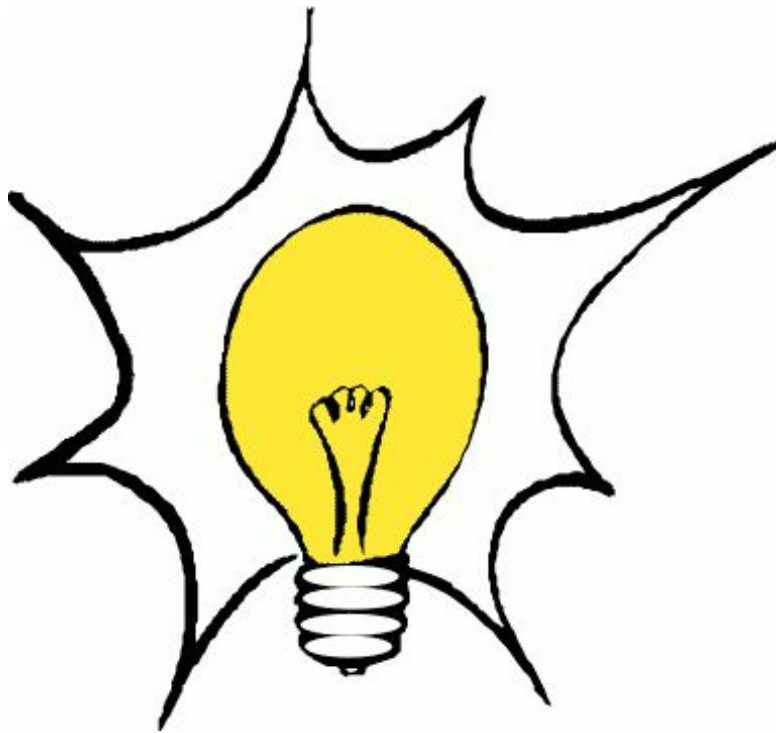

ALE 1 - Design Documentation

Saturday, 06.04.2019



Summary

The following document contains information about the implementation of the functionalities and features in the software application. For each assignment I will make a description, comment on the additional features that I have added and when necessary mention the teammate that I have worked with. I will also describe the software design, the GUI and the tests that I have implemented. The last chapter is represented by a conclusion.

Personal Details

Amira Catalina Cruceru

363690

Summary	1
Personal Details	1
Assignment 1: Parse + Tree	3
Sources	5
Assignment 2: Truth table + Hash code	5
Sources	6
Assignment 3: Simplify	7
Sources	7
Assignment 4: Normalize	8
Sources:	8
Introduction 5: Nandify	9
Sources:	9
Software Design	10
GUI	11
Testing	16
Proposition Tests	16
Service Tests	17
Truth Table Tests	17
Conclusions and future implementations	17

Assignment 1: Parse + Tree

In this chapter I will describe the first assignment which consisted of parsing the logical propositions and creating the tree. Also, extra details about the initiation phase of the project have been added in this chapter.

I chose to create this project in C# as it is the most convenient programming language for me, due to the experience that I have gained during University. In the application the ASCII prefix has been used for the operators.

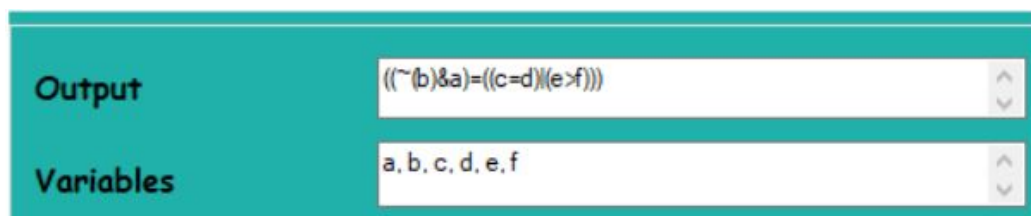
Operators (logical notations):

- ★ '¬' (negation) which is actually '~' in ASCII
- ★ '⇒' (implication) which is actually '>' in ASCII
- ★ '⇔' (bi implication) which is actually '=' in ASCII
- ★ '∧' (conjunction) which is actually '&' in ASCII
- ★ '∨' (disjunction) which is actually '|' in ASCII

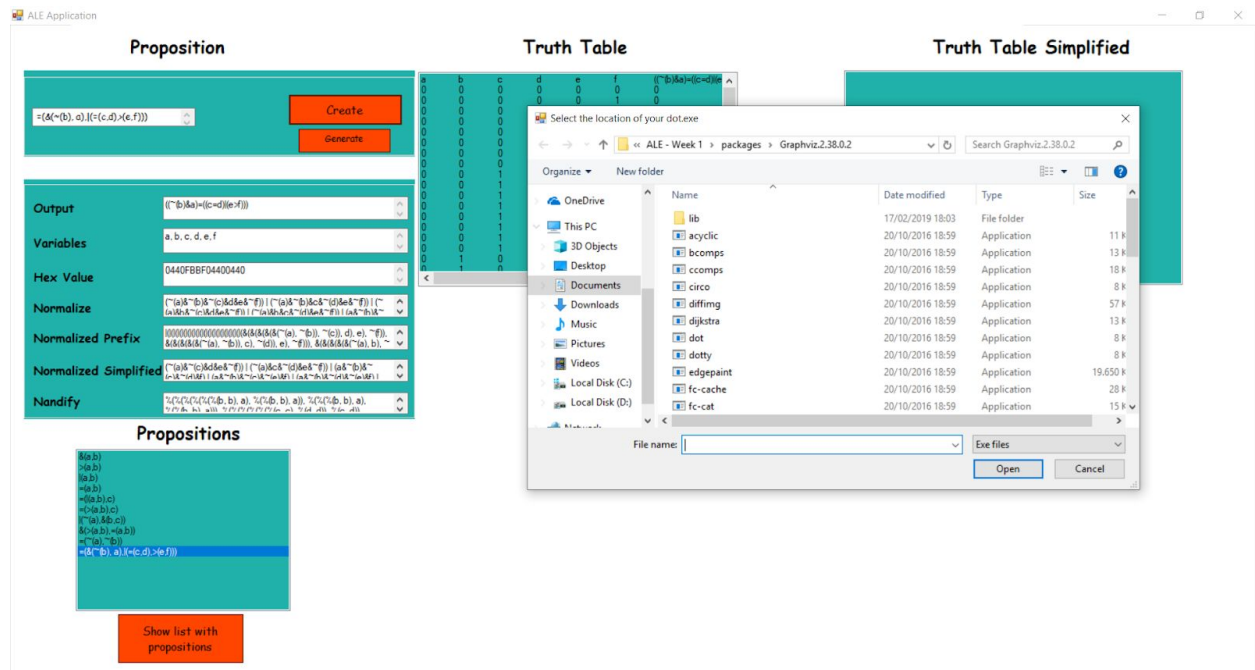
Operands: represented by letters from 'a' to 'z'

