

Оглавление

[Введение 3](#_Toc531539508)

[Интерфейс программы 4](#_Toc531539509)

[Ввод и вывод данных 5](#_Toc531539510)

[Логическая клавиатура и другие варианты введения формул 5](#_Toc531539511)

[Сохранение результатов (в файл) 7](#_Toc531539512)

[Возможности программы 8](#_Toc531539513)

[Работа с введенными формулами 8](#_Toc531539514)

[Предварительная проверка правильности формулы 8](#_Toc531539515)

[Построение таблицы истинности 8](#_Toc531539516)

[Проверка выполнимости 10](#_Toc531539517)

[Построение нормальных форм (СКНФ и СДНФ) 10](#_Toc531539518)

[Получение информации из базы данных логических формул 13](#_Toc531539519)

[Генерация логических формул 20](#_Toc531539520)

[Дополнительные возможности 23](#_Toc531539521)

[Настройки символов функций 23](#_Toc531539522)

[Получение информации о программе 25](#_Toc531539523)

[Приведение результата к международному формату представления (SAT) 10](#_Toc531539524)

[Дополнительные файлы, необходимые для работы программы 26](#_Toc531539525)

[База данных логических формул 26](#_Toc531539526)

[Вывод 27](#_Toc531539527)

# [Введение](#_top)

Основной задачей данной курсовой работы является реализация программы «Редактор, анализатор и генератор булевых формул». Передо мной поставлено несколько задач:

* создание логической клавиатуры
* реализация приведения результата программы к международному внутреннему формату представления
* осуществление синтаксического анализа формул
* создание базы данных логических формул
* получение доступа к характеристикам логических формул из базы данных, индексация и поиск нужных формул в БД
* генерация логических формул с нужными характеристиками

Программа создана на языке C# с помощью интерфейса программирования приложений «Windows Forms»

# [Интерфейс программы](#_top)

Интерфейс программы является классическим, с несколькими формами, дружелюбным к пользователю: программа информирует обо всех ошибках, возникающих в процессе, пользователю предложены разнообразные инструкции для работы с программой.

Кроме того, пользователь может настроить программу по своему пожеланию: изменить шрифт, символы функций.

# [Ввод и вывод данных](#_top)

## [Логическая клавиатура и другие варианты введения формул](#_top)

В программе предусмотрено несколько вариантов ввода нужной формулы. В любом из предложенных способов после нажатии кнопки «Обработка булевой формулы» из текста удалятся все пробелы для удобства последующей обработки.

1. Введение с использованием кнопок («логической клавиатуры»)

Используя кнопки справа от поля ввода, вы можете ввести формулу, состоящую из основных (элементарных) булевых функций, а также из переменных, нуля или единицы. Вы можете изменить символы для обозначения логических сущностей (см. подробнее в пункте [Настройки](#_Настройки_значков_функций))

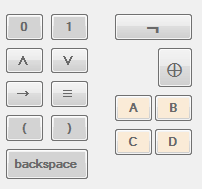


Рисунок 2 Логическая клавиатура

Добавление символа в richTextBox1 реализовано с помощью метода «Apply\_Character». Он принимает в качестве аргумента символ операции (который получает из свойства button.Text при нажатии на кнопку), и может обрабатывать два случая: курсор установлен в неком месте в richTextBox1 (тогда новый символ будет вставлен слева от курсора), либо курсора там нет (новый символ будет вставлен в конце).

public void Apply\_Character(string character)

{

if (flag == true)

{

richTextBox1.Text = richTextBox1.Text.Insert(richTextBox1.SelectionStart, character);

flag = false;

}

else

richTextBox1.Text += character;

}

Значение флажка становится истинным, если курсор установлен в richTextBox1, и вновь становится ложным после нажатия кнопки и вызова метода «Apply\_Character».

private void richTextBox1\_Enter(object sender, EventArgs e)

{

flag = true;

}

1. Введение с клавиатуры

Введение с клавиатуры имеет некоторые особенности. При нажатии кнопки «Правила введения формул» пользователь может ознакомиться с ними. Пользователя предупреждают о том, что «неизвестные символы» (не занесенный в базу в качестве символов операций) могут не обработаться программой. Кроме того, в качестве переменных воспринимаются только символы латинского алфавита (за исключением букв “V” и “v”). Учитывая эти ограничения, лучше воздержаться от этого способа ввода формулы.

