Sworn declaration

I hereby declare, under oath, that this homework has been my independent work and has not been aided with any prohibited means. I declare, to the best of my knowledge and belief, that all passages taken from published and unpublished sources or documents have been reproduced whether as original, slightly changed or in thought, have been mentioned as such at the corresponding places of the thesis, by citation, where the extent of the original quotes is indicated.

A homework that violates the above statement will be rejected.

This is readme about how to use the Boolean expression solver that I built using Z3 solver.

There are 2 classes of components involved in this:

- The syntax nodes and tree classes which represent the parsed expression
- The actual syntax tree evaluator class called “BooleanExpressionsProver” which takes a “SyntaxTree” class and proves if it’s satisfiable

The possible operations are:

- Addition binary operation represented by the “add” symbol
- Substraction binary operation represented by the “sub” symbol
- Multiplication binary operation represented by the “mul” symbol
- Division binary operation represented by the “div” symbol
- Boolean and binary operation represented by the “and” symbol
- Boolean or binary operation represented by the “or” symbol
- Boolean not unary operation represented by the “not” symbol
- Greater than binary operation represented by “gt” symbol
- Greater or equal than binary operation represented by the “gte” symbol
- Lesser than binary operation represented by the “ls” symbol
- Lesser or equal than binary operation represented by the “lse” symbol
- Minus unary operation represented by the symbol “min”
- Block syntax representing an operator or other expression surrounded by “(“ and “)”
- Any domain unary operation represented by the symbol “any”
- Exists domain unary operation represented by the symbol “exists”

I used words instead of the proper symbols for addition and comparison operations because it was easier to parse them. Too lazy to fix them but they work properly

For example an addition operation will look like “x add y”. A block operator will look like this “(a add b) gt 16”. I also applied the greater than operation “gt”. These operations can be chained so for example a greater than operator can take a block operator as the left argument.

When you pass an expression to the parser it creates a SyntaxTree out of it. It contains the root node. The root node can be a BlockSyntaxNode or any other binary or unary operation.

The syntax nodes types are:

- “LiteralSyntaxNode” which represents mostly a keyword such as the keywords for the operators like “and”, “or”. It can also be a parenthesis.
- “SymbolSyntaxNode” which represents variables in the expression. It has an important property called the “Id” which is used to identify the symbol uniquely
- “BlockExpressionSyntaxNode” which represents another expression enclosed by parentheses “(“ and “)”. Used to change the way the SyntaxTree looks like and how it will be evaluated
- “UnaryExpressionSyntaxNode” which represents a unary operation like Boolean “not” or arithmetic “min” operations
- “BinaryExpressionSyntaxNode” which represents a binary operation like arithmetic operation “add”
- “ConstantValueSyntaxNode” which represents a hardcoded constant value in the expression like a number “1231” or a Boolean value like “true”
- “ValueSyntaxNode” which is represents the base type for all the syntax nodes which return another value like the “add” operation or another “SymbolSyntaxNode”. Has a special property called “DomainValue” which represent the domain of a variable inside the ”ValueSyntaxNode”. A “SymbolSyntaxNode” or a “BlockExpressionSyntaxNode” are derived from “ValueSyntaxNode”
- “DomainValueSyntaxNode” which represents a special node to restrict a variable inside a “ValueSyntaxNode”. It can contain the “any” or “exists” operations

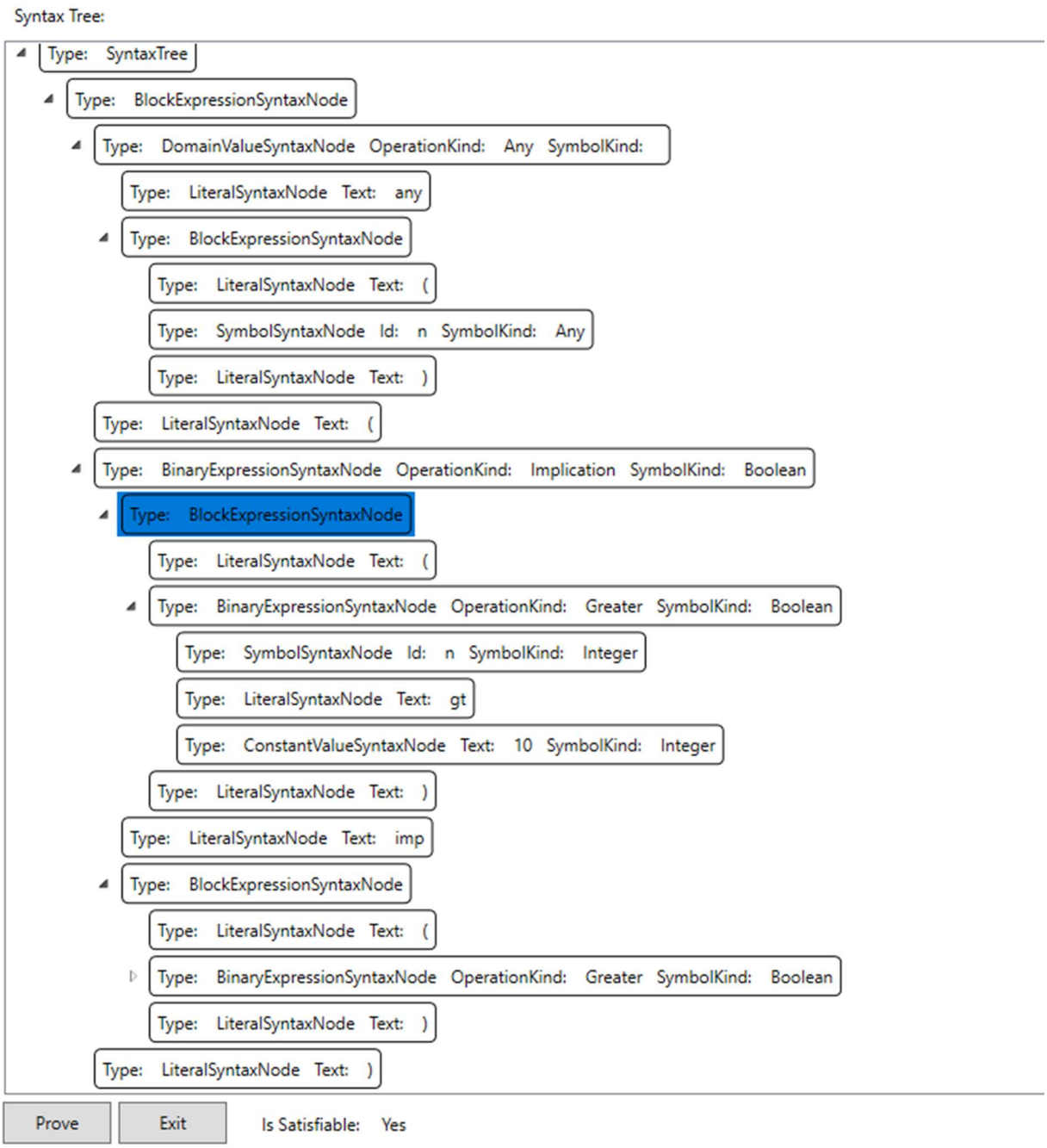
A “SyntaxTree” is built using the nodes mentioned above. The nodes mentioned above can have child nodes. For example a “BinaryExpressionSyntaxNode” has as children the left value, the actual operation symbol or literal called “LiteralSyntaxNode” and the right value.

The left and right values are any kind of syntax nodes derived from “ValueSyntaxNode”

I used the C# compiler tools codenamed “Roslyn” for another project and they had the same kind of names and structure. I got the idea from them but used my own classes and my own parser. The actual C# language has hundreds of syntax nodes. And the parser has 20000 lines in total. The parser is not generated by any sort of tool and is manually written. You can see the individual commits in GitHub:
https://github.com/dotnet/roslyn/blob/master/src/Compilers/CSharp/Portable/Parser/LanguageParser.cs

Now in the application that I made you can actually see the SyntaxTree structure in the expandable treeview.

