Научная работа

Миграция программного кода Pascal – Basic - C

Автор: **Винокуров Денис**,

ученик 11 «Б» класса

МБОУ гимназия
им. И.А. Бунина,
город Воронеж

Руководитель: **Головин Дмитрий Владимирович**,

учитель информатики
и ИКТ

Воронеж, 2013

Содержание

Введение 2

1 Актуальность и практическое применение мигратора 3

2 Функциональное описание мигратора 4

2.1 Основа работы приложения 4

2.2 Пользовательский интерфейс приложения 5

3 Структурные принципы обработки кода 8

3.1 Соглашения описания программы 8

3.2 Некоторые основные конструкции программы 8

3.3 Входные данные 10

3.4 Синтаксический анализ 15

3.5 Миграция 17

3.6 Выходные данные 28

4 Возможности мигратора 29

4.1 Что мигратор уже умеет 29

4.2 Что мигратор пока не умеет 29

4.3 Перспективы 31

Приложение 1 Примеры миграции простых алгоритмических конструкций 32

1. Ввод-вывод 32
2. Условный переход 33
3. Цикл со счетчиком 34
4. Цикл с предусловием 35
5. Функции 37

Приложение 2 Листинг программы 39

Введение

Со времени создания первых программируемых машин человечество придумало более двух тысяч языков программирования. Каждый год их число увеличивается. Некоторыми языками умеет пользоваться только небольшое число их собственных разработчиков, другие становятся известны миллионам людей.

Языки программирования представлены в виде набора спецификаций, определяющих их синтаксис и семантику. Даже у самых популярных языков этот «фундамент» сильно отличается друг от друга. И, зачастую, даже опытному программисту бывает нелегко разобраться в синтаксисе другого языка программирования.

Создание универсального мигратора кода, «переводчика» с одного языка программирования на другой, могло бы решить некоторые проблемы программистов, связанные с пониманием и воспроизводством кода некоторых программ. Именно этот механизм и будет рассмотрен в данной работе.

Создать универсальный мигратор для любых языков практически невозможно, вероятно, это и не нужно на данном этапе. Но хотелось бы иметь механизм перевода кода хотя бы для наиболее популярных из них. Речь идет о языках Pascal, Basic и C. И такой механизм мне удалось создать. Представляю его вашему вниманию. Описываемая ниже программа-мигратор является результатом практической части моих исследований. Все программные коды, представленные в работе, ни у кого не были заимствованы, и написаны мной собственноручно, поэтому являются авторскими.

1 Актуальность и практическое применение мигратора

В наше время многообразие языков программирования является одной из важнейших проблем, усложняющих кооперированную работу специалистов, и ее решение происходит разными способами.

Во-первых, создаются стандарты разработки в той или иной области. Но не всегда очевидно, использование какого языка программирования является наилучшим.

Поэтому существует еще один способ решения описанной проблемы – создание платформ, способных интегрировать коды, написанные на различных языках, и представлять его в виде единой программы (например, .NET Framework). Но такие платформы в настоящий момент поддерживают далеко не все многообразие существующих языков. Однако многие универсальные библиотеки уже были написаны, и их использование значительно облегчило бы работу многим программистам, не знающим исходного языка. Именно в таких случаях применим рассматриваемый мигратор.

Он способен «переводить» код между тремя языками: Pascal, C и Basic. Каждый из этих языков имеет уникальный синтаксис, а главное – изначальное применение. Данный мигратор, в перспективе, позволит использовать совместно мощь «низкоуровневых» функций C, простоту синтаксиса Basic и множественные математические функции библиотек Pascal.

2 Функциональное описание мигратора

2.1 Основа работы приложения

Язык программирования — это формальная знаковая система, предназначенная для записи компьютерных программ. Язык программирования определяет набор лексических, синтаксических и семантических правил, задающих внешний вид программы и действия, которые выполнит исполнитель (компьютер) под её управлением.

Любой язык программирования состоит из последовательностей допустимых символов, имеющих смысл для транслятора – **лексем**. Лексемы любого языка подразделяются на четыре типа:

разделители **–** служат для разделения стоящих рядом лексем и выделения смысловых блоков кода (например, символ пробела, символ конца строки);

ключевые слова **–** зарезервированные лексемы языка программирования, предназначенные для определения контекста стоящих рядом лексем, определения последовательности выполнения алгоритма и т.д. (например, **for**, **while**);

типы переменных **–** лексемы, служащие для определения типа переменных (например, **integer, string**);

переменные – слова, определяемые пользователем и дающие условные названия ячейкам памяти.

