5.3. A Theorem Prover for the Propositional Calculus

The program presented in this section: (i) parses a given input string denoting a propositional formula, (ii) generates the internal tree which represents that formula, and (iii) tells us whether or not that formula is a tautology. In case it is not a tautology the program returns the values of the propositional variables for which the formula evaluates to `false`.

The productions of the context-free grammar which generates the input string are the following ones:

`p`	::=	`c | c>p`	(`p` is a propositional formula)
`c`	::=	`l | l+c |lxc`	(`c` is a condition)
`l`	::=	`a | -l`	(`l` is a literal)
`a`	::=	`'0' | ... | '9' | (b)`	(`a` is an atom)

The axiom is `p`. The propositional variables are denoted by the numbers `0,...,9` and thus, we assume that there are at most ten of them.

As in Section 5.2, `t` stands for `true`, `f` stands for `false`, `-` stands for `not`, `+` stands for `or`, `x` stands for `and`, and `>` stands for `implies`. No blanks are allowed before, in between, and after the input string. We assume that: (i) `+` and `x` bind more than `>`, and (ii) `-` binds more than `+` and `x`. We also assume right associativity for subexpressions involving `+` and `x`. Parentheses override priorities.

The tree which corresponds to the input string is then printed as a string which is generated by the following productions from the axiom `p`:

```
p ::= n | (-p) | (p + p) | (p x p) | (p > p)
n ::= 0 | 1 |...| 9
```

Thus, if we execute the following two commands:

```
javac PropositionalTheoremProver.java
java  PropositionalTheoremProver
```

we get:

```
Input a propositional formula according to the given grammar, please.
The variables should be between 0 and 3.
```

Indeed, in our implementation we set the value of the variable `DIM` (which tells us the maximum number of the propositional variables occurring in the input formula) to 4. Thus, our propositional variables are: `0,1,2,3`. The value of `DIM` can be at most `10` . Then, if we type the following input string:

```
0>(1+1)+-0x-2
```

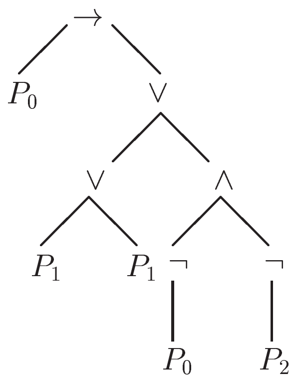
we get:

```
Parse tree of the propositional formula:
(0 > ((1 + 1) + ((-0) x (-2))))

Is it a tautology?  NO.
The falsity assignment is:
```

```
0=true; 1=false; 2=false; 3=false;
```

Indeed, the string `0 > ((1 + 1) + ((-0) x (-2)))` represents according to our conventions, the propositional formula $P_0 \to ((P_1 \vee P_1) \vee (\neg P_0 \wedge \neg P_2))$, where we have used the more familiar symbols $\to$, $\wedge$, $\vee$, and $\neg$ for denoting implication, conjunction, disjunction, and negation, respectively, and we have used the propositional variables P_0, P_1, and P_2, instead of `0`, `1`, and `2`, respectively. That formula can be depicted as a tree as follows:



