CMPE534 Automated Deduction

Term Project

Theorem Prover for Propositional Logic

135394 Eser Laroux Ozvataf
13.01.2014
Eastern Mediterranean University

CONTENTS

Theorem Prover for Prop. Logic

1 INTRODUCTION

This term project is designed to be a basic interpreter for proving theorems in propositional logic. It is basically accepts propositional statements in a specific syntax in order to prove its validity. For non-valid statements, all falsifying valuations will be extracted from counter-examples.

1.1 REQUIREMENTS:

- Reading a propositional formula to be proven from a file.
- Proving given proposition in Gentzen System G'.
- Output of deduction tree.
- Validity check of given proposition.
- Output of falsifying valuations for non-valid proposition.

1.2 ASSUMPTIONS:

- Since Gentzen System G' is going to be used for proving, the theorem-prover needs to recognize sequents properly.

- To construct the deduction tree for each proposition, system will need customized data structures supports branching.

- Also an evaluator is needed for output of falsifying valuations which will be extracted from counter-example nodes.

2 SOLUTION OF THE PROBLEM

2.1 ENVIRONMENT

The solution is implemented in C# programming language and its compiled binary works as CLI (command-line interface) application on Windows, OS X and GNU/Linux platforms.

In C#'s default toolkit, there is no support to work with language of propositional logic. Therefore, it was first thing to do construct a parser which will recognize the propositional symbols, connectives and some auxiliary symbols.

2.2 Recognition of Propositional Language

These symbols are defined in a "registry" with its token characteristics which is being used by parser (see Table 1). The registry can be extendible during both compile-time and run-time.

Symbols	Type	Precedence	Is Right Associative	Value	Closes With
!, not	Not	1	Yes		
&, and	And	4	No		
|, or	Or	5	No		
>, implies	Implication	6	No		
=, equals	Equivalence	6	No		
[	Parenthesis	101	No		]
(	Parenthesis	101	No		)
f, 0, false	Constant	0	No	False	
t, 1, true	Constant	0	No	True	

Table 1: Registry

Each type specified in registry is coupled with a class definition in the code. It is a design consideration which allows extendibility of the registry by dynamically linking produced executable from a new .NET assembly.

All of these "proposition member" types implements IMember interface in class design. But some of the types inherits its properties and methods from their related base classes such as Connective and Symbol.

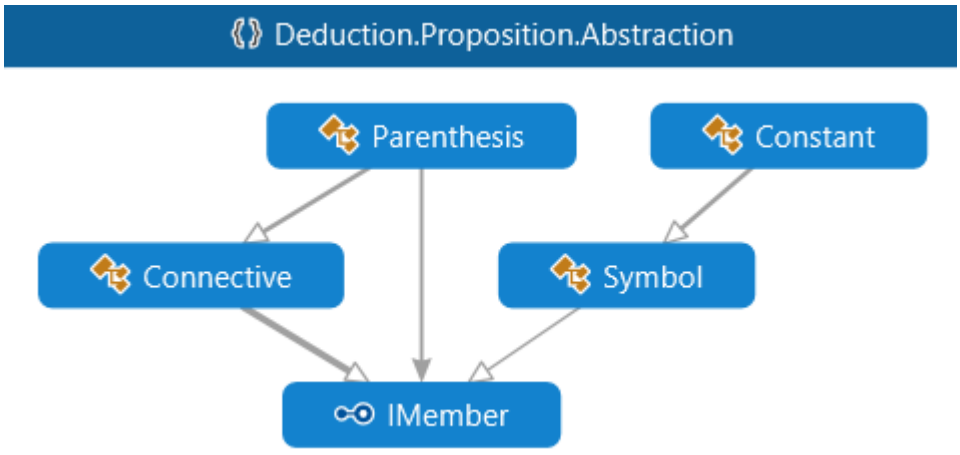

Figure 1: Class Diagram

2.2.1 Data Structure

Each Symbol is basically a propositional symbol. Constant inherits from Symbol, but it also a constant value (true or false according to logical setup).

Propositional connectives which are defined in the registry (Not, And, Or, Implication, Equivalence and Parenthesis) derived from abstract class named Connective which is designed to be n-ary.

Apart from the other types, Connective is some kind of a tree structure which contains some other propositional member types in it.

For example, “A ∧ (B ∨ C)” proposition will be structured like in Figure 2.

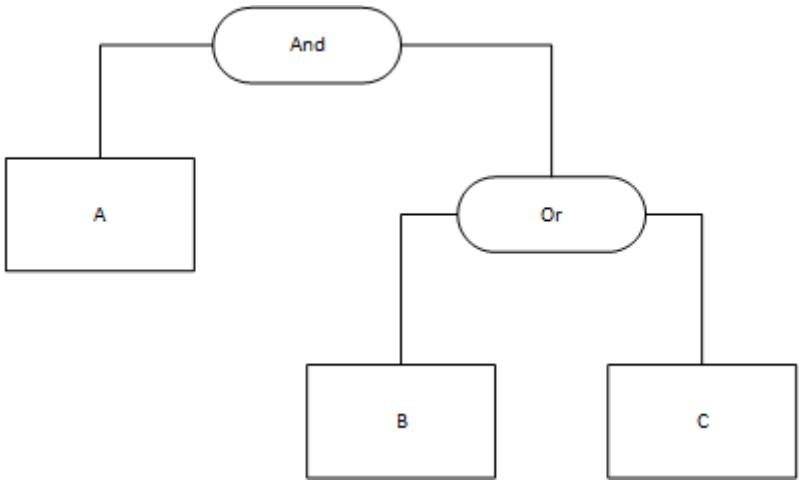

Figure 2: Data Structure of Connectives

Since it is complete a class in code, connectives also have some methods named Operation and Simplify.

Operation method basically evaluates given parameters within the function of related connective. For example: the method call And.Operation(true, false) returns false since “true ^ false” is false.

Simplify method derives a new propositional member if simplification is possible. For example: the method call Or.Simplify(A, true) returns true since “any value ∨ true” is always true.

2.2.2 Parsing Propositional Formula

To parse user input according to registry, shunting-yard algorithm is implemented to convert infix notation to desired data structure.

It is done in two phases. Lexical analyzer identifies tokens and links them to their related definition in the registry.

Then, parser constructs abstract syntax tree (AST) to produce data structures.

2.2.3 Parsing Sequents

Reading sequents are separated from the main parser and handled by a different component called SequentReader which is designed as the factory class of Sequents.

Sequents are basically splits two sides of sequent and considers these sides are collection of propositional members.

$P_1, P_2, P_3, \ldots, P_n \rightarrow Q_1, Q_2, Q_3, \ldots, Q_m$

Figure 3: Sample Sequent

2.3 Proving with Gentzen System G'

In the design, proving procedure is done by components named Prover and Falsifier. Prover will execute the theorem-prover algorithm and then all available counter-example branches need to be collected to execute Falsifier which will list all falsifying valuations.

2.3.1 Prover

Prover procedure has 3 methods, Search, Expand and ScanRules.

Search basically operates a breadth-first search starting from the root node to the deepest nodes of sequent tree. At the beginning of the procedure, there will be only the sequent parsed from user input. Search procedure will fold all possible branches of the sequent, which is at the top of the queue, with the help of the expand procedure, then it will add these branches to the bottom of the queue.

Expand procedure scans both sides of the current sequent and applies available rules in sequent order. In the design, these rules are divided in groups which occurs branching and replacing. Branching rules forks sequent node in two and only one branching rule is going to be applied at the same expand call. Replacing rules can be called at the same time, they will only replace the original sequent member with the expanded one.

ScanRules is called by Expand method, it is called for both sides of sequent since the rules changes depending on the side.

The operation of these procedures stops digging branches if the related branch is an axiom or atomic.

2.3.2 Falsifier

Falsifier gets an atomic sequent as input and tries to falsify it. First, it generates all possible valuations of propositional symbols which are extracted in sequent. Then, it eliminates non-falsifying valuations in order to find the falsifying ones.

Since only counter-examples are passed to falsifier, performance effect of brute-forcing all valuations is minimized.

2.4 User Interaction

The system uses Command Line Interface in order to interact with the user.

