ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

Кафедра вычислительных систем

КУРСОВАЯ РАБОТА

по дисциплине «Технологии разработки программного обеспечения»

на тему « Интерпретатор языка программирования Машины Тьюринга »

Выполнил:
ст. гр. ИП-814
Иванов К.А.

Проверил:
доц., к.ф.-м.н. Пудов С. Г.

Новосибирск, 2019

Содержание:

Введение и постановка задачи

Задание курсового проекта “интерпретатор языка программирования Машины Тьюринга”.
Цель была создать похожую программу, которая представлена на сайте, без доступа к интернету, и тем самым увеличить максимальную скорость выполнения зависящую от конкретного компьютера.

Техническое задание

Wiki на GitHub`e

Интерпретатор языка программирования Машины Тьюринга.

Программа предназначена в учебных целях и преследует следующие задачи:

---

1. Повышение уровня написания программ студентов, путём изучения новых путей решения поставленных задач.
2. Обучение начальным навыкам командной работы.
3. Решение поставленного задания. *Интерпретирование кода Тьюринга, написанного в программе.

Описание проекта:

Интерпретатор языка машины Тьюринга поможет ознакомится студентам с абстрактным исполнителем и тем, что любой алгоритм возможно реализовать используя простейшие алгоритмы без использования памяти. Сам продукт создается, как альтернативное решение машины Тьюринга без привязки к интернету и будет обладать схожим интерфейсом, с представленным ниже по ссылке, для удобства. http://morphett.info/turing/ - Turing machine simulator. Симулятор, который предлагают использовать студентам.

1. Оконное приложение.
2. Ввод кода в поле предназначенное для этого.
3. Сохранение кода в текстовый файл.
4. Загрузка кода из текстового файла.
5. Выполнение написанного кода.

6. Визуальное отображение происходящего. Бесконечная в обе стороны лента.
7. Смена скорости выполнения два режима.
8. Вывод ошибок.
9. Примеры программ в этом репозитории.

Функционал:

Программа будет с интерактивным интерфейсом. В поле ввода можно будет писать код алгоритма, с последующем выполнением и визуальным отображением на ленте бесконечной влево и вправо.

Синтаксис:

1. <текущая позиция> <текущий символ> <новый символ> <направление> <новая позиция> - пример, a 0 1 r b . Текущая и новая позиции могут содержать любые символы, но начало программы обязательно с 0.
2. Текущий и новый символы должны быть единичны или '_'(пустое место) - пример, a 0 1 r b . a 0 12 r b - является ошибкой
3. Направление содержит один символ l - двигаться влево, r - двигаться вправо или * - не менять направление. Всё движение смещает каретку на один символ в ленте.
4. Символ ";" является для обозначения комментария, всё написанное после не будет учитываться при выполнении программы.
5. После встречи кодового слова "halt" программа будет останавливать выполнение. Например, "halt" или "halt-<любые символы>".

Семантика языка:

010101 - лента до выполнения.

0 0 1 r 1 - на текущей позиции, если текущий символ '0' совпадает с символом в ленте '0', то меняем на новый символ '1', переводим каретки по направлению 'r' (вправо) на один символ и переходим к позиции '1'.

110101 - лента после выполнения.

Если позиции '1' не найдено или при позиции '1' не совпадает ни один из текущих символов с символом в ленте, то произойдёт ошибка и выполнение прекратиться.

Пример ошибки:

0 0 1 r 1

1 0 1 r 0

В ленте "010101" на второй позиции находится '1', в условиях выполнения такое не предусмотрено, программа завершиться с ошибкой.

Обработка:

При запуске программы будет происходить построчное считывание кода. На этом этапе будет происходить проверка на соответствие синтаксису, а так же обработка кода. Сначала **Лексический анализ, а после ***синтаксический анализ.

Если нет ошибок написанный код будет выполняться.

Сама программа будет написана при использовании языка программирования C++.

Запись в файл будет производится по-средствам библиотек языка C++.

Графическое оформление с применением фреймворка QT;

Приложения:

*Интерпретирование - построчный анализ и последующее выполнение программы.

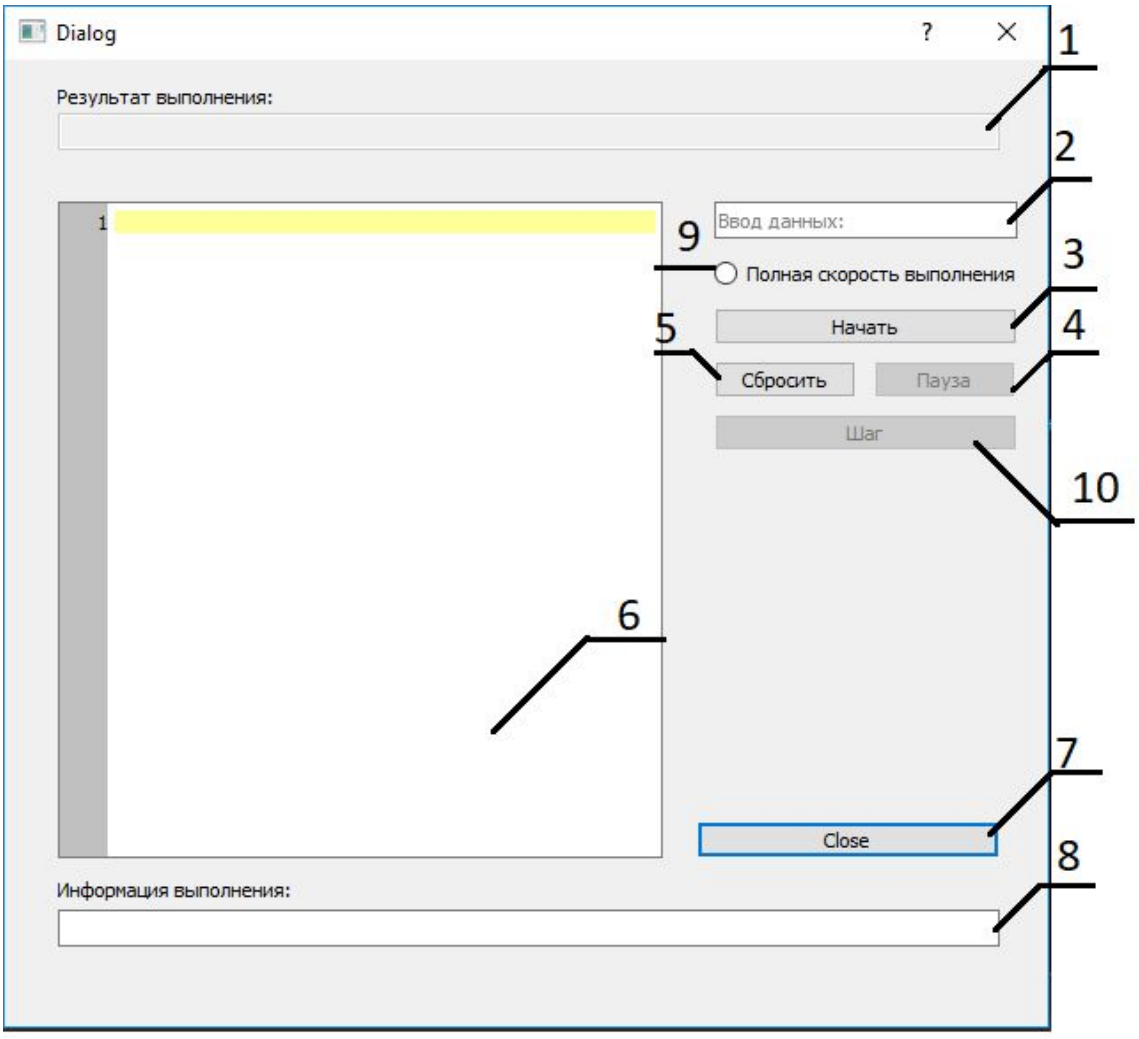
**Лексический анализ - процесс аналитического разбора входной последовательности символов на распознанные группы.