The value of that propositional formula is `false` if $P_0 =$`true`, and $P_1 = P_2 =$ `false`.

```
/**
 * ========================================================================
 *                      PROPOSITIONAL THEOREM PROVER
 *
 * Grammar G for the input propositional formula p
 *
 *    propositional formulas  p  ::=  c   | c>p               <-- propParse
 *    conditions              c  ::=  l   | l+c | lxc         <-- condParse
 *    literals                l  ::=  a   | -l                <-- litParse
 *    atoms                   a  ::= '0' | ... | '9' | (p)   <-- atomParse
 *
 * Propositional variables are named: '0', ..., '9'.
 * To speed up the theorem prover, the value of DIM gives us the maximum
 * number of propositional variables actually present in the formula.
 * If DIM = 4, then the variables are: '0', '1', '2', and '3'.
 * (see line (1) below). DIM should be at most 10.
 *
 * - stands for not, + stands for or, x stands for and, > stands for implies
 * The grammar should not be left-recursive.
 *
 * Internal tree:
 * -------------
 * tree =
 * '0' | ... | '9' | (-tree) | (tree x tree) | (tree + tree) | (tree > tree)
 * (Leaf)     (Leaf)  (Unary)     (Binary)        (Binary)        (Binary)
 *
 * The value of a boolean expression:       value ::= true | false
 * ------------
 * Priorities: + and x bind more than >
 *             - binds more than + or x
 *             Right associativity for + and x
```

```
 *             Parentheses override priorities.
 * No blanks are allowed before, in between, and after the input string.
 *
 * The propositional formula in input should be a string generated by
 * the grammar G and then it is parsed and evaluated for all assignments
 * of the propositional variables.
 * The word 'node' is assumed to be equivalent to the word 'tree'.
 * ---------
 * The use of "C2 extends C1" keeps for C2 the operations of the class C1.
 * This may not be so good, but it allows the use of objects of the class C1
 * and then the use of "instanceof" to test whether or not an object of the
 * class C1 is indeed of the class C2.
 * ---------
 * Line (2) below checks that the input propositional formula is terminated
 * by a carriage-return '\n'.
 * ==========================================================================
 */
import java.io.*;
/**
 * ==========================================================================
 *                  Class which constructs a Node
 *
 * This class is used for constructing a Node of a binary tree.
 * By default, a Node is a binary node.
 * A Node can be extended for constructing a Node which is
 * either a binary node, or a unary node, or a leaf node.
 * ==========================================================================
 */
class Node {
   public Object  value;      // value field
   public Node    left;       // reference to the left  child node
   public Node    right;      // reference to the right child node
                                        // constructor
  /**
   * @param v : value at the node
   * @param l : reference to the left  child node
   * @param r : reference to the right child node
   */                                   // constructor of a binary node
   public Node( Object v, Node l, Node r) {
      if ( v == null )  throw new IllegalArgumentException( );
      value  = v;
      left   = l;
      right  = r;
   }
}
/**
 * ==========================================================================
 *                Class which constructs a binary node with two child nodes
 * ==========================================================================
 */
class BinaryNode extends Node {
  /**                             // constructor of binary node
   * @param v : value at the node
   * @param l : reference to the left  child node
   * @param r : reference to the right child node
   */
   public BinaryNode(Object v, Node l, Node r) {
```

```
      super(v, l, r);
   }
  /**
   * @return a string representing the tree whose root is the given node
   */
   public String toString(){
      return "("+ left.toString() +" "+ value +" "+ right.toString() +")";
   }
}
/**
 * ==========================================================================
 *                Class which constructs a unary node with one child node
 * ==========================================================================
 */
class UnaryNode extends Node {
  /**                             // constructor of unary node
   * @param v     : value at the node
   * @param child : reference to the child node
   */

   public UnaryNode(Object v, Node child) {
          super(v, child, null);
   }
  /**
   * @return a string representing the tree whose root is the given node
   */
     public String toString() {
          return "(" + value + left.toString() + ")";
     }
}
/**
 * ==========================================================================
 *                Class which constructs a leaf node
 * ==========================================================================
 */
class LeafNode extends Node {
  /**                                   // constructor of a leaf