For example if we have the expression “any (n) ((n gt 10) imp (n gt 5))” the expanded generated SyntaxTree will appear like this in the user interface:



You can see how a node has children. The nodes that are shifted one step to the right are children from the first node from the top which is shifted less to the right as them.

You can also see a label called “Is Satisfiable” which shows the result of the evaluation of the syntax tree. In case of expression in our example it is satisfiable. If n is greater than 10, then of course n will be also greater than 5.

The “any” and “exists” domain operators are really tricky because they get attached to the first available value expression to their right. To force them to get attached to the proper value expression we need to add a BlockSyntaxNode with parenthesis to the right as we added in the example.

If we have another expression like this: “any (x) x add 5” it will apply the “any” domain operator to the x symbol and not the entire add operation. We need to encompass the “add” operation inside a “BlockSyntaxNode” like this: “any (x) (x add 5)”.

I tried with really complex expressions and I was amazed that it actually worked.

For example if we have the expression: “any (n) (any (m) (((n gt 10) and (m gt 15)) imp ((n add 15 add m) gt 30)))” then it returns that it is satisfiable. Why? Because if we have any two numbers, one greater than 10 and one greater than 15, then if we add them and add 15 to that it will always be greater than 30.

On the other hand with the expression: “any (n) (any (m) (((n gt 10) and (m gt 15)) imp ((n add 15 add m) gt 10)))” it returns false because the sum of any two numbers, one greater than 15 and the other greater than 15 to which we add 15 is not always greater than 100.

It also works with plain Boolean expressions. For example if we have “(x and not x)” then it will prove that it will always return false.