***Синтаксический анализ - преобразование входных данных в структурированный формат.

Описание выполненного проекта

Выполненный проект достиг поставленных целей, таких как обучение студентов командному взаимодействию через удаленный репозиторий с помощью GitHub. А также развитие студентов, которое они получали при изучении новых функций и взаимного взаимодействия.
Программа является законченной и представляет из себя диалоговое окно с возможностями взаимодействия такими, как написание кода или пошаговое выполнение.

Интерфейс.



Программа в завершенном состоянии выглядит так (рис. 1):

рис. 1

Интерфейс состоит из 6 кнопок и 4 поля ввода/вывода:
1 - представление бесконечной в обе стороны ленты. Слева граница для 0 позиции и лента сдвигается вправо.
2 - поле входных данных. Может быть пустым.

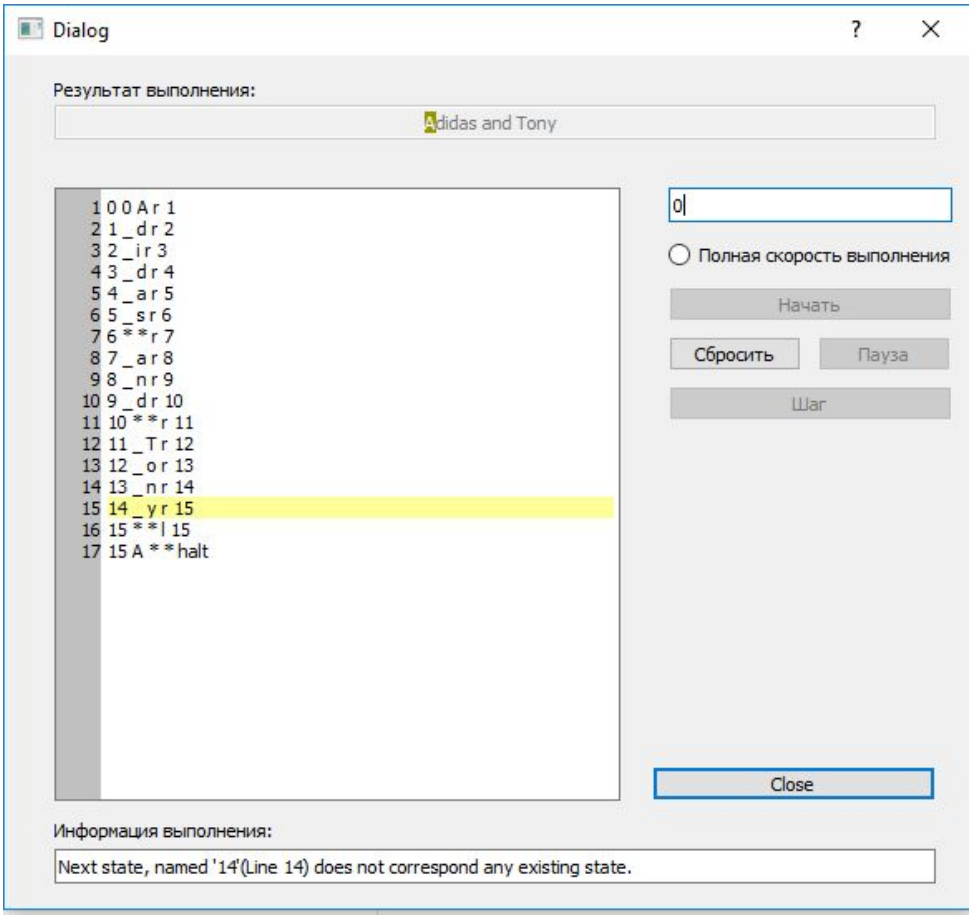
3 - кнопка запуска программы. Состоит из двух положений: “Начать” и “Продолжить”. При первичном нажатии программа выполняет один шаг и жмёт дальнейших действий: “Шаг” или “Продолжить”. Так же заполняет ленту данными из входных данных.
4 - кнопка остановки выполнения и ожидания дальнейших действий.
5 - кнопка сброса в первоначальное положение. Перенос входных данных в ленту.
6 - поле для ввода кода. Простой code-editor с нумерацией строк и подсветкой текущей строки.
7 - кнопка выхода из программы.
8 - поле для вывода ошибок при выполнении. Без ошибок не заполняется.
9 - Кнопка переключения скорости от задержки в 50 мс между действиями, до выполнения в полную доступную мощность компьютера.
10 - кнопка шага. Проводит одну итерацию, не может начинать выполнение.

