Assignment 4: Logical Expressions, Nand and Nor Logic, and Logical Simplification

Due Date: 30.6.24

Introduction

In this assignment, we will put aside our game for a little while, and delve instead into the magical world of logics. We will implement a system that can represent nested logical expressions that include variables, evaluate their values for specific variable assignments, convert them, and simplify the results.

In doing so we will work in a recursive framework, see some more examples of polymorphism, and practice the use of inheritance and class hierarchies for sharing common code.

Part 1 – Logical Expressions

Our goal is to represent logical expressions such as:

$$\sim ((T \wedge (x \vee y)) \oplus x)$$

Where T is a value of "True", the $\sim$, $\vee$, $\wedge$, $\oplus$ symbols denote the "not", "or", "and", and "xor" operators respectively, and x and y are variables.

The unary expressions are:

- `Var("x")` indicating that x is a variable.
- `Not(x)` indicating the negation of the value of x.

The binary expressions are:

- `Or(x,y)` indicating the "or" of x, y.
- `And(x,y)` indicating the "and" of x, y.
- `Xor(x,y)` indicating the "xor" of x, y.

This link might be helpful: Wikipedia page for Boolean operations in Hebrew.

We also have a `Val(F)` expression indicating the logical value "False".

Assuming we represent each of the atomic expressions as a Class of the same name that takes its arguments in the constructor, we can create the expression above in Java using:

```
Expression e = new Not(
                  new Xor(
                     new And(
                        new Val(true),
                        new Or(
                           new Var("x"),
                           new Var("y")
                        )
                     ),
                     new Var("x")
                  )
               );
```

The tree is given below: Figure-1

Note that all the nodes in the tree are expressions (according to the ‘Expression‘ interface): Figure-2

Similarly, we could represent the expression $(x \wedge y) \oplus T$ as follows:

```
Expression e2 = new Xor(new And(new Var("x"), new Var("y")), new Val(true));
```

Once we have an expression, we would like to be able to:

- **Get a nice and readable string representation:**

  ```
  String s = e2.toString();
  System.out.println(s);
  ```

  Should print $((x \wedge y) \oplus T)$

- **Ask about the variables in the expression:** (this example uses generics)

  ```
  List<String> vars = e2.getVariables();
  for (String v : vars) {
      System.out.println(v);
  }
  ```

  Should print

  ```
  x
  y
  ```

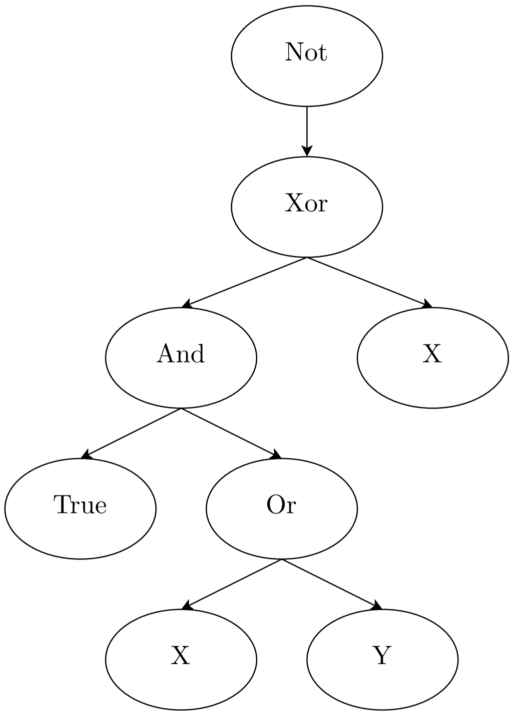

Figure 1: Expression tree

- **Assign values to variables:**

```
Expression e3 = e2.assign("y", e2);
System.out.println(e3);
// ((x & ((x & y) ^ T)) ^ T)
e3 = e3.assign("x", new Val(false));
System.out.println(e3);
// ((F & ((F & y) ^ T)) ^ T)
```

  In the first assign the variable y was assigned the Expression $(x \wedge y) \oplus T$, while in the second assign the variable x was assigned the Expression **False**.