How to run this. This is a Visual Studio 2015 project. You need Visual Studio 2015 for this and a Windows based computer with the x64 version of windows, the 64 bit version of windows. Inside the archive at the path “Tema1B_FMSE\bin\Debug” there is an executable called “Tema1B_FMSE.exe”. Run this. It also has a list of predefined examples that you can use.

```
using System.Collections.Generic;
using System.Threading;

namespace Tema1B_FMSE.SyntaxNodes
{
    public class BinaryExpressionSyntaxNode : ValueSyntaxNode
    {
        public LiteralSyntaxNode Operator { get; set; }

        public ValueSyntaxNode LeftValue { get; set; }

        public ValueSyntaxNode RightValue { get; set; }

        public EOperationKinds OperationKind { get; set; }

        public override void AssignChild(SyntaxNode child)
        {
            if (LeftValue == null)
            {
                var valueSyntaxNode = (ValueSyntaxNode) child;
                LeftValue = valueSyntaxNode;
                valueSyntaxNode.Parent = this;

                StartIndex = valueSyntaxNode.StartIndex;
                EndIndex = valueSyntaxNode.EndIndex;

                return;
            }

            if (Operator == null)
            {
                var literalSyntaxNode = (LiteralSyntaxNode)child;

                OperationKind =
OperatorsInformation.MappedOperationKinds[literalSyntaxNode.LiteralValue];
                Operator = literalSyntaxNode;
                literalSyntaxNode.Parent = this;
                EndIndex = literalSyntaxNode.EndIndex;
                AssignSymbolKindToThis();

                return;
            }

            if (RightValue == null)
            {
                var valueSyntaxNode = (ValueSyntaxNode)child;
                RightValue = valueSyntaxNode;
                valueSyntaxNode.Parent = this;
                EndIndex = valueSyntaxNode.EndIndex;
                _isFinishedReading = true;

                AssignSymbolKindsToValues();
```

```
            }
        }

        private void AssignSymbolKindToThis()
        {
            SymbolKind = OperatorsInformation.MappedOperationResults[OperationKind];
        }

        private void AssignSymbolKindsToValues()
        {
            var symbolKind =
OperatorsInformation.MappedOperationInputs[OperationKind];

            LeftValue.SymbolKind = symbolKind;
            RightValue.SymbolKind = symbolKind;
        }

        public override IEnumerable<SyntaxNode> Children
        {
            get
            {
                var children = new List<SyntaxNode>();

                if (DomainValue != null)
                {
                    children.Add(DomainValue);
                }

                children.Add(LeftValue);
                children.Add(Operator);
                children.Add(RightValue);

                return children;
            }
        }
    }
}


using System.Collections.Generic;

namespace Tema1B_FMSE.SyntaxNodes
{
    public class BlockExpressionSyntaxNode : ValueSyntaxNode
    {
        public LiteralSyntaxNode LeftParanthesis { get; set; }

        public LiteralSyntaxNode RightParanthesis { get; set; }

        public ValueSyntaxNode InnerValue { get; set; }

        public override void AssignChild(SyntaxNode child)
        {
            if (LeftParanthesis == null)
            {
                var literalSyntaxNode = (LiteralSyntaxNode) child;

                LeftParanthesis = literalSyntaxNode;
                literalSyntaxNode.Parent = this;
                StartIndex = literalSyntaxNode.StartIndex;
                EndIndex = literalSyntaxNode.EndIndex;
```

```
                return;
            }

            if (InnerValue == null)
            {
                var valueSyntaxNode = (ValueSyntaxNode) child;

                InnerValue = valueSyntaxNode;
                valueSyntaxNode.Parent = this;
                EndIndex = valueSyntaxNode.EndIndex;

                return;
            }

            if (RightParanthesis == null)
            {
                var literalSyntaxNode = (LiteralSyntaxNode) child;

                RightParanthesis = literalSyntaxNode;
                literalSyntaxNode.Parent = this;
                EndIndex = literalSyntaxNode.EndIndex;
                _isFinishedReading = true;
            }
        }

        public override IEnumerable<SyntaxNode> Children
        {
            get
            {
                var children = new List<SyntaxNode>();

                if (DomainValue != null)
                {
                    children.Add(DomainValue);
                }

                children.Add(LeftParanthesis);
                children.Add(InnerValue);
                children.Add(RightParanthesis);

                return children;
            }
        }
    }
}


using System.Collections.Generic;

namespace Tema1B_FMSE.SyntaxNodes
{
    public class ConstantValueSyntaxNode : ValueSyntaxNode
    {
        public override void AssignChild(SyntaxNode child)
        {
        }

        public string LiteralValue { get; set; }

        public object Value
        {
            get
```

```
            {
                if (SymbolKind == ESymbolKinds.Boolean)
                    return bool.Parse(LiteralValue);

                return int.Parse(LiteralValue);
            }
        }

        public override IEnumerable<SyntaxNode> Children
        {
            get
            {
                return new List<SyntaxNode>();
            }
        }
    }
}


using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Tema1B_FMSE.SyntaxNodes
{
    public class DomainValueSyntaxNode : SyntaxNode
    {
        public LiteralSyntaxNode Operator { get; set; }

        public EOperationKinds OperationKind { get; set; }

        public ValueSyntaxNode InnerValue { get; set; }


        public override void AssignChild(SyntaxNode child)
        {
            if (Operator == null)
            {
                var literalSyntaxNode = (LiteralSyntaxNode)child;

                Operator = literalSyntaxNode;
                literalSyntaxNode.Parent = this;
                StartIndex = literalSyntaxNode.StartIndex;
                EndIndex = literalSyntaxNode.EndIndex;

                OperationKind =
OperatorsInformation.MappedOperationKinds[literalSyntaxNode.LiteralValue];

                return;
            }

            if (InnerValue == null)
            {
                ValueSyntaxNode valueSyntaxNode = (ValueSyntaxNode)child;

                InnerValue = valueSyntaxNode;
                valueSyntaxNode.Parent = this;
                EndIndex = valueSyntaxNode.EndIndex;
                _isFinishedReading = true;
            }
```

```
        }

        public override IEnumerable<SyntaxNode> Children
        {
            get
            {
                return new List<SyntaxNode> { Operator, InnerValue };
            }
        }
    }
}
```

```
namespace Tema1B_FMSE.SyntaxNodes
{
    public enum EOperationKinds
    {
        None = 0,
        Not = 1,
        And = 2,
        Or = 3,
        Implication = 4,
        Add = 5,
        Substract = 6,
        Multiply = 7,
        Divide = 8,
        Less = 9,
        LessOrEqual = 10,
        Equal = 11,
        GreaterOrEqual = 12,
        Greater = 13,
        Minus = 14,
        Any = 15,
        Exists = 16
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Tema1B_FMSE.SyntaxNodes
{
    public enum ESymbolKinds
    {
        Any =  0,
        Boolean = 1,
        Integer = 2
    }
}
```

```
using System.Collections.Generic;


namespace Tema1B_FMSE.SyntaxNodes
{
    public class LiteralSyntaxNode : SyntaxNode
    {
        public LiteralSyntaxNode()
        {
            _isFinishedReading = true;
        }

        public override void AssignChild(SyntaxNode child)
        {

        }

        public override IEnumerable<SyntaxNode> Children
        {
            get
            {
                return new List<SyntaxNode>();
            }
        }

        public string LiteralValue { get; set; }
    }
}
```

```
namespace Tema1B_FMSE.SyntaxNodes
{
    using System.Collections.Generic;

    public class OperatorsInformation
    {
        public static Dictionary<string, EOperationKinds> MappedOperationKinds =
            new Dictionary<string, EOperationKinds>
                {
                    { "add", EOperationKinds.Add },
                    { "mul", EOperationKinds.Multiply },
                    { "and", EOperationKinds.And },
                    { "imp", EOperationKinds.Implication },
                    { "or", EOperationKinds.Or },
                    { "not", EOperationKinds.Not },
                    { "div", EOperationKinds.Divide },
                    { "sub", EOperationKinds.Substract },
                    { "ls", EOperationKinds.Less },
                    { "lse", EOperationKinds.LessOrEqual },
                    { "eq", EOperationKinds.Equal },
                    { "gte", EOperationKinds.GreaterOrEqual },
                    { "gt", EOperationKinds.Greater },
                    { "min", EOperationKinds.Minus },
                    { "any", EOperationKinds.Any },
                    { "exists", EOperationKinds.Exists }
                };
```

```
        public static Dictionary<EOperationKinds, ESymbolKinds> MappedOperationResults
=
            new Dictionary<EOperationKinds, ESymbolKinds>
                {