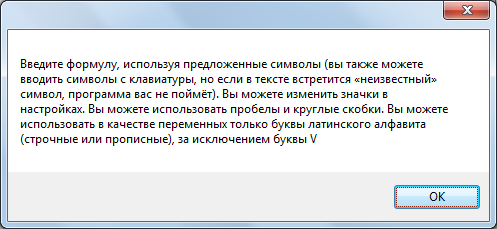


Рисунок 3 Правила введения формул

1. Получение формулы из сайта

При нажатии кнопки «Загрузить из файла» откроется системный диалог открытия файла. Данная функция программы ориентирована на использование файлов формата .txt

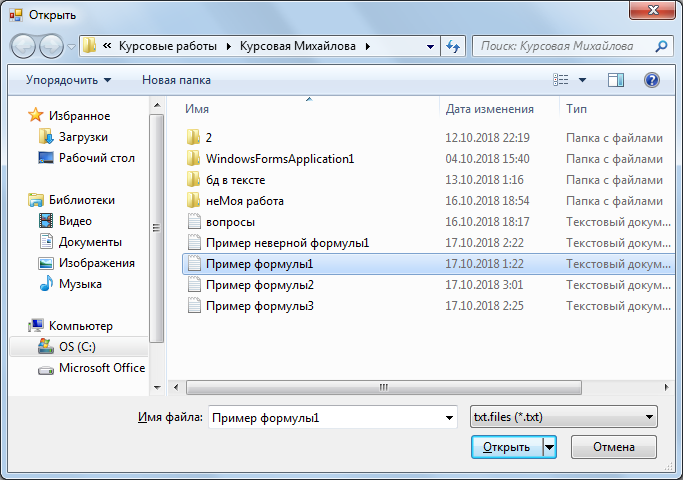


Рисунок 4 Системный диалог выбора файла

## [Сохранение результатов (в файл)](#_top)

В программе есть возможность сохранить результаты работы программы в файл формата .txt с помощью стандартного saveFileDialog.

# [Возможности программы](#_top)

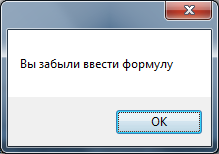
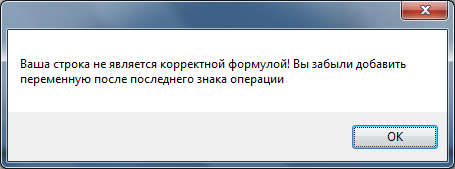
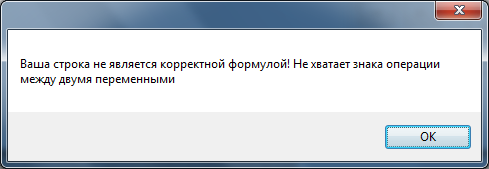
## [Работа с введенными формулами](#_top)

### Предварительная проверка правильности формулы

Перед работой с булевой формулой важно проверить корректность ее введения. Это реализуется с помощью специального метода Correct(). Этот метод способен выявить и сообщить пользователю о следующих проблемах:

* несоответствие количество скобок (например, количество открывающих больше количества закрывающих)
* отсутствие знака операции между двумя переменными
* отсутствие переменной после знака операции
* формула не введена

После любой из ошибок выводится MessageBox, информирующий об ошибке.

### Построение таблицы истинности

Построение таблицы истинности выполняется с помощью метода TruthTableBuilding().

Алгоритм составления таблицы истинности данной функции состоит из следующих этапов:

1. Вычисление количества переменных функции с помощью метода NumOfVar() (не встроенный метод)
2. Создание двумерной таблицы с количеством столбцов, равным количеству переменных num\_of\_var, и количеством строк, равным 2num\_of\_var

|  |  |
| --- | --- |
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

1. Заполнение двумерного массива нулями и единицами по порядку. Пример заполнения массива для двух переменных:
2. Из каждой строки данной таблицы формируется строка значений (например, «0001», тогда x1, x2, x3 будут нулями, x4 – единицей). Эта строка передается в качестве параметра методу Computation()

Computation() представляет собой метод, который:

1. Заменяет все переменные нулями или единицами. Переменные берутся по порядку (например, формула AvBvC при входном параметре 011 изменится на 0v1v1)
2. Высчитывается количество скобок с помощью метода NumberOfBrackets();
3. Далее происходит «вычленение» самой внутренней скобки. Для этого в цикле проходим от j = 0 до последней открывающей скобки (для этого использую метод IndexOfInternalBracket()). Далее нахожу индекс следующей за ней «)». Далее с помощью этих данных обрезаю содержимое самой внутренней скобки.
4. Содержимое самой внутренней скобки передаю в метод ComputationInBracket(). После получения результата заменяю всю внутреннюю скобку вместе с «(» и «)» и перехожу к следующей итерации.