Classes created for this assignment: Node.cs, Components.cs, Proposition.cs, Service.cs. (the classes will be explained in the **Software Design** chapter).

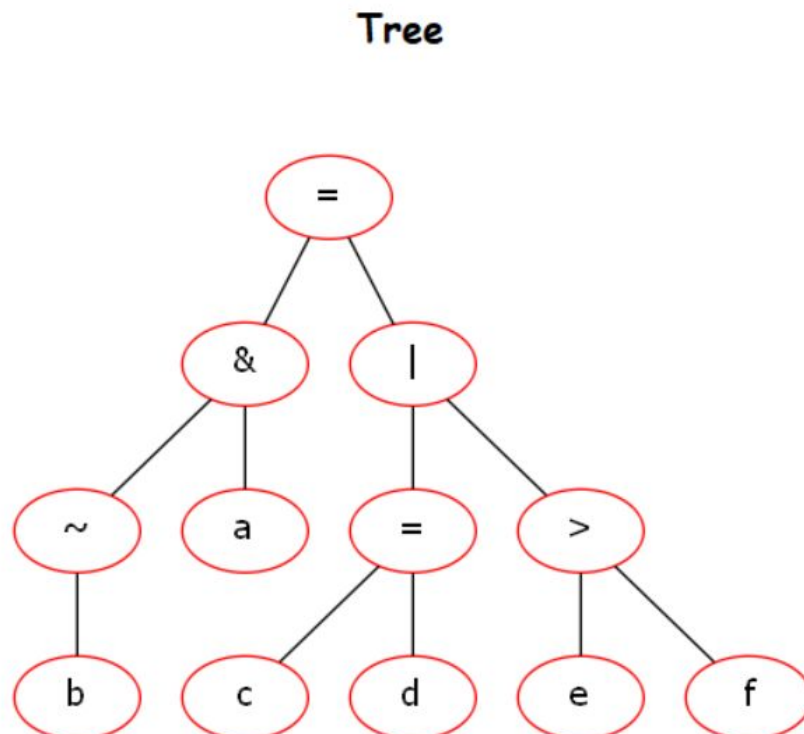
In this picture you can notice the parsing ('Output' label) of the proposition present in the text box and the variables ('Variables' label) that are ordered alphabetically.



In order to generate the tree, you need to choose a file where to read the graph from.



In this picture you can notice the tree created from the proposition $\neg(b) \wedge a \vee ((c \vee d) \wedge (e \vee f))$ which is the same as $((\neg(b) \wedge a) \vee ((c \vee d) \wedge (e \vee f)))$.



Sources

- URL: <http://graphviz.org/>
- Website Title: Graphviz
- Article Title: Graph Visualization Software
- URL: <https://www.sanfoundry.com/cpp-program-implement-expression-tree-prefix/>
- Website Title: Sanfoundry
- Article Title: C++ Program to Construct an Expression Tree for a Given Prefix Expression
- Date Published: July 06, 2017

Assignment 2: Truth table + Hash code

For the second assignment I had to create three extra classes: TruthTable and TruthTableComponents(TruthTableRow & TruthTableStructure). The TruthTable class contains all the truth table logic, the TruthTableRow contains two constructors and a method used for simplified (see [Simplify](#)) and the TruthTableStructure class contains the structure of the truth table (description and rows).

In the following image you can see the truth table of the following proposition: $=(&(\sim(b), a), |(=(c,d), >(e,f)))$.

Truth Table

a	b	c	d	e	f	$((\sim(b) \& a) = ((c = d) \& (e > f)))$
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	0	1	0	1	0
0	0	0	1	1	0	1
0	0	0	1	1	1	0
0	0	1	0	0	0	0
0	0	1	0	0	1	0
0	0	1	0	1	0	1
0	0	1	0	1	1	0
0	0	1	1	0	0	0
0	0	1	1	0	1	0
0	0	1	1	1	0	0
0	0	1	1	1	1	0
0	1	0	0	0	0	0
0	1	0	0	0	1	0

The hash code was calculated by getting the last row's values and read them from bottom to top. The hash code value appears in a text box in the following image.

Proposition

Create

Generate

Output	<input style="width: 90%;" type="text" value="((¬(b)&a)¬((c=d)¬(e>f)))"/>
Variables	<input style="width: 90%;" type="text" value="a, b, c, d, e, f"/>
Hex Value	<input style="width: 90%;" type="text" value="0440FBBF04400440"/>

Sources

- URL: <https://github.com/iasjem/truth-table-java>
- Website Title: GitHub
- Article Title: iasjem/truth-table-java
- Date Published: February 02, 2018

- URL: <https://stackoverflow.com/questions/16340/how-do-i-generate-a-hashcode-from-a-byte-array-in-c>
- Website Title: Stack Overflow
- Article Title: How do I generate a hashcode from a byte array in C#?