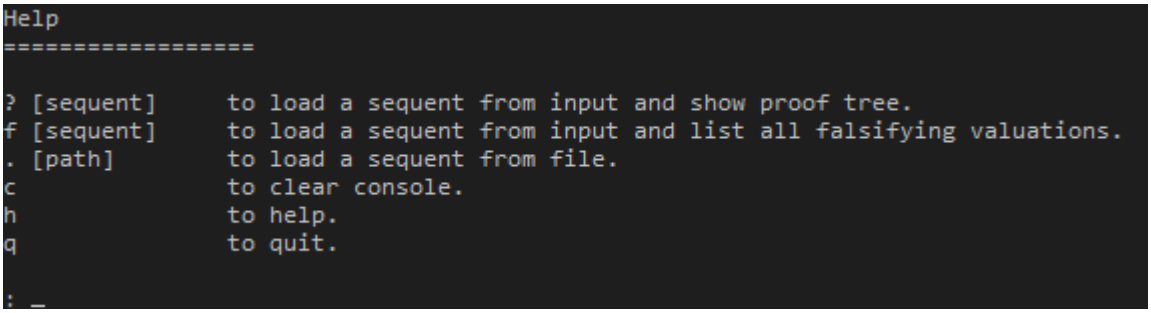

Figure 4: Command Line Interface

A help and command prompt will be displayed at program start. User actions are determined by a single character command and following parameters.

2.4.1 Prove command

To try to prove a sequent "?" command is needed to be executed.

Sample execution #1:

```
: ? A > B

Deduction tree of: A > B


sequent = -> (A > B)

sequent = A -> B

          ** counter-example node **


Formula is not valid, details below.

+ Counter-examples:

  .. A -> B


+ Falsifying valuations:

  .. Valuation #1:

  ..                A -> t

  ..                B -> f
```

This output shows as given formula "A > B" (A implies B) has one counter-example node "A -> B" and valuation A = true, B = false makes the formula falsified.

Sample execution #2:

```
: ? A -> A | B
Deduction tree of: A -> A | B


sequent = A -> (A | B)
sequent = A -> A, B
          ** axiom node **


Formula is valid.
```

This output shows as given formula is valid.

2.4.2 Load from file command

To try to prove set of sequents, which is going to be loaded from a file, "." command is needed to be executed with the file path.

Sample file content:

```
((!Q & P) & (P > Q) & (T > S) & (R and S)) > T
(((Q & P) & ((P > Q) & ((T > S) & (R | S)))) > T)
A | !A
A & !A
(A implies B) & (B > A)
A = B
// A equals B
```

Empty lines and lines prefixed with the // and # characters, will be skipped.

Sample execution:

```
: . prop.txt
Reading file: prop.txt


------------------------------------------------
Deduction tree of: ((!Q & P) & (P > Q) & (T > S) & (R and S)) > T


sequent = -> (((((!Q & P) & (P > Q)) & (T > S)) & (R & S)) > T)
sequent = ((((!Q & P) & (P > Q)) & (T > S)) & (R & S)) -> T
sequent = (((!Q & P) & (P > Q)) & (T > S)), (R & S) -> T
```

```
sequent = R, S, ((!Q & P) & (P > Q)), (T > S) -> T
        sequent = (!Q & P), (P > Q), R, S -> T, T
                sequent = !Q, P, R, S -> P, T, T
                          ** axiom node **

                sequent = Q, !Q, P, R, S -> T, T
                sequent = Q, P, R, S -> Q, T, T
                          ** axiom node **

        sequent = S, (!Q & P), (P > Q), R, S -> T
                sequent = !Q, P, S, R, S -> P, T
                          ** axiom node **

                sequent = Q, !Q, P, S, R, S -> T
                sequent = Q, P, S, R, S -> Q, T
                          ** axiom node **

Formula is valid.

------------------------------------------------
Deduction tree of: (((Q & P) & ((P > Q) & ((T > S) & (R | S)))) > T)

sequent = -> (((Q & P) & ((P > Q) & ((T > S) & (R | S)))) > T)
sequent = ((Q & P) & ((P > Q) & ((T > S) & (R | S)))) -> T
sequent = (Q & P), ((P > Q) & ((T > S) & (R | S))) -> T
sequent = (P > Q), ((T > S) & (R | S)), Q, P -> T
        sequent = (T > S), (R | S), Q, P -> P, T
                  ** axiom node **

        sequent = (T > S), (R | S), Q, Q, P -> T
                sequent = (R | S), Q, Q, P -> T, T
                        sequent = R, Q, Q, P -> T, T
                                  ** counter-example node **

                        sequent = S, Q, Q, P -> T, T
```

```
                                ** counter-example node **


              sequent = S, (R | S), Q, Q, P -> T
                      sequent = R, S, Q, Q, P -> T
                                ** counter-example node **


                      sequent = S, S, Q, Q, P -> T
                                ** counter-example node **


Formula is not valid, details below.
+ Counter-examples:
  .. R, Q, Q, P -> T, T
  .. S, Q, Q, P -> T, T
  .. R, S, Q, Q, P -> T
  .. S, S, Q, Q, P -> T


+ Falsifying valuations:
  .. Valuation #1:
  ..                R -> t
  ..                Q -> t
  ..                P -> t
  ..                T -> f
  .. Valuation #2:
  ..                S -> t
  ..                Q -> t
  ..                P -> t
  ..                T -> f
  .. Valuation #3:
  ..                R -> t
  ..                S -> t
  ..                Q -> t
  ..                P -> t
  ..                T -> f
  .. Valuation #4:
  ..                S -> t
```

```
  ..                  Q -> t
  ..                  P -> t
  ..                  T -> f



------------------------------------------------
Deduction tree of: A | !A

sequent = -> (A | !A)
sequent = -> A, !A
sequent = A -> A
          ** axiom node **

Formula is valid.

------------------------------------------------
Deduction tree of: A & !A

sequent = -> (A & !A)
        sequent = -> A
                  ** counter-example node **

        sequent = -> !A
        sequent = A ->
                  ** counter-example node **

Formula is not valid, details below.
+ Counter-examples:
  .. -> A
  .. A ->

+ Falsifying valuations:
  .. Valuation #1:
  ..                  A -> f
  .. Valuation #2:
```

```
  ..                  A -> t



------------------------------------------------
Deduction tree of: (A implies B) & (B > A)


sequent = -> ((A > B) & (B > A))
        sequent = -> (A > B)
        sequent = A -> B
                  ** counter-example node **


        sequent = -> (B > A)
        sequent = B -> A
                  ** counter-example node **


Formula is not valid, details below.
+ Counter-examples:
  .. A -> B
  .. B -> A


+ Falsifying valuations:
  .. Valuation #1:
  ..                  A -> t
  ..                  B -> f
  .. Valuation #2:
  ..                  B -> t
  ..                  A -> f



------------------------------------------------
Deduction tree of: A = B


sequent = -> (A = B)
        sequent = -> (A > B)
        sequent = A -> B
```

```
                  ** counter-example node **


        sequent = -> (B > A)

        sequent = B -> A

                  ** counter-example node **


Formula is not valid, details below.

+ Counter-examples:

  .. A -> B

  .. B -> A


+ Falsifying valuations:

  .. Valuation #1:

  ..                  A -> t

  ..                  B -> f

  .. Valuation #2:

  ..                  B -> t

  ..                  A -> f
```

This output shows as given formula is valid.

2.4.3 Clear screen command

To clear the contents of the command line, “c” command is needed to be executed.

2.4.4 Help command

To display brief description of commands, “h” command is needed to be executed.

2.4.5 Quit command

To terminate the execution of the program, “q” command is needed to be executed.

3 CONCLUSION

To produce deduction tree of given propositions, a system is designed in an imperative language environment. Extensibility and modularity were the key considerations during the system design.

Apart from C#, this design started to be implemented in C++ and Java. And all the work are open-source and stored on online repository (https://github.com/larukedi/cmpe534) with 45 commits since 29th November 2013.