Демонстрация «разделения» формулы на отдельные скобки и передачи каждой скобки методу ComputationInBracket()

for (int i = 0; i < (int)((double)NumberOfBrackets(param) / 2); )

{

string buffbegin = buff; //

int l = 0;

for (int j = IndexOfInternalBracket(buff) + 1; j < buff.Length; j++) //иду от момента, где начинается самая внутренняя "("

//до момента, где находится соответствующая ей ")" (т.е. следующая ")" за данной "(")

{

if (buff.IndexOf(")", j) == j)

break;

l++;

}

buff = buff.Substring(IndexOfInternalBracket(buff) + 1, l); //обрезаю строку buff так, чтобы осталось только содержимое скобки

//ДЕЙСТВИЯ В СКОБКАХ

int index = 0;

while ((index = buffbegin.IndexOf("(" + buff + ")", index) + 1) != 0)

i++;

buff = buffbegin.Replace("(" + buff + ")", ComputationInBracket(buff)); /\*заменяю в текущей строке (сохранена в начале итерации в исполнения строки ("(" и ")" удаляю тоже \*/

message = message + buff + Environment.NewLine;

}

if (NumberOfBrackets(buff) == 0)

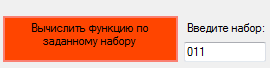
{

message = message + ComputationInBracket(buff) + Environment.NewLine;

return CharToBool(Convert.ToChar(ComputationInBracket(buff)));

}

ComputationInBracket() представляет собой метод в котором по очереди (в порядке их приоритета: сначала отрицание, затем дизъюнкция и т.д.) проводятся логические операции. Строка сразу же перезаписывается с результатом логической операции (например, “0v1&0” изменится сначала на “0v0”, а затем на “0”).



Кроме построения таблицы истинности, можно вычислить значение функции только на заданном наборе переменных. Тогда заданный набор (если он совпадает с количеством переменных) передается сразу в метод Computation().

### Проверка выполнимости

Проверка выполнимости совершается элементарно – с помощью метода (встроенного для string) .IndexOf проверяется, встречаются ли в таблице истинности данной функции “1” и “0” и делается вывод

### Построение нормальных форм (СКНФ и СДНФ)

В программе есть возможность построения совершенных нормальных форм функций. Рассмотрим эту возможность на примере СДНФ:

В метод public string ConstructionOfDisjunctiveNormalForm(string function) подается формула. Сначала с помощью пользовательских методов ListOfVariable() и TruthTableBuilding() создается список переменных и строка-таблица истинности. Далее мне необходимо рассмотреть каждую строку таблицы переменных, которой соответствует «1» в таблице истинности. Иными словами, в нижеследующей таблице мне нужно рассмотреть только выделенные коричневым цветом строки (голубым указана таблица истинности)

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Для того, чтобы это сделать, я могла бы воспользоваться приёмом, который я использовала в методе TruthTableBuilding() и сгенерировать двумерный массив, который бы служил списком наборов переменных. Однако я захотела изучить более простой (как мне показалось) способ – работа с каждым набором переменных как с двоичным числом. Действительно, раз каждый набор “00..0”, “00..1”, … “11..1” является двоичным представлением чисел от 0 до N (N – число переменных в формуле), гораздо легче просто идти в цикле по каждому значению от 0 до N, тут же преобразовывая это значение в двоичный вид, вместо того, чтобы создавать отдельный двумерный массив.

Так я и сделала. Для каждого встреченной единицы в таблице истинности я:

1. Узнаю, какая по счету эта единица в таблице истинности (узнаю ее индекс i)
2. Перевожу i в двоичный вид. В случае, если в двоичном виде будет меньше символов, чем количества переменных (например, 2 переходит в “10” при том, что каждый набор должен содержать значения для трех переменных: “000”, “010” и т.д.), дополняю его нулями впереди
3. Далее если в наборе j-ый символ – единица, к результирующей строке добавляется j-ый символ из набора переменных. Если j-ый символ – ноль, j-ый символ из набора переменных добавляется с отрицанием
4. После каждой итерации добавляется “∨” (с отдельной обработкой граничных дизъюнкций).

for (int i = 0; i < truth\_table.Length; i++)

{

if (truth\_table[i] == '1')