                    { EOperationKinds.Add, ESymbolKinds.Integer },
                    { EOperationKinds.Multiply, ESymbolKinds.Integer },
                    { EOperationKinds.And, ESymbolKinds.Boolean },
                    { EOperationKinds.Implication, ESymbolKinds.Boolean },
                    { EOperationKinds.Or, ESymbolKinds.Boolean },
                    { EOperationKinds.Not, ESymbolKinds.Boolean },
                    { EOperationKinds.Divide, ESymbolKinds.Integer },
                    { EOperationKinds.Substract, ESymbolKinds.Integer },
                    { EOperationKinds.Less, ESymbolKinds.Boolean },
                    { EOperationKinds.LessOrEqual, ESymbolKinds.Boolean },
                    { EOperationKinds.Equal, ESymbolKinds.Boolean },
                    { EOperationKinds.GreaterOrEqual, ESymbolKinds.Boolean },
                    { EOperationKinds.Greater, ESymbolKinds.Boolean },
                    { EOperationKinds.Minus, ESymbolKinds.Integer },
                    { EOperationKinds.Any, ESymbolKinds.Any },
                    { EOperationKinds.Exists, ESymbolKinds.Any }
                };

        public static Dictionary<EOperationKinds, ESymbolKinds> MappedOperationInputs
=
            new Dictionary<EOperationKinds, ESymbolKinds>
                {
                    { EOperationKinds.Add, ESymbolKinds.Integer },
                    { EOperationKinds.Multiply, ESymbolKinds.Integer },
                    { EOperationKinds.And, ESymbolKinds.Boolean },
                    { EOperationKinds.Implication, ESymbolKinds.Boolean },
                    { EOperationKinds.Or, ESymbolKinds.Boolean },
                    { EOperationKinds.Not, ESymbolKinds.Boolean },
                    { EOperationKinds.Divide, ESymbolKinds.Integer },
                    { EOperationKinds.Substract, ESymbolKinds.Integer },
                    { EOperationKinds.Less, ESymbolKinds.Integer },
                    { EOperationKinds.LessOrEqual, ESymbolKinds.Integer },
                    { EOperationKinds.Equal, ESymbolKinds.Integer },
                    { EOperationKinds.GreaterOrEqual, ESymbolKinds.Integer },
                    { EOperationKinds.Greater, ESymbolKinds.Integer },
                    { EOperationKinds.Minus, ESymbolKinds.Integer },
                    { EOperationKinds.Any, ESymbolKinds.Any },
                    { EOperationKinds.Exists, ESymbolKinds.Any }
                };

        public static List<EOperationKinds> UnaryOperations = new
List<EOperationKinds>
                                                                {

EOperationKinds.Minus,

EOperationKinds.Not,

EOperationKinds.Any,

EOperationKinds.Exists
                                                                };

        public static Dictionary<string, EOperationKinds> MappedBinaryOperationKinds =
            new Dictionary<string, EOperationKinds>
                {
                    { "add", EOperationKinds.Add },
```

```
                    { "mul", EOperationKinds.Multiply },
                    { "and", EOperationKinds.And },
                    { "imp", EOperationKinds.Implication },
                    { "or", EOperationKinds.Or },
                    { "div", EOperationKinds.Divide },
                    { "sub", EOperationKinds.Substract },
                    { "ls", EOperationKinds.Less },
                    { "lse", EOperationKinds.LessOrEqual },
                    { "eq", EOperationKinds.Equal },
                    { "gte", EOperationKinds.GreaterOrEqual },
                    { "gt", EOperationKinds.Greater }
                };

        public static Dictionary<string, EOperationKinds> MappedUnaryOperationKinds =
            new Dictionary<string, EOperationKinds>
                {
                    { "not", EOperationKinds.Not },
                    { "min", EOperationKinds.Minus },
                };

        public static List<string> MappedDomainOperations = new List<string> { "any",
"exists" };
    }
}
```

```
using System;
using System.Collections.Generic;

namespace Tema1B_FMSE.SyntaxNodes
{
    public class SymbolSyntaxNode : ValueSyntaxNode
    {
        public string Id { get; set; }

        public override void AssignChild(SyntaxNode child)
        {
            if (!(child is SymbolSyntaxNode))
            {
                throw new SyntaxTreeParserException("Invalid character for symbol",
StartIndex);
            }

            var symbolSyntaxNode = (SymbolSyntaxNode) child;

            if (!_isFinishedReading)
            {
                if (symbolSyntaxNode.Id == " " || symbolSyntaxNode.Id == "and" ||
symbolSyntaxNode.Id == "or" ||
                    symbolSyntaxNode.Id == "imp" || symbolSyntaxNode.Id == "not")
                {
                    _isFinishedReading = true;
                }
                else
                {
                    Id += symbolSyntaxNode.Id;
                    EndIndex = symbolSyntaxNode.EndIndex;
                }
```

```
            }
        }

        public override IEnumerable<SyntaxNode> Children
        {
            get
            {
                return new List<SyntaxNode>();
            }
        }
    }
}
```

```
using System.Collections.Generic;

namespace Tema1B_FMSE.SyntaxNodes
{
    public abstract class SyntaxNode
    {
        protected bool _isFinishedReading;

        public SyntaxNode Parent { get; set; }

        public bool IsFinishedReading
        {
            get { return _isFinishedReading; }
            set { _isFinishedReading = value; }
        }

        public int StartIndex { get; set; }

        public int EndIndex { get; set; }

        public abstract void AssignChild(SyntaxNode child);

        public abstract IEnumerable<SyntaxNode> Children { get; }

        public string Name
        {
            get { return GetType().Name; }
        }
    }
}
```

```
using System.Collections.Generic;

namespace Tema1B_FMSE.SyntaxNodes
{
    public class SyntaxTree : SyntaxNode
    {
        public SyntaxNode RootValue { get; set; }
```

```
        public override void AssignChild(SyntaxNode child)
        {
            RootValue = child;
        }

        public override IEnumerable<SyntaxNode> Children
        {
            get
            {
                return new List<SyntaxNode> { RootValue };
            }
        }
    }
}
```

```
using System.Collections.Generic;

namespace Tema1B_FMSE.SyntaxNodes
{
    public class SyntaxTreeParser
    {
        private Stack<SyntaxNode> _availableSyntaxNodes = new Stack<SyntaxNode>();
        private string _expressionText;
        private int _indexInExpression;

        public SyntaxTree Parse(string expressionText)
        {
            Initialize(expressionText);
            ParseExpression();
            FinishAndPopLastSymbolSyntaxNodeIfNeeded();

            return new SyntaxTree {RootValue = (ValueSyntaxNode)
_availableSyntaxNodes.Pop()};
        }

        private void ParseExpression()
        {
            while (CanReadMore())
            {
                var nextFoundToken = ReadNextToken();

                if (LandedOnCloseParanthesis(nextFoundToken) ||
LandedOnOpenParanthesis(nextFoundToken)
                    || LandedOnDomainOperator(nextFoundToken) != null ||
LandedOnOperator(nextFoundToken) != null)
                {
                    var literalSyntaxNode = new LiteralSyntaxNode
                                            {
                                                StartIndex = _indexInExpression,
                                                EndIndex =
                                                    _indexInExpression +
nextFoundToken.Length,

                                                LiteralValue = nextFoundToken
                                            };
```

```
                    PushNewSyntaxNode(literalSyntaxNode);
                    continue;
                }

                if (LandedOnConstant(nextFoundToken) != null)
                {
                    var constantValueSyntaxNode = new ConstantValueSyntaxNode
                    {
                        LiteralValue = nextFoundToken,
                        StartIndex = _indexInExpression,
                        EndIndex = _indexInExpression + nextFoundToken.Length
                    };

                    PushNewSyntaxNode(constantValueSyntaxNode);
                    continue;
                }

                var symbolSyntaxNode = new SymbolSyntaxNode
                {
                    StartIndex = _indexInExpression,
                    EndIndex = _indexInExpression + nextFoundToken.Length,
                    Id = nextFoundToken
                };
                PushNewSyntaxNode(symbolSyntaxNode);
            }
        }

        private void PushNewSyntaxNode(SyntaxNode syntaxNode)
        {
            _indexInExpression = syntaxNode.EndIndex;

            if (syntaxNode is LiteralSyntaxNode)
            {
                var literalSyntaxNode = (LiteralSyntaxNode)syntaxNode;

                if (LandedOnBinaryOperator(literalSyntaxNode.LiteralValue))
                {
                    FinishAndPopLastSymbolSyntaxNodeIfNeeded();

                    var binaryExpressionSyntaxNode = new BinaryExpressionSyntaxNode();

                    var valueSyntaxNode =
(ValueSyntaxNode)_availableSyntaxNodes.Pop();
                    valueSyntaxNode.IsFinishedReading = true;

