVisitor<T> v) { return v.visit(this); }
}

class Group extends Shape { /\* ... \*/
 @Override <T> T accept(ShapeVisitor<T> v) { return v.visit(this); }
}

- (b) class FindColor implements ShapeVisitor<Boolean> {
 private Color color;
 public FindColor(Color color) { this.color = color; }
 @Override public Boolean visit(Point p) {
 return p.getColor().equals(this.color);
 }
 @Override public Boolean visit(Rectangle r) {
 return r.getColor().equals(this.color);
 }
 @Override public Boolean visit(Group g) {
 if (g.getColor().equals(this.color))
 return Boolean.TRUE;
 for (Shape s : g.getShapes())
 if (s.accept(this))
 return Boolean.TRUE;
 return Boolean.FALSE;
 }
}

- (c) class CalculateBoundingBox implements ShapeVisitor<BoundingBox> {
 @Override public BoundingBox visit(Point p) {
 final int px = p.getX();
 final int py = p.getY();
 return new BoundingBox(px, py, px, py);
 }
 @Override public BoundingBox visit(Rectangle r) {
 return new BoundingBox(r.getX1(), r.getY1(), r.getX2(), r.getY2());
 }
 @Override public BoundingBox visit(Group g) {
 Shape[] shapes = g.getShapes();
 assert shapes.length>0;
 BoundingBox result = shapes[0].accept(this);
 for (int i=1; i<shapes.length; ++i)
 result = result.union(shapes[i].accept(this));
 return result;
 }
}

# luglio 2013

4. Considerare i package ast e visitor che implementano abstract syntax tree e visite su di essi per espressioni aritmetiche formate a partire da variabili, literal interi, l'operatore binario di addizione e quello unario di sottrazione.

```

package ast;
import visitor.Visitor;
public interface Exp {
    Iterable<Exp> getChildren();
    void accept(Visitor v);
}

package ast;
import static java.util.Arrays.asList;
public abstract class AbsExp implements Exp {
    private final Iterable<Exp> children;
    protected AbsExp(Exp... children) {
        // completare
    }
    @Override
    public Iterable<Exp> getChildren() {
        // completare
    }
}

package ast;
import visitor.Visitor;
public class IdentExp extends AbsExp implements Variable {
    private final String name;
    public IdentExp(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void accept(Visitor v) {
        // completare
    }
}

package ast;
import visitor.Visitor;
public class NumLit extends AbsExp {
    final private int value;
    public NumLit(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    public void accept(Visitor v) {
        // completare
    }
}

package ast;
import visitor.Visitor;
public class AddExp extends AbsExp {
    public AddExp(Exp exp1, Exp exp2) {
        // completare
    }
    public void accept(Visitor v) {
        // completare
    }
}

package ast;
import visitor.Visitor;
public class MinusExp extends AbsExp {
    public MinusExp(Exp exp) {
        // completare
    }
    public void accept(Visitor v) {
        // completare
    }
}

```

(a) Completare le definizioni delle classi AbsExp, IdentExp, NumLit, AddExp e MinusExp.

(b) Date le seguenti dichiarazioni di classe e interfaccia, completare la definizione delle classi ContainVarVisitor e PrefixVisitor.

```

package visitor;
import ast.*;
public interface Visitor {
    void visit(IdentExp e);
    void visit(NumLit e);
    void visit(AddExp e);
    void visit(MinusExp e);
}

package visitor;
public abstract class AbstractVisitor<T> implements Visitor {
    protected T result;
    public T getResult() {
        return result;
    }
}

package visitor;
import java.util.Iterator;
import ast.*;
public class ContainVarVisitor extends AbstractVisitor<Boolean> {
    private Variable var;
    public void setVar(Variable var) {
        this.var = var;
    }
    public ContainVarVisitor(Variable var) {
        this.var = var;
    }
    @Override
    public void visit(AddExp exp) {
        // completare
    }

    @Override
    public void visit(IdentExp exp) {
        // completare
    }

    @Override
    public void visit(MinusExp exp) {
        // completare
    }

    @Override
    public void visit(NumLit exp) {
        // completare
    }
}

package visitor;
import java.util.Iterator;
import ast.*;
public class PrefixVisitor extends AbstractVisitor<String> {
    public void visit(AddExp exp) {
        // completare
    }
    public void visit(IdentExp exp) {
        // completare
    }
    public void visit(MinusExp exp) {
        // completare
    }
    public void visit(NumLit exp) {
        // completare
    }
}