{

if (is\_begin)

is\_begin = false;

else

result += "∨";

string binary\_buff = Convert.ToString(i, 2);

while (binary\_buff.Length != list\_of\_var.Length)

{

binary\_buff = "0" + binary\_buff;

}

char[] binary\_buff\_char = binary\_buff.ToCharArray(); // бинарное представление конкретного i

for (int j = 0; j < binary\_buff\_char.Length - 1; j++)

{

if (binary\_buff\_char[j] == '1')

result += list\_of\_var[j] + "·";

if (binary\_buff\_char[j] == '0')

result += "¬" + list\_of\_var[j] + "·";

}

for (int j = binary\_buff\_char.Length - 1; j < binary\_buff\_char.Length; j++)

{

if (binary\_buff\_char[j] == '1')

result += list\_of\_var[j];

if (binary\_buff\_char[j] == '0')

result += "¬" + list\_of\_var[j];

}

}

}

Отдельно обработаны случаи невыполнимых и тождественно истинных функций. Вместо СКНФ и СДНФ возвращаются «0» или «1».

### Приведение результата к международному формату представления (SAT)

Это происходит очень простым способом: все символы операций меняются на стандартные, а переменные заменяются на x1, x10, x11 и т.д. (в соответствии с требованиями SAT)

for (int i = 0; i < list\_of\_var.Length; i++)

{

text = text.Replace(new string(new char[] {list\_of\_var[i]}), "x" + Convert.ToString(i,2));

}

## [Получение информации из базы данных логических формул](#_top)

Для реализаций следующих функций необходимо зайти в форму «База данных булевых функций».

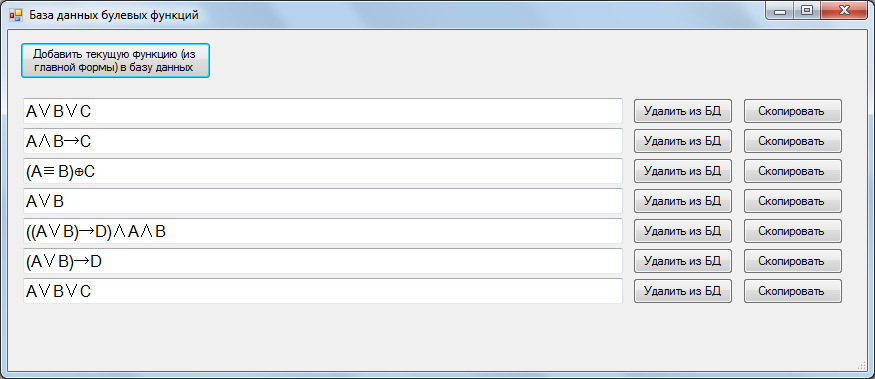


Рисунок 5 Форма "БД булевых функций"

В данной форме реализованы следующие функции:

1. получение и загрузка на форму функций из базы данных
2. добавление новой записи в базу данных
3. удаление записи из базы данных
4. передача записи БД в главную форму

Рассмотрим все функции по порядку:

**1. Получение и загрузка на форму функций из базы данных**

База данных хранится в бинарном файле. Подробнее можно прочитать в разделе «».

Для того, чтобы корректно считать некоторое заранее не известное количество формул, удобно будет для записи формул использовать связный список. Опишу структуру связного списка в классах LinkedList и Node, также добавив туда методы Add и Count, которые нам в дальнейшем понадобятся:

public class Node<T>

{

public Node(T data)

{

Data = data;

}

public T Data { get; set; }

public Node<T> Next { get; set; }

}

public class LinkedList<T> : IEnumerable<T> // односвязный список

{

Node<T> head; // головной/первый элемент

Node<T> tail; // последний/хвостовой элемент

int count; // количество элементов в списке

// добавление элемента

public void Add(T data)

{

Node<T> node = new Node<T>(data);

if (head == null)

head = node;

else

tail.Next = node;

tail = node;

count++;

}

// удаление элемента

public bool Remove(T data)

{

Node<T> current = head;

Node<T> previous = null;

while (current != null)

{

if (current.Data.Equals(data))

{

// Если узел в середине или в конце

if (previous != null)

{

// убираем узел current, теперь previous ссылается не на current, а на current.Next

previous.Next = current.Next;

// Если current.Next не установлен, значит узел последний,

// изменяем переменную tail

if (current.Next == null)

tail = previous;

}

else

{

// если удаляется первый элемент

// переустанавливаем значение head

head = head.Next;

// если после удаления список пуст, сбрасываем tail

if (head == null)

tail = null;

