|  |
| --- |
| CMPE534 Automated Deduction |
| Term Project |
| Theorem Prover for Propositional Logic |

|  |
| --- |
| 135394 Eser Laroux Ozvataf  13.01.2014 |

Theorem Prover for Prop. Logic

# 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.

## 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.

## 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.

# Solution of the Problem

## 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.

## 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.

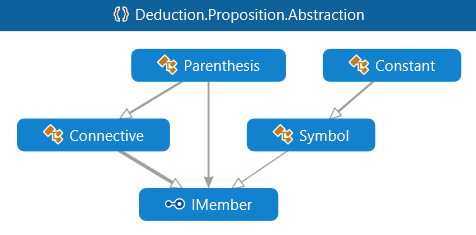


Figure 1: Class Diagram

### 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.

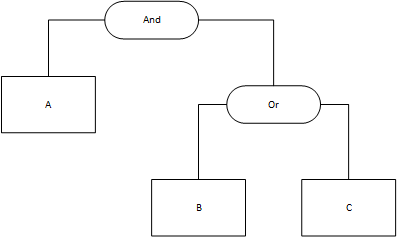


Figure 2: Data Structure of Connectives

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.

### 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.

### 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.

|  |
| --- |
| P1, P2, P3, …, Pn  –> Q1, Q2, Q3, …, Qm |

Figure 3: Sample Sequent

## 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.

### 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.

### 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.

## User Interaction