                    binaryExpressionSyntaxNode.AssignChild(valueSyntaxNode);
                    binaryExpressionSyntaxNode.AssignChild(literalSyntaxNode);


PushSyntaxNodeClosingPreviousIfAvailable(binaryExpressionSyntaxNode);

                    return;
                }

                if (LandedOnUnaryOperator(literalSyntaxNode.LiteralValue))
                {
                    var unaryExpressionSyntaxNode = new UnaryExpressionSyntaxNode();
                    unaryExpressionSyntaxNode.AssignChild(literalSyntaxNode);


AssignValueNodeToLastNodePopingIfNeeded(unaryExpressionSyntaxNode);
```

```
PushSyntaxNodeClosingPreviousIfAvailable(unaryExpressionSyntaxNode);

                    return;
                }

                if (LandedOnDomainOperator(literalSyntaxNode.LiteralValue) != null)
                {
                    var domainValueSyntaxNode = new DomainValueSyntaxNode();
                    domainValueSyntaxNode.AssignChild(literalSyntaxNode);

                    PushSyntaxNodeClosingPreviousIfAvailable(domainValueSyntaxNode);
                }

                if (literalSyntaxNode.LiteralValue == "(")
                {
                    PushSyntaxNodeClosingPreviousIfAvailable(literalSyntaxNode);

                    return;
                }

                if (literalSyntaxNode.LiteralValue == ")")
                {
                    FinishAndPopLastSymbolSyntaxNodeIfNeeded();
                    var valueSyntaxNode = (ValueSyntaxNode)
_availableSyntaxNodes.Pop();
                    var previousLiteralSyntaxNode = (LiteralSyntaxNode)
_availableSyntaxNodes.Pop();

                    valueSyntaxNode.IsFinishedReading = true;
                    previousLiteralSyntaxNode.IsFinishedReading = true;

                    var blockExpressionSyntaxNode = new BlockExpressionSyntaxNode();

                    blockExpressionSyntaxNode.AssignChild(previousLiteralSyntaxNode);
                    blockExpressionSyntaxNode.AssignChild(valueSyntaxNode);
                    blockExpressionSyntaxNode.AssignChild(literalSyntaxNode);
                    blockExpressionSyntaxNode.SymbolKind = valueSyntaxNode.SymbolKind;


AssignValueNodeToLastNodePopingIfNeeded(blockExpressionSyntaxNode);

                    if (blockExpressionSyntaxNode.Parent == null)
                    {

PushSyntaxNodeClosingPreviousIfAvailable(blockExpressionSyntaxNode);
                    }

                    return;
                }
            }

            if (syntaxNode is ConstantValueSyntaxNode)
            {
                var constantValueSyntaxNode = (ConstantValueSyntaxNode)syntaxNode;

                if (_availableSyntaxNodes.Count > 0 &&
!_availableSyntaxNodes.Peek().IsFinishedReading)
                {
                    AssignValueNodeToLastNodePopingIfNeeded(constantValueSyntaxNode);
```

```
                    if (constantValueSyntaxNode.Parent == null)
                    {
                        _availableSyntaxNodes.Push(syntaxNode);
                    }

                    return;
                }
            }


if (syntaxNode is SymbolSyntaxNode)
            {
                var symbolSyntaxNode = (SymbolSyntaxNode)syntaxNode;

                if (symbolSyntaxNode.Id == " ")
                {
                    return;
                }

                if (_availableSyntaxNodes.Count > 0 &&
!_availableSyntaxNodes.Peek().IsFinishedReading)
                {
                    AssignValueNodeToLastNodePopingIfNeeded(symbolSyntaxNode);

                    _availableSyntaxNodes.Push(syntaxNode);

                    return;
                }

                if (_availableSyntaxNodes.Count > 0 && _availableSyntaxNodes.Peek() is
SymbolSyntaxNode)
                {
                    AssignValueNodeToLastNodePopingIfNeeded(symbolSyntaxNode);

                    return;
                }

                _availableSyntaxNodes.Push(symbolSyntaxNode);
            }
        }

        private void AssignValueNodeToLastNodePopingIfNeeded(ValueSyntaxNode
valueSyntaxNode)
        {
            if (_availableSyntaxNodes.Count > 0)
            {
                _availableSyntaxNodes.Peek().AssignChild(valueSyntaxNode);

                if (_availableSyntaxNodes.Peek().IsFinishedReading &&
_availableSyntaxNodes.Peek().Parent != null)
                {
                    _availableSyntaxNodes.Pop();
                }
            }
        }

        private void PushSyntaxNodeClosingPreviousIfAvailable(SyntaxNode
syntaxNodeToPush)
        {
            if (_availableSyntaxNodes.Count > 0)
            {
                _availableSyntaxNodes.Peek().IsFinishedReading = true;
```

```
                PopSymbolSyntaxNodeIfUsedAndFinished();
            }

            if (_availableSyntaxNodes.Count > 0 && (_availableSyntaxNodes.Peek() is
DomainValueSyntaxNode) && _availableSyntaxNodes.Peek().IsFinishedReading &&
(syntaxNodeToPush is ValueSyntaxNode))
            {
                var domainValueSyntaxNode =
(DomainValueSyntaxNode)_availableSyntaxNodes.Peek();
                var valueSyntaxNodeToPush = (ValueSyntaxNode)syntaxNodeToPush;

                if (domainValueSyntaxNode.OperationKind == EOperationKinds.Any ||
domainValueSyntaxNode.OperationKind == EOperationKinds.Exists)
                {
                    valueSyntaxNodeToPush.AssignDomainValue(domainValueSyntaxNode);
                }

                _availableSyntaxNodes.Pop();
            }

            _availableSyntaxNodes.Push(syntaxNodeToPush);
        }

        private void FinishAndPopLastSymbolSyntaxNodeIfNeeded()
        {
            if (_availableSyntaxNodes.Peek() is SymbolSyntaxNode &&
_availableSyntaxNodes.Peek().Parent != null)
            {
                var lastSymbolSyntaxNode = _availableSyntaxNodes.Pop();
                lastSymbolSyntaxNode.IsFinishedReading = true;
            }
        }

        private void PopSymbolSyntaxNodeIfUsedAndFinished()
        {
            if (_availableSyntaxNodes.Peek() is SymbolSyntaxNode &&
_availableSyntaxNodes.Peek().Parent != null)
            {
                _availableSyntaxNodes.Pop();
            }
        }

        private string ReadNextToken()
        {
            string op = null;

            if ((op = LandedOnOperator(_expressionText, _indexInExpression)) != null)
            {
                return op;
            }

            if (LandedOnCloseParanthesis(_expressionText, _indexInExpression))
            {
                return _expressionText.Substring(_indexInExpression, 1);
            }

            if (LandedOnOpenParanthesis(_expressionText, _indexInExpression))
            {
                return _expressionText.Substring(_indexInExpression, 1);
            }

            string constantValue = null;
```

```
            if ((constantValue = LandedOnConstant(_expressionText,
_indexInExpression)) != null)
            {
                return constantValue;
            }

            return _expressionText.Substring(_indexInExpression, 1);
        }

        private string LandedOnConstant(string expressionText, int indexInExpression =
0)
        {
            if (char.IsDigit(expressionText[indexInExpression]) ||
expressionText[indexInExpression] == '+' ||
                expressionText[indexInExpression] == '-')
            {
                var startIndex = indexInExpression;

                while (startIndex < expressionText.Length &&
(char.IsDigit(expressionText[startIndex]) ||
                       expressionText[startIndex] == '+' ||
                       expressionText[startIndex] == '-'))
                {
                    startIndex++;
                }

                return expressionText.Substring(indexInExpression, startIndex -
indexInExpression);
            }


            if (indexInExpression + 5 <= expressionText.Length)
            {
                var falseTest = expressionText.Substring(indexInExpression, 5);

                if (falseTest == "false")
                {
                    return falseTest;
                }
            }

            if (indexInExpression + 3 <= expressionText.Length)
            {
                var trueTest = expressionText.Substring(indexInExpression, 3);

                if (trueTest == "true")
                {
                    return trueTest;
                }
            }

            return null;
        }

        private string LandedOnOperator(string expression, int startIndex = 0)
        {
            foreach (var operatorCode in
OperatorsInformation.MappedOperationKinds.Keys)
            {
                if (startIndex + operatorCode.Length <= expression.Length)
                {
```

```
                    var op = expression.Substring(startIndex, operatorCode.Length);

                    if (op == operatorCode)
                    {
                        return operatorCode;
                    }
                }
            }

            return null;
        }

        private bool LandedOnBinaryOperator(string expression, int startIndex = 0)
        {
            foreach (var operatorCode in
OperatorsInformation.MappedBinaryOperationKinds.Keys)
            {
                if (startIndex + operatorCode.Length <= expression.Length)
                {
                    var op = expression.Substring(startIndex, operatorCode.Length);

                    if (op == operatorCode)
                    {
                        return true;