}

count--;

return true;

}

previous = current;

current = current.Next;

}

return false;

}

public int Count { get { return count; } }

public bool IsEmpty { get { return count == 0; } }

// очистка списка

public void Clear()

{

head = null;

tail = null;

count = 0;

}

////// реализация интерфейса IEnumerable

IEnumerator IEnumerable.GetEnumerator()

{

return ((IEnumerable)this).GetEnumerator();

}

IEnumerator<T> IEnumerable<T>.GetEnumerator()

{

Node<T> current = head;

while (current != null)

{

yield return current.Data;

current = current.Next;

}

}

}

Связный список, класс которого мы создали, поддерживает интерфейс IEnumerable, и, так как мы задали собственную логику перебора в методе GetEnumerator, теперь мы можем использовать foreach для элементов этой коллекции.

Мы создали необходимую структуру, теперь мы можем записать в нее функции из БД. Подключаем using System.IO, и далее с использованием объекта BinaryReader считываем каждую функцию в связный список. Теперь осталось реализовать вывод этих формул на форму. Он будет производиться в таком формате: textbox с необходимой функцией, справа от него две кнопки с надписями «Удалить» и «Скопировать».

a[i].func = new TextBox()

{

Name = "TextBox" + i.ToString(),

Location = new Point(15, 68 + 30 \* i),

Text = s,

Size = new Size(600, 26),

Font = new Font("Microsoft Sans Serif", 12F, System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(204)))

};

this.Controls.Add(a[i].func);

Для этого необходимо реализовать динамическое создание кнопок с заданными параметрами. Для этого создаю еще один класс Threesome:

public struct Threesome

{

public TextBox func;

public Button copy;

public Button delete;

}

После динамического создания кнопок, я добавляю к ним метод Click:

a[i].delete.Click += Delete;

Каждая «тройка» является элементом массива a. Полный код:

private void Информация\_о\_булевых\_функциях\_Load(object sender, EventArgs e1)

{

LinkedList<string>functions = new LinkedList<string>();

try

{

// создаем объект BinaryReader

using (BinaryReader reader = new BinaryReader(File.Open(path + "\\database.bin", FileMode.Open)))

{

// пока не достигнут конец файла

// считываем каждое значение из файла

while (reader.PeekChar() > -1)

{

functions.Add(reader.ReadString());

}

Threesome[] a = new Threesome[functions.Count];

int i = 0;

foreach (string s in functions)

{

a[i].func = new TextBox()

{

Name = "TextBox" + i.ToString(),

Location = new Point(15, 68 + 30 \* i),

Text = s,

Size = new Size(600, 26),

Font = new Font("Microsoft Sans Serif", 12F, System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((byte)(204)))

};

this.Controls.Add(a[i].func);

a[i].delete = new Button()

{

Name = "DeleteTextBox" + i.ToString(),

Location = new Point(15 + 600 + 10, 68 + 30 \* i),

Size = new Size(100, 26),

Text = "Удалить из БД",

};

this.Controls.Add(a[i].delete);

a[i].delete.Click += Delete;

a[i].copy = new Button()

{

Name = "CopyTextBox" + i.ToString(),

Location = new Point(15 + 600 + 10 + 100 + 10, 68 + 30 \* i),

Size = new Size(100, 26),

Text = "Скопировать"

};

this.Controls.Add(a[i].copy);

a[i].copy.Click += Copy;

i++;

}

}

}

catch (Exception e)

{

MessageBox.Show(e.Message);

}

}

**Передача записи БД в главную форму**

Теперь нужно определить методы Copy и Delete. Начнем с Copy. Приведем параметр sender (он сообщает информацию об объекте, который вызвал событие) явно к типу Button. Это нам необходимо, чтобы получить номер кнопки, с которой связана нужная нам формула. Считываем из БД все записи, но нужна нам будет только запись, соответствующая номеру. Как только она появляется, нам необходимо вызвать функцию, которая изменит текст richTextBox1 в главной форме. Чтобы передать данные в другую форму, воспользуемся известным нам приемом (аналогичный использовался в [Настройки](#_Настройки_символов_функций)) создания класса делегата в общем namespace.

public static class Data2

{

public delegate void MyEvent(string data);

public static MyEvent EventHandler;

}

Соответствующий делегату метод записан в конструкторе Form1:

Data2.EventHandler = new Data2.MyEvent(change\_main\_textbox);

Метод change\_main\_textbox записан в классе Form1:

void change\_main\_textbox(string param)

