Assignment Logical Formulas

Object Orientation

Spring 2020

The Visitor Pattern

1 Learning Goals

In this assignment we ask you to design and implement a representation for logical formulas in Java. After having completed this assignment, you should be able to:

- Implement classes to represent logical formulas, where
  - A suitable interface or abstract class is defined that serves as a base for all nodes in your syntax tree.
  - Each logical connective is implemented as an extension of this base class.
- Introduce new operations using the visitor pattern.
- Utilize UML class and sequence diagrams to design an object oriented program.

2 Warming Up With Generics

This is a warm-up exercise. It will not be graded.

In the start project you find a class Pair and a test case PairTest. The second test case fails with a type cast error. Pair stores its components as Objects, so whenever you get a component from a pair, you have to type cast it in order to use it. This can lead to run time errors if done inconsistently.

By using generics, you can turn these kind of run time errors into compile time errors. Your task: Make the Pair class generic `Pair<F, S>`. When you do everything correctly, the second test case will give two compile errors: “Cannot cast from Boolean to String” and ”Cannot cast from String to Boolean”.

3 Logical Formulas

This assignment is about representing and manipulating formulas of propositional logic.

A *logical formula* consists of the constants *true* and *false*, atomic propositions, logical operators $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies) and $\neg$ (not). Atomic propositions are like variables in programming languages. Atomic propositions can be any string, but we usually use single capital letters like A, B, P, Q. The abstract syntax of logical formulas is given by the following grammar.

$$F ::= \textbf{true} \mid \textbf{false} \mid \text{Atomic} \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Rightarrow F_2 \mid \neg F$$

Examples for logical formulas are $A \Rightarrow B$, $\neg A \vee B$, $A \Rightarrow (A \vee B) \wedge (C \Rightarrow \neg D)$. In these examples A, B, C, D are atomic propositions.

Formulas are represented in Java by trees, where the nodes correspond to the syntactic elements. Such trees are called *abstract syntax trees*. The leaves in such a tree correspond to constants or atomic propositions and internal nodes represent the operators. This means that some internal nodes have 2 children and others 1 child.

The representation of formulas using syntax trees is similar to the numerical expressions in assignment 5. There is one interface for all the different nodes, and concrete classes that implement this interface for each type of node. There are two major differences compared to assignment 5.

1. In the numerical expressions, operations were added to the base interface as abstract methods, and implemented in each subclass. In this assignment, the base interface is left unchanged. All operations are implemented as visitors.
2. A single concrete class is used to represent all binary operators. It uses the *strategy pattern* to realize their different behaviors.

The visitor pattern has two ingredients. First the visitor interface, in this assignment called `FormulaVisitor`, and second an abstract method `accept` in the formula base class.

The base interface for logical formulas looks as follows. All of the concrete nodes must implement this interface.

```
public interface Formula {
    void accept( FormulaVisitor visitor ) ;
}
```

The `FormulaVisitor` interface has one `visit` method for each concrete node. Note: The visit methods are overloaded, that's why they can all have the same name. It is possible to give them different names, like `visitNot`, `visitBinOp`, and so on.

```
public interface FormulaVisitor {
    void visit( Not form );
    void visit( BinaryOperator form );
    // and so on
}
```

All of the concrete nodes in the syntax tree are classes that implement `Formula`. Here is an example for a class of the *not* operator.

```
public class Not implements Formula  {
    private Formula operand;

    public Not( Formula oper ) {
        this.operand = oper;
    }

    public Formula getOperand() {
        return operand;
    }

    public void accept( FormulaVisitor v ) {
```

```
        v.visit( this );
    }
}
```

This definition should be self explanatory. The `accept` method is boilerplate code that always looks like this.

All binary operators should be represented by a single class `BinaryOperator` which has an operator `BinOp` as an attribute. This is an example of the strategy pattern. `BinOp` should be an enum that implements `BinaryOperator<Boolean>` from the Java standard library. Every `BinOp` should have a string representation, a precedence, and an evaluation function.

4 Operations on Logical Formulas

You should implement two visitors for logical formulas: a pretty-printer and an evaluator.