                    }
                }
            }

            return false;
        }

        private bool LandedOnUnaryOperator(string expression, int startIndex = 0)
        {
            foreach (var operatorCode in
OperatorsInformation.MappedUnaryOperationKinds.Keys)
            {
                if (startIndex + operatorCode.Length <= expression.Length)
                {
                    var op = expression.Substring(startIndex, operatorCode.Length);

                    if (op == operatorCode)
                    {
                        return true;
                    }
                }
            }

            return false;
        }

        private string LandedOnDomainOperator(string expression, int startIndex = 0)
        {
            foreach (var operatorCode in OperatorsInformation.MappedDomainOperations)
            {
                if (startIndex + operatorCode.Length <= expression.Length)
                {
                    var op = expression.Substring(startIndex, operatorCode.Length);

                    if (op == operatorCode)
                    {
                        return op;
                    }
```

```
                }
            }

            return null;
        }

        private bool LandedOnOpenParanthesis(string expression, int startIndex = 0)
        {
            return expression[startIndex] == '(';
        }

        private bool LandedOnCloseParanthesis(string expression, int startIndex = 0)
        {
            return expression[startIndex] == ')';
        }

        private void Initialize(string expressionText)
        {
            _expressionText = expressionText;
            _availableSyntaxNodes = new Stack<SyntaxNode>();
            _indexInExpression = 0;
        }

        private bool CanReadMore()
        {
            return _indexInExpression < _expressionText.Length;
        }
    }
}
```

```
using System;

namespace Tema1B_FMSE.SyntaxNodes
{
    public class SyntaxTreeParserException : Exception
    {
        public SyntaxTreeParserException(string message, int index) : base(message + "
at:" + index)
        {
        }
    }
}
```

```
using System.Collections.Generic;

namespace Tema1B_FMSE.SyntaxNodes
{
    public class UnaryExpressionSyntaxNode : ValueSyntaxNode
```

```
    {
        public LiteralSyntaxNode Operator { get; set; }

        public EOperationKinds OperationKind { get; set; }

        public ValueSyntaxNode InnerValue { get; set; }
        public override void AssignChild(SyntaxNode child)
        {
            if (Operator == null)
            {
                var literalSyntaxNode = (LiteralSyntaxNode) child;

                OperationKind =
OperatorsInformation.MappedUnaryOperationKinds[literalSyntaxNode.LiteralValue];

                Operator = literalSyntaxNode;
                literalSyntaxNode.Parent = this;
                StartIndex = literalSyntaxNode.StartIndex;
                EndIndex = literalSyntaxNode.EndIndex;

                return;
            }

            if (InnerValue == null)
            {
                var valueSyntaxNode = (ValueSyntaxNode) child;

                InnerValue = valueSyntaxNode;
                valueSyntaxNode.Parent = this;
                EndIndex = valueSyntaxNode.EndIndex;
                _isFinishedReading = true;

                AssignSymbolKindsToValues();
            }
        }

        private void AssignSymbolKindsToValues()
        {
            var symbolKind =
OperatorsInformation.MappedOperationInputs[OperationKind];

            InnerValue.SymbolKind = symbolKind;
        }

        public override IEnumerable<SyntaxNode> Children
        {
            get
            {
                return new List<SyntaxNode> { Operator, InnerValue };
            }
        }
    }
}
```

```
namespace Tema1B_FMSE.SyntaxNodes
{
    using System.Collections.Generic;
    using System.Windows.Documents;

    public abstract class ValueSyntaxNode : SyntaxNode
    {
        private DomainValueSyntaxNode _domainValue;

        public ESymbolKinds SymbolKind { get; set; }

        public DomainValueSyntaxNode DomainValue
        {
            get
            {
                return _domainValue;
            }
        }

        public override IEnumerable<SyntaxNode> Children
        {
            get
            {
                var children = new List<SyntaxNode>();
                children.Add(DomainValue);
                return children;
            }
        }

        public void AssignDomainValue(DomainValueSyntaxNode domainValue)
        {
            _domainValue = domainValue;
            domainValue.Parent = this;
        }
    }
}
```

```
namespace Tema1B_FMSE
{
    using System.Collections.Generic;

    using Microsoft.Z3;

    using Tema1B_FMSE.SyntaxNodes;

    public class BooleanExpressionsProver
    {
        private Dictionary<string, BoolExpr> _createdBoolSymbols;

        private Dictionary<string, IntExpr> _createdIntSymbols;

        private Dictionary<string, Symbol> _createdSymbols = new Dictionary<string,
Symbol>();

        private Context _currentContext;
```

```
        private BoolExpr _expressionTree;

        public bool IsSatisfiable(SyntaxTree syntaxTree)
        {
            Initialize();
            ConstructProof(syntaxTree);

            return CheckIfIsSatisfiable();
        }

        private bool CheckIfIsSatisfiable()
        {
            var solver = _currentContext.MkSolver();
            solver.Assert(_expressionTree);
            var status = solver.Check();

            if (status == Status.SATISFIABLE) return true;

            return false;
        }

        private void ConstructProof(SyntaxTree syntaxTree)
        {
            _expressionTree = (BoolExpr)ConstructExpression(syntaxTree.RootValue);
        }

        private Symbol ConstructSymbol(SyntaxNode syntaxNode)
        {
            if (syntaxNode is BlockExpressionSyntaxNode)
            {
                var blockExpressionSyntaxNode = (BlockExpressionSyntaxNode)syntaxNode;
                return ConstructSymbol(blockExpressionSyntaxNode.InnerValue);
            }

            if (syntaxNode is SymbolSyntaxNode)
            {
                 var symbolSyntaxNode = (SymbolSyntaxNode)syntaxNode;

                if (_createdSymbols.ContainsKey(symbolSyntaxNode.Id)) return
_createdSymbols[symbolSyntaxNode.Id];

                var symbolInt = _currentContext.MkSymbol(symbolSyntaxNode.Id);
                var intExpr = _currentContext.MkIntConst(symbolInt);

                _createdIntSymbols[symbolSyntaxNode.Id] = intExpr;
                _createdSymbols[symbolSyntaxNode.Id] = symbolInt;

                return symbolInt;
            }

            return null;
        }

        private Expr ConstructExpression(SyntaxNode syntaxNode)
        {
            if (syntaxNode is BlockExpressionSyntaxNode)
            {
                var blockExpressionSyntaxNode = (BlockExpressionSyntaxNode)syntaxNode;
```

```
                return
WrapAroundDomainIfNeeded(ConstructExpression(blockExpressionSyntaxNode.InnerValue),
blockExpressionSyntaxNode);
            }

            if (syntaxNode is SymbolSyntaxNode)
            {
                var symbolSyntaxNode = (SymbolSyntaxNode)syntaxNode;

                if (symbolSyntaxNode.SymbolKind == ESymbolKinds.Boolean)
                {
                    if (_createdBoolSymbols.ContainsKey(symbolSyntaxNode.Id)) return
WrapAroundDomainIfNeeded(_createdBoolSymbols[symbolSyntaxNode.Id], symbolSyntaxNode);

                    Symbol symbolBool = null;
                    if (_createdSymbols.ContainsKey(symbolSyntaxNode.Id))
                    {
                        symbolBool = _createdSymbols[symbolSyntaxNode.Id];
                    }
                    else
                    {
                        symbolBool = _currentContext.MkSymbol(symbolSyntaxNode.Id);
                        _createdSymbols[symbolSyntaxNode.Id] = symbolBool;
                    }

                    var boolExpr = _currentContext.MkBoolConst(symbolBool);
                    _createdBoolSymbols[symbolSyntaxNode.Id] = boolExpr;


                    return WrapAroundDomainIfNeeded(boolExpr, symbolSyntaxNode);
                }

                if (_createdIntSymbols.ContainsKey(symbolSyntaxNode.Id)) return
WrapAroundDomainIfNeeded(_createdIntSymbols[symbolSyntaxNode.Id], symbolSyntaxNode);


                Symbol symbolInt = null;

                if (_createdSymbols.ContainsKey(symbolSyntaxNode.Id))
                {
                    symbolInt = _createdSymbols[symbolSyntaxNode.Id];
                }
                else
                {
                    symbolInt = _currentContext.MkSymbol(symbolSyntaxNode.Id);
                    _createdSymbols[symbolSyntaxNode.Id] = symbolInt;
                }

                var intExpr = _currentContext.MkIntConst(symbolInt);
                _createdIntSymbols[symbolSyntaxNode.Id] = intExpr;
                _createdSymbols[symbolSyntaxNode.Id] = symbolInt;
                return WrapAroundDomainIfNeeded(intExpr, symbolSyntaxNode);
            }

            if (syntaxNode is ConstantValueSyntaxNode)
            {
                var constantValueSyntaxNode = (ConstantValueSyntaxNode)syntaxNode;

                if (constantValueSyntaxNode.Value is bool) return
_currentContext.MkBool((bool)constantValueSyntaxNode.Value);
```

```