- **Evaluate its value for a given variable assignment to values:** (this example uses a mapping)

```
Map<String, Boolean> assignment = new TreeMap<>();
assignment.put("x", true);
```

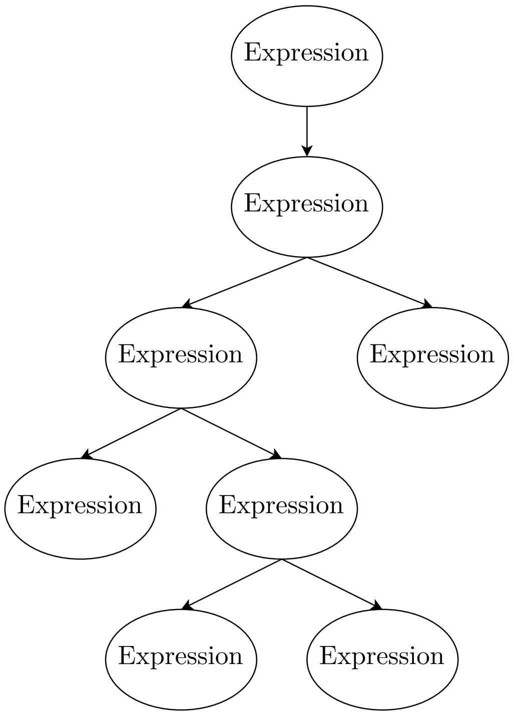

Figure 2: Expression tree

```
        assignment.put("y", false);
        Boolean value = e2.evaluate(assignment);
        System.out.println("The result is: " + value);
```

Should print `The result is: true`

What you need to implement

In the first part, begin with a simple interface called `Expression`:
(this interface uses generics and map - read more in the appendix section)

```
public interface Expression {
    // Evaluate the expression using the variable values provided
    // in the assignment, and return the result. If the expression
    // contains a variable which is not in the assignment, an exception
    // is thrown.
    Boolean evaluate(Map<String, Boolean> assignment) throws Exception;
    // A convenience method. Like the `evaluate(assignment)` method above,
    // but uses an empty assignment.
    Boolean evaluate() throws Exception;

    // Returns a list of the variables in the expression.
    List<String> getVariables();

    // Returns a nice string representation of the expression.
    String toString();

    // Returns a new expression in which all occurrences of the variable
    // var are replaced with the provided expression (Does not modify the
    // current expression).
    Expression assign(String var, Expression expression)
}
```

You should write the following classes, each of them corresponding to an atomic expression, and each of them should implement the Expression interface: `Val`, `Var` - representing truth values and variables. Unary expressions - `Not`. Binary expressions: `And`, `Or`, `Xor`, `Nand`, `Nor`, `Xnor`.

The string representations are as follows:

- And$(x, y) = (x \wedge y)$
- Or$(x, y) = (x \vee y)$
- Xor$(x, y) = (x \oplus y)$
- Nand$(x, y) = (x \text{ A } y)$
- Nor$(x, y) = (x \text{ V } y)$
- Xnor$(x, y) = (x \# y)$
- Not$(x) = \sim (x)$

You have to use the exact same symbols, since the assignment will be checked with an automatic test as well.

`Val` should have a constructor accepting a `Boolean`.
`Var` should have a constructor accepting a `String`.
The unary expressions should have a constructor accepting an `Expression`.
The binary expressions should have a constructor accepting two `Expressions`.

The implementation will make heavy use of recursion. For example, in order to evaluate an expression, you need to first evaluate its sub-expressions and then apply some function to the results, with the base cases being the evaluation of the `Var` and `Val` expressions.

1 Class Hierarchy

You should also implement the abstract base classes `BaseExpression`, `UnaryExpression` and `BinaryExpression`, and have the different expression classes inherit from them, according to the following hierarchy: Figure 3