4 SOURCE CODE

```
// GentzenPrime/Abstraction/BranchDistribution.cs

namespace Deduction.GentzenPrime.Abstraction
{
    public enum BranchDistribution : int
    {
        ToLeft,
        ToRight,
        Both
    }
}

// GentzenPrime/Abstraction/RuleOperation.cs

using Deduction.Proposition.Abstraction;

namespace Deduction.GentzenPrime.Abstraction
{
    public struct RuleOperation
    {
        private readonly BranchDistribution branchDistribution;
        private readonly SequentDirection sequentDirection;
        private readonly IMember member;
        private readonly bool isModified;

        public RuleOperation(BranchDistribution branchDistribution, SequentDirection
sequentDirection, IMember member, bool isModified)
        {
            this.branchDistribution = branchDistribution;
            this.sequentDirection = sequentDirection;
            this.member = member;
            this.isModified = isModified;
        }

        public BranchDistribution BranchDistribution
        {
            get
            {
                return this.branchDistribution;
            }
        }

        public SequentDirection SequentDirection
        {
            get
            {
                return this.sequentDirection;
            }
        }

        public IMember Member
        {
            get
            {
                return this.member;
```

```
            }
        }

        public bool IsModified
        {
            get
            {
                return this.isModified;
            }
        }
    }
}


// GentzenPrime/Abstraction/Sequent.cs

using System;
using System.Collections.Generic;
using Deduction.Proposition.Abstraction;

namespace Deduction.GentzenPrime.Abstraction
{
    public class Sequent : ICloneable
    {
        public const string SEQUENT_SEPARATOR = "->";
        public const string ITEM_SEPARATOR = ",";

        protected List<IMember> left;
        protected List<IMember> right;

        public Sequent()
        {
            this.left = new List<IMember>();
            this.right = new List<IMember>();
        }

        public List<IMember> Left
        {
            get
            {
                return this.left;
            }
        }

        public List<IMember> Right
        {
            get
            {
                return this.right;
            }
        }

        public bool IsAxiom()
        {
            if (this.Left.Count > 0)
            {
                foreach (IMember leftMember in this.Left)
                {
```

```
                    if (leftMember is Constant && (leftMember as Constant).Value ==
false)
                    {
                        return true;
                    }

                    foreach (IMember rightMember in this.Right)
                    {
                        if (rightMember is Constant && (rightMember as Constant).Value ==
true)
                        {
                            return true;
                        }

                        if (leftMember.Equals(rightMember))
                        {
                            return true;
                        }
                    }
                }
            }
            else
            {
                foreach (IMember rightMember in this.Right)
                {
                    if (rightMember is Constant && (rightMember as Constant).Value ==
true)
                    {
                        return true;
                    }
                }
            }

            return false;
        }

        public bool IsAtomic()
        {
            foreach (IMember leftMember in this.Left)
            {
                if (!leftMember.IsAtomic)
                {
                    return false;
                }
            }

            foreach (IMember rightMember in this.Right)
            {
                if (!rightMember.IsAtomic)
                {
                    return false;
                }
            }

            return true;
        }

        public object Clone()
```

```
        {
            Sequent clone = new Sequent();

            for (int i = 0; i < this.Left.Count; i++)
            {
                clone.Left.Add(this.Left[i].Clone() as IMember);
            }

            for (int i = 0; i < this.Right.Count; i++)
            {
                clone.Right.Add(this.Right[i].Clone() as IMember);
            }

            return clone;
        }
    }
}


// GentzenPrime/Abstraction/SequentDirection.cs

namespace Deduction.GentzenPrime.Abstraction
{
    public enum SequentDirection : int
    {
        Left,
        Right
    }
}

// GentzenPrime/Abstraction/Tree{T}.cs

using System;
using System.Collections.Generic;

namespace Deduction.GentzenPrime.Abstraction
{
    public class Tree<T>
    {
        protected T content;
        protected Tree<T> parent;
        protected readonly LinkedList<Tree<T>> children;

        public Tree(T content, Tree<T> parent = null)
        {
            this.content = content;
            this.parent = parent;
            this.children = new LinkedList<Tree<T>>();
        }

        public T Content
        {
            get
            {
                return this.content;
            }
            set
            {
```

```
            this.content = value;
        }
    }

    public Tree<T> Parent
    {
        get
        {
            return this.parent;
        }
        set
        {
            this.parent = value;
        }
    }

    public LinkedList<Tree<T>> Children
    {
        get
        {
            return this.children;
        }
    }

    public Tree<T> this[int index]
    {
        get
        {
            foreach (Tree<T> child in this.Children)
            {
                if (--index == 0)
                {
                    return child;
                }
            }

            return null;
        }
    }

    public void AddChild(T content)
    {
        this.children.AddLast(new Tree<T>(content, this));
    }

    public void Traverse(Action<T, int> action, int depth = 0)
    {
        action(this.Content, depth);
        foreach (Tree<T> child in this.Children)
        {
            child.Traverse(action, depth + 1);
        }
    }

    public void BreadthFirstNodes(Queue<Tree<T>> queue)
    {
        foreach (Tree<T> child in this.Children)
        {
```

```
                queue.Enqueue(child);
            }

            foreach (Tree<T> child in this.Children)
            {
                child.BreadthFirstNodes(queue);
            }
        }
    }
}


// GentzenPrime/Processors/Falsifier.cs

using System.Collections.Generic;
using Deduction.GentzenPrime.Abstraction;
using Deduction.Proposition.Abstraction;
using Deduction.Proposition.Parsing;
using Deduction.Proposition.Processors;

namespace Deduction.GentzenPrime.Processors
{
    public class Falsifier
    {
        protected readonly Registry registry;

        public Falsifier(Registry registry)
        {
            this.registry = registry;
        }

        public List<Dictionary<string, string>> Falsify(Sequent sequent)
        {
            List<Dictionary<string, string>> result = new List<Dictionary<string,
string>>();

            Substitutor substitutor = new Substitutor(this.registry);
            List<string> table = new List<string>();

            foreach (IMember member in sequent.Left)
            {
                substitutor.GetSymbols(member, ref table);
            }

            foreach (IMember member in sequent.Right)
            {
                substitutor.GetSymbols(member, ref table);
            }

            List<Dictionary<string, string>> allPossibleValuations = new
List<Dictionary<string, string>>();
            for (int i = 0; i < (1 << table.Count); i++)
            {
                Dictionary<string, string> row = new Dictionary<string, string>();
                for (int j = 0; j < table.Count; j++)
                {
                    row.Add(table[j], (i & (1 << j)) > 0 ? "t" : "f");
                }
```

```
                allPossibleValuations.Add(row);
            }

            foreach (Dictionary<string, string> valuation in allPossibleValuations)
            {
                bool leftSide = true;
                foreach (IMember member in sequent.Left)
                {
                    IMember resultMember = substitutor.Substitute(member, valuation);
                    if (resultMember is Constant)
                    {
                        leftSide = leftSide && (resultMember as Constant).Value;
                    }
                }

                bool rightSide = false;
                foreach (IMember member in sequent.Right)
                {
                    IMember resultMember = substitutor.Substitute(member, valuation);
                    if (resultMember is Constant)
                    {
                        rightSide = rightSide || (resultMember as Constant).Value;
                    }
                }

                if (leftSide && !rightSide)
                {
                    result.Add(valuation);
                }
            }

            return result;
        }
    }
}


// GentzenPrime/Processors/Prover.cs

using System.Collections.Generic;
using Deduction.GentzenPrime.Abstraction;
using Deduction.Proposition.Abstraction;
using Deduction.Proposition.Members;
using Deduction.Proposition.Parsing;

namespace Deduction.GentzenPrime.Processors
{
    public class Prover
    {
        protected readonly Registry registry;

        public Prover(Registry registry)
        {
            this.registry = registry;
        }

        public void Search(Tree<Sequent> tree)
```

```
        {
            Queue<Tree<Sequent>> queue = new Queue<Tree<Sequent>>();
            queue.Enqueue(tree);

            while (queue.Count > 0)
            {
                Tree<Sequent> current = queue.Dequeue();

                if (current.Content.IsAxiom())
                {
                    continue;
                }

                this.Expand(current);

                foreach (Tree<Sequent> node in current.Children)
                {
                    queue.Enqueue(node);
                }
            }
        }

        public void Expand(Tree<Sequent> sequentNode)
        {
            List<RuleOperation> ruleOperations = new List<RuleOperation>();
            bool branched = false;

            this.ScanForLeftRules(ref ruleOperations, ref branched,