- URL: <https://github.com/pedrooaugusto/truth-table-generator>
- Website Title: GitHub
- Article Title: pedrooaugusto/truth-table-generator
- Date Published: October 25, 2017

- URL: <http://qaru.site/questions/430508/convert-long-string-of-binary-to-hex-c>
- Website Title: qaru.site
- Article Title: Преобразование длинной строки двоичного кода в hex C#
- Date Published: April 10, 2011

Assignment 3: Simplify

In my opinion the simplification was the hardest and for this one I had thought it through together with Denisa Apostol. For this assignment I used Quine–McCluskey algorithm and I have updated the classes: TruthTable and TruthTableRow. An example of the simplification can be found below.

Truth Table Simplified

a	b	c	d	e	f	$((\neg(b) \& a) = ((c=d)(e>f)))$
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	0	1	0	1	0
0	0	0	1	1	1	0
0	0	1	0	0	0	0
0	0	1	0	0	1	0
0	0	1	0	1	1	0
0	0	1	1	0	0	0
0	0	1	1	0	1	0
0	0	1	1	1	0	0
0	0	1	1	1	1	0
0	1	0	0	0	0	0
0	1	0	0	0	1	0
0	1	0	0	1	0	0
0	1	0	0	1	1	0
0	1	0	1	0	0	0
0	1	0	1	0	1	0
0	1	0	1	1	1	0
0	1	1	0	0	0	0
0	1	1	0	0	1	0
0	1	1	0	1	0	0
0	1	1	0	1	1	0
0	1	1	1	0	0	0
0	1	1	1	0	1	0
0	1	1	1	1	1	0
1	0	0	0	0	0	0
1	0	0	0	0	1	0
1	0	0	0	1	0	0
1	0	0	0	1	1	0
1	0	0	1	0	0	0
1	0	0	1	0	1	0
1	0	0	1	1	1	0
1	0	1	0	0	0	0
1	0	1	0	0	1	0
1	0	1	0	1	0	0
1	0	1	0	1	1	0
1	0	1	1	0	0	0
1	0	1	1	0	1	0
1	0	1	1	1	1	0
1	1	0	0	0	0	0
1	1	0	0	0	1	0
1	1	0	0	1	0	0
1	1	0	0	1	1	0
1	1	0	1	0	0	0
1	1	0	1	0	1	0
1	1	0	1	1	1	0
1	1	1	0	0	0	0
1	1	1	0	0	1	0
1	1	1	0	1	0	0
1	1	1	0	1	1	0
1	1	1	1	0	0	0
1	1	1	1	0	1	0
1	1	1	1	1	1	0
1	1	1	1	1	1	0

Sources

- URL: <https://github.com/branchwelder/simplylogical>
- Website Title: GitHub
- Article Title: branchwelder/simplylogical
- Date Published: February 21, 2017

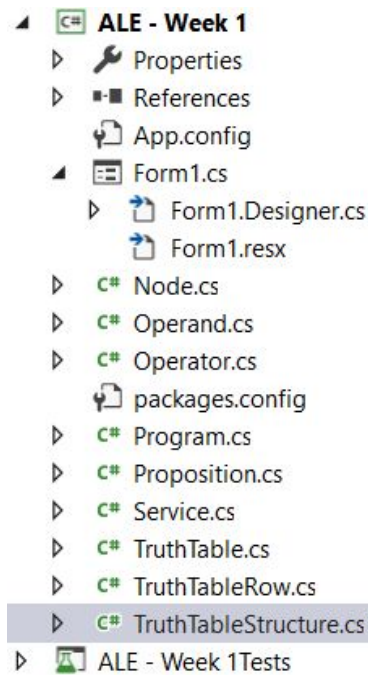
Sources:

Software Design

In the software design I have the project (ALE - Week 1) which contains the logic of the project and the form application (User Interface - detailed in the [GUI](#) chapter).

The logic of the application consists of the following classes:

- ★ **Node** - This class contains information about the node itself.
- ★ **Components** - Class containing the constructor of the operator, the constructor of the operands and a method for setting the true or false value needed for the truth table.
- ★ **Proposition** - Class containing the logic of creating the propositions, checking if one of the inputted characters is a letter/operator/negation, creating the tree, checking the truth sign for the truth table and creating the nandify version of the proposition.
- ★ **TruthTable** - This class contains all the logic behind the truth table, simplified truth table and normalization matters.
- ★ **TruthTableComponents**- TruthTableRow: This class contains two constructors, one with a list of rows as parameter and one with the result and a values list as parameters. It also contains a method for comparing the rows in order to help creating the simplified table. TruthTableStructure: This class contains two lists, one for storing the rows and one for storing the description (proposition - in order to show it on the first line of the table) and one constructor which has as parameters the two lists.
- ★ **Service** - The service class is used for posting all the functionalities provided by the other classes regarding the logic of the application.



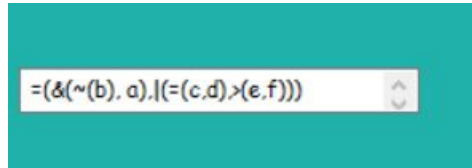
GUI

In terms of GUI I chose to arrange the elements in such a way that all of them fit without being too crowded. On the GUI page On the form, information about the truth table, simplified truth table, hex value, normalize, nandify, tree and a list with propositions can be found.