                var currentExpression =
_currentContext.MkInt((int)constantValueSyntaxNode.Value);
                return WrapAroundDomainIfNeeded(currentExpression,
constantValueSyntaxNode);
            }

            if (syntaxNode is BinaryExpressionSyntaxNode)
            {
                var binaryExpressionSyntaxNode =
(BinaryExpressionSyntaxNode)syntaxNode;

                if (binaryExpressionSyntaxNode.OperationKind == EOperationKinds.And)
                {
                    var leftExpression =
(BoolExpr)ConstructExpression(binaryExpressionSyntaxNode.LeftValue);
                    var rightExpression =
(BoolExpr)ConstructExpression(binaryExpressionSyntaxNode.RightValue);

                    var currentExpression = _currentContext.MkAnd(leftExpression,
rightExpression);
                    return WrapAroundDomainIfNeeded(currentExpression,
binaryExpressionSyntaxNode);
                }

                if (binaryExpressionSyntaxNode.OperationKind == EOperationKinds.Or)
                {
                    var leftExpression =
(BoolExpr)ConstructExpression(binaryExpressionSyntaxNode.LeftValue);
                    var rightExpression =
(BoolExpr)ConstructExpression(binaryExpressionSyntaxNode.RightValue);

                    var currentExpression = _currentContext.MkOr(leftExpression,
rightExpression);
                    return WrapAroundDomainIfNeeded(currentExpression,
binaryExpressionSyntaxNode);
                }

if (binaryExpressionSyntaxNode.OperationKind == EOperationKinds.Implication)
                {
                    var leftExpression =
(BoolExpr)ConstructExpression(binaryExpressionSyntaxNode.LeftValue);
                    var rightExpression =
(BoolExpr)ConstructExpression(binaryExpressionSyntaxNode.RightValue);

                    var currentExpression = _currentContext.MkImplies(leftExpression,
rightExpression);
                    return WrapAroundDomainIfNeeded(currentExpression,
binaryExpressionSyntaxNode);
                }

                if (binaryExpressionSyntaxNode.OperationKind == EOperationKinds.Add)
                {
                    var leftExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.LeftValue);
                    var rightExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.RightValue);

                    var currentExpression = _currentContext.MkAdd(leftExpression,
rightExpression);
                    return WrapAroundDomainIfNeeded(currentExpression,
binaryExpressionSyntaxNode);
                }
```

```
                if (binaryExpressionSyntaxNode.OperationKind ==
EOperationKinds.Divide)
                {
                    var leftExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.LeftValue);
                    var rightExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.RightValue);

                    var currentExpression = _currentContext.MkDiv(leftExpression,
rightExpression);
                    return WrapAroundDomainIfNeeded(currentExpression,
binaryExpressionSyntaxNode);
                }

                if (binaryExpressionSyntaxNode.OperationKind ==
EOperationKinds.Multiply)
                {
                    var leftExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.LeftValue);
                    var rightExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.RightValue);

                    var currentExpression = _currentContext.MkMul(leftExpression,
rightExpression);
                    return WrapAroundDomainIfNeeded(currentExpression,
binaryExpressionSyntaxNode);
                }

                if (binaryExpressionSyntaxNode.OperationKind ==
EOperationKinds.Substract)
                {
                    var leftExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.LeftValue);
                    var rightExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.RightValue);

                    var currentExpression = _currentContext.MkSub(leftExpression,
rightExpression);
                    return WrapAroundDomainIfNeeded(currentExpression,
binaryExpressionSyntaxNode);
                }

                if (binaryExpressionSyntaxNode.OperationKind == EOperationKinds.Less)
                {
                    var leftExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.LeftValue);
                    var rightExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.RightValue);

                    var currentExpression = _currentContext.MkLt(leftExpression,
rightExpression);
                    return WrapAroundDomainIfNeeded(currentExpression,
binaryExpressionSyntaxNode);
                }

                if (binaryExpressionSyntaxNode.OperationKind ==
EOperationKinds.LessOrEqual)
                {
                    var leftExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.LeftValue);
                    var rightExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.RightValue);
```

```
                    var currentExpression = _currentContext.MkLe(leftExpression,
rightExpression);
                    return WrapAroundDomainIfNeeded(currentExpression,
binaryExpressionSyntaxNode);
                }

                if (binaryExpressionSyntaxNode.OperationKind == EOperationKinds.Equal)
                {
                    var leftExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.LeftValue);
                    var rightExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.RightValue);

                    var currentExpression = _currentContext.MkEq(leftExpression,
rightExpression);
                    return WrapAroundDomainIfNeeded(currentExpression,
binaryExpressionSyntaxNode);
                }

                if (binaryExpressionSyntaxNode.OperationKind ==
EOperationKinds.GreaterOrEqual)
                {
                    var leftExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.LeftValue);
                    var rightExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.RightValue);

                    var currentExpression = _currentContext.MkGe(leftExpression,
rightExpression);
                    return WrapAroundDomainIfNeeded(currentExpression,
binaryExpressionSyntaxNode);
                }

                if (binaryExpressionSyntaxNode.OperationKind ==
EOperationKinds.Greater)
                {
                    var leftExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.LeftValue);
                    var rightExpression =
(IntExpr)ConstructExpression(binaryExpressionSyntaxNode.RightValue);

                    var currentExpression = _currentContext.MkGt(leftExpression,
rightExpression);
                    return WrapAroundDomainIfNeeded(currentExpression,
binaryExpressionSyntaxNode);
                }
            }

            if (syntaxNode is UnaryExpressionSyntaxNode)
            {
                var unaryExpressionSyntaxNode = (UnaryExpressionSyntaxNode)syntaxNode;
                var innerExpression =
(BoolExpr)ConstructExpression(unaryExpressionSyntaxNode.InnerValue);

                if (unaryExpressionSyntaxNode.OperationKind == EOperationKinds.Not)
return _currentContext.MkNot(innerExpression);
            }

            return null;
        }
```

```
        public Expr WrapAroundDomainIfNeeded(Expr currentExpression, ValueSyntaxNode
valueSyntaxNode)
        {
            if (valueSyntaxNode.DomainValue != null)
            {
                var domainExpression =
ConstructExpression(valueSyntaxNode.DomainValue.InnerValue);

                if (valueSyntaxNode.DomainValue.OperationKind == EOperationKinds.Any)
                {
                    currentExpression = _currentContext.MkForall(new Expr[] {
domainExpression },  currentExpression);
                }
                else if (valueSyntaxNode.DomainValue.OperationKind ==
EOperationKinds.Exists)
                {
                    currentExpression = _currentContext.MkExists(new Expr[] {
domainExpression }, currentExpression);
                }
            }

            return currentExpression;
        }

        private void Initialize()
        {
            _currentContext = new Context(new Dictionary<string, string> { { "mbqi",
"true" } });
            _createdBoolSymbols = new Dictionary<string, BoolExpr>();
            _createdIntSymbols = new Dictionary<string, IntExpr>();
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Tema1B_FMSE
{
    public class ExpressionExample
    {
        public string Text { get; set; }

        public ExpressionExample(string text)
        {
            Text = text;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using Tema1B_FMSE.SyntaxNodes;
using Microsoft.Z3;

namespace Tema1B_FMSE
{
    using System.Collections.ObjectModel;

    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            ExpressionExamples.Add(new ExpressionExample("x and not x"));
            ExpressionExamples.Add(new ExpressionExample("(x and y and not z) imp (x
and not z)"));
            ExpressionExamples.Add(new ExpressionExample("(x or not x) imp (x and not
x)"));
            ExpressionExamples.Add(new ExpressionExample("any (n) ((n gt 10) imp (any
(n) (n gt 15)))"));
            ExpressionExamples.Add(new ExpressionExample("any (n) ((n gt 10) imp (n ls
0))"));
            ExpressionExamples.Add(new ExpressionExample("any (n) ((n gt 10) imp (n gt
5))"));
            ExpressionExamples.Add(new ExpressionExample("any (n) ((n gt 10) imp ((n
add 15) gt 40))"));