sequentNode.Content.Left);
            this.ScanForRightRules(ref ruleOperations, ref branched,
sequentNode.Content.Right);

            Sequent leftBranch = new Sequent();
            Sequent rightBranch = new Sequent();
            int modifiedMembers = 0;
            int branchDiffs = 0;

            foreach (RuleOperation ruleOperation in ruleOperations)
            {
                if (ruleOperation.IsModified)
                {
                    modifiedMembers++;

                    if (ruleOperation.BranchDistribution != BranchDistribution.Both)
                    {
                        branchDiffs++;
                    }
                }

                if (ruleOperation.BranchDistribution == BranchDistribution.Both ||
ruleOperation.BranchDistribution == BranchDistribution.ToLeft)
                {
                    if (ruleOperation.SequentDirection == SequentDirection.Left)
                    {
                        leftBranch.Left.Add(ruleOperation.Member);
                    }
                    else
                    {
```

```
                        leftBranch.Right.Add(ruleOperation.Member);
                    }
                }

                if (ruleOperation.BranchDistribution == BranchDistribution.Both ||
ruleOperation.BranchDistribution == BranchDistribution.ToRight)
                {
                    if (ruleOperation.SequentDirection == SequentDirection.Left)
                    {
                        rightBranch.Left.Add(ruleOperation.Member);
                    }
                    else
                    {
                        rightBranch.Right.Add(ruleOperation.Member);
                    }
                }
            }

            if (modifiedMembers > 0)
            {
                sequentNode.AddChild(leftBranch);

                if (branchDiffs > 0)
                {
                    sequentNode.AddChild(rightBranch);
                }
            }
        }

        protected void ScanForLeftRules(ref List<RuleOperation> operations, ref bool
branched, List<IMember> members)
        {
            foreach (IMember member in members)
            {
                if (member is Not)
                {
                    Not not = member as Not;
                    operations.Insert(0, new RuleOperation(BranchDistribution.Both,
SequentDirection.Right, not.Parameters[0], true));
                }
                else if (member is And)
                {
                    And and = member as And;
                    operations.InsertRange(0,
                        new RuleOperation[]
                        {
                            new RuleOperation(BranchDistribution.Both,
SequentDirection.Left, and.Parameters[0], true),
                            new RuleOperation(BranchDistribution.Both,
SequentDirection.Left, and.Parameters[1], true)
                        }
                    );
                }
                else if (!branched && member is Or)
                {
                    Or or = member as Or;
                    operations.Insert(0, new RuleOperation(BranchDistribution.ToLeft,
SequentDirection.Left, or.Parameters[0], true));
```

```
                    operations.Insert(0, new RuleOperation(BranchDistribution.ToRight,
SequentDirection.Left, or.Parameters[1], true));

                    branched = true;
                }
                else if (!branched && member is Implication)
                {
                    Implication implication = member as Implication;
                    operations.Insert(0, new RuleOperation(BranchDistribution.ToLeft,
SequentDirection.Right, implication.Parameters[0], true));
                    operations.Insert(0, new RuleOperation(BranchDistribution.ToRight,
SequentDirection.Left, implication.Parameters[1], true));

                    branched = true;
                }
                else if (member is Equivalence)
                {
                    Equivalence equivalence = member as Equivalence;
                    operations.InsertRange(0,
                        new RuleOperation[]
                        {
                            new RuleOperation(BranchDistribution.Both,
SequentDirection.Left, new Implication(equivalence.Parameters[0],
equivalence.Parameters[1]), true),
                            new RuleOperation(BranchDistribution.Both,
SequentDirection.Left, new Implication(equivalence.Parameters[1],
equivalence.Parameters[0]), true)
                        }
                    );
                }
                else
                {
                    operations.Add(new RuleOperation(BranchDistribution.Both,
SequentDirection.Left, member, false));
                }
            }
        }

        protected void ScanForRightRules(ref List<RuleOperation> operations, ref bool
branched, List<IMember> members)
        {
            foreach (IMember member in members)
            {
                if (member is Not)
                {
                    Not not = member as Not;
                    operations.Insert(0, new RuleOperation(BranchDistribution.Both,
SequentDirection.Left, not.Parameters[0], true));
                }
                else if (!branched && member is And)
                {
                    And and = member as And;
                    operations.Insert(0, new RuleOperation(BranchDistribution.ToLeft,
SequentDirection.Right, and.Parameters[0], true));
                    operations.Insert(0, new RuleOperation(BranchDistribution.ToRight,
SequentDirection.Right, and.Parameters[1], true));

                    branched = true;
```

```
                }
                else if (member is Or)
                {
                    Or or = member as Or;
                    operations.InsertRange(0,
                        new RuleOperation[]
                        {
                            new RuleOperation(BranchDistribution.Both,
SequentDirection.Right, or.Parameters[0], true),
                            new RuleOperation(BranchDistribution.Both,
SequentDirection.Right, or.Parameters[1], true)
                        }
                    );
                }
                else if (member is Implication)
                {
                    Implication implication = member as Implication;
                    operations.Insert(0, new RuleOperation(BranchDistribution.Both,
SequentDirection.Left, implication.Parameters[0], true));
                    operations.Insert(0, new RuleOperation(BranchDistribution.Both,
SequentDirection.Right, implication.Parameters[1], true));
                }
                else if (!branched && member is Equivalence)
                {
                    Equivalence equivalence = member as Equivalence;
                    operations.Insert(0, new RuleOperation(BranchDistribution.ToLeft,
SequentDirection.Right, new Implication(equivalence.Parameters[0],
equivalence.Parameters[1]), true));
                    operations.Insert(0, new RuleOperation(BranchDistribution.ToRight,
SequentDirection.Right, new Implication(equivalence.Parameters[1],
equivalence.Parameters[0]), true));

                    branched = true;
                }
                else
                {
                    operations.Add(new RuleOperation(BranchDistribution.Both,
SequentDirection.Right, member, false));
                }
            }
        }
    }
}


// GentzenPrime/Processors/SequentReader.cs

using System;
using Deduction.GentzenPrime.Abstraction;
using Deduction.Proposition.Parsing;

namespace Deduction.GentzenPrime.Processors
{
    public class SequentReader
    {
        public static Sequent Read(Registry registry, string prop)
        {
```

```
            string[] propParts = prop.Split(new string[] { Sequent.SEQUENT_SEPARATOR },
2, StringSplitOptions.None);

            if (propParts.Length < 2)
            {
                propParts = new string[] { "", propParts[0] };
            }

            Sequent sequent = new Sequent();
            Parser parser = new Parser(registry);

            for (int i = 0; i < 2; i++)
            {
                string[] sequentParts = propParts[i].Split(new string[]
{ Sequent.ITEM_SEPARATOR }, StringSplitOptions.RemoveEmptyEntries);

                foreach (string sequentPart in sequentParts)
                {
                    Lexer lexer = new Lexer(registry, sequentPart);
                    var tokens = lexer.Analyze();
                    var root = parser.Parse(tokens);

                    if (i == 0)
                    {
                        sequent.Left.Add(root);
                    }
                    else
                    {
                        sequent.Right.Add(root);
                    }
                }
            }

            return sequent;
        }
    }
}


// Proposition/Abstraction/Connective.cs

using System;
using System.Collections.Generic;
using Deduction.Proposition.Parsing;

namespace Deduction.Proposition.Abstraction
{
    public abstract class Connective : IMember
    {
        protected readonly List<IMember> parameters;

        public Connective(params IMember[] parameters)
        {
            this.parameters = new List<IMember>(parameters);
        }

        public Connective(IEnumerable<IMember> parameters)
        {
```

```
    this.parameters = new List<IMember>(parameters);
}

public List<IMember> Parameters
{
    get
    {
        return this.parameters;
    }
}

public bool IsAtomic
{
    get
    {
        return false;
    }
}

public abstract int ParameterCount
{
    get;
}

public abstract bool Operation(bool[] values);

public virtual IMember Simplify(Registry registry)
{
    return null;
}

public override bool Equals(object obj)
{
    Connective connective = obj as Connective;
    if (connective == null || !this.GetType().IsInstanceOfType(connective))
    {
        return false;
    }

    int length = this.Parameters.Count;
    if (length != connective.Parameters.Count)
    {
        return false;
    }

    for (int i = length - 1; i >= 0; i--)
    {
        if (!this.Parameters[i].Equals(connective.Parameters[i]))
        {
            return false;
        }
    }

    return true;
}

public override int GetHashCode()
{
```

```
            unchecked
            {
                return this.GetType().GetHashCode() + this.Parameters.GetHashCode();
            }
        }

        public object Clone()
        {
            Connective clone = Activator.CreateInstance(this.GetType(), new IMember[0])