{

richTextBox1.Text = param;

}

Полный код Copy:

private void Copy(object sender, EventArgs e1)

{

var button = (Button)sender;

try

{

using (BinaryReader reader = new BinaryReader(File.Open(path + "\\database.bin", FileMode.Open)))

{

for (int i = 0; reader.PeekChar() > -1; i++)

{

if (i == Convert.ToInt32(button.Name.Replace("CopyTextBox", string.Empty)))

{

Data2.EventHandler(reader.ReadString());

break;

}

reader.ReadString();

}

}

}

catch (Exception e)

{

MessageBox.Show(e.Message);

}

}

1. **Удаление записи из БД**

Аналогично определяем Delete. Идея метода заключается в том, чтобы сначала считать все данные из БД, а потом перезаписать в БД все функции, кроме той, которая соответствует нажатой кнопке «Удалить». Кроме этого, было необходимо реализовать «переоткрытие» формы (для ее обновления), для этого добавлена строка this.DialogResult = DialogResult.Retry . В это время в главной форме код открытия формы «База данных» имеет следующий вид:

private void информацияОВсехБулевыхФункцияхToolStripMenuItem\_Click(object sender, EventArgs e)

{

Информация\_о\_булевых\_функциях dialog = new Информация\_о\_булевых\_функциях(richTextBox1.Text);

while (dialog.ShowDialog() == DialogResult.Retry)

{

dialog.Close();

dialog = new Информация\_о\_булевых\_функциях(richTextBox1.Text);

}

}

Таким образом, в случае, если пользователь нажал кнопку «крестик» (закрыл форму сам), форма закроется обычным образом. Если же пользователь нажал на одну из кнопок «Удалить», программа поймет это по свойству DialogResult.Retry и закроет форму, сразу же создав ее новый экземпляр.

Полный код метода Delete:

private void Delete(object sender, EventArgs e1)

{

var button = (Button)sender;

LinkedList<string>functions = new LinkedList<string>();

try

{

// создаем объект BinaryReader

using (BinaryReader reader = new BinaryReader(File.Open(path + "\\database.bin", FileMode.Open)))

{

// пока не достигнут конец файла

// считываем каждое значение из файла

while (reader.PeekChar() > -1)

{

functions.Add(reader.ReadString());

}

}

File.Delete(path + "\\database.bin");

using (BinaryWriter writer = new BinaryWriter(File.Open(path + "\\database.bin", FileMode.CreateNew)))

{

int i = 0;

foreach (string s in functions)

{

if (i != Convert.ToInt32(button.Name.Replace("DeleteTextBox", string.Empty)))

writer.Write(s);

i++;

}

}

}

catch (Exception e)

{

MessageBox.Show(e.Message);

}

this.DialogResult = DialogResult.Retry;

}

**4. Добавление новой записи в БД**

Эта функция реализует добавление формулы из главной формы в БД. Она имеет только одну особенность: формула перед добавлением в БД проходит проверку в методе Correct (подробнее вы можете узнать в разделе . Для этого делегату Data1.EventHandler присваивается ссылка на метод Correct.

После добавления новой записи происходит повторная загрузка формы.

private void button1\_Click(object sender, EventArgs e)

{

using (BinaryWriter writer = new BinaryWriter(File.Open(path + "\\database.bin", FileMode.Append)))

{

if (Data1.EventHandler(func\_from\_main\_form))

{

writer.Write(func\_from\_main\_form);

}

else

MessageBox.Show("Невозможно добавить запись в БД, так как в главной форме записана не формула");

}

Информация\_о\_булевых\_функциях\_Load(null, null);

}

## [Генерация логических формул](#_top)

Реализованы несколько типов генерации формул: с заданным кол-вом переменных и с заданной таблицей истинности. Пользователь может вводить любой из этих параметров (или оба), или не вводить ни одного – тогда программа сгенерирует функцию случайным образом.

Перед любой генерацией происходит проверка условий в методе CorrectnessOfConditions() (например, пользователь может задать таблицу истинности «0001» при количестве переменных «5» - программа совершит ему об ошибке).

Методы генераций условно разделю на 3 вида и рассмотрю каждый из них.