            ExpressionExamples.Add(new ExpressionExample("any (n) ((n gt 10) imp ((n
add 15) gt 23))"));
            ExpressionExamples.Add(new ExpressionExample("(x and not x) or x"));
            ExpressionExamples.Add(new ExpressionExample("any (n) (any (m) (((n gt 10)
and (m gt 15)) imp ((n add 15 add m) gt 100)))"));
            ExpressionExamples.Add(new ExpressionExample("any (n) (any (m) (((n gt 10)
and (m gt 15)) imp ((n add 15 add m) gt 30)))"));
        }

        public ObservableCollection<ExpressionExample> ExpressionExamples { get; } =
new ObservableCollection<ExpressionExample>();


        private void BtnProve_OnClick(object sender, RoutedEventArgs e)
        {
            var syntaxTreeParser = new SyntaxTreeParser();
```

```
            var parsedSyntaxTree = syntaxTreeParser.Parse(txtExpression.Text);

            trViewSyntaxTree.ItemsSource = new List<SyntaxNode> {parsedSyntaxTree};

            var booleanExpressionsProver = new BooleanExpressionsProver();

            var isSatifiable =
booleanExpressionsProver.IsSatisfiable(parsedSyntaxTree);

            if (isSatifiable)
            {
                lblIsSatisfiableVal.Content = "Yes";
            }
            else
            {
                lblIsSatisfiableVal.Content = "No";
            }
        }

        private void Control_OnMouseDoubleClick(object sender, MouseButtonEventArgs e)
        {
            var label = (Label)sender;
            var item = (ExpressionExample)label.DataContext;
            txtExpression.Text = item.Text;
        }
    }
}
```

```
<Window x:Class="Tema1B_FMSE.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:Tema1B_FMSE"
        xmlns:syntaxNodes="clr-namespace:Tema1B_FMSE.SyntaxNodes"
        mc:Ignorable="d"
        Title="MainWindow" Height="543.287" Width="782.333">
    <Grid>
        <Button x:Name="btnProve" Content="Prove" HorizontalAlignment="Left"
Margin="10,0,0,9" Width="75"
                Click="BtnProve_OnClick" Height="29" VerticalAlignment="Bottom" />
        <TextBox x:Name="txtExpression" Height="24" Margin="11,41,9,0"
TextWrapping="Wrap"
                 Text="any (n) ((n gt 10) imp (n gt 5))" VerticalAlignment="Top" />
        <Button x:Name="btnExit" Content="Exit" HorizontalAlignment="Left"
Margin="90,0,0,9" Width="75" Height="29"
                VerticalAlignment="Bottom" />
        <Label x:Name="lblLegend" Content="Legend: and=and, or=or, not=not,
implication=imp " Margin="0,0,10,9"
               HorizontalAlignment="Right" Width="286" Height="27"
VerticalAlignment="Bottom" />
        <Label x:Name="lblExpression" Content="Expression:" HorizontalAlignment="Left"
Margin="11,10,0,0"
               VerticalAlignment="Top" />
        <TreeView x:Name="trViewSyntaxTree" Margin="11,245,10,43">
```

```
            <TreeView.Resources>
                <HierarchicalDataTemplate DataType="{x:Type
syntaxNodes:BinaryExpressionSyntaxNode }"
                                          ItemsSource="{Binding Children}">
                    <Border BorderBrush="#FF000000" BorderThickness="1,1,1,1"
CornerRadius="4,4,4,4" Margin="2">
                        <WrapPanel>
                            <Label Content="Type: " />
                            <Label Content="{Binding Path=Name}" />
                            <Label Content="OperationKind: " />
                            <Label Content="{Binding Path=OperationKind}" />
                            <Label Content="SymbolKind: " />
                            <Label Content="{Binding Path=SymbolKind}" />
                        </WrapPanel>
                    </Border>
                </HierarchicalDataTemplate>
                <HierarchicalDataTemplate DataType="{x:Type
syntaxNodes:UnaryExpressionSyntaxNode }"
                                          ItemsSource="{Binding Children}">
                    <Border BorderBrush="#FF000000" BorderThickness="1,1,1,1"
CornerRadius="4,4,4,4" Margin="2">
                        <WrapPanel>
                            <Label Content="Type: " />
                            <Label Content="{Binding Path=Name}" />
                            <Label Content="OperationKind: " />
                            <Label Content="{Binding Path=OperationKind}" />
                            <Label Content="SymbolKind: " />
                            <Label Content="{Binding Path=SymbolKind}" />
                        </WrapPanel>
                    </Border>
                </HierarchicalDataTemplate>
                <HierarchicalDataTemplate DataType="{x:Type
syntaxNodes:DomainValueSyntaxNode }"
                                          ItemsSource="{Binding Children}">
                    <Border BorderBrush="#FF000000" BorderThickness="1,1,1,1"
CornerRadius="4,4,4,4" Margin="2">
                        <WrapPanel>
                            <Label Content="Type: " />
                            <Label Content="{Binding Path=Name}" />
                            <Label Content="OperationKind: " />
                            <Label Content="{Binding Path=OperationKind}" />
                            <Label Content="SymbolKind: " />
                            <Label Content="{Binding Path=SymbolKind}" />
                        </WrapPanel>
                    </Border>
                </HierarchicalDataTemplate>
                <HierarchicalDataTemplate DataType="{x:Type
syntaxNodes:LiteralSyntaxNode }"
                                          ItemsSource="{Binding Children}">
                    <Border BorderBrush="#FF000000" BorderThickness="1,1,1,1"
CornerRadius="4,4,4,4" Margin="2">
                        <WrapPanel>
                            <Label Content="Type: " />
                            <Label Content="{Binding Path=Name}" />
                            <Label Content="Text: " />
                            <Label Content="{Binding Path=LiteralValue}" />
                        </WrapPanel>
                    </Border>
                </HierarchicalDataTemplate>
                <HierarchicalDataTemplate DataType="{x:Type
syntaxNodes:ConstantValueSyntaxNode }"
                                          ItemsSource="{Binding Children}">
```

```
                    <Border BorderBrush="#FF000000" BorderThickness="1,1,1,1"
CornerRadius="4,4,4,4" Margin="2">
                        <WrapPanel>
                            <Label Content="Type: " />
                            <Label Content="{Binding Path=Name}" />
                            <Label Content="Text: " />
                            <Label Content="{Binding Path=LiteralValue}" />
                            <Label Content="SymbolKind: " />
                            <Label Content="{Binding Path=SymbolKind}" />
                        </WrapPanel>
                    </Border>
                </HierarchicalDataTemplate>
                <HierarchicalDataTemplate DataType="{x:Type
syntaxNodes:SymbolSyntaxNode }"
                                          ItemsSource="{Binding Children}">
                    <Border BorderBrush="#FF000000" BorderThickness="1,1,1,1"
CornerRadius="4,4,4,4" Margin="2">
                        <WrapPanel>
                            <Label Content="Type: " />
                            <Label Content="{Binding Path=Name}" />
                            <Label Content="Id: " />
                            <Label Content="{Binding Path=Id}" />
                            <Label Content="SymbolKind: " />
                            <Label Content="{Binding Path=SymbolKind}" />
                        </WrapPanel>
                    </Border>
                </HierarchicalDataTemplate>
                <HierarchicalDataTemplate DataType="{x:Type syntaxNodes:SyntaxNode }"
ItemsSource="{Binding Children}">
                    <Border BorderBrush="#FF000000" BorderThickness="1,1,1,1"
CornerRadius="4,4,4,4" Margin="2">
                        <WrapPanel>
                            <Label Content="Type: " />
                            <Label Content="{Binding Path=Name}" />
                        </WrapPanel>
                    </Border>
                </HierarchicalDataTemplate>
            </TreeView.Resources>
        </TreeView>
        <Label x:Name="lblSyntaxTree" Content="Syntax Tree:"
HorizontalAlignment="Left" Margin="13,214,0,0"
               VerticalAlignment="Top" />
        <Label x:Name="lblIsSatisfiable" Content="Is Satisfiable:"
HorizontalAlignment="Left" Margin="184,0,0,10"
               Height="26" VerticalAlignment="Bottom" />
        <Label x:Name="lblIsSatisfiableVal" Content="" HorizontalAlignment="Left"
Margin="267,0,0,10" Height="26"
               VerticalAlignment="Bottom" />
        <ListBox x:Name="lstExamples" Height="108" Margin="11,101,10,0"
                 VerticalAlignment="Top"
                 ItemsSource="{Binding RelativeSource={RelativeSource
Mode=FindAncestor,
                                                       AncestorType={x:Type
Window}},
                                                       Path=ExpressionExamples}">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <Label Content="{Binding Text}"
MouseDoubleClick="Control_OnMouseDoubleClick" />
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
```

```
        <Label x:Name="lblExamples" Content="Examples (double click to insert):"
HorizontalAlignment="Left"
               Margin="13,70,0,0" VerticalAlignment="Top" />
    </Grid>
</Window>
```