Достоинства и недостатки*.

Плюсы:
1. Автономность.
2. Скорость выполнения.
3. Сохранение кода в файл.

Минусы:
1. Отсутствие выделения текущего шага.
2. Отсутствие шагов назад.
3. Отсутствие подсветки ошибки.

*Сравнение проводилось с программой представленной на [сайте](#) так, как была примером для выполнения данной курсовой работы.



Пример выполнения программы по написанию никнеймов. рис. 2

Личный вклад в проект.

Проект делился на две части, т.к. участников двое.
Части “внешняя” и “внутренняя”. Моя часть была “внешней”, т.е. мой вклад заключается в создании оконного приложения.
Взаимодействие пользователя с программой и программы с программой отслеживается и на основе поступающих сигналов происходят соответствующие действия: активация, дизактивация кнопок, сохранение написанного кода в файл, вывод данных полученных из “внутренней” части в нужное поле, обработка некоторых ошибок.

Приложение. Текст программы.

//файл EndlessTape.h

```
#ifndef ENDLESSTAPE_H_INCLUDED
#define ENDLESSTAPE_H_INCLUDED

#include <cstdint>
#include <utility>

class EndlessTape
{
public:
	enum{SYMBOLS_IN_CHUNK = 32};

private:
	struct DataChunk
	{
		char Symbols[SYMBOLS_IN_CHUNK];
		 DataChunk(): Symbols{}, PrevChunk(nullptr), NextChunk(nullptr){}
		 char & operator[](unsigned int i){return Symbols[i];}
		DataChunk * PrevChunk;
		DataChunk * NextChunk;
	};

	  uint8_t PositionInChunk;
	  DataChunk * GlobalPosition;

	  uint32_t ChunksNumber;
	  DataChunk * FirstChunk;
	  DataChunk * LastChunk;

	  int8_t LastShift;
	  void Stay(){LastShift = NONE;}
	  void MoveLeft();
	  void MoveRight();
	  void PutSymbol(char Symbol){(*GlobalPosition)[PositionInChunk] = Symbol;}
```

```
        friend class Program;

public:
        enum Shift: int8_t{LEFT = -1, NONE, RIGHT};
        EndlessTape & operator=(EndlessTape & CopiedTape);
        EndlessTape & operator=(EndlessTape && MovedTape);
        void operator=(const char * String);

        EndlessTape();
        EndlessTape(EndlessTape & CopiedTape): EndlessTape(){operator=(CopiedTape);}
        EndlessTape(EndlessTape && MovedTape):
        EndlessTape(){operator=(std::move(MovedTape));}
        EndlessTape(const char * String): EndlessTape(){operator=(String);}
        ~EndlessTape();
        void ResetPosition();
        const char * GetCurrentSymbol(){return &(*GlobalPosition)[PositionInChunk];}
        int8_t GetLastShift(){return LastShift;}
};

#endif // ENDLESSTAPE_H_INCLUDED
-------------------------------
//file EndlessTape.cpp
#include "mydialog.h"
#include <cstring>

void EndlessTape::MoveLeft()
{
        if(PositionInChunk == 0)
        {
                if(!GlobalPosition->PrevChunk)
                {
                        GlobalPosition->PrevChunk = new DataChunk;
                        GlobalPosition->PrevChunk->NextChunk = GlobalPosition;
                        FirstChunk = GlobalPosition->PrevChunk;
                        ChunksNumber++;
                }

                GlobalPosition = GlobalPosition->PrevChunk;
                PositionInChunk = SYMBOLS_IN_CHUNK - 1;
        }
        else
                PositionInChunk--;
```

```
		LastShift = LEFT;
}


void EndlessTape::MoveRight()
{
	if(PositionInChunk == SYMBOLS_IN_CHUNK - 1)
	{
		if(!GlobalPosition->NextChunk)
		{
			GlobalPosition->NextChunk = new DataChunk;
			GlobalPosition->NextChunk->PrevChunk = GlobalPosition;
			LastChunk = GlobalPosition->NextChunk;
			ChunksNumber++;
		}
		GlobalPosition = GlobalPosition->NextChunk;
		PositionInChunk = 0;
	}
	else
		PositionInChunk++;
	LastShift = RIGHT;
}

EndlessTape::EndlessTape()
{
	PositionInChunk = 0;
	LastShift = NONE;

	ChunksNumber = 1;
	GlobalPosition = new DataChunk;
	FirstChunk = GlobalPosition;
	LastChunk = GlobalPosition;
}

void EndlessTape::operator=(const char * String)
{
	size_t StringLength = strlen(String);

	DataChunk * CurrentChunk;
	while(FirstChunk)
	{
		CurrentChunk = FirstChunk->NextChunk;
```

```
                delete FirstChunk;
                FirstChunk = CurrentChunk;
        }

        PositionInChunk = 0;
        ChunksNumber = 1;
        GlobalPosition = FirstChunk = new DataChunk;
        for(CurrentChunk = FirstChunk; StringLength > SYMBOLS_IN_CHUNK;
StringLength -= SYMBOLS_IN_CHUNK, String += SYMBOLS_IN_CHUNK)
        {
                strncpy(CurrentChunk->Symbols, String, SYMBOLS_IN_CHUNK);
                CurrentChunk->NextChunk = new DataChunk;
                CurrentChunk->NextChunk->PrevChunk = CurrentChunk;
                CurrentChunk = CurrentChunk->NextChunk;
                ChunksNumber++;
        }
        strncpy(CurrentChunk->Symbols, String, StringLength);
        LastChunk = CurrentChunk;
}

EndlessTape::~EndlessTape()
{
        DataChunk * Temp;
        while(FirstChunk)
        {
                Temp = FirstChunk->NextChunk;
                delete FirstChunk;
                FirstChunk = Temp;
        }
}

EndlessTape & EndlessTape::operator=(EndlessTape & CopiedTape)
{
        PositionInChunk = CopiedTape.PositionInChunk;
        ChunksNumber = CopiedTape.ChunksNumber;

        FirstChunk = new DataChunk;
        strncpy(FirstChunk->Symbols, CopiedTape.FirstChunk->Symbols,
SYMBOLS_IN_CHUNK);
        if(CopiedTape.GlobalPosition == CopiedTape.FirstChunk)
                GlobalPosition = FirstChunk;
```

```
	DataChunk * i = FirstChunk;
	for(DataChunk * j = CopiedTape.FirstChunk->NextChunk; j; i = i->NextChunk, j = j->NextChunk)
	{
		i->NextChunk = new DataChunk;
		i->NextChunk->PrevChunk = i;
		strncpy(i->NextChunk->Symbols, j->Symbols, SYMBOLS_IN_CHUNK);
		if(CopiedTape.GlobalPosition == j)
			GlobalPosition = i->NextChunk;
	}
	LastChunk = i;

	return *this;
}

EndlessTape & EndlessTape::operator=(EndlessTape && MovedTape)
{
	PositionInChunk = MovedTape.PositionInChunk;
	GlobalPosition = MovedTape.GlobalPosition;
	MovedTape.GlobalPosition = nullptr;
	ChunksNumber = MovedTape.ChunksNumber;
	FirstChunk = MovedTape.FirstChunk;
	 LastChunk = MovedTape.LastChunk;
	MovedTape.FirstChunk = MovedTape.LastChunk = nullptr;

return *this;
}

void EndlessTape::ResetPosition()
{
	PositionInChunk = 0;
	GlobalPosition = FirstChunk;
	LastShift = NONE;
}
```