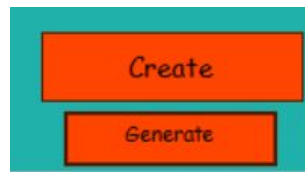
The screenshot displays the ALE Application interface with four main panels:

- Proposition:** Contains a text input field with the expression $\neg(a \rightarrow (b, c)) \vee ((c, d) \rightarrow (e, f))$, a "Create" button, and a "Generate" button. Below this is an "Output" section showing the same expression, a "Variables" list (a, b, c, d, e, f), a "Hex Value" (0440FBFB04400440), and sections for "Normalize", "Normalized Prefix", "Normalized Simplified", and "Nandify".
- Truth Table:** A table with 6 columns (a, b, c, d, e, f) and 64 rows, showing the truth values for the proposition. The title is "Truth Table".
- Truth Table Simplified:** A simplified version of the truth table, showing only the rows where the proposition is true. The title is "Truth Table Simplified".
- Tree:** A logic tree diagram showing the hierarchical structure of the proposition. The root node is "=", which branches into "&" and "|". "&" branches into "~" and "a", which further branch into "b" and "c". "|" branches into "=" and ">", which further branch into "d", "e", and "f".

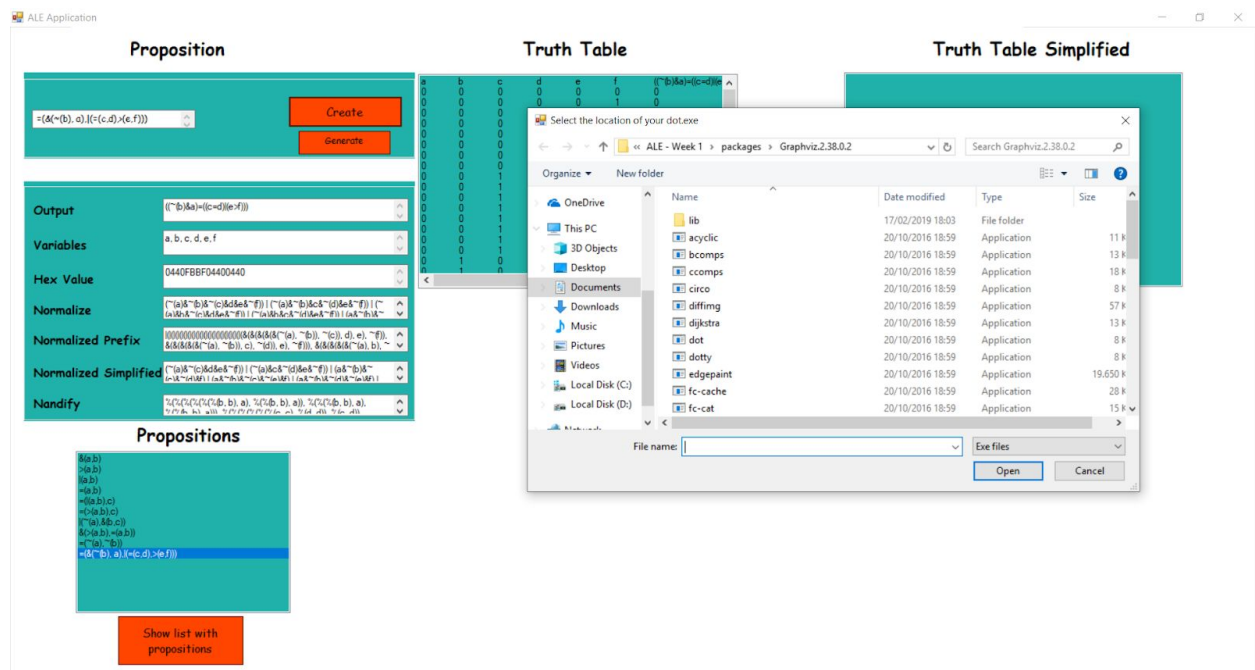
In the following image there is a text box where the proposition is inputed.



On the form, there are two buttons one (Create) for generating the proposition, variables, hex value, normalization, nandify, tree and truth table and one button (Generate) for generating the simplified version of the truth table.



When pressing the 'Create' button, a console it's opening asking for choosing the file where to read the information about the tree from.



In order to make it easier, I also created a list with propositions to automatically update into the input box of the proposition. You can do this by first pressing the button 'Show list with propositions' and then pressing one of them from the list. The chosen proposition will pop up in the input box.

Propositions

$\&(a,b)$
 $>(a,b)$
 $!(a,b)$
 $=(a,b)$
 $=(!(a,b),c)$
 $=(>(a,b),c)$
 $!(\sim(a),\&(b,c))$
 $\&>(>(a,b),=(a,b))$
 $=(\sim(a),\sim(b))$
 $=(\&(\sim(b), a),!(=(c,d),>(e,f)))$

Show list with
propositions

On the form, there are two list boxes for the truth table and the simplified version of the truth table. The lists can be scrolled both vertically and horizontally.