```

- la classe ContainVarVisitor controlla se l'espressione visitata contiene una data variabile. Esempio:

```

Exp exp = new AddExp(new AddExp(new NumLit(0), new IdentExp("z")),
    new MinusExp(new AddExp(new IdentExp("x"), new IdentExp("y"))));
ContainVarVisitor cv = new ContainVarVisitor(new IdentExp("x"));
exp.accept(cv);
cv.visit(new IdentExp("z"));
exp.accept(cv);
assert cv.getResult();
cv.setVar(new IdentExp("w"));
exp.accept(cv);
assert cv.getResult();
cv.setVar(new IdentExp("w"));
exp.accept(cv);
assert !cv.getResult();

```

- la classe PrefixVisitor genera la stringa corrispondente alla forma pollica prefissa della espressione visitata. Esempio:

```

Exp exp = new AddExp(new AddExp(new NumLit(0), new IdentExp("z")),
    new MinusExp(new AddExp(new IdentExp("x"), new IdentExp("y"))));
PrefixVisitor pv = new PrefixVisitor();
exp.accept(pv);
assert pv.getResult().equals("+ 0 z - x y");

```

```

public abstract class AbsExp implements Exp { public class IdentExp extends AbsExp implements Variable {
    private final String name;
    private final Iterable<Exp> children;
    protected AbsExp(Exp... children) {
        this.children = asList(children);
    }
    @Override
    public Iterable<Exp> getChildren() {
        return children;
    }
}

public class NumLit extends AbsExp {
    final private int value;
    public NumLit(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}

public class AddExp extends AbsExp {
    public AddExp(Exp exp1, Exp exp2) {
        super(exp1, exp2);
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}

public class MinusExp extends AbsExp {
    public MinusExp(Exp exp) {
        super(exp);
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}

public class ContainVarVisitor extends AbstractVisitor<Boolean> {
    private Variable var;
    public void setVar(Variable var) {
        this.var = var;
    }
    public ContainVarVisitor(Variable var) {
        this.var = var;
    }
    @Override
    public void visit(AddExp exp) {
        Iterator<Exp> it = exp.getChildren().iterator();
        it.next().accept(this);
        if (result)
            return;
        it.next().accept(this);
    }

    @Override
    public void visit(IdentExp exp) {
        result = var.getName().equals(exp.getName());
    }

    @Override
    public void visit(MinusExp exp) {
        exp.getChildren().iterator().next().accept(this);
    }

    @Override
    public void visit(NumLit exp) {
        result = false;
    }
}

public class PrefixVisitor extends AbstractVisitor<String> {
    public void visit(AddExp exp) {
        Iterator<Exp> it = exp.getChildren().iterator();
        it.next().accept(this);
        String res = result;
        it.next().accept(this);
        result = "+" + res + " " + result;
    }

    public void visit(IdentExp exp) {
        result = exp.getName();
    }

    public void visit(MinusExp exp) {
        Iterator<Exp> it = exp.getChildren().iterator();
        it.next().accept(this);
        result = "-" + result;
    }

    public void visit(NumLit exp) {
        result = String.valueOf(exp.getValue());
    }
}

```

# giugno 2013

4. Considerare i package ast e visitor che implementano abstract syntax tree e visite su di essi per espressioni aritmetiche formate a partire da variabili, literal interi e gli operatori binari di addizione e moltiplicazione.

```

package ast;
import java.util.List;
import visitor.Visitor;
public interface Exp {
    List<Exp> getChildren();
    void accept(Visitor v);
}

package ast;
public interface Variable extends Exp {
    String getName();
}

package ast;
import static java.util.Arrays.asList;
import java.util.List;
public abstract class AbsExp implements Exp {
    private final List<Exp> children;
    protected AbsExp(Exp... children) {
        // completare
    }
    @Override
    public List<Exp> getChildren() {
        // completare
    }
}

package ast;
import visitor.Visitor;
public class IdentExp extends AbsExp implements Variable {
    private final String name;
    public IdentExp(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void accept(Visitor v) {
        // completare
    }
}

package ast;
import visitor.Visitor;
public class NumLit extends AbsExp {
    final private int value;
    public NumLit(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    public void accept(Visitor v) {
        // completare
    }
}

package ast;
import visitor.Visitor;
public class AddExp extends AbsExp {
    public AddExp(Exp exp1, Exp exp2) {
        // completare
    }
    public void accept(Visitor v) {
        // completare
    }
}

package ast;
import visitor.Visitor;
public class MulExp extends AbsExp {
    public MulExp(Exp exp1, Exp exp2) {
        // completare
    }
    public void accept(Visitor v) {
        // completare
    }
}

```

(a) Completare le definizioni delle classi AbsExp, IdentExp, NumLit, AddExp e MulExp.