as Connective;

            for (int i = 0; i < clone.ParameterCount; i++)
            {
                clone.Parameters.Add(this.Parameters[i].Clone() as IMember);
            }

            return clone;
        }
    }
}


// Proposition/Abstraction/Constant.cs

using System;

namespace Deduction.Proposition.Abstraction
{
    public class Constant : Symbol
    {
        protected readonly bool value;

        public Constant(string letter, bool value) : base(letter)
        {
            this.value = value;
        }

        public bool Value
        {
            get
            {
                return this.value;
            }
        }

        public override bool Equals(object obj)
        {
            Constant constant = obj as Constant;
            if (constant == null)
            {
                return false;
            }

            return (this.Value == constant.Value);
        }

        public override int GetHashCode()
        {
```

```
            return this.value.GetHashCode();
        }

        public override object Clone()
        {
            return Activator.CreateInstance(this.GetType(), this.Letter, this.Value);
        }
    }
}


// Proposition/Abstraction/IMember.cs

using System;

namespace Deduction.Proposition.Abstraction
{
    public interface IMember : ICloneable
    {
        bool IsAtomic { get; }
    }
}


// Proposition/Abstraction/Parenthesis.cs

using System.Collections.Generic;

namespace Deduction.Proposition.Abstraction
{
    public class Parenthesis : Connective
    {
        public Parenthesis(params IMember[] parameters)
            : base(parameters)
        {
        }

        public Parenthesis(IEnumerable<IMember> parameters)
            : base(parameters)
        {
        }

        public override int ParameterCount
        {
            get
            {
                return 0;
            }
        }

        public override bool Operation(bool[] values)
        {
            return false;
        }
    }
}
```

```
// Proposition/Abstraction/Symbol.cs

using System;

namespace Deduction.Proposition.Abstraction
{
    public class Symbol : IMember
    {
        protected readonly string letter;

        public Symbol(string letter)
        {
            this.letter = letter;
        }

        public string Letter
        {
            get
            {
                return this.letter;
            }
        }

        public bool IsAtomic
        {
            get
            {
                return true;
            }
        }

        public override bool Equals(object obj)
        {
            Symbol symbol = obj as Symbol;
            if (symbol == null)
            {
                return false;
            }

            return (this.Letter == symbol.Letter);
        }

        public override int GetHashCode()
        {
            return this.letter.GetHashCode();
        }

        public virtual object Clone()
        {
            return Activator.CreateInstance(this.GetType(), this.Letter);
        }
    }
}


// Proposition/Members/And.cs

using System.Collections.Generic;
```

```
using Deduction.Proposition.Abstraction;
using Deduction.Proposition.Parsing;

namespace Deduction.Proposition.Members
{
    public class And : Connective
    {
        public And(params IMember[] parameters)
            : base(parameters)
        {
        }

        public And(IEnumerable<IMember> parameters)
            : base(parameters)
        {
        }

        public override int ParameterCount
        {
            get
            {
                return 2;
            }
        }

        public override bool Operation(bool[] values)
        {
            return values[0] && values[1];
        }

        public override IMember Simplify(Registry registry)
        {
            Symbol firstParameter = this.Parameters[0] as Symbol;
            RegistryMember firstParameterSymbol = null;
            if (firstParameter != null)
            {
                firstParameterSymbol =
registry.GetMemberBySymbolChar(firstParameter.Letter);
                if (firstParameterSymbol != null && firstParameterSymbol.Value == false)
                {
                    return new Constant("f", false);
                }
            }

            Symbol secondParameter = this.Parameters[1] as Symbol;
            RegistryMember secondParameterSymbol = null;
            if (secondParameter != null)
            {
                secondParameterSymbol =
registry.GetMemberBySymbolChar(secondParameter.Letter);
                if (secondParameterSymbol != null && secondParameterSymbol.Value ==
false)
                {
                    return new Constant("f", false);
                }
            }

            if (firstParameter != null && secondParameter != null)
```

```
            {
                if (firstParameter.Letter == secondParameter.Letter)
                {
                    return new Symbol(firstParameter.Letter);
                }
            }

            return null;
        }
    }
}


// Proposition/Members/Equivalence.cs

using System.Collections.Generic;
using Deduction.Proposition.Abstraction;

namespace Deduction.Proposition.Members
{
    public class Equivalence : Connective
    {
        public Equivalence(params IMember[] parameters)
            : base(parameters)
        {
        }

        public Equivalence(IEnumerable<IMember> parameters)
            : base(parameters)
        {
        }

        public override int ParameterCount
        {
            get
            {
                return 2;
            }
        }

        public override bool Operation(bool[] values)
        {
            return !(values[0] && !values[1]) && !(!values[0] && values[1]);
        }
    }
}


// Proposition/Members/Implication.cs

using System.Collections.Generic;
using Deduction.Proposition.Abstraction;

namespace Deduction.Proposition.Members
{
    public class Implication : Connective
    {
        public Implication(params IMember[] parameters)
```

```
            : base(parameters)
        {
        }

        public Implication(IEnumerable<IMember> parameters)
            : base(parameters)
        {
        }

        public override int ParameterCount
        {
            get
            {
                return 2;
            }
        }

        public override bool Operation(bool[] values)
        {
            return !(values[0] && !values[1]);
        }
    }
}


// Proposition/Members/Not.cs

using System.Collections.Generic;
using Deduction.Proposition.Abstraction;

namespace Deduction.Proposition.Members
{
    public class Not : Connective
    {
        public Not(params IMember[] parameters)
            : base(parameters)
        {
        }

        public Not(IEnumerable<IMember> parameters)
            : base(parameters)
        {
        }

        public override int ParameterCount
        {
            get
            {
                return 1;
            }
        }

        public override bool Operation(bool[] values)
        {
            return !values[0];
        }
    }
}
```

```
// Proposition/Members/Or.cs

using System.Collections.Generic;
using Deduction.Proposition.Abstraction;
using Deduction.Proposition.Parsing;

namespace Deduction.Proposition.Members
{
    public class Or : Connective
    {
        public Or(params IMember[] parameters)
            : base(parameters)
        {
        }

        public Or(IEnumerable<IMember> parameters)
            : base(parameters)
        {
        }

        public override int ParameterCount
        {
            get
            {
                return 2;
            }
        }

        public override bool Operation(bool[] values)
        {
            return values[0] || values[1];
        }

        public override IMember Simplify(Registry registry)
        {
            Symbol firstParameter = this.Parameters[0] as Symbol;
            RegistryMember firstParameterSymbol = null;
            if (firstParameter != null)
            {
                firstParameterSymbol =
registry.GetMemberBySymbolChar(firstParameter.Letter);
                if (firstParameterSymbol != null && firstParameterSymbol.Value == true)
                {
                    return new Constant("t", true);
                }
            }

            Symbol secondParameter = this.Parameters[1] as Symbol;
            RegistryMember secondParameterSymbol = null;
            if (secondParameter != null)
            {
                secondParameterSymbol =
registry.GetMemberBySymbolChar(secondParameter.Letter);
                if (secondParameterSymbol != null && secondParameterSymbol.Value == true)
                {
                    return new Constant("t", true);
                }
            }
```

```
            if (firstParameter != null && secondParameter != null)
            {
                if (firstParameter.Letter == secondParameter.Letter)
                {
                    return new Symbol(firstParameter.Letter);
                }
            }

            return null;
        }
    }
}


// Proposition/Parsing/Lexer.cs

using System.Collections.Generic;

namespace Deduction.Proposition.Parsing
{
    public class Lexer
    {
        protected readonly Registry registry;
        protected readonly string line;
        protected int currentPosition;

        public Lexer(Registry registry, string line)
        {
            this.registry = registry;
            this.line = line;
            this.currentPosition = 0;
        }

        public List<Token> Analyze()
        {
            List<Token> final = new List<Token>();

            this.Reset();

            while (true)
            {
                string curr = this.GetNext();
                if (curr == null)
                {
                    break;
                }

                RegistryMember registryMember = registry.GetMemberBySymbolChar(curr);
                Token token = new Token(curr, registryMember);

                final.Add(token);
            }

            return final;
        }

        protected string GetNext()
```

```
        {
            if (this.currentPosition >= this.line.Length)
            {
                return null;
            }

            while (char.IsWhiteSpace(this.line[this.currentPosition]))
            {
                if (++this.currentPosition >= this.line.Length)
                {