   * @param v : value at the node
   */
   public LeafNode(Object v) {
           super(v, null, null);
   }
  /**
   * @return a string representing the leaf node
   */
   public String toString() {
          return "" + value;
   }
}
/**
 * ==========================================================================
 *                Class ParseException
 * ==========================================================================
 */
class ParseException extends Exception {
                         // constructor of a new instance of ParseException
   public ParseException() {
```

```
         super("parse exception!");
   }
}
/**
 * ==========================================================================
 *                Class EvalException
 * ==========================================================================
 */
class EvalException extends Exception {
                         // constructor of a new instance of EvalException
   public EvalException() {
          super("eval exception!");
   }
}
// --------------------------------------------------------------------------
public class PropositionalTheoremProver {
// ==========================================================================
    /**
     * This class PropositionalTheoremProver transforms the input string
     * into a tree of type Node.
     * The string encodes the tree as indicated by the grammar G.
     * @param  string: the string which encodes the given tree
     * @return the root of the tree generated by the parsing
     * @throws IOException or ParseException or EvalException.
     */
// Three static fields of the class PropositionalTheoremProver
                        // DIM: maximum number of the distinct variables
                        // in the propositional formula
   public static final int DIM = 4;                                 // <-- (1)
                        // At most DIM propositional variables in the
                        // propositional formula. By default, every
                        // element of tValues is initialized to "false"
   public static boolean [] tValues = new boolean [DIM];

   public static char c;
                        // inputLine is a sequence of characters,
                        // which are read one at a time, by
                        // the method read() of the class StringReader

   public static Node propParse(BufferedReader inputLine)
                                  throws IOException, ParseException {
      Node node1 = condParse(inputLine);
      if ( c == '>' ) {
         c = (char)inputLine.read();
         Node node2 = propParse(inputLine);
         return new BinaryNode(new Character('>'), node1, node2);
      } else { return node1; }
   }


   public static Node condParse(BufferedReader inputLine)
                                  throws IOException, ParseException {
      Node node1 = litParse(inputLine);
      if ( c == 'x' ) {
         c = (char)inputLine.read();
         Node node2 = condParse(inputLine);
```

```
         return new BinaryNode(new Character('x'), node1, node2);
      } else
      if ( c == '+' )   {
         c = (char)inputLine.read();
         Node node2 = condParse(inputLine);
         return new BinaryNode(new Character('+'), node1, node2);
      } else { return node1; }
   }

   public static Node litParse(BufferedReader inputLine)
                               throws IOException, ParseException {
      Node node = null;
      if ( c == '-' ) {
         c = (char)inputLine.read();
         Node node1 = litParse(inputLine);
         node = new UnaryNode(new Character('-'), node1);
      } else { node = atomParse(inputLine); };
      return node;
   }

   public static Node atomParse(BufferedReader inputLine)
                               throws IOException, ParseException {
      Node node1 = null;
      if (('0' <= c) && (c <= '9')) {
         node1 = new LeafNode(new Character(c));
         c = (char)inputLine.read();
         return node1;
      } else {
      if ( c=='(' ) {
         c = (char)inputLine.read();
         node1 = propParse(inputLine);
         if ( c==')' ) { c = (char)inputLine.read(); } // reads past ')'
         else { throw new ParseException(); };
         return node1;
         };
      };
      throw new ParseException();
   }

// ---------------------------------------------------------------------------
  /** Evaluation of the tree which represents the input boolean expression
   *  @param tree : tree to be evaluated
   *  @return the boolean value of the tree
   */
   public static boolean evalTree(Node tree)
                                throws EvalException {
// ---------------------- RECURSIVE VERSION --------------------------------
 if ( tree instanceof LeafNode ) {                                         //
                              // from the character to the integer: -48 //
     return tValues[((Character)(((LeafNode)tree).value)).charValue()-48];//
      } else                                                               //
      if ( tree instanceof UnaryNode ) {                                   //