**Генерация без таблицы истинности**

Самый простой вид генерации: программа случайным образом добавляет сначала «¬» (знак отрицания) или ничего, затем переменную из массива переменных или «0» или «1», затем знак операции, затем повторяет операцию до тех пор пока не запишет все переменные (их количество либо определяется пользователем, либо генерируется случайным образом).

string generated\_function = string.Empty;

for (int i = 0, k = 0; k < num\_of\_var; i++ )

{

if (i % 2 == 0)

{

int randomize1 = rnd.Next(0, 2);

if (randomize1 == 0)

{

generated\_function += "¬";

}

int randomize = rnd.Next(0, 6);

if (randomize < 5)

{

generated\_function += letters\_short[rnd.Next(letters\_short.Length)]; k++;

}

if (randomize == 5)

generated\_function += null\_and\_one[rnd.Next(null\_and\_one.Length)];

}

else

{

generated\_function += operations\_short[rnd.Next(operations\_short.Length)];

}

}

**Генерация с таблицей истинности: построение нормальных форм**

В этом варианте строится СКНФ или СДНФ по такому же принципу, как в форме «Обработка булевой функции»

**Генерация с таблицей истинности: случайная формула.**

Общий принцип этого способа: «сначала генерируется формула, затем находится ее таблица истинности, если она не совпадает с заданной таблицей, создается другая».

Естественно, данный способ является очень долгим (что самое плохое – непредсказуемо долгим). Поэтому возникла необходимость, во-первых, демонстрировать пользователю, что приложение не зависло и работа происходит, а во-вторых, дать возможность пользователю в любой момент отменить генерацию.

Для этого было принято решение убрать выполнение метода Generation() в фоновый поток с помощью контрола backgroundWorker.

При нажатии кнопки «Генерация» кнопка «Прекратить генерацию» становится доступна для нажатия и запускается фоновая операция.

button7.Enabled = true;

backgroundWorker1.RunWorkerAsync();

В качестве фоновой операции выступает Generation()

private void backgroundWorker1\_DoWork(object sender, DoWorkEventArgs e)

{

e.Result = Generation();

}

По завершении работы метода кнопка «Прекратить генерацию» вновь «отключается».

private void backgroundWorker1\_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)

{

button7.Enabled = false;

}

Теперь при генерации форма не «зависает», пользователь в любой момент может ее закрыть или прекратить генерацию и ввести новые параметры.

Однако для пользователя важно знать, как продвигается генерация. Для осуществления этого необходимо каждую новую сгенерированную функцию добавлять в окошко «Перебор вариантов». Из фонового потока этого сделать нельзя, так как все контролы формы принадлежат основному потоку. Для доступа к textBox’ам основной формулы воспользуемся методом .Invoke().

Для начала объявим делегат.

delegate void Del(string text, TextBox textBox);

Далее в фоновом потоке создадим универсальные анонимные методы (универсальные они потому, что подходят для изменения текста любого textBox’а).

Del change\_textbox\_delegate = new Del((s, textBox) => textBox.Text = s);

Del add\_textbox\_delegate = new Del((s, textBox) => textBox.AppendText(s));

С помощью данных анонимных методов (точнее, их делегатов) можно использовать Invoke(). Рассмотрим их работу на примере добавления функции в окошко «Перебор вариантов». К textBox4 применяется Invoke, получающий в качестве первого параметра делегат со ссылкой на функцию добавления текста в текстбокс, и в качестве второго параметра входные данные для делегата.

if (textBox4.InvokeRequired)

{

// textBox4.Text += generated\_function + Environment.NewLine;

// добавление функции в окно "Перебор вариантов"

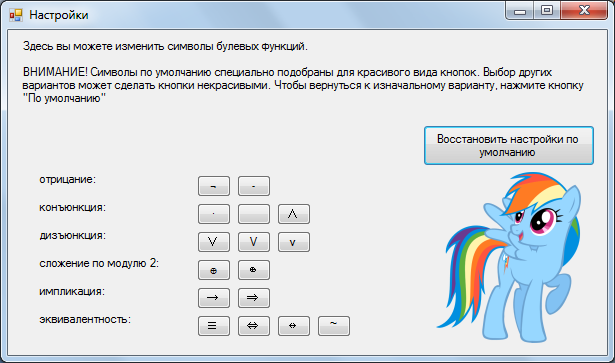
textBox4.Invoke(add\_textbox\_delegate, generated\_function + Environment.NewLine, textBox4);

}

Также мы провели проверку InvokeRequired для обеспечения потокобезопасности.

## [Дополнительные возможности](#_top)

### [Настройки символов функций](#_top)



Пользование этим пунктом меню интуитивно понятно.