                    return null;
                }
            }

            if (char.IsLetterOrDigit(this.line[this.currentPosition]))
            {
                string final = string.Empty;

                do
                {
                    final += this.line[this.currentPosition];

                    if (++this.currentPosition >= this.line.Length)
                    {
                        return final;
                    }
                } while (char.IsLetterOrDigit(this.line[this.currentPosition]));

                return final;
            }

            return this.line[this.currentPosition++].ToString();
        }

        protected void Reset()
        {
            this.currentPosition = 0;
        }
    }
}


// Proposition/Parsing/Parser.cs

using System;
using System.Collections.Generic;
using Deduction.Proposition.Abstraction;

namespace Deduction.Proposition.Parsing
{
    public class Parser
    {
        protected readonly Registry registry;

        protected List<Token> tokens;

        protected int currentPosition;
        protected Stack<Token> connectives;
```

```
        protected Stack<IMember> members;

        public Parser(Registry registry)
        {
            this.registry = registry;
        }

        public IMember Parse(List<Token> tokens)
        {
            Stack<RegistryMember> parentheses = new Stack<RegistryMember>();

            this.tokens = tokens;
            this.Reset();

            while (true)
            {
                // read a token
                Token curr = this.GetNext();
                if (curr == null)
                {
                    break;
                }

                // if the token is a connective
                if (curr.RegistryMember != null &&
typeof(Connective).IsAssignableFrom(curr.RegistryMember.Type))
                {
                    // if the token is an opening parenthesis also,
                    if (typeof(Parenthesis).IsAssignableFrom(curr.RegistryMember.Type))
                    {
                        // push it onto parantheses stack.
                        parentheses.Push(curr.RegistryMember);
                    }
                    // otherwise,
                    else
                    {
                        // loop while there is an connective token at the top of the
stack,
                        while (this.connectives.Count > 0)
                        {
                            Token topConnective = this.connectives.Peek();

                            // if it's left-associative,
                            if (!topConnective.RegistryMember.IsRightAssociative)
                            {
                                // check precedence is less than or equal to current
token's precedence.
                                if (curr.RegistryMember.Precedence <
topConnective.RegistryMember.Precedence)
                                {
                                    break;
                                }
                            }
                            // if it's right-associative,
                            else
                            {
                                // check precedence is less than current token's
precedence.
```

```
                                if (curr.RegistryMember.Precedence <=
topConnective.RegistryMember.Precedence)
                                {
                                    break;
                                }
                            }

                            // pop the connective at the top of the stack,
                            // make a tree node with it.
                            this.MakeNodeTree();
                        }
                    }

                    // push the token onto the connectives stack.
                    this.connectives.Push(curr);
                }
                // if the token is closing parenthesis,
                else if (parentheses.Count > 0 && parentheses.Peek().ClosesWith ==
curr.Content)
                {
                    // pop the last paranthesis off the parantheses stack,
                    RegistryMember lastParenthesis = parentheses.Pop();

                    // loop until the token at the top of the stack is the opening
parenthesis,
                    while (this.connectives.Count > 0 &&
this.connectives.Peek().Content != lastParenthesis.SymbolChar)
                    {
                        // pop connectives off the stack,
                        // to make tree nodes with it.
                        this.MakeNodeTree();
                    }

                    this.connectives.Pop();
                }
                else if (curr.RegistryMember != null &&
typeof(Constant).IsAssignableFrom(curr.RegistryMember.Type))
                {
                    Constant constant = new Constant(curr.Content,
curr.RegistryMember.Value.Value);
                    this.members.Push(constant);
                }
                else
                {
                    Symbol symbol = new Symbol(curr.Content);
                    this.members.Push(symbol);
                }
            }

            while (this.connectives.Count > 0)
            {
                this.MakeNodeTree();
            }

            return this.members.Pop();
        }

        protected void MakeNodeTree()
```

```
        {
            Token poppedConnective = this.connectives.Pop();
            Connective connective =
(Connective)Activator.CreateInstance(poppedConnective.RegistryMember.Type, new
IMember[0]);

            IMember[] connectiveParameters = new IMember[connective.ParameterCount];
            for (int i = connective.ParameterCount - 1; i >= 0; i--)
            {
                IMember poppedMember = this.members.Pop();
                connectiveParameters[i] = poppedMember;
            }
            connective.Parameters.AddRange(connectiveParameters);

            this.members.Push(connective);
        }

        protected Token GetNext()
        {
            if (this.currentPosition >= this.tokens.Count)
            {
                return null;
            }

            return this.tokens[this.currentPosition++];
        }

        protected void Reset()
        {
            this.currentPosition = 0;

            this.connectives = new Stack<Token>();
            this.members = new Stack<IMember>();
        }
    }
}


// Proposition/Parsing/Registry.cs

using System;
using System.Collections.Generic;

namespace Deduction.Proposition.Parsing
{
    public class Registry
    {
        protected readonly List<RegistryMember> members;

        public Registry()
        {
            this.members = new List<RegistryMember>();
        }

        public List<RegistryMember> Members
        {
            get
            {
```

```
                return this.members;
            }
        }

        public void AddMembers(params RegistryMember[] members)
        {
            this.Members.AddRange(members);
        }

        public RegistryMember GetMemberByType(Type type)
        {
            foreach (RegistryMember registryMember in this.Members)
            {
                if (registryMember.Type == type)
                {
                    return registryMember;
                }
            }

            return null;
        }

        public RegistryMember GetMemberBySymbolChar(string symbolChar)
        {
            foreach (RegistryMember registryMember in this.Members)
            {
                if (registryMember.SymbolChar == symbolChar)
                {
                    return registryMember;
                }

                if (registryMember.Aliases != null)
                {
                    foreach (string alias in registryMember.Aliases)
                    {
                        if (alias == symbolChar)
                        {
                            return registryMember;
                        }
                    }
                }
            }

            return null;
        }
    }
}


// Proposition/Parsing/RegistryMember.cs

using System;

namespace Deduction.Proposition.Parsing
{
    public class RegistryMember
    {
        // fields
```

```
        protected readonly string symbolChar;
        protected readonly Type type;
        protected readonly int precedence;
        protected readonly int parameters;
        protected readonly bool? value;
        protected readonly string[] aliases;
        protected readonly string closesWith;
        protected readonly bool isRightAssociative;

        // constructors
        public RegistryMember(string symbolChar, Type type, int precedence = 10, int
parameters = 0, bool? value = null, string[] aliases = null, string closesWith = null,
bool isRightAssociative = false)
        {
            this.symbolChar = symbolChar;
            this.type = type;
            this.precedence = precedence;
            this.parameters = parameters;
            this.value = value;
            this.aliases = aliases;
            this.closesWith = closesWith;
            this.isRightAssociative = isRightAssociative;
        }

        // properties
        public string SymbolChar
        {
            get
            {
                return this.symbolChar;
            }
        }

        public Type Type
        {
            get
            {
                return this.type;
            }
        }

        public int Precedence
        {
            get
            {
                return this.precedence;
            }
        }

        public int Parameters
        {
            get
            {
                return this.parameters;
            }
        }

        public bool? Value
```

```
        {
            get
            {
                return this.value;
            }
        }

        public string[] Aliases
        {
            get
            {
                return this.aliases;
            }
        }

        public string ClosesWith
        {
            get
            {
                return this.closesWith;
            }
        }

        public bool IsRightAssociative
        {
            get
            {
                return this.isRightAssociative;
            }
        }
    }
}


// Proposition/Parsing/Token.cs

namespace Deduction.Proposition.Parsing
{
    public class Token
    {
        // fields
        protected readonly string content;
        protected readonly RegistryMember registryMember;