//file Program.h

```
#ifndef PROGRAM_H_INCLUDED
#define PROGRAM_H_INCLUDED

#include <cstdint>
#include <cstring>
```

```
class EndlessTape;



class Program
{
private:

	struct ProgramUnit
	{
		char Key;
		char SetTo;
		void (EndlessTape::*TapeMove)();
		uint16_t NextState;
	};

	ProgramUnit ** ProgramData;
	uint8_t * StatesEntriesCount;
	uint16_t StatesCount;
	char ** StatesNames;
	enum{HALT = 0xFFFF};
	uint16_t CurrentState;
	bool Halted;

	bool ProgramIsValid;
	char * ErrorString;

	size_t WordLen(const char * String);
	bool WordCmp(const char * String1, const char * String2);
	void Sort(char ** Strings, size_t n, size_t * Numbers);

public:
	enum{ERROR = false, SUCCESS = true};

	Program(Program &) = delete;
	Program(Program &&) = delete;
	Program & operator=(Program &) = delete;
	Program & operator=(Program &&) = delete;

	Program(): ProgramData(nullptr), StatesEntriesCount(nullptr), StatesCount(0),
StatesNames(nullptr), CurrentState(0), Halted(false), ProgramIsValid(false),
ErrorString(nullptr){}
```

```
	~Program();

	bool InitProgram(char ** ProgramString, size_t LinesCount);
	void ResetState(){Halted = false; CurrentState = 0;}
	bool IsHalted(){return Halted;}

	bool Execute(EndlessTape & TapeForExecution);

	const char * GetError(){return ErrorString? ErrorString: ProgramIsValid? Halted?
"Program has been halted!": "All is good.": "Program has not been initialized!";}
};

#endif // PROGRAM_H_INCLUDED
```