           return !(evalTree(((UnaryNode)tree).left));                     //
      } else                                                               //
      if ( tree instanceof BinaryNode ) {                                  //
           boolean left  = evalTree(((BinaryNode)tree).left);              //
           boolean right = evalTree(((BinaryNode)tree).right);             //
```

```
        if (((Character)(((BinaryNode)tree).value)).charValue() == '+')//
           { return left || right; } else                                 //
        if (((Character)(((BinaryNode)tree).value)).charValue() == 'x')//
           { return left && right; } else                                 //
        if (((Character)(((BinaryNode)tree).value)).charValue() == '>')//
           { return (!left) || right; }                                   //
     };                                                                   //
     throw new EvalException();                                           //
// --------------------- END OF RECURSIVE VERSION ------------------------
  }
// -----------------------------------------------------------------------
   public static void main(String[] args)
                               throws IOException, ParseException {
      boolean answer = true;
      boolean bval   = true;

      System.out.println("Input a propositional formula according to the "
                        + "given grammar, please.");
      System.out.println("The variables should be between 0 and "
                        + (DIM-1) +".");
      Node tree = null;
      try {    // from String to StringReader and to BufferedReader
          BufferedReader inputLine =
                    new BufferedReader(new InputStreamReader(System.in));
          c = (char)inputLine.read();
          tree = propParse(inputLine);
          if (c != '\n') { throw new ParseException(); };       // <--- (2)
      } catch ( IOException e ) {
          e.printStackTrace();
          System.out.println("IOException!");
      } catch ( ParseException pex ) {
          System.out.println(pex.getMessage());
          return;
      };
  // ------ (begin 0)
      if (tree != null) {
          System.out.println("\nParse tree of the propositional formula:\n"
                             +tree);
          System.out.print("\nIs it a tautology? ");
/** -----------------------------------------------------------------------
 * The following fragment computes the dispositions for tValues[0],...,
 * tValues[DIM-1] from false,...,false to true,...,true until we finds one
 * disposition which makes bval == false, if any. In that case the given
 * formula is NOT a tautology. If, otherwise, all dispositions make
 * bval == true, then the given formula is a tautology.
 * -----------------------------------------------------------------------
*/
          for (int i = 0; i < (int)Math.pow(2,DIM); i++) {
             try {
               bval = evalTree(tree);
             } catch ( EvalException evalex ) {
                System.out.println(evalex.getMessage());
             };
             if (!bval) { answer = false; break; }
             else {
                for (int k = 0; k < DIM; k++) {
```

```
                    if (tValues[k]) { tValues[k] = false; }
                    else { tValues[k] = true; break; }
                }
             }
          };
// --------------------------------------------------------
          if (answer) { System.out.println("YES."); } else {
             System.out.println("NO.\nThe falsity assignment is:\n");
             for (int k = 0; k < DIM; k++) {
                System.out.print(k + "=" + tValues[k] + "; ");};
             System.out.println();
          }
// --------------------------------------------------------
       }
   // ------ (end 0)
   }
}
/**
 * input:  output:
 * ------------------------------------------------------------------------
 * javac PropositionalTheoremProver.java
 * java  PropositionalTheoremProver
 *
 *     Input a propositional formula according to the given grammar, please.
 *     The variables should be between 0 and 9.
 * 0>1>0
 *
 *     Parse of the propositional formula:
 *     (0 > (1 > 0))
 *
 *     Is it a tautology? YES.
 * ------------------------------------------------------------------------
 *     Input a propositional formula according to the given grammar, please.
 *     The variables should be between 0 and 9.
 * 0>-1
 *
 *     Parse of the propositional formula:
 *     (0 > (-1))
 *
 *     Is it a tautology? NO.
 *     The falsity assignment is:
 *
 *     0=true; 1=true; 2=false; 3=false;
 * ------------------------------------------------------------------------
 *     Input a propositional formula according to the given grammar, please.
 *     The variables should be between 0 and 9.
 * 0>(1+1)+-0x-2
 *
 *     Parse tree of the propositional formula:
 *     (0 > ((1 + 1) + ((-0) x (-2))))
 *
 *     Is it a tautology? NO.
 *     The falsity assignment is:
 *
 *     0=true; 1=false; 2=false; 3=false;
 * ------------------------------------------------------------------------
 */
```