        // constructors
        public Token(string content, RegistryMember registryMember)
        {
            this.content = content;
            this.registryMember = registryMember;
        }

        // properties
        public string Content
        {
            get
            {
                return this.content;
            }
```

```
        }

        public RegistryMember RegistryMember
        {
            get
            {
                return this.registryMember;
            }
        }
    }
}


// Proposition/Processors/Dumper.cs

using System.Collections.Generic;
using System.Text;
using Deduction.Proposition.Abstraction;
using Deduction.Proposition.Parsing;

namespace Deduction.Proposition.Processors
{
    public class Dumper
    {
        protected readonly Registry registry;

        public Dumper(Registry registry)
        {
            this.registry = registry;
        }

        public string Dump(params IMember[] input)
        {
            return this.Dump(input as IEnumerable<IMember>);
        }

        public string Dump(IEnumerable<IMember> input)
        {
            StringBuilder output = new StringBuilder();

            foreach (IMember inputItem in input)
            {
                if (output.Length > 0)
                {
                    output.Append(", ");
                }

                if (inputItem is Symbol)
                {
                    output.Append((inputItem as Symbol).Letter);
                }
                else if (inputItem is Connective)
                {
                    Connective connective = (inputItem as Connective);
                    RegistryMember registryMember =
registry.GetMemberByType(connective.GetType());

                    if (connective.ParameterCount == 1)
```

```
                    {
                        output.Append(registryMember.SymbolChar);

                        if (!connective.Parameters[0].IsAtomic)
                        {
                            output.Append("(");
                        }

                        output.Append(this.Dump(connective.Parameters[0]));

                        if (!connective.Parameters[0].IsAtomic)
                        {
                            output.Append(")");
                        }
                    }
                    else
                    {
                        output.Append("(");
                        for (int i = 0; i < connective.ParameterCount; i++)
                        {
                            if (i != 0)
                            {
                                output.Append(" ");
                                output.Append(registryMember.SymbolChar);
                                output.Append(" ");
                            }

                            output.Append(this.Dump(connective.Parameters[i]));
                        }
                        output.Append(")");
                    }
                }
            }

            return output.ToString();
        }
    }
}


// Proposition/Processors/Simplifier.cs

using Deduction.Proposition.Abstraction;
using Deduction.Proposition.Parsing;

namespace Deduction.Proposition.Processors
{
    public class Simplifier
    {
        protected readonly Registry registry;

        public Simplifier(Registry registry)
        {
            this.registry = registry;
        }

        public IMember Simplify(IMember node)
        {
```

```
            IMember final = node.Clone() as IMember;

            if (final is Connective)
            {
                Connective sourceConnective = final as Connective;

                for (int i = 0; i < sourceConnective.ParameterCount; i++)
                {
                    sourceConnective.Parameters[i] =
this.Simplify(sourceConnective.Parameters[i]);
                }

                IMember simplified = sourceConnective.Simplify(this.registry);
                if (simplified != null)
                {
                    return simplified;
                }

                return sourceConnective;
            }

            return final;
        }
    }
}


// Proposition/Processors/Substitutor.cs

using System;
using System.Collections.Generic;
using Deduction.Proposition.Abstraction;
using Deduction.Proposition.Parsing;

namespace Deduction.Proposition.Processors
{
    public class Substitutor
    {
        protected readonly Registry registry;

        public Substitutor(Registry registry)
        {
            this.registry = registry;
        }

        public IMember Substitute(IMember node, Dictionary<string, string> table)
        {
            IMember final = node.Clone() as IMember;

            if (final is Connective)
            {
                Connective sourceConnective = final as Connective;
                RegistryMember sourceRegistryMember =
this.registry.GetMemberByType(sourceConnective.GetType());

                if (table.ContainsKey(sourceRegistryMember.SymbolChar))
                {
```

```
                    RegistryMember targetRegistryMember =
this.registry.GetMemberBySymbolChar(table[sourceRegistryMember.SymbolChar]);
                    sourceConnective =
Activator.CreateInstance(targetRegistryMember.Type, sourceConnective.Parameters) as
Connective;
                }

                for (int i = 0; i < sourceConnective.ParameterCount; i++)
                {
                    sourceConnective.Parameters[i] =
this.Substitute(sourceConnective.Parameters[i], table);
                }

                return sourceConnective;
            }

            if (final is Symbol)
            {
                Symbol sourceSymbol = final as Symbol;

                if (table.ContainsKey(sourceSymbol.Letter))
                {
                    RegistryMember targetRegistryMember =
this.registry.GetMemberBySymbolChar(table[sourceSymbol.Letter]);

                    if (targetRegistryMember != null &&
typeof(Constant).IsAssignableFrom(targetRegistryMember.Type))
                    {
                        sourceSymbol =
Activator.CreateInstance(targetRegistryMember.Type, targetRegistryMember.SymbolChar,
targetRegistryMember.Value) as Constant;
                    }
                    else
                    {
                        sourceSymbol = Activator.CreateInstance(sourceSymbol.GetType(),
table[sourceSymbol.Letter]) as Symbol;
                    }
                }

                return sourceSymbol;
            }

            throw new Exception();
        }

        public void GetSymbols(IMember node, ref List<string> table)
        {
            if (node is Connective)
            {
                Connective sourceConnective = node as Connective;
                RegistryMember sourceRegistryMember =
this.registry.GetMemberByType(sourceConnective.GetType());

                for (int i = 0; i < sourceConnective.ParameterCount; i++)
                {
                    this.GetSymbols(sourceConnective.Parameters[i], ref table);
                }
```

```
                return;
            }

            if (node is Constant)
            {
                return;
            }

            if (node is Symbol)
            {
                Symbol sourceSymbol = node as Symbol;

                if (!table.Contains(sourceSymbol.Letter))
                {
                    table.Add(sourceSymbol.Letter);
                }
            }
        }
    }
}


// Tests/ConsoleOut/PropositionTests.cs

using System.Collections.Generic;
using System.IO;
using Deduction.Proposition.Parsing;
using Deduction.Proposition.Processors;

namespace Deduction.Tests.ConsoleOut
{
    public class PropositionTests
    {
        public static void Test(TextWriter output, Registry registry, string prop)
        {
            Lexer lexer = new Lexer(registry, prop);
            var tokens = lexer.Analyze();

            Parser parser = new Parser(registry);
            var rootOfTree = parser.Parse(tokens);

            Substitutor substitutor = new Substitutor(registry);
            var table = new Dictionary<string, string>()
            {
                // { "&", "=" },
                { "Second", "First" },
                { "B", "t" },
            };
            var assigned = substitutor.Substitute(rootOfTree, table);

            Simplifier simplifier = new Simplifier(registry);
            var simplified = simplifier.Simplify(assigned);

            Dumper dumper = new Dumper(registry);
            output.WriteLine("proposition               = {0}", prop);
            output.WriteLine("Dumper.Dump()             = {0}", dumper.Dump(rootOfTree));
            output.WriteLine("Substitutor.Substitute()  = {0}", dumper.Dump(assigned));
            output.WriteLine("Simplifier.Simplify()     = {0}", dumper.Dump(simplified));
```

```
            output.WriteLine();
        }
    }
}


// Commands.cs

using System;
using System.Collections.Generic;
using System.IO;
using Deduction.GentzenPrime.Abstraction;
using Deduction.GentzenPrime.Processors;
using Deduction.Proposition.Parsing;
using Deduction.Proposition.Processors;

namespace Deduction
{
    public class Commands
    {
        protected Registry registry;
        protected TextWriter textWriter;

        public Commands(Registry registry, TextWriter textWriter)
        {
            this.registry = registry;
            this.textWriter = textWriter;
        }

        public bool Interprete(TextReader textReader)
        {
            this.textWriter.Write(": ");

            string line = textReader.ReadLine();
            string[] words = line.Split(new char[] { ' ', '\t' }, 2,
StringSplitOptions.RemoveEmptyEntries);

            if (words.Length > 0)
            {
                switch (words[0])
                {
                    case "q":
                        return false;

                    case "h":
                        this.Help();
                        break;

                    case ".":
                        if (words.Length < 2)
                        {
                            this.textWriter.WriteLine("Required parameter is missing,
type 'h' for help.");
                            this.textWriter.WriteLine();
                            break;
                        }

                        this.LoadFromFileProof(words[1]);
```

```
                        break;

                    case "?":
                        if (words.Length < 2)