//file Program.cpp

```
#include "mydialog.h"

size_t Program::WordLen(const char * String)
{
	size_t i = 0;
	while(String[i] != ' ')i++;
	return i;
}

bool Program::WordCmp(const char * String1, const char * String2)
{
	if(WordLen(String1) != WordLen(String2))
		return false;
	for(size_t i = 0; String1[i] != ' '; i++)
		if(String1[i] != String2[i])
			return false;
	 return true;
}

Program::~Program()
{
	if(ProgramData)
		for(uint16_t i = 0; i < StatesCount; i++)
			delete [] ProgramData[i];

	if(StatesNames)
		for(uint16_t i = 0; i < StatesCount; i++)
```

```
                        delete [] StatesNames[i];

        delete [] ProgramData;
        delete [] StatesNames;
        delete [] StatesEntriesCount;
        delete [] ErrorString;
}

static size_t itos(uint32_t Num, char * Str)
{
        uint64_t NumCpy = Num, Numl = 1;
        while(NumCpy /= 10)
                Numl++;
        Str[Numl--] = '\0';
        do
                Str[Numl--] = Num % 10 + 48;
        while(Num /= 10);
        return strlen(Str);
}

bool Program::InitProgram(char ** ProgramString, size_t LinesCount)
{
        size_t LineBufferSize = 16;
        this->~Program();
        CurrentState = 0;
        StatesCount = 1;
        ProgramIsValid = false;
        Halted = false;

        size_t * LinesNumbers = new size_t[LinesCount];
        for(size_t i = 0; i < LinesCount; i++)
                LinesNumbers[i] = i + 1;

        char * StringTemp;
        for(size_t i = 0, j, StringLength; i < LinesCount; i++)
        {
                j = 0;
                while(ProgramString[i][j] == ' ' || ProgramString[i][j] == '\t')
                        j++;

                if(j > 0)
                {
```

```
                StringLength = strlen(ProgramString[i] + j) + 1;
                StringTemp = new char[StringLength];

                strncpy(StringTemp, ProgramString[i] + j, StringLength);
                strncpy(ProgramString[i], StringTemp, StringLength);

                delete [] StringTemp;
        }

        if(ProgramString[i][0] == ';' || ProgramString[i][0] == '\n')
                ProgramString[i][0] = '\0';
}

Sort(ProgramString, LinesCount, LinesNumbers);

size_t EmptyStringsShift = 0;
while(ProgramString[EmptyStringsShift][0] == '\0')
        EmptyStringsShift++;

int64_t BadSyntax = -1;
for(size_t i = EmptyStringsShift, j, k; i < LinesCount; i++)
{
        j = 0;
        while(  ProgramString[i][j] != '\n' && ProgramString[i][j] != ' ' &&
                ProgramString[i][j] != ';' && ProgramString[i][j] != '\t' &&
                ProgramString[i][j] != '\0')
                j++;

        if(ProgramString[i][j] != ' ')
        {
                BadSyntax = LinesNumbers[i];
                break;
        }

        k = 1;
        while(k < 7)
        {
                if(     ProgramString[i][j+k] == '\n' || ProgramString[i][j+k] == ' ' ||
                        ProgramString[i][j+k] == ';' || ProgramString[i][j+k] == '\t' ||
                        ProgramString[i][j+k] == '\0')
                        {
                        BadSyntax = LinesNumbers[i];
```

```
                break;
                }
        k++;
        if(ProgramString[i][j+k] != ' ' && ProgramString[i][j+k] != '\t')
        {
                BadSyntax = LinesNumbers[i];
                break;
        }
        k++;
}

if(BadSyntax > 0)break;

j += 7;
if(     ProgramString[i][j] == '\n' || ProgramString[i][j] == ' ' ||
        ProgramString[i][j] == ';' || ProgramString[i][j] == '\t' ||
        ProgramString[i][j] == '\0')
        {
                BadSyntax = LinesNumbers[i];
                break;
        }

while(  ProgramString[i][j] != '\n' && ProgramString[i][j] != ' ' &&
        ProgramString[i][j] != ';' && ProgramString[i][j] != '\t' &&
        ProgramString[i][j] != '\0')
        j++;

k = j;
while(  ProgramString[i][j] != '\n' && ProgramString[i][j] != ';' &&
        ProgramString[i][j] != '\0')
        if(ProgramString[i][j] != ' ' && ProgramString[i][j] != '\t')
        {
                const char * ErrorStr1 = "Unexpected symbol(";
                const char * ErrorStr2 = ") on line ";
                const char * ErrorStr3 = ".";

                char LineNumBuffer[LineBufferSize];
                ErrorString = new char[strlen(ErrorStr1) + 1 + strlen(ErrorStr2)
                                + itos(LinesNumbers[i], LineNumBuffer)
                                + strlen(ErrorStr3) + 1]{};

                strcpy(ErrorString, ErrorStr1);
```

```
                        size_t ErrorStringShift = strlen(ErrorStr1);
                        strncpy(ErrorString + ErrorStringShift,
                                                        ProgramString[i] + j, 1);

                        ErrorStringShift++;
                        strcpy(ErrorString + ErrorStringShift, ErrorStr2);

                        ErrorStringShift += strlen(ErrorStr2);
                        strcpy(ErrorString + ErrorStringShift, LineNumBuffer);

                        ErrorStringShift += strlen(LineNumBuffer);
                        strcpy(ErrorString + ErrorStringShift, ErrorStr3);

                        delete [] LinesNumbers;

                        return ERROR;
                }
                else
                        j++;

        ProgramString[i][k] = '\0';
}

if(BadSyntax > 0)
{
        const char * ErrorStr1 = "Command on line ";
        const char * ErrorStr2 = " not correspond to Turing
                                                machine command syntax.";

        char LineNumBuffer[LineBufferSize];
        ErrorString = new char[strlen(ErrorStr1) + itos(BadSyntax, LineNumBuffer)
                                + strlen(ErrorStr2) + 1]{};

        strcpy(ErrorString, ErrorStr1);

        size_t ErrorStringShift = strlen(ErrorStr1);
        strcpy(ErrorString + ErrorStringShift, LineNumBuffer);

        ErrorStringShift += strlen(LineNumBuffer);
        strcpy(ErrorString + ErrorStringShift, ErrorStr2);
```

```
		delete [] LinesNumbers;

		return ERROR;
}

if(strncmp(ProgramString[EmptyStringsShift], "0 ", 2))
{
		bool StartStateNotFounded = true;
		for(size_t i = EmptyStringsShift + 1; i < LinesCount; i++)
			if(!strncmp(ProgramString[i], "0 ", 2))
			{