The following program is like the above one, but it uses a graphical interface. It can be run by typing the following commands:

```
javac PropositionalTheoremProver.java
javac PropositionalTheoremProverGUI.java
java  PropositionalTheoremProverGUI
```

Here is the program for parsing a given propositional formula and checking whether or not it is a tautology.

```
/**
 * ==========================================================================
 *                       PROPOSITIONAL THEOREM PROVER
 *                          Graphical User Iterface
 *
 * Grammar G for the input propositional formula p
 *
 *    propositional formulas  p  ::=  c  | c>p               <-- propParse
 *    conditions              c  ::=  l  | l+c | lxc         <-- condParse
 *    literals                l  ::=  a  | -l                <-- litParse
 *    atoms                   a  ::= '0' | ... | '9' | (p)   <-- atomParse
 *
 * Propositional variables are named: '0', ..., '9'.
 * To speed up the theorem prover, the value of DIM gives us the maximum
 * number of propositional variables actually present in the formula.
 * If DIM = 4, then the variables are: '0', '1', '2', and '3'.
 * (see lines (1) below). DIM should be at most 10.
 *
 * - stands for not, + stands for or, x stands for and, > stands for implies
 * The grammar should not be left-recursive.
 *
 * Internal tree:
 * -------------
 * tree =
 * '0' | ... | '9' | (-tree) | (tree x tree) | (tree + tree) | (tree > tree)
 * (Leaf)     (Leaf)   (Unary)     (Binary)        (Binary)        (Binary)
 *
 * The value of a boolean expression:       value ::= true | false
 * ------------
 * Priorities: + and x bind more than >
 *             - binds more than + or x
 *             Right associativity for + and x
 *             Parentheses override priorities.
 * No blanks are allowed before, in between, and after the input string.
 *
 * A tree is input as a string generated by the grammar G and then
 * it is parse and evaluated.
 *
 * This program uses the class PropositionalTheoremProver in the file named
 * PropositionalTheoremProver.java. The file PropositionalTheoremProver.java
 * should be stored in the same folder where is stored this file
 * PropositionalTheoremProverGUI.java.
 * ------------
 * Line (2) below checks that the input propositional formula is terminated
 * by a carriage-return '\n'.
 *
 * @author Corrado Di Pietro. Modifications by Alberto Pettorossi.
```

```
 * ==========================================================================
 */

import java.io.*;
import javax.swing.JFrame;

// ------------------------------------------------------
public class PropositionalTheoremProverGUI extends JFrame {
    /** Creates new form PropositionalTheoremProverGUI */
    public PropositionalTheoremProverGUI() {
        initComponents();
    }
    /** This method initComponents() is called from within the constructor
     *  to initialize the form.
     *  WARNING: Do NOT modify this code. The content of this method is
     *  always regenerated by the Form Editor.
     */
    private void initComponents() {//GEN-BEGIN:initComponents
        jLabel1      = new javax.swing.JLabel();
        jLabel2      = new javax.swing.JLabel();
        jLabel3      = new javax.swing.JLabel();
        jLabel4      = new javax.swing.JLabel();
        txInForm     = new javax.swing.JTextField();
        txParsedForm = new javax.swing.JTextField();
        txAnswTaut   = new javax.swing.JTextField();
        btParse      = new javax.swing.JButton();
        menuBar      = new javax.swing.JMenuBar();
        fileMenu     = new javax.swing.JMenu();
        exitMenuItem = new javax.swing.JMenuItem();

        getContentPane().setLayout(null);
        setTitle("Propositional Theorem Prover");
        setFont(new java.awt.Font("Default", 0, 10));
        setLocationRelativeTo(null);
        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent evt) {
                exitForm(evt);
            }
        });
// -------------------------------------------------------------
        jLabel1.setBackground(new java.awt.Color(204, 204, 0));
        jLabel1.setFont(new java.awt.Font("Default", 0, 10));
        jLabel1.setText("<HTML>\n<PRE>\n"
        +" Grammar for the input propositional formula p:\n"
        +" prop:   p ::= c | c>p      cond:   c ::= l | l+c | lxc\n"
        +" lit:    l ::= a | -l"
        +"       atom:   a ::= '0' | ... | '3' | (p)\n"          // <--- (1)
        +" Internal tree: tree ::= '0' | ... | '3' | (-tree)\n"  // <--- (1)
        +"                        | (tree x tree) | (tree + tree) "
        +"| (tree > tree)\n"
        +" Abbreviations: f/false  t/true  -/not  x/and  +/or  >/implies"
        +"<PRE>\n</HTML>");
        jLabel1.setVerticalAlignment(javax.swing.SwingConstants.TOP);
        jLabel1.setOpaque(true);
        getContentPane().add(jLabel1);
        jLabel1.setBounds(5, 5, 425, 97);
```

```
// -----------------------------------------------------------
        jLabel2.setText("Input Formula:");
        getContentPane().add(jLabel2);
        jLabel2.setBounds(10, 104, 130, 16);
        getContentPane().add(txInForm);
        txInForm.setBounds(130, 104, 300, 20);
// -----------------------------------------------------------
        jLabel3.setText("Parsed Formula:");
        getContentPane().add(jLabel3);
        jLabel3.setBounds(10, 144, 130, 16);
        txParsedForm.setBackground(new java.awt.Color(204, 255, 204));
        getContentPane().add(txParsedForm);
        txParsedForm.setBounds(130, 144, 300, 20);
// -----------------------------------------------------------