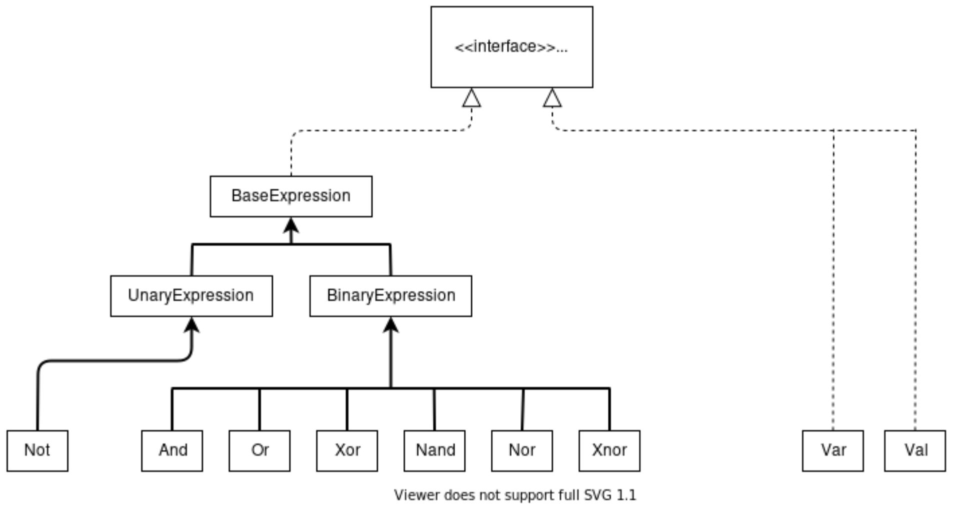

Figure 3: UML class diagram of expression classes

Try to put shared code in the base classes instead of the leaf classes. Example for candidate methods that can be in the base classes are `Boolean evaluate()` and `List<String> getVariables()`. You can add whatever non-public methods you want to the class hierarchy in order to help with code sharing.

How to Approach This

You need to implement many classes. One way to approach this would be to start with only a subset of the classes, for example only `Var`, `Val`, `And`, `Or` and `Not`. Once these are working, see if you can move some of the shared code to the base classes `UnaryExpression`, `BinaryExpression` and `BaseExpression`. Then, go ahead and implement the rest of the expression classes.

Test Your Code

Create a class with a `main` method that creates some nested expressions (for example `Expression e` as defined above) and then prints them, evaluates them, and asks for the variables in them.

Part 2 – Nandify and Norify

We can now create expressions, get their variables, and evaluate them with given variable assignments.

In this part, we will also convert them to logically equal expressions according to this logic:

- Wikipedia page for Nand Logic
- Wikipedia page for Nor Logic

Add the following methods to the Expression interface:

```
public interface Expression {
   // ... as before

   // Returns the expression tree resulting from converting all the operations to the
   // logical Nand operation.

   Expression nandify();

   // Returns the expression tree resulting from converting all the operations to the
   // logical Nor operation.

   Expression norify();
}
```

Part 3 – Simplification

Logical expressions can be quite messy and contain many ”redundant” parts.
For example:

```
Expression e = new Xor(new And(new Var("x"), new Val(false)),
new Or(new Var("y"), new Val(false)));
System.out.println(e);
// the result is:
// ((x & F) ^ (y | F))
```

This is correct, but can be really hard to read. We need to ”simplify” the expression to make it more friendly to humans.

We will add another method to the Expression interface. This method will return a new expression which is a simplified version of the current one.

```
public interface Expression {
   // ... as before

   // Returned a simplified version of the current expression.
   Expression simplify();
}
```

Example usage:

```
Expression e = new Xor(new And(new Var("x"), new Val(false)),
new Or(new Var("y"), new Val(false)));
System.out.println(e);
// the result is:
// ((x & F) ^ (y | F))
System.out.println(e.simplify());
// the result is:
// y
```

You need to support the following simplifications:

- $x \wedge 1 = x$
- $x \wedge 0 = 0$
- $x \wedge x = x$
- $x \vee 1 = 1$
- $x \vee 0 = x$
- $x \vee x = x$
- $x \oplus 1 = \sim(x)$
- $x \oplus 0 = x$

- $x \oplus x = 0$
- $xA1 =\sim (x)$
- $xA0 = 1$
- $xAx =\sim (x)$
- $xV1 = 0$
- $xV0 =\sim (x)$
- $xVx =\sim (x)$
- $x\#x = 1$
- An expression without variables evaluates to its result. For example:

  $$((T \wedge T) \vee F) \oplus T \Rightarrow F$$

  Note that X here stands for **any** expression, not just a variable.
- These should be recursive, so that, for example:

  $$\text{And}(\text{Xnor}(x, x), y) \Rightarrow y$$

What to Submit

Your code should include at least the following classes, interfaces, and abstract classes: `Val`, `Var`, `And`, `Or`, `Xor`, `Nand`, `Nor`, `Xnor`, `Not`, `Expression`, `BaseExpression`, `BinaryExpression`, `UnaryExpression`.

You should also include a class called `ExpressionsTest` including a `main` method that will:

1. Create an expression with at least three variables.
2. Print the expression.
3. Print the value of the expression with an assignment to every variable.
4. Print the Nandified version of the expression.
5. Print the Norified version of the expression.
6. Print the simplified version of the expression.

Each printing should be performed on its own line, do not add extra text, and do not add spaces between the lines.