Truth Table

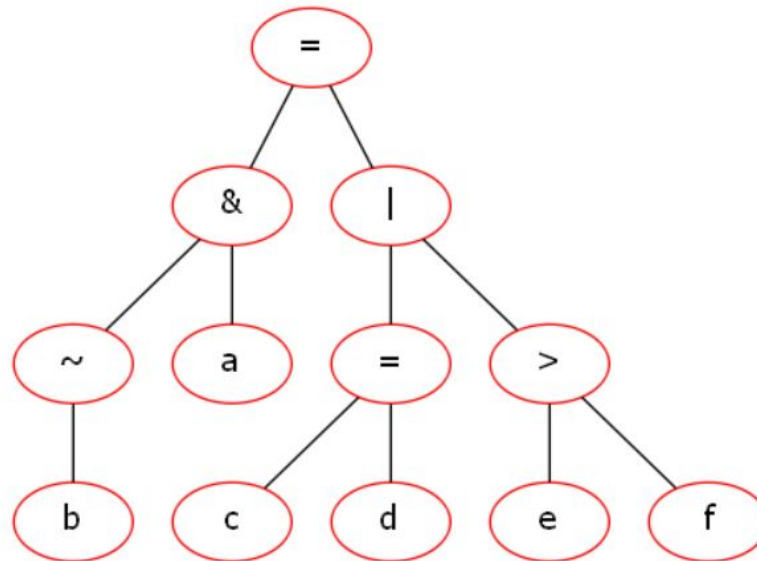
a	b	c	d	e	f	$((\neg(b) \& a) = ((c=d) \& (e > f)))$
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	0	1	0	1	0
0	0	0	1	1	0	1
0	0	0	1	1	1	0
0	0	1	0	0	0	0
0	0	1	0	0	1	0
0	0	1	0	1	0	1
0	0	1	0	1	1	0
0	0	1	1	0	0	0
0	0	1	1	0	1	0
0	0	1	1	1	0	0
0	0	1	1	1	1	0
0	1	0	0	0	0	0
0	1	0	0	0	1	0

Truth Table Simplified

a	b	c	d	e	f	$((\neg(b) \& a) = ((c=d) \& (e > f)))$
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	0	1	0	1	0
0	0	0	1	1	1	0
0	0	1	0	0	0	0
0	0	1	0	0	1	0
0	0	1	0	1	1	0
0	0	1	1	0	0	0
0	0	1	1	0	1	0
0	0	1	1	1	0	0
0	0	1	1	1	1	0
0	1	0	0	0	0	0
0	1	0	0	0	1	0
0	1	0	0	1	0	0
0	1	0	0	1	1	0
0	1	0	1	0	0	0

The tree is shown in a picture box like in the following image.

Tree



All other assignments (variables, hex value, normalization and nandify) are shown in the following box. The text boxes can be scrolled vertically.

Proposition

=(&(~(b), a),!(=(c,d)>(e,f)))

CreateGenerate

[illegible]

Testing

In order to test the application, I have created several unit tests. For covering as much as I could of the app I created tests for simple and complicated propositions, errors, different operators and all functionalities except the tree. I also added a print screen of the code coverage that I managed to test (72%) and what percent is still left for testing(28%).

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
363690_DESKTOP-PNPSI78 2019-0...	648	28,00%	1666	72,00%
ale - week 1.exe	648	41,20%	925	58,80%
ALE__Week_1	643	41,01%	925	58,99%
Form1	538	100,00%	0	0,00%
Node	1	10,00%	9	90,00%
Operand	3	18,75%	13	81,25%
Operator	0	0,00%	2	100,00%
Program	5	100,00%	0	0,00%
Proposition	58	16,67%	290	83,33%
Service	32	14,81%	184	85,19%
Service.<>c	0	0,00%	15	100,00%
TruthTable	6	1,88%	313	98,12%
TruthTable.<>c	0	0,00%	14	100,00%
TruthTable.<>c_Display...	0	0,00%	2	100,00%
TruthTable.<>c_Display...	0	0,00%	4	100,00%
TruthTable.<>c_Display...	0	0,00%	4	100,00%
TruthTableRow	0	0,00%	65	100,00%
TruthTableRow.<>c	0	0,00%	2	100,00%
TruthTableStructure	0	0,00%	8	100,00%
ALE__Week_1.Properties	5	100,00%	0	0,00%
ale - week 1tests.dll	0	0,00%	741	100,00%

Proposition Tests

In the following image there are all the tests I have for the proposition.

PropositionTests (10)	38 ms
PropositionTestBigProposition	< 1 ms
PropositionTestBiimplication	< 1 ms
PropositionTestConjunction	< 1 ms
PropositionTestDisjunction	< 1 ms
PropositionTestErrorWhenEmpty	1 ms
PropositionTestErrorWhenWrongOperator	< 1 ms
PropositionTestImplication	< 1 ms
PropositionTestNegation	34 ms
PropositionTestNormalProposition	< 1 ms
PropositionTestWithSpaces	< 1 ms

Service Tests

In the following image there are all the tests I have for the service.

▲ ✓ ServiceTests (8)	67 ms
✓ GetVariablesTest	11 ms
✓ HexTest	24 ms
✓ NandifyBigProposi...	< 1 ms
✓ NandifyNegationT...	< 1 ms
✓ NandifyNormalProp...	3 ms
✓ NormalizedPrefixT...	< 1 ms
✓ NormalizedSimplifi...	24 ms
✓ NormalizedTest	2 ms

Truth Table Tests

In the following image there are all the tests that I have for the truth table.

▲ ✓ TruthTableTests (8)	42 ms
✓ TruthTableBigProposition	1 ms
✓ TruthTableNormalProposition	< 1 ms
✓ TruthTableSimpleProposition	1 ms
✓ TruthTableSimplifiedBigProposition	1 ms
✓ TruthTableSimplifiedNormalProposition	1 ms
✓ TruthTableSimplifiedSimpleProposition	< 1 ms
✓ TruthTableSimplifiedTautology	35 ms
✓ TruthTableTatology	< 1 ms

Conclusions and future implementations

This course taught me how to use all the knowledge that I gathered during the logic mathematics classes. It was not an easy course, as a lot of hours needed to be put in, in order to finish the application.

I think that the application can be improved by letting the user input propositions in infix and postfix forms as well in the input box. Of course, the application can still be improved with more error messages and a better structure of the software design.