(b) Date le seguenti dichiarazioni di classe e interfaccia, completare la definizione delle classi FreeVarVisitor e SimplifyVisitor.

```

package visitor;
import ast.*;
public interface Visitor {
    void visit(IdentExp e);
    void visit(NumLit e);
    void visit(AddExp e);
    void visit(MulExp e);
}

package visitor;
public abstract class AbstractVisitor<T> implements Visitor {
    protected T result;
    Public T getResult() {
        return result;
    }
}

package visitor;
import java.util.HashSet;
public class FreeVarVisitor extends AbstractVisitor<Set<String>> {
    result = new HashSet<>();
}
@Override
public void visit(IdentExp exp) {
    // completare
}
@Override
public void visit(NumLit exp) {
    // completare
}
@Override
public void visit(AddExp exp) {
    // completare
}
@Override
public void visit(MulExp exp) {
    // completare
}

package visitor;
import java.util.List;
public class SimplifyVisitor extends AbstractVisitor<Exp> {
    private boolean isLit(Exp exp, int val) {
        return exp instanceof NumLit && ((NumLit) exp).getValue() == val;
    }
    @Override
    public void visit(IdentExp exp) {
        // completare
    }
    @Override
    public void visit(NumLit exp) {
        // completare
    }
    @Override
    public void visit(AddExp exp) {
        // completare
    }
    @Override
    public void visit(MulExp exp) {
        // completare
    }
}

```

- la classe FreeVarVisitor calcola l'insieme dei nomi di tutte le variabili contenute nell'espressione. Esempio:

```

Exp exp = new AddExp(new NumLit(0), new MulExp(new AddExp(new IdentExp("x"), new IdentExp("y")), new AddExp(new IdentExp("y"), new IdentExp("z"))));
FreeVarVisitor fvv = new FreeVarVisitor();
exp.accept(fvv);
Set<String> names = fvv.getResult();
assert names.contains("x");
assert names.contains("y");
assert names.contains("z");
assert names.size() == 3;

```

- la classe SimplifyVisitor semplifica l'espressione applicando le identità

$$\begin{aligned} e + 0 &= e & e + e &= 0 + e \\ e \cdot 1 &= e & e = 1 \cdot e \end{aligned}$$

Esempio:

```

SimplifyVisitor sv = new SimplifyVisitor();
Exp exp = new MulExp(new AddExp(new NumLit(1), new NumLit(0)), new MulExp(new IdentExp("x"), new NumLit(1)));
exp.accept(sv);
exp = sv.getResult();
assert exp instanceof IdentExp;
assert ((IdentExp) exp).getName().equals("x");

```

```

public abstract class AbsExp implements Exp {
    public class IdentExp extends AbsExp implements Variable {
        private final String name;
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}

public class NumLit extends AbsExp {
    final private int value;
    public NumLit(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}

public class AddExp extends AbsExp {
    public AddExp(Exp exp1, Exp exp2) {
        super(exp1, exp2);
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}

public class MulExp extends AbsExp {
    public MulExp(Exp exp1, Exp exp2) {
        super(exp1, exp2);
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}

public class FreeVarVisitor extends AbstractVisitor<Set<String>> {
    public FreeVarVisitor() {
        result = new HashSet<>();
    }
    private void genVisit(Exp exp) {
        for (Exp e : exp.getChildren())
            e.accept(this);
    }
    public void visit(MulExp exp) {
        genVisit(exp);
    }
    public void visit(IdentExp exp) {
        result.add(exp.getName());
    }
    public void visit(NumLit exp) {
        genVisit(exp);
    }
    public void visit(AddExp exp) {
        genVisit(exp);
    }
}

public class SimplifyVisitor extends AbstractVisitor<Exp> {
    private boolean isLit(Exp exp, int val) {
        return exp instanceof NumLit && ((NumLit) exp).getValue() == val;
    }
    @Override
    public void visit(MulExp exp) {
        List<Exp> exps = exp.getChildren();
        exps.get(0).accept(this);
        Exp exp1 = result;
        exps.get(1).accept(this);
        if (isLit(result, 1))
            result = exp1;
        else if (!isLit(exp1, 1))
            result = new MulExp(exp1, result);
    }
    @Override
    public void visit(IdentExp exp) {
        result = exp;
    }
    @Override
    public void visit(NumLit exp) {
        result = exp;
    }
    @Override
    public void visit(AddExp exp) {
        List<Exp> exps = exp.getChildren();
        exps.get(0).accept(this);
        Exp exp1 = result;
        exps.get(1).accept(this);
        if (isLit(result, 0))
            result = exp1;
        else if (!isLit(exp1, 0))
            result = new AddExp(exp1, result);
    }
}