Гораздо более интересна реализация этого пункта в программе. Дело в том, что для смены значков кнопок (а значит, изменения свойства Text у кнопки) необходимо реализовать передачу данных между формами. Необходимо передать свойство.Text у кнопки с формы «Настройки» в свойство .Text у соответственной кнопки с главной формы.

В классе Form1 определяю метод для смены символов кнопок (num – номер кнопки, которая будет меняться в форме Form1, param – параметр-символ, на который изменится символ кнопки)

void change\_character(int num, string param)

{

switch (num)

{

case 5:

{

button5.Text = param;

break;

}

case 6:

{

button6.Text = param;

break;

}

case 8:

{

button8.Text = param;

break;

}

case 7:

{

button7.Text = param;

break;

}

case 9:

{

button9.Text = param;

break;

}

case 12:

{

button12.Text = param;

break;

}

}

}

Теперь необходимо сделать так, чтобы данным методом можно было пользоваться из другой формы. Для этого в основном пространстве имён (в Program.cs) создаю отдельный класс для делегата. Этот делегат будет доступен в любой форме программы. Задаю делегату необходимую конфигурацию: он не должен ничего возвращать, и должен принимать параметры типа int и string.

public static class Data

{

public delegate void MyEvent(int number, string data);

public static MyEvent EventHandler;

}

Теперь я могу вернуться в класс Form1 и в конструкторе формы прописать

public Form1()

{

InitializeComponent();

Data.EventHandler = new Data.MyEvent(change\_character);

//остальной код

}

Теперь Data.EventHandler – делегат, который является ссылкой на метод change\_character, и который я могу использовать в другой форме (форме «Настройки»). Рассмотрим смену символа импликации с «→» на «⇒» с помощью делегата. При нажатии на кнопку №14 (кнопку, содержащую символ «⇒»), происходит событие:

private void button14\_Click(object sender, EventArgs e)

{

Data.EventHandler(9, button14.Text);

}

Таким образом, я поменяла символ кнопки №9 (кнопки импликации) в главной форме на символ кнопки №14 в форме «Настройки»

При нажатии на кнопку «Восстановить настройки» все символы заменяются на стандартные

private void button1\_Click(object sender, EventArgs e)

{

Data.EventHandler(8, "∧"); Data.EventHandler(7, "∨");

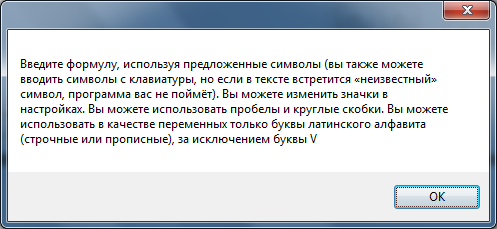
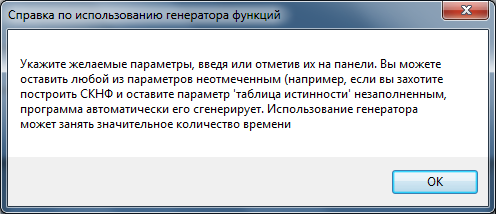
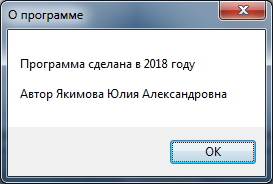
Data.EventHandler(9, "→"); Data.EventHandler(12, "≡");

Data.EventHandler(6, "ⴲ"); Data.EventHandler(5, "¬");

}

### [Получение информации о программе](#_top)

Почти в каждой форме есть возможность получения справочных данных, например, правил введения формул, информации об авторе программы и др.

# [Дополнительные файлы, необходимые для работы программы](#_top)

## [База данных логических формул](#_top)

База данных представляет собой обычный бинарных файл, в который записываются формулы с помощью методов, описанных в пункте .

# [Вывод](#_top)

Основная цель курсовой работы выполнена – я создала работоспособную и полезную для студентов, изучающих дискретную математику или матлогику, программу-редактор булевых функций.

Работая над данной программой, я узнала и освоила новые приёмы программирования на C#:

1. динамическое создание элементов управления (кнопок и текстбоксов) с заданными параметрами и методами
2. разные способы передачи данных между формами
3. реализация линейных списков на C#
4. использование бинарных файлов в C#, создание базы данных в виде бинарного файла
5. многопоточное программирование (с помощью специального класса BackgroundWorker), включая методы приостановки потоков, доступа к контролам основного потока из фонового потока и т.д.
6. создание XML-комментариев
7. работа с двоичными числами, перевод чисел между разными системами счисления
8. анонимные методы, лямбда-выражения, делегаты