The pretty-printer should print a given formula to standard output, omitting parentheses when it is safe. For this, every operator should have a precedence. The precedence of the operators are, from high to low: $\neg, \wedge, \vee, \Rightarrow$. This means that $\neg$ binds stronger than $\wedge$, and so on. To determine if parentheses are needed, you have to compare the precedence of an operator with that of its parent. The details are left for you to figure out. Hint: pass the precedence of the current operator as an extra argument to the recursive calls to `accept`. For operators with the same precedence, you should always print parenthesis to be safe.

The evaluator needs an environment for atomic propositions, just as the evaluator for numerical expression needs for variables. Where variables were assigned integers, atomic propositions are assigned Booleans. This means that the evaluator needs a `Map<String,Boolean>` to evaluate formulas. For example, in the environment $[P \mapsto \textbf{false}]$, the formula $\neg P$ should evaluate to **true**.

The `visit` methods of the evaluator have a return value, while the ones of the pretty-printer need an additional argument. This means that the type signature can not be just `public void visit(Not f)`. To introduce the flexibility you need, you should make `FormulaVisitor` generic.

`Void` is the class representation of the keyword `void` standing for the empty type. `Void` functions should always `return null`.

Here are some lines of code to give you inspiration. You should figure out the rest for yourself.

```
public interface FormulaVisitor<Result,AdditionalArg> {
    Result visit(Not form, AdditionalArg a);
}

public interface Formula {
    public <R,A> R accept(FormulaVisitor<R,A> visitor, A a);
}

public class PrintVisitor
    implements FormulaVisitor<Void,Integer>
{
  private StringBuilder result;
```

```
  public String getResult() { return result.toString(); }
}

public class EvaluateVisitor
    implements FormulaVisitor<Boolean, Void> { }
```

5 Your Tasks

In this assignment you have to draw two UML diagrams in steps 1 and 7. You have to hand in these diagrams. Either draw them on paper and submit a **photo as jpeg**, or use a UML tool and **export as pdf**. No other format will be accepted.

If you already have a UML drawing program you like, feel free to use it. Just make sure it can **export to pdf**. If you do not know which tool to use, we suggest Visual Paradigm. You can use it for free if you sign up with your email address. `https://online.visual-paradigm.com/subscribe.jsp`

1. Before you begin with your implementation, draw one UML class diagram of all the classes you need.
2. Implement the nodes for the syntax tree as described above. You need an interface and four concrete classes for the different nodes in the syntax tree. The classes should be called `Atom`, `Constant`, `BinaryOperator`, and `Not`. Hint: there can only be exactly two instances of Constant.
3. For every concrete node, add a corresponding `visit` method to `FormulaVisitor`.
4. Create a class `PrintVisitor` that implements `FormulaVisitor` and transforms a formula to a String.
5. Create a class `EvaluateVisitor` that implements `FormulaVisitor` and evaluates formulas in a given environment.
6. The project template comes with test cases. Make sure that all test cases pass.
7. Draw a sequence diagram for printing the formula $\neg P$, given in the following code. Only include calls to `accept` and `visit`. Use the diagram in fig. 1 as a starting point.

```
Atom f2 = new Atom("P");
Not f1 = new Not(f2);
PrintVisitor v = new PrintVisitor();
f1.accept(v);
```

6 Submit Your Project

To submit your project, follow these steps. You have to submit three files: The java project, the class diagram, and the sequence diagram.

1. Submit the diagrams either as **jpeg or pdf**. No other format is accepted.
2. Use the **project export** feature of NetBeans to create a zip file of your entire project: File → Export Project → To ZIP.

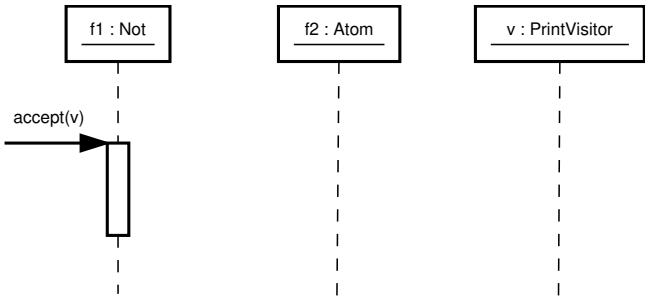

Figure 1: Template for the sequence diagram you have to draw.

3. **Submit this zip file on Brightspace**. Do not submit individual Java files. Do not submit any other archive format. Only one person in your group has to submit it. Submit your project before the deadline, which can be found on Brightspace.