```

# giugno 2012

3. Considerare le seguenti classi e interfacce:

- `Exp, AbstractExp, NumLit, AddExp, MulExp`: classi e interfacce che definiscono l'abstract syntax tree di un'espressione di tipo `int`, costruita a partire da costanti numeriche (`NumLit`), e dagli operatori di addizione (`AddExp`) e moltiplicazione (`MulExp`).
- `Visitor, AbstractVisitor`: interfaccia e classe astratta che definiscono un generico visitor di `Exp`.
- `EvalVisitor`: classe che implementa un visitor che restituisce come risultato il valore in cui si valuta l'espressione visitata.
- `SwapVisitor`: classe che implementa un visitor che restituisce come risultato una nuova espressione ottenuta da quella visitata sostituendo gli operatori di addizione con quelli di moltiplicazione e viceversa.

```
public interface Exp {
    void accept(Visitor v);
}

public abstract class AbstractExp implements Exp {
    private final Exp[] children;
    public AbstractExp(Exp... children) {
        this.children = children;
    }
    public Exp[] getChildren() {
        return children;
    }
}

public final class NumLit extends AbstractExp {
    private final int value;
    public NumLit(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    @Override
    public void accept(Visitor v) {
        // completare
    }
}

public final class AddExp extends AbstractExp {
    AddExp(Exp exp1, Exp exp2) {
        super(exp1, exp2);
    }
    @Override
    public void accept(Visitor v) {
        // completare
    }
}

public final class MulExp extends AbstractExp {
    MulExp(Exp exp1, Exp exp2) {
        super(exp1, exp2);
    }
    @Override
    public void accept(Visitor v) {
        // completare
    }
}

public interface Visitor {
    // completare
}

public abstract class AbstractVisitor<T> implements Visitor {
    protected T result;
    public T getResult() {
        return result;
    }
}

public class EvalVisitor extends AbstractVisitor<Integer> {
    // completare
}

public class SwapVisitor extends AbstractVisitor<Exp> {
    // completare
}
```

- Completare i metodi `accept` delle classi `NumLit`, `AddExp` e `MulExp`.
- Completare la definizione dell'interfaccia `Visitor`.
- Completare la definizione della classe `EvalVisitor`.
- Completare la definizione della classe `SwapVisitor`.

```
(a) public final class NumLit extends AbstractExp {
    private final int value;
    public NumLit(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    @Override
    public void accept(Visitor v) {
        v.visit(this);
    }
}

public final class AddExp extends AbstractExp {
    AddExp(Exp exp1, Exp exp2) {
        super(exp1, exp2);
    }
    @Override
    public void accept(Visitor v) {
        v.visit(this);
    }
}

public final class MulExp extends AbstractExp {
    MulExp(Exp exp1, Exp exp2) {
        super(exp1, exp2);
    }
    @Override
    public void accept(Visitor v) {
        v.visit(this);
    }
}

(c) public class EvalVisitor extends AbstractVisitor<Integer> {
    @Override
    public void visit(NumLit e) {
        result = e.getValue();
    }
    @Override
    public void visit(AddExp e) {
        Exp[] children = e.getChildren();
        children[0].accept(this);
        int res0 = result;
        children[1].accept(this);
        result += res0;
    }
    @Override
    public void visit(MulExp e) {
        Exp[] children = e.getChildren();
        children[0].accept(this);
        int res0 = result;
        children[1].accept(this);
        result *= res0;
    }
}

(b) public interface Visitor {
    void visit(NumLit e);
    void visit(AddExp e);
    void visit(MulExp e);
}

(d) public class SwapVisitor extends AbstractVisitor<Exp> {
    @Override
    public void visit(NumLit e) {
        result = new NumLit(e.getValue());
    }
    @Override
    public void visit(AddExp e) {
        Exp[] children = e.getChildren();
        children[0].accept(this);
        Exp res0 = result;
        children[1].accept(this);
        result = new MulExp(res0, result);
    }
    @Override
    public void visit(MulExp e) {
        Exp[] children = e.getChildren();
        children[0].accept(this);
        Exp res0 = result;
        children[1].accept(this);
        result = new AddExp(res0, result);
    }
}
```