Appendix 1

In the following image I have tested the negations.

```
[TestMethod]
0 references
public void PropositionTestNegation()
{
    string input = "~(A)";
    string output = "~(A)";
    Proposition check = new Proposition(input);
    Assert.AreEqual(output, check.ToString());
}
```

In the following image I have tested a simple proposition containing a conjunction.

```
[TestMethod]
0 references
public void PropositionTestConjunction()
{
    string input = "&(A,B)";
    string output = "(A&B)";
    Proposition check = new Proposition(input);
    Assert.AreEqual(output, check.ToString());
}
```

In the following image I have tested a simple proposition containing a disjunction.

```
[TestMethod]
0 references
public void PropositionTestDisjunction()
{
    string input = "|(A,B)";
    string output = "(A|B)";
    Proposition check = new Proposition(input);
    Assert.AreEqual(output, check.ToString());
}
```

In the following image I have tested a simple proposition containing an implication.

```

[TestMethod]
0 references
public void PropositionTestImplication()
{
    string input = ">(A,B)";
    string output = "(A>B)";
    Proposition check = new Proposition(input);
    Assert.AreEqual(output, check.ToString());
}

```

In the following image I have tested a simple proposition containing a bi implication.

```

[TestMethod]
0 references
public void PropositionTestBiimplication()
{
    string input = "=(A,B)";
    string output = "(A=B)";
    Proposition check = new Proposition(input);
    Assert.AreEqual(output, check.ToString());
}

```

In the following image I have tested a more complicated proposition containing two operators and three variables.

```

[TestMethod]
0 references
public void PropositionTestNormalProposition()
{
    string input = "=(A,|(C,B))";
    string output = "(A=(C|B))";
    Proposition check = new Proposition(input);
    Assert.AreEqual(output, check.ToString());
}

```

In the following image I have tested a proposition which contains spaces.

```
[TestMethod]
0 references
public void PropositionTestWithSpaces()
{
    string input = "=( | (A ,B), &(C ,D))";
    string output = "((A|B)=(C&D))";
    Proposition check = new Proposition(input);
    Assert.AreEqual(output, check.ToString());
}
```

In the following image I have tested a 'wrong operator' input to see if the error message is triggered.

```
[TestMethod]
0 references
public void PropositionTestErrorWhenWrongOperator()
{
    string input = "+(A,B)";
    string output = "Ups, you did something wrong!";
    Proposition check = new Proposition(input);
    Assert.AreEqual(output, check.ToString());
}
```

In the following image I have tested an empty input to see if the error message is triggered.

```
[TestMethod]
0 references
public void PropositionTestErrorWhenEmpty()
{
    string input = "";
    string output = "Ups, you did something wrong!";
    Proposition check = new Proposition(input);
    Assert.AreEqual(output, check.ToString());
}
```

In the following image I have tested if the correct number of variables is returned from a proposition.

```

[TestMethod]
0 references
public void GetVariablesTest()
{
    string input = "=&(~(b), a),|(<=(c,d),>(e,f)))";
    Proposition check = new Proposition(input);
    Assert.AreEqual(6, check.GetVariables().Count());
}

```

In the following image I have tested if the correct hexadecimal output is calculated.

```

[TestMethod]
0 references
public void HexTest()
{
    string input = "&(a,b)";
    string output = "08";

    Service check = new Service();
    check.Proposition(input);
    var truthTableHex = check.TruthTableHex();

    Assert.AreEqual(output, truthTableHex);
}

```

In the following image I have tested if the correct normalized output is given.

```

[TestMethod]
0 references
public void NormalizedTest()
{
    string input = "(>(a,b),c)";
    string output = "(\~(a)&\~(b)&c) | (\~(a)&b&c) | (a&\~(b)&\~(c)) | (a&b&c)";

    Service check = new Service();
    check.Proposition(input);
    var normalized = check.GetNormalized();

    Assert.AreEqual(output, normalized);
}

```

In the following image I have tested if the correct normalized form of the prefix is given.

```

[TestMethod]
0 references
public void NormalizedPrefixTest()
{
    string input = "(>(a,b),c)";
    string output = "!(|(|(&(\~(a), \~(b)), c), &(\~(a), b), c)), &(\~(a), \~(b)), \~(c))), &(\~(a), b), c))";

    Service check = new Service();
    check.Proposition(input);
    var normalizedPrefix = check.GetNormalizedPrefix();

    Assert.AreEqual(output, normalizedPrefix);
}

```

In the following image I have tested if the correct normalized form of the simplify is given.

```

[TestMethod]
0 references
public void NormalizedSimplifiedTest()
{
    string input = "(>(a,b),c)";
    string output = "(\~(a)&c) | (b&c)";

    Service check = new Service();
    check.Proposition(input);
    var normalizedSimplified = check.GetNormalizedSimplified();

    Assert.AreEqual(output, normalizedSimplified);
}

```


In the following image I have tested if the correct nandified form of a normal proposition was generated correctly.

```
[TestMethod]
public void NandifyNormalPropositionTest()
{
    string input = "|(|(a,b),c)";
    string output = "%(%%(%%(a, a), %(b, b)), %(a, a), %(b, b))), %(c, c))";

    Service check = new Service();
    check.Proposition(input);
    string nandify = check.Nandify();
    Assert.AreEqual(output, nandify);
}
```

In the following image I have tested if the correct nandified form of a more complicated proposition was generated correctly.

```
[TestMethod]
public void NandifyBigPropositionTest()
{
    string input = "&(&(~(b), a),|(&(c,d),>(e,f)))";
    string output = "%(%%(%%(%%(b, b), a), %(b, b), a)), %(b, b), a)), %(b, b), a))), %(%%(%%(c, c), %(d, d)), " +
        "%(c, d)), %(c, c), %(d, d)), %(c, d)), %(e, %(f, f)), %(e, %(f, f))), %(%%(%%(c, c), %(d, d)), %(c, d)), " +
        "%(%%(c, c), %(d, d)), %(c, d)), %(e, %(f, f)), %(e, %(f, f))))), %(%%(b, b), a), %(b, b), a)), %(%%(c, c), " +
        "%(d, d)), %(c, d)), %(c, c), %(d, d)), %(c, d)), %(e, %(f, f)), %(e, %(f, f))))))";

    Service check = new Service();
    check.Proposition(input);
    string nandify = check.Nandify();
    Assert.AreEqual(output, nandify);
}
```