String Representation Rules

- Truth values are a single capital letter: T, F
- Spaces and parentheses in the binary expressions &, $|,\hat{},A,V,\#$:(x $\wedge y$), $(x \vee y)$, $(x \oplus y)$, (x A y), (x V y), $(x \# y)$
- Parentheses (but no spaces) in $\sim$: $\sim(x)$ (and have double parentheses if they come from the inner expression)

Notes

- You are not allowed to use downcasting.
- You are not allowed to use the function `instanceof`.
- The assignment will be checked **automatically**, please submit it with correct printing format, otherwise you'll get 0.
- Since the test files cannot be uploaded to Submit, two test files are attached to help you find errors in your code:
  - `ExpressionTest` runs several tests and prints `true` for each successful test. If you pass the tests, you should see 5 prints of true.
  - `ExpressionOutput` prints the outputs of the previous tests to check what your output was in case you made a mistake.

  It's important to emphasize, these tests are designed to assist you and do not guarantee a 100% score.

Questions and Answers

- **Question:** What should double negation look like?
- **Answer:** $\sim(\sim(F))$
- **Question:** Should an expression containing only `Var` or `Val` be printed with or without parentheses?
- **Answer:** For this purpose, in this case:

  ```
  Expression e = new Xor(new And(new Var("x"), new Val(false)),
  new Or(new Var("y"), new Val(false)));
  System.out.println(e);
  ```

  The result will be: $((y \vee F) \oplus (x \wedge F))$

.1 Map

A `Mapping` or a `Dictionary` is a very common and useful abstraction when designing and writing programs.

The Mapping allows you to create a mapping from keys to values, and efficient retrieval of values based on their keys. Maps are usually implemented using data structures like binary search trees or hash-maps (which you will learn about in the data-structures course) but can also be implemented using linked lists – they will just be less efficient.

The interface for a Map allows you to:

- associate a key with a value,
- check if a key exists in the map,
- get the value associated with a specific key.

In the Java collections library, the Map interface represents a mapping, and it has several implementations, among them "HashMap" and "TreeMap" (the first is implemented using a hash-based data structure, and the second is implemented using a tree-based data structure).

You create and use a mapping as follows:

```
// You need the following imports
import java.util.Map;
import java.util.TreeMap;
// ...

// Now pretend we are inside a code block:
// The following lines will create a mapping where the keys are of type
// String and the values are of type Point.
Map<String, Point> m = new TreeMap<String, Point>();

// This is how we associate values with keys.
m.put("p1", new Point(1, 1));
m.put("p2", new Point(2, 3));

// This is how we check if a key exists, and get
// the value associated with a key.
if (m.containsKey("p1")) {
   Point p = m.get("p1");
}

Point p2 = m.get("baaa"); // p2 is null.
m.isEmpty(); // returns false
m.clear();
m.isEmpty(); // returns true
```

For further details, see the documentation for other available methods.

Valid Keys

The `TreeMap` and `HashMap` implementations both rely on the `.equals()` method of the keys being correct. In addition, `TreeMap` requires that the keys be comparable (according to the `Comparable` interface or a supplied `Comparator`), and `HashMap` relies on a correctly implemented `.hashCode()`. Built-in Java objects such as `String`, `Integer`, `Double` are all comparable and have proper `hashCode`, but if you want to use your own objects as keys, you must make sure to implement these methods yourself.

Valid Values

Any Java Object can be a value. This means that primitive types are **not** valid values, so the mapping must be created with one of the corresponding object types, for example:

```
Map<String, double> m = new HashMap<String, double>(); // Impossible!
Map<String, Double> m = new HashMap<String, Double>(); // Good.
```

However, once created you can pass the primitive type to '.put()' and return it from '.get()', it will be automagically converted to and from the object type.

```
m.put("a",1.0);   // this works
double d = m.get("a"); // this works
double e = m.get("b"); // NullPointerException!
```

.2 Generics

Java collection classes can be parameterized by a type, through the use of generics. We will learn more about generics later in the class. But you do not need to know much about generics in order to use them, so here is an example.

```
import java.util.List;
import java.util.LinkedList;
public class C {
  public static void main(String[] args) {
    // This creates a LinkedList containing only Strings.
    List<String> l = new LinkedList<String>();
    // Let's add some elements:
    l.add("ABC");
    l.add("DEF");
    // Since we declared the type upon construction, we can now
    // get elements of type string without casting.
    String s = l.get(1);
    // And we cannot add non-Strings
    l.add(new java.util.Random()); // This will fail at complile time.
    l.add(new C()); // This will fail at complile time.
  }
}
```