				StartStateNotFounded = false;
				break;
			}

		if(StartStateNotFounded)
		{
			const char * ErrorStr = "Start state(0) was not found.";
			ErrorString = new char[strlen(ErrorStr) + 1]{};
			strcpy(ErrorString, ErrorStr);

			delete [] LinesNumbers;

			return ERROR;
		}
}

const char * PrevString = ProgramString[EmptyStringsShift];
for(size_t i = EmptyStringsShift + 1; i < LinesCount; i++)
{
		if(!WordCmp(PrevString, ProgramString[i]))
		{
			StatesCount++;
			PrevString = ProgramString[i];
		}
}

StatesEntriesCount = new uint8_t[StatesCount]{};
StatesEntriesCount[0]++;
size_t WordSize = WordLen(ProgramString[EmptyStringsShift]);
StatesNames = new char *[StatesCount];
StatesNames[0] = new char[WordSize + 1];
```

```
strncpy(StatesNames[0], ProgramString[EmptyStringsShift], WordSize);
StatesNames[0][WordSize] = '\0';

PrevString = ProgramString[EmptyStringsShift];
for(size_t i = EmptyStringsShift + 1, j = 0; i < LinesCount; i++)
{
	if(!WordCmp(PrevString, ProgramString[i]))
	{
		j++;
		PrevString = ProgramString[i];
		WordSize = WordLen(ProgramString[i]);
		StatesNames[j] = new char[WordSize + 1];
		strncpy(StatesNames[j], ProgramString[i], WordSize);
		StatesNames[j][WordSize] = '\0';
	}
	StatesEntriesCount[j]++;
}

ProgramData = new ProgramUnit *[StatesCount]{};
bool HaltFound = false, StateNotFound;
for(size_t i = 0, Line = EmptyStringsShift, StringShift; i < StatesCount; i++)
{
ProgramData[i] = new ProgramUnit[StatesEntriesCount[i]];
if(!strcmp(StatesNames[i], "0"))
	CurrentState = i;
for(size_t j = 0; j < StatesEntriesCount[i]; j++, Line++)
{
	StringShift = 0;
	StringShift += WordLen(ProgramString[Line] + StringShift) + 1;
	ProgramData[i][j].Key = ProgramString[Line][StringShift] == '_'?
						0: ProgramString[Line][StringShift];
	for(size_t k = 0; k < j; k++)
		if(ProgramData[i][j].Key == ProgramData[i][k].Key)
		{
			const char * ErrorStr1 = "Multiple entries for symbol \'";
			const char * ErrorStr2 = "\' in state, named \'";
			const char * ErrorStr3 = "\'.";

			ErrorString = new char[strlen(ErrorStr1) + 1 + strlen(ErrorStr2)
				+ strlen(StatesNames[i]) + strlen(ErrorStr3) + 1]{};

			strcpy(ErrorString, ErrorStr1);
```

```
                size_t ErrorStringShift = strlen(ErrorStr1);
                ErrorString[ErrorStringShift] = ProgramData[i][j].Key?
                                                ProgramData[i][j].Key: '_';

                ErrorStringShift++;
                strcpy(ErrorString + ErrorStringShift, ErrorStr2);

                ErrorStringShift += strlen(ErrorStr2);
                strcpy(ErrorString + ErrorStringShift, StatesNames[i]);

                ErrorStringShift += strlen(StatesNames[i]);
                strcpy(ErrorString + ErrorStringShift, ErrorStr3);

                delete [] LinesNumbers;

                return ERROR;
        }
StringShift += 2;
ProgramData[i][j].SetTo = ProgramString[Line][StringShift] == '_'?
                                0: ProgramString[Line][StringShift];
StringShift += 2;
switch(ProgramString[Line][StringShift])
{
        case '*':
                ProgramData[i][j].TapeMove = EndlessTape::Stay;
                break;

        case 'r':
                ProgramData[i][j].TapeMove = EndlessTape::MoveRight;
                break;

        case 'l':
                ProgramData[i][j].TapeMove = EndlessTape::MoveLeft;
                break;

        default:
                const char * ErrorStr1 = "Symbol \'";
                const char * ErrorStr2 = "\' in the command on line ";
                const char * ErrorStr3 = " does not match any direction control
character(r/l/*).";
```

```
                    char LineNumBuffer[LineBufferSize];
                    ErrorString = new char[strlen(ErrorStr1) + 1 + strlen(ErrorStr2)
                                + itos(LinesNumbers[Line], LineNumBuffer)
                                + strlen(ErrorStr3) + 1]{};

                    strcpy(ErrorString, ErrorStr1);

                    size_t ErrorStringShift = strlen(ErrorStr1);
                    ErrorString[ErrorStringShift] =
                                        ProgramString[Line][StringShift];

                    ErrorStringShift++;
                    strcpy(ErrorString + ErrorStringShift, ErrorStr2);

                    ErrorStringShift += strlen(ErrorStr2);
                    strcpy(ErrorString + ErrorStringShift, LineNumBuffer);

                    ErrorStringShift += strlen(LineNumBuffer);
                    strcpy(ErrorString + ErrorStringShift, ErrorStr3);

                    delete [] LinesNumbers;

                    return ERROR;
}
StringShift += 2;
StateNotFound = true;
if(!strncmp(ProgramString[Line] + StringShift, "halt", 4))
{
        HaltFound = true;
        StateNotFound = false;
        ProgramData[i][j].NextState = HALT;
}
else
        for(size_t k = 0; k < StatesCount; k++)
                if(!strcmp(ProgramString[Line]
                                        + StringShift, StatesNames[k]))
                {
                        StateNotFound = false;
                        ProgramData[i][j].NextState = k;
                        break;
                }
```

```
        if(StateNotFound)
        {
                const char * ErrorStr1 = "Next state, named \'";
                const char * ErrorStr2 = "\'(Line ";
                const char * ErrorStr3 = ") does not correspond any existing state.";

                char LineNumBuffer[LineBufferSize];
                ErrorString = new char[strlen(ErrorStr1) + strlen(ProgramString[Line]
                                        + StringShift) + strlen(ErrorStr2)
                                        + itos(LinesNumbers[Line], LineNumBuffer)
                                        + strlen(ErrorStr3) + 1]{};

                strcpy(ErrorString, ErrorStr1);

                size_t ErrorStringShift = strlen(ErrorStr1);
                strcpy(ErrorString + ErrorStringShift, ProgramString[Line]
                                                                + StringShift);

                ErrorStringShift += strlen(ProgramString[Line] + StringShift);
                strcpy(ErrorString + ErrorStringShift, ErrorStr2);

                ErrorStringShift += strlen(ErrorStr2);