        jLabel4.setText("Is it a tautology?");
        getContentPane().add(jLabel4);
        jLabel4.setBounds(10, 184, 140, 16);
        txAnswTaut.setBackground(new java.awt.Color(102, 255, 102));
        getContentPane().add(txAnswTaut);
        txAnswTaut.setBounds(130, 184, 300, 20);
// -----------------------------------------------------------
        btParse.setText("Parse and Evaluate");
        btParse.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btParseActionPerformed(evt);
                }
            });
        getContentPane().add(btParse);
        btParse.setBounds(160, 220, 158, 26);
// -----------------------------------------------------------
        fileMenu.setText("File");
        exitMenuItem.setText("Exit");
        exitMenuItem.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                exitMenuItemActionPerformed(evt);
                }
            });

        fileMenu.add(exitMenuItem);
        menuBar.add(fileMenu);
        setJMenuBar(menuBar);

        java.awt.Dimension screenSize =
             java.awt.Toolkit.getDefaultToolkit().getScreenSize();
        setBounds((screenSize.width-444)/2,
                  (screenSize.height-300)/2, 444, 300);
    } //GEN-END:initComponents

    private void btParseActionPerformed(java.awt.event.ActionEvent evt) {
      //GEN-FIRST:event_btParseActionPerformed
      // Add your handling code here:
      // This part is like the main of the class PropositionalTheoremProver
      // in the file PropositionalTheoremProver.java.
           boolean answer = true;
           boolean bval   = true;
           int DIM        = PropositionalTheoremProver.DIM;      // <--- (1)
```

```
          Node tree = null;
          try {    // from String to StringReader and to BufferedReader.
                   // Added an extra '\n' at the end of the input for
                   // terminating the input as in the
                   // PropositionalTheoremProver class.
           BufferedReader inputLine
            = new BufferedReader(
              new StringReader((txInForm.getText()) + '\n'));
           PropositionalTheoremProver.c = (char)inputLine.read();
           tree=PropositionalTheoremProver.propParse(inputLine);// <--- (2)
           if (PropositionalTheoremProver.c != '\n')
                   { throw new ParseException(); };
          } catch ( IOException e ) {
              e.printStackTrace();
              txParsedForm.setText("IOException!");
          } catch ( ParseException pex ){
              txParsedForm.setText(pex.getMessage());
              return;
          };
          // ------ (begin Parse and Evaluate)
          if (tree != null) {
              txParsedForm.setText(tree.toString());
// -------------------------------------------------------
          for (int i = 0; i < (int)Math.pow(2,DIM);
                          i++) {
             try {
               bval = PropositionalTheoremProver.evalTree(tree);
             } catch ( EvalException evalex ) {
               txAnswTaut.setText(evalex.getMessage());
             };
             if (!bval) { answer = false; break; }
             else {
                for (int k = 0; k < DIM; k++) {
                    if (PropositionalTheoremProver.tValues[k])
                         { PropositionalTheoremProver.tValues[k] = false; }
                    else { PropositionalTheoremProver.tValues[k] = true;
                           break; }
                }
             }
          };
//         for (int i=0; i<DIM; i++) { System.out.print(" "+i
 //              +": "+tValues[i]);};
// -------------------------------------------------------
          if (answer) { txAnswTaut.setText("YES."); } else {
             String outstring = "NO. False if: ";
             for (int k = 0; k < DIM; k++) {
             outstring = outstring + k + "="
                  + ((PropositionalTheoremProver.tValues[k])?"t; ":"f; ");}
             txAnswTaut.setText(outstring);
          }
// -------------------------------------------------------
          }
          // ------ (end Parse and Evaluate)
    } //GEN-LAST:event_btParseActionPerformed

    private void
```

```
                exitMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
      //GEN-FIRST:event_exitMenuItemActionPerformed
        System.exit(0);
    } //GEN-LAST:event_exitMenuItemActionPerformed

    /** Exit the Application */
    private void exitForm(java.awt.event.WindowEvent evt) {
      //GEN-FIRST:event_exitForm
        System.exit(0);
    } //GEN-LAST:event_exitForm

    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        new PropositionalTheoremProverGUI().show();
    }

    // Variables declaration - do not modify   //GEN-BEGIN:variables
    private javax.swing.JMenuItem exitMenuItem;
    private javax.swing.JMenu fileMenu;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JLabel jLabel3;
    private javax.swing.JLabel jLabel4;
    private javax.swing.JMenuBar menuBar;
    private javax.swing.JTextField txInForm;
    private javax.swing.JTextField txParsedForm;
    private javax.swing.JTextField txAnswTaut;
    private javax.swing.JButton btParse;
    // End of variables declaration            //GEN-END:variables
}

/**
 * input:              output:
 * ----------------------------------------------------------------------
 * javac PropositionalTheoremProver.java       // This compilation is needed
 * javac PropositionalTheoremProverGUI.java
 * java  PropositionalTheoremProverGUI
 *
 *                (see the figures below)
 * ----------------------------------------------------------------------
 */
```