In the following image I have tested if the correct nandified form of a negation was generated correctly.

```
[TestMethod]
public void NandifyNegationTest()
{
    string input = "~(a)";
    string output = "%(a, a)";

    Service check = new Service();
    check.Proposition(input);
    string nandify = check.Nandify();
    Assert.AreEqual(output, nandify);
}
```

In the following image I have tested if the **simplified truth table** for a tautology is generated correctly.

```
[TestMethod]
0 references
public void TruthTableSimplifiedTatology()
{
    string input = ">(&(a,b),&(a,b))";

    Service check = new Service();
    check.Proposition(input);
    var truthTable = check.GetSimplifiedTable();

    Assert.AreEqual(1, truthTable.Rows.Count);
    Assert.AreEqual("((a&b)>(a&b))", truthTable.Description.Last());
    Assert.AreEqual("*", truthTable.Rows[0].Values[0]);
    Assert.AreEqual("*", truthTable.Rows[0].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[0].Result);
}
```


In the following image I have tested if the **simplified truth table** for a normal length proposition is generated correctly.

```
[TestMethod]
0 references
public void TruthTableSimplifiedNormalProposition()
{
    string input = "(>(a,b),c)";

    Service check = new Service();
    check.Proposition(input);
    var truthTable = check.GetSimplifiedTable();

    Assert.AreEqual(6, truthTable.Rows.Count);
    Assert.AreEqual("((a>b)=c)", truthTable.Description.Last());

    Assert.AreEqual("0", truthTable.Rows[0].Values[0]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[1]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[2]);
    Assert.AreEqual("0", truthTable.Rows[0].Result);

    Assert.AreEqual("0", truthTable.Rows[1].Values[0]);
    Assert.AreEqual("1", truthTable.Rows[1].Values[1]);
    Assert.AreEqual("0", truthTable.Rows[1].Values[2]);
    Assert.AreEqual("0", truthTable.Rows[1].Result);

    Assert.AreEqual("1", truthTable.Rows[2].Values[0]);
    Assert.AreEqual("0", truthTable.Rows[2].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[2].Values[2]);
    Assert.AreEqual("0", truthTable.Rows[2].Result);

    Assert.AreEqual("1", truthTable.Rows[3].Values[0]);
    Assert.AreEqual("1", truthTable.Rows[3].Values[1]);
    Assert.AreEqual("0", truthTable.Rows[3].Values[2]);
    Assert.AreEqual("0", truthTable.Rows[3].Result);

    Assert.AreEqual("0", truthTable.Rows[4].Values[0]);
    Assert.AreEqual("1", truthTable.Rows[4].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[4].Values[2]);
    Assert.AreEqual("1", truthTable.Rows[4].Result);

    Assert.AreEqual("1", truthTable.Rows[5].Values[0]);
    Assert.AreEqual("1", truthTable.Rows[5].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[5].Values[2]);
    Assert.AreEqual("1", truthTable.Rows[5].Result);
}
```

In the following image I have tested if the **simplified truth table** for a simple proposition is correctly generated.

```
[TestMethod]
0 references
public void TruthTableSimplifiedSimpleProposition()
{
    string input = ">(a,b)";

    Service check = new Service();
    check.Proposition(input);
    var truthTable = check.GetSimplifiedTable();

    Assert.AreEqual(3, truthTable.Rows.Count);
    Assert.AreEqual("(a>b)", truthTable.Description.Last());

    Assert.AreEqual("1", truthTable.Rows[0].Values[0]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[1]);
    Assert.AreEqual("0", truthTable.Rows[0].Result);

    Assert.AreEqual("0", truthTable.Rows[1].Values[0]);
    Assert.AreEqual("*", truthTable.Rows[1].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[1].Result);

    Assert.AreEqual("*", truthTable.Rows[2].Values[0]);
    Assert.AreEqual("1", truthTable.Rows[2].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[2].Result);
}
```

In the following image I have tested if the **simplified truth table** for a complicated proposition is generated correctly.

```
[TestMethod]
0 references
public void TruthTableSimplifiedBigProposition()
{
    string input = "=(~(b), a),|=(c,d),>(e,f)))";

    Service check = new Service();
    check.Proposition(input);
    var truthTable = check.GetSimplifiedTable();

    Assert.AreEqual(58, truthTable.Rows.Count);
    Assert.AreEqual("((~(b)&a)=((c=d)|(e>f)))", truthTable.Description.Last());

    Assert.AreEqual("0", truthTable.Rows[0].Values[0]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[1]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[2]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[3]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[4]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[5]);
    Assert.AreEqual("0", truthTable.Rows[0].Result);

    Assert.AreEqual("1", truthTable.Rows[53].Values[0]);
    Assert.AreEqual("0", truthTable.Rows[53].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[53].Values[2]);
    Assert.AreEqual("*", truthTable.Rows[53].Values[3]);
    Assert.AreEqual("0", truthTable.Rows[53].Values[4]);
    Assert.AreEqual("0", truthTable.Rows[53].Values[5]);
    Assert.AreEqual("1", truthTable.Rows[53].Result);

    Assert.AreEqual("1", truthTable.Rows[57].Values[0]);
    Assert.AreEqual("0", truthTable.Rows[57].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[57].Values[2]);
    Assert.AreEqual("1", truthTable.Rows[57].Values[3]);
    Assert.AreEqual("1", truthTable.Rows[57].Values[4]);
    Assert.AreEqual("*", truthTable.Rows[57].Values[5]);
    Assert.AreEqual("1", truthTable.Rows[57].Result);
}
```