                        {
                            this.textWriter.WriteLine("Required parameter is missing,
type 'h' for help.");
                            this.textWriter.WriteLine();
                            break;
                        }

                        this.LoadFromInputProof(words[1]);
                        break;

                    case "f":
                        if (words.Length < 2)
                        {
                            this.textWriter.WriteLine("Required parameter is missing,
type 'h' for help.");
                            this.textWriter.WriteLine();
                            break;
                        }

                        this.LoadFromInputFalsify(words[1]);
                        break;

                    case "c":
                        if (this.textWriter == Console.Out)
                        {
                            Console.Clear();
                        }
                        else
                        {
                            for (int i = 25; i >= 0; i--)
                            {
                                this.textWriter.WriteLine();
                            }
                        }
                        this.Help();
                        break;

                    default:
                        this.textWriter.WriteLine("Invalid input, type 'h' for help.");
                        this.textWriter.WriteLine("Hint: to prove a propositional
formula, use '? {0}' command.", line);
                        this.textWriter.WriteLine();
                        break;
                }
            }

            return true;
        }

        public void Help()
        {
            this.textWriter.WriteLine("Help");
            this.textWriter.WriteLine("==================");
            this.textWriter.WriteLine();
```

```
            this.textWriter.WriteLine("? [sequent]     to load a sequent from input and
show proof tree.");
            this.textWriter.WriteLine("f [sequent]     to load a sequent from input and
list all falsifying valuations.");
            this.textWriter.WriteLine(". [path]        to load a sequent from file.");
            this.textWriter.WriteLine("c               to clear console.");
            this.textWriter.WriteLine("h               to help.");
            this.textWriter.WriteLine("q               to quit.");
            this.textWriter.WriteLine();
        }

        public void LoadFromFileProof(string path)
        {
            if (!File.Exists(path))
            {
                this.textWriter.WriteLine("File not found - {0}", path);
                this.textWriter.WriteLine();
                return;
            }

            this.textWriter.WriteLine("Reading file: {0}", path);
            this.textWriter.WriteLine();

            string[] lines = File.ReadAllLines(path);
            foreach (string line in lines)
            {
                string trimmedLine = line.Trim();
                if (trimmedLine.Length == 0 || trimmedLine.StartsWith("//") ||
trimmedLine.StartsWith("#"))
                {
                    continue;
                }

                this.textWriter.WriteLine("------------------------------------------------
---");
                this.LoadFromInputProof(line);
            }
        }

        public void LoadFromInputProof(string sequentLine)
        {
            this.textWriter.WriteLine("Deduction tree of: {0}", sequentLine);
            this.textWriter.WriteLine();

            Sequent sequent = SequentReader.Read(this.registry, sequentLine);
            if (sequent == null)
            {
                this.textWriter.WriteLine("Not a valid sequent. Ex: A & B -> B | C, D");
                this.textWriter.WriteLine();
                return;
            }

            Tree<Sequent> root = new Tree<Sequent>(sequent);

            Prover prover = new Prover(registry);
            prover.Search(root);

            Falsifier falsifier = new Falsifier(this.registry);
```

```
            List<Dictionary<string, string>> falsifyingValuations = new
List<Dictionary<string, string>>();

            Dumper dumper = new Dumper(registry);

            int depth = 0;
            List<string> counterExamples = new List<string>();
            Action<Tree<Sequent>> dumpAction = null;
            dumpAction = delegate(Tree<Sequent> seq)
            {
                string indentation = new string('\t', depth);

                string output = (dumper.Dump(seq.Content.Left) + " -> " +
dumper.Dump(seq.Content.Right)).Trim();
                this.textWriter.WriteLine("{0}sequent = {1}", indentation, output);
                //this.textWriter.WriteLine("{0}sequent.isAxiom()      = {1}",
indentation, seq.Content.IsAxiom());
                //this.textWriter.WriteLine("{0}sequent.isAtomic()     = {1}",
indentation, seq.Content.IsAtomic());
                if (seq.Content.IsAxiom())
                {
                    this.textWriter.WriteLine("{0}          ** axiom node **",
indentation);
                    this.textWriter.WriteLine();
                }
                else if (seq.Content.IsAtomic())
                {
                    counterExamples.Add(output);

                    this.textWriter.WriteLine("{0}          ** counter-example node **",
indentation);
                    this.textWriter.WriteLine();

                    List<Dictionary<string, string>> valuations =
falsifier.Falsify(seq.Content);
                    foreach (Dictionary<string, string> valuation in valuations)
                    {
                        falsifyingValuations.Add(valuation);
                    }
                }

                if (seq.Children.Count >= 2)
                {
                    depth++;
                }

                foreach (Tree<Sequent> child in seq.Children)
                {
                    dumpAction(child);
                }

                if (seq.Children.Count >= 2)
                {
                    depth--;
                }
            };

            dumpAction(root);
```

```
            if (counterExamples.Count > 0)
            {
                this.textWriter.WriteLine("Formula is not valid, details below.");

                this.textWriter.WriteLine("+ Counter-examples:");
                foreach (string counterExample in counterExamples)
                {
                    this.textWriter.Write("  .. ");
                    this.textWriter.WriteLine(counterExample);
                }
                this.textWriter.WriteLine();

                this.textWriter.WriteLine("+ Falsifying valuations:");
                for (int i = 0; i < falsifyingValuations.Count; i++)
                {
                    this.textWriter.WriteLine("  .. Valuation #{0}:", i + 1);
                    foreach (KeyValuePair<string, string> valuation in
falsifyingValuations[i])
                    {
                        this.textWriter.WriteLine("  ..                {0} -> {1}",
valuation.Key, valuation.Value);
                    }
                }
                this.textWriter.WriteLine();
            }
            else
            {
                this.textWriter.WriteLine("Formula is valid.");
            }

            this.textWriter.WriteLine();
        }

        public void LoadFromInputFalsify(string sequentLine)
        {
            Sequent sequent = SequentReader.Read(this.registry, sequentLine);
            if (sequent == null)
            {
                this.textWriter.WriteLine("Not a valid sequent. Ex: A & B -> B | C, D");
                this.textWriter.WriteLine();
                return;
            }

            Falsifier falsifier = new Falsifier(this.registry);
            List<Dictionary<string, string>> valuations = falsifier.Falsify(sequent);

            if (valuations.Count > 0)
            {
                this.textWriter.WriteLine("Falsifying valuations of: {0}", sequentLine);

                for (int i = 0; i < valuations.Count; i++)
                {
                    this.textWriter.WriteLine(".. Valuation #{0}:", i + 1);
                    foreach (KeyValuePair<string, string> valuation in valuations[i])
                    {
                        this.textWriter.WriteLine("..                {0} -> {1}",
valuation.Key, valuation.Value);
```

```
                    }
                }
            }
            else
            {
                this.textWriter.WriteLine("Falsifying needs atomic input. To prove a
sequent use '? {0}' command instead.", sequentLine);
            }

            this.textWriter.WriteLine();
        }
    }
}


// Program.cs

using System;
using Deduction.Proposition.Abstraction;
using Deduction.Proposition.Members;
using Deduction.Proposition.Parsing;

namespace Deduction
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Registry registry = new Registry();
            registry.AddMembers(
                new RegistryMember("!", typeof(Not), precedence: 1, isRightAssociative:
true, aliases: new string[] { "not" }),
                new RegistryMember("&", typeof(And), precedence: 4, aliases: new string[]
{ "and" }),
                new RegistryMember("|", typeof(Or), precedence: 5, aliases: new string[]
{ "or" }),
                new RegistryMember(">", typeof(Implication), precedence: 6, aliases: new
string[] { "implies" }),
                new RegistryMember("=", typeof(Equivalence), precedence: 6, aliases: new
string[] { "equals" }),

                new RegistryMember("[", typeof(Parenthesis), precedence: 101, closesWith:
"]"),
                new RegistryMember("(", typeof(Parenthesis), precedence: 101, closesWith:
")"),

                new RegistryMember("f", typeof(Constant), precedence: 0, value: false,
aliases: new string[] { "0", "false" }),
                new RegistryMember("t", typeof(Constant), precedence: 0, value: true,
aliases: new string[] { "1", "true" })
            );

            // proposition tests including parsing, valuation, simplification.
            // string prop = "(((Anne & A) & B) & (B & C)) | (!C & D | D | D) | !!!(!f) |
f | f | t and D";
            //string prop = "(First | Second) & (A | B) & C";
            //PropositionTests.Test(Console.Out, registry, prop);
```

```
            Commands commands = new Commands(registry, Console.Out);
            commands.Help();

            while (commands.Interprete(Console.In))
            {
            }
        }
    }
}
```