                strcpy(ErrorString + ErrorStringShift, LineNumBuffer);

                ErrorStringShift += strlen(LineNumBuffer);
                strcpy(ErrorString + ErrorStringShift, ErrorStr3);

                delete [] LinesNumbers;

                return ERROR;
        }
}
}

if(!HaltFound)
{
        const char * ErrorStr = "Halt state was not found.";
        ErrorString = new char[strlen(ErrorStr) + 1]{};
        strcpy(ErrorString, ErrorStr);

        delete [] LinesNumbers;
```

```
                return ERROR;
        }

        ProgramIsValid = true;

        delete [] LinesNumbers;

        return SUCCESS;
}

void Program::Sort(char ** Strings, size_t n, size_t * Numbers)
{
        const size_t CiuraSteps[9] = {701, 301, 132, 57, 23, 10, 4, 1, 0};
        char * Cashe;
        size_t NumberCashe;
        for(size_t i = 0, d = CiuraSteps[i]; d != 0; d = CiuraSteps[++i])
                for(size_t i = d, j; i < n; i++)
                {
                        Cashe = Strings[i];
                        NumberCashe = Numbers[i];
                        for(j = i; j >= d; j -= d)
                        {
                                if(strcmp(Cashe, Strings[j-d]) < 0)
                                {
                                        Strings[j] = Strings[j-d];
                                        Numbers[j] = Numbers[j-d];
                                }
                                else
                                        break;
                        }
                        Strings[j] = Cashe;
                        Numbers[j] = NumberCashe;
                }
}

bool Program::Execute(EndlessTape & TapeForExecution)
{
        if(!ProgramIsValid || Halted)
                return ERROR;

        char KeyForCheck = *TapeForExecution.GetCurrentSymbol();
        if(KeyForCheck == ' ')
```

```
        KeyForCheck = 0;

int16_t KeyFinded = -1;
for(uint8_t i = 0; i < StatesEntriesCount[CurrentState]; i++)
{
        if(ProgramData[CurrentState][i].Key == KeyForCheck)
        {
                KeyFinded = i;
                break;
        }
        else
                if(ProgramData[CurrentState][i].Key == '*')
                        KeyFinded = i;

}

if(KeyFinded > -1)
{
        if(ProgramData[CurrentState][KeyFinded].SetTo != '*')
TapeForExecution.PutSymbol(ProgramData[CurrentState][KeyFinded].SetTo);
        (TapeForExecution.*ProgramData[CurrentState][KeyFinded].TapeMove)();
        CurrentState = ProgramData[CurrentState][KeyFinded].NextState;
}
else
{
        const char * ErrorStr1 = "State, named ";
        const char * ErrorStr2 = " has don't have entry for \'";
        const char * ErrorStr3 = "\'.";

        ErrorString = new char[strlen(ErrorStr1) + strlen(StatesNames[CurrentState])
                                + strlen(ErrorStr2) + 1 + strlen(ErrorStr3) + 1]{};

        strcpy(ErrorString, ErrorStr1);

        size_t ErrorStringShift = strlen(ErrorStr1);
        strcpy(ErrorString + ErrorStringShift, StatesNames[CurrentState]);

        ErrorStringShift += strlen(StatesNames[CurrentState]);
        strcpy(ErrorString + ErrorStringShift, ErrorStr2);

        ErrorStringShift += strlen(ErrorStr2);
        ErrorString[ErrorStringShift] = KeyForCheck? KeyForCheck: '_';
```

```
                ErrorStringShift++;
                strcpy(ErrorString + ErrorStringShift, ErrorStr3);

                return ERROR;
        }

        if(CurrentState == HALT)
        Halted = true;

        return SUCCESS;
}

//file mydialog.h
#ifndef MYDIALOG_H
#define MYDIALOG_H

#include "customplaintext.h"
#include "ui_mydialog.h"
#include "EndlessTape.h"
#include "Program.h"
#include <QFile>
#include <QString>
#include <QTextStream>
#include <QRegExp>
#include <QPlainTextEdit>
#include <QObject>
#include <QPainter>
#include <QTextBlock>
#include <QList>
#include <QAbstractScrollArea>
#include <fstream>
#include <cstring>

using namespace std;


class MyDialog : public QDialog, public Ui::Dialog
{
        Q_OBJECT

public:
```

```
        explicit MyDialog(QWidget *parent = nullptr);
        void Proccess();

private:
        QWidget *lineNumberArea;
        CustomPlainText PlainText;

private slots:
        void on_Start_clicked();
        void on_FullSpeed_clicked();
        void on_Pause_clicked();
        void on_Reset_clicked();
        void on_Step_clicked();
};



#endif // MYDIALOG_H

//file mydialog.cpp
#include "mydialog.h"
#include <QtWidgets>

int ActivFase = 0; // 0 - first, 1 - next, 2 - pause, 3 -step
bool b_FullSpeed = false;
bool b_Reset = false;
void MyDialog::Proccess()
{
        Start->setText("Продолжить");
        QFile file("text.txt");
        QTextStream writeStream(&file);
        file.open(QIODevice::WriteOnly | QIODevice::Text);
        QString temp = Input->text();
        Output->setText(temp);
        Output->setSelection(0,1);
        QString to_file = plainTextEdit->toPlainText();
        QStringList strList = to_file.split(QRegExp("[\n]"));
        writeStream << to_file << endl;
        file.close();// first stade

        if(to_file.isEmpty())
        {
```

```
        ErrorLine->setText("Code is empty.");
        return;
}
string code;
string memory;
size_t dlina_stroki;

int ListSize = strList.size();
char **MassivChar = new char *[ListSize];
for(int i = 0; i < ListSize; i++)
{
        memory = strList.at(i).toStdString();
        dlina_stroki = memory.length();
        MassivChar[i] = new char[dlina_stroki + 1];
        strcpy(MassivChar[i], memory.c_str());
        MassivChar[i][dlina_stroki] = '\0';
}
//second stade
int x = 0;
EndlessTape Tape;
Program program;
const char * CurrentBukva;
Output->setSelection(x,1);
if(!program.InitProgram( MassivChar , ListSize ))
{
        ErrorLine->setText(program.GetError());

        for(int i = 0; i < ListSize; i++)
                delete [] MassivChar[i];

        delete []MassivChar;

        return;
}
Tape = temp.toStdString().c_str();
while(!program.IsHalted())
{
         while(ActivFase == 2)
        {
                QEventLoop loop;
                QTimer timer;
                timer.setInterval(100);
```

```
                connect (&timer, SIGNAL(timeout()), &loop, SLOT(quit()));
                timer.start();
                loop.exec();