In the following image I have tested if the **truth table** for a simple proposition is generated correctly.

```
[TestMethod]
0 references
public void TruthTableSimpleProposition()
{
    string input = "(a|b)";

    Service check = new Service();
    check.Proposition(input);
    var truthTable = check.GetTable();

    Assert.AreEqual(4, truthTable.Rows.Count);
    Assert.AreEqual("(a|b)", truthTable.Description.Last());

    Assert.AreEqual("0", truthTable.Rows[0].Values[0]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[1]);
    Assert.AreEqual("0", truthTable.Rows[0].Result);

    Assert.AreEqual("0", truthTable.Rows[1].Values[0]);
    Assert.AreEqual("1", truthTable.Rows[1].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[1].Result);

    Assert.AreEqual("1", truthTable.Rows[2].Values[0]);
    Assert.AreEqual("0", truthTable.Rows[2].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[2].Result);

    Assert.AreEqual("1", truthTable.Rows[3].Values[0]);
    Assert.AreEqual("1", truthTable.Rows[3].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[3].Result);
}
```


In the following image I have tested if the **truth table** of a big proposition is generated correctly.

```
[TestMethod]
0 references
public void TruthTableBigProposition()
{
    string input = "=(~(b), a), |=(c,d), >(e,f))";

    Service check = new Service();
    check.Proposition(input);
    var truthTable = check.GetTable();

    Assert.AreEqual(64, truthTable.Rows.Count);
    Assert.AreEqual("((~(b)&a)=((c=d)|(e>f)))", truthTable.Description.Last());

    Assert.AreEqual("0", truthTable.Rows[0].Values[0]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[1]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[2]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[3]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[4]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[5]);
    Assert.AreEqual("0", truthTable.Rows[0].Result);

    Assert.AreEqual("1", truthTable.Rows[62].Values[0]);
    Assert.AreEqual("1", truthTable.Rows[62].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[62].Values[2]);
    Assert.AreEqual("1", truthTable.Rows[62].Values[3]);
    Assert.AreEqual("1", truthTable.Rows[62].Values[4]);
    Assert.AreEqual("0", truthTable.Rows[62].Values[5]);
    Assert.AreEqual("0", truthTable.Rows[62].Result);

    Assert.AreEqual("1", truthTable.Rows[63].Values[0]);
    Assert.AreEqual("1", truthTable.Rows[63].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[63].Values[2]);
    Assert.AreEqual("1", truthTable.Rows[63].Values[3]);
    Assert.AreEqual("1", truthTable.Rows[63].Values[4]);
    Assert.AreEqual("1", truthTable.Rows[63].Values[5]);
    Assert.AreEqual("0", truthTable.Rows[63].Result);
}
```

In the following image I have tested if the **truth table** for a tautology is generated correctly.

```
[TestMethod]
0 references
public void TruthTableTatology()
{
    string input = ">(&(a,b),&(a,b))";

    Service check = new Service();
    check.Proposition(input);
    var truthTable = check.GetTable();

    Assert.AreEqual(4, truthTable.Rows.Count);
    Assert.AreEqual("((a&b)>(a&b))", truthTable.Description.Last());

    Assert.AreEqual("0", truthTable.Rows[0].Values[0]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[0].Result);

    Assert.AreEqual("0", truthTable.Rows[1].Values[0]);
    Assert.AreEqual("1", truthTable.Rows[1].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[1].Result);

    Assert.AreEqual("1", truthTable.Rows[2].Values[0]);
    Assert.AreEqual("0", truthTable.Rows[2].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[2].Result);

    Assert.AreEqual("1", truthTable.Rows[3].Values[0]);
    Assert.AreEqual("1", truthTable.Rows[3].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[3].Result);
}
```

In the following image I have tested if the **truth table** of a normal length proposition is correctly generated.

```
[TestMethod]
0 references
public void TruthTableNormalProposition()
{
    string input = "(>(a,b),c)";

    Service check = new Service();
    check.Proposition(input);
    var truthTable = check.GetTable();

    Assert.AreEqual(8, truthTable.Rows.Count);
    Assert.AreEqual("((a>b)=c)", truthTable.Description.Last());

    Assert.AreEqual("0", truthTable.Rows[0].Values[0]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[1]);
    Assert.AreEqual("0", truthTable.Rows[0].Values[2]);
    Assert.AreEqual("0", truthTable.Rows[0].Result);

    Assert.AreEqual("0", truthTable.Rows[1].Values[0]);
    Assert.AreEqual("0", truthTable.Rows[1].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[1].Values[2]);
    Assert.AreEqual("1", truthTable.Rows[1].Result);

    Assert.AreEqual("0", truthTable.Rows[2].Values[0]);
    Assert.AreEqual("1", truthTable.Rows[2].Values[1]);
    Assert.AreEqual("0", truthTable.Rows[2].Values[2]);
    Assert.AreEqual("0", truthTable.Rows[2].Result);

    Assert.AreEqual("0", truthTable.Rows[3].Values[0]);
    Assert.AreEqual("1", truthTable.Rows[3].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[3].Values[2]);
    Assert.AreEqual("1", truthTable.Rows[3].Result);

    Assert.AreEqual("1", truthTable.Rows[7].Values[0]);
    Assert.AreEqual("1", truthTable.Rows[7].Values[1]);
    Assert.AreEqual("1", truthTable.Rows[7].Values[2]);
    Assert.AreEqual("1", truthTable.Rows[7].Result);
}
```