                if(b_Reset)
                return;
        }
        if(!b_FullSpeed)
        {
                QEventLoop loop;
                QTimer timer;
                timer.setInterval(50);
                connect (&timer, SIGNAL(timeout()), &loop, SLOT(quit()));
                timer.start();
                loop.exec();
        }
        if(b_Reset)
                return;
        CurrentBukva = Tape.GetCurrentSymbol();
        if(!program.Execute(Tape))
        {
                ErrorLine->setText(program.GetError());
                break;
        }
        if(x < 0)
        {
                temp.prepend(*CurrentBukva);
                 x = 0;
        }
        else if(x > temp.size())
                temp.append(*CurrentBukva);
        else
                temp[x] = (*CurrentBukva);

        Output->setText(temp);
        Output->setSelection(x,1);
        x += Tape.GetLastShift();



        if(ActivFase == 3)
        ActivFase = 2;
}
```

```
        if(program.IsHalted())
        {
                Pause->setEnabled(false);
                Start->setEnabled(false);
                Step->setEnabled(false);
        }
}

CustomPlainText::CustomPlainText(QWidget *parent): QPlainTextEdit(parent)
{
        lineNumberArea = new LineNumberArea(this);

        connect(this, SIGNAL(cursorPositionChanged()), this,
        SLOT(highlightCurrentLine()));
        connect(this, SIGNAL(updateRequest(QRect,int)), this,
        SLOT(updateLineNumberArea(QRect,int)));
        connect(this, SIGNAL(blockCountChanged(int)), this,
        SLOT(updateLineNumberAreaWidth()));

        updateLineNumberAreaWidth();
        highlightCurrentLine();
}

void CustomPlainText::CustomSetViewportMargins(int left, int top, int right, int bottom)
{
        setViewportMargins(left, top, right, bottom);
}

void CustomPlainText::CustomSetViewportMargins(const QMargins & margins)
{
        setViewportMargins(margins);
}

MyDialog::MyDialog(QWidget *parent) : QDialog (parent)
{

        setupUi(this);
        Output->setEnabled(false);
        Output->setStyleSheet("QLineEdit { selection-background-color: rgb(150, 150, 00);
}");
        Output->setSelection(0,1);
}
```

```
int CustomPlainText::lineNumberAreaWidth()
{
	int digits = 1;
	int max = qMax(1, 1000);
	while (max >= 10) {
		max /= 10;
		digits++;
	}

	int space = 3 + fontMetrics().horizontalAdvance(QLatin1Char('9')) * digits;

	return space;
}

void CustomPlainText::updateLineNumberAreaWidth()
{
	setViewportMargins(lineNumberAreaWidth(), 0, 0, 0);
}

void CustomPlainText::updateLineNumberArea(const QRect &rect, int dy)
{
	if (dy)
		lineNumberArea->scroll(0, dy);
	else
		lineNumberArea->update(0, rect.y(), lineNumberArea->width(), rect.height());

	if (rect.contains(viewport()->rect()))
		updateLineNumberAreaWidth();
}

void CustomPlainText::resizeEvent(QResizeEvent *e)
{
	QPlainTextEdit::resizeEvent(e);

	QRect cr = contentsRect();
	lineNumberArea->setGeometry(QRect(cr.left(), cr.top(), lineNumberAreaWidth(),
cr.height()));
```

```
}

void CustomPlainText::highlightCurrentLine()
{
	QList<QTextEdit::ExtraSelection> extraSelections;

	if (!isReadOnly())
	{
		QTextEdit::ExtraSelection selection;

		QColor lineColor = QColor(Qt::yellow).lighter(160);

		selection.format.setBackground(lineColor);
		selection.format.setProperty(QTextFormat::FullWidthSelection, true);

		selection.cursor = textCursor();
		selection.cursor.clearSelection();
		extraSelections.append(selection);
	}

	setExtraSelections(extraSelections);
}

void CustomPlainText::highlightErrorLine(int line)
{
	QList<QTextEdit::ExtraSelection> extraSelections;

	if (!isReadOnly()) {
		QTextEdit::ExtraSelection selection;

		QColor lineColor = QColor(Qt::red).lighter(160);

		selection.format.setBackground(lineColor);
		selection.format.setProperty(QTextFormat::FullWidthSelection, true);
		selection.cursor.movePosition(QTextCursor::StartOfWord);
		selection.cursor.setPosition(line);
		selection.cursor = textCursor();
		selection.cursor.clearSelection();
		extraSelections.append(selection);
	}

	setExtraSelections(extraSelections);
```

```
}

void CustomPlainText::lineNumberAreaPaintEvent(QPaintEvent *event)
{
	QPainter painter(lineNumberArea);
	painter.fillRect(event->rect(), Qt::lightGray);

	QTextBlock block = firstVisibleBlock();
	int blockNumber = block.blockNumber();
	int top = (int) blockBoundingGeometry(block).translated(contentOffset()).top();
	int bottom = top + (int) blockBoundingRect(block).height();
	while (block.isValid() && top <= event->rect().bottom())
	{
		if (block.isVisible() && bottom >= event->rect().top())
		{
			QString number = QString::number(blockNumber + 1);
			painter.setPen(Qt::black);
			painter.drawText(0, top, lineNumberArea->width(),
			fontMetrics().height(),
			Qt::AlignRight, number);
		}

		block = block.next();
		top = bottom;
		bottom = top + (int) blockBoundingRect(block).height();
		blockNumbe++r;
	}
}

void MyDialog::on_Start_clicked()
{
	if(ActivFase == 0)
	{
		ActivFase = 3;
		Pause->setEnabled(true);
		b_Reset = false;
		Proccess();
	}
	else if(ActivFase == 2)
	{
		Start->setText("Начать");
		ActivFase = 1;
```

```
                Start->setEnabled(false);
                Pause->setEnabled(true);
        }
        else if(ActivFase == 3)
        {
        Start->setText("Начать");
        ActivFase = 1;
        Start->setEnabled(false);
        Pause->setEnabled(true);
        }
}

void MyDialog::on_FullSpeed_clicked()
{
        if(b_FullSpeed)
                b_FullSpeed = false;
        else
                b_FullSpeed = true;
}

void MyDialog::on_Pause_clicked()
{
        ActivFase = 2;
        Start->setText("Продолжить");
        Pause->setEnabled(false);
        Start->setEnabled(true);
        Step->setEnabled(true);
}

void MyDialog::on_Reset_clicked()
{
        QString temp = Input->text();
        Output->repaint();
        Output->setText(temp);
        ActivFase = 0;
        Start->setEnabled(true);
        Start->setText("Начать");
        Step->setEnabled(true);
        b_Reset = true;
        Output->setSelection(0,1);
        ErrorLine->clear();
}
```

```
void MyDialog::on_Step_clicked()
{
	Start->setText("Продолжить");
	Start->setEnabled(true);
	Pause->setEnabled(false);
	ActivFase = 3;
}
```