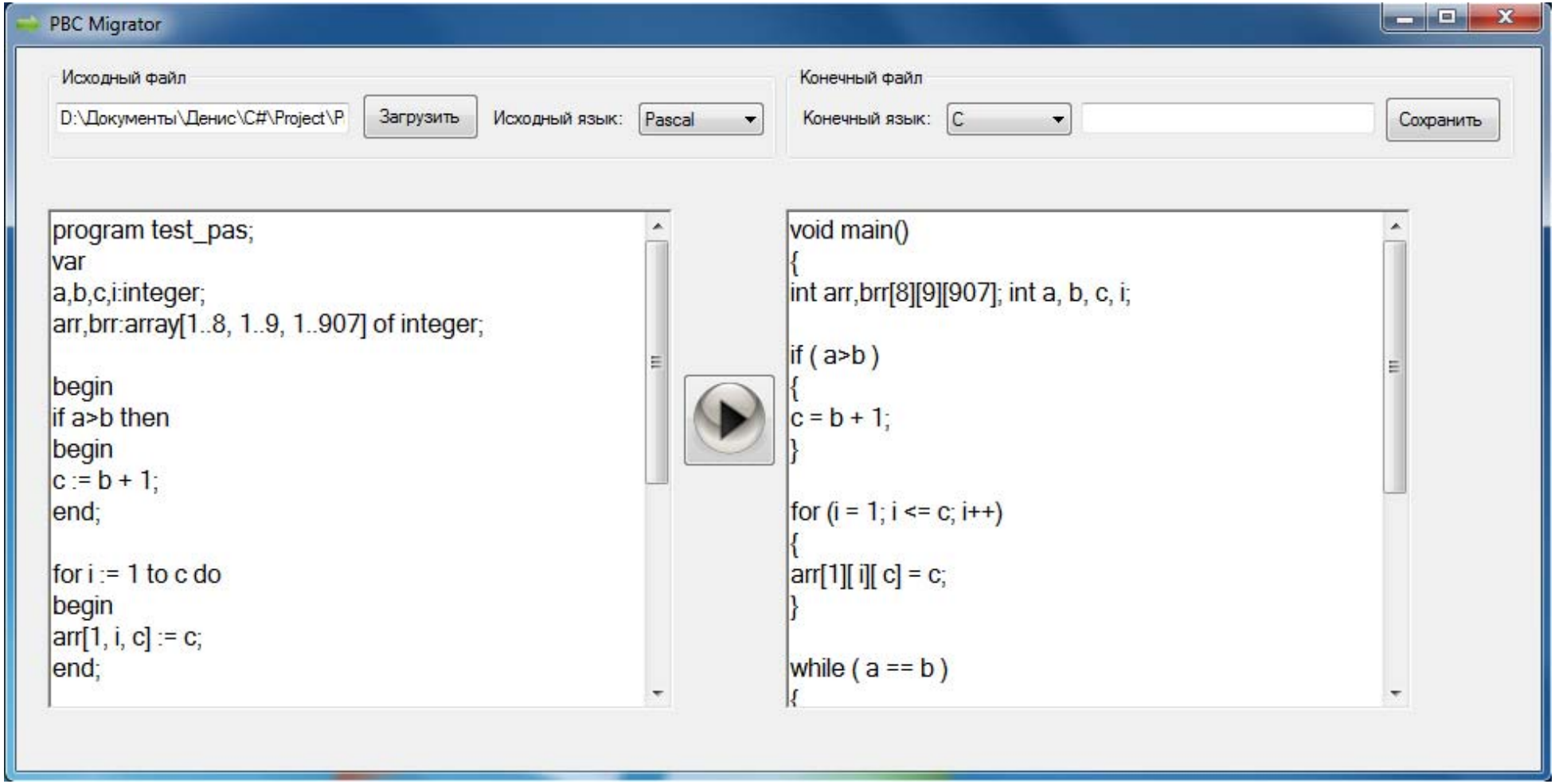
Лексемы, в свою очередь, составляют линейные последовательности, связанные формальной грамматикой языка, - **синтаксические конструкции**. На их обработке и основан принцип работы данного мигратора.

Мигратор получает код исходной программы, разбивает его на лексемы, сравнивает их со словарем исходного языка, расположенном в XML-файле. Затем происходит выделение среди лексем составных синтаксических конструкций, их трансляция в конструкции конечного языка. На последнем этапе осуществляется трансляция лексем исходного языка в лексемы конечного языка, в соответствии со словарем конечного языка, и их вывод на форму пользователя.

2.2 Пользовательский интерфейс приложения

Форма пользователя, являясь частью графического интерфейса, предоставляет пользователю доступ к функциям программы. Данная форма создана с использованием библиотеки **Windows Forms** и средств языка **C#**. Она предоставляет следующие возможности:

- загрузка файла исходной программы;
- ввод исходной программы с клавиатуры;
- начало миграции этого файла;
- отображение конечного кода программы;
- редактирование конечного кода программы;
- сохранение полученного кода в файл конечного языка.


PBC Migrator
Исходный файл
D:\Документы\Денис\C#\Project\P
Загрузить
Исходный язык:
Pascal
Конечный файл
Конечный язык:
C
Сохранить
program test_pas;
var
a,b,c,i:integer;
arr,brr:array[1..8, 1..9, 1..907] of integer;
begin
if a>b then
begin
c := b + 1;
end;
for i := 1 to c do
begin
arr[1, i, c] := c;
end;
void main()
{
int arr,brr[8][9][907]; int a, b, c, i;
if (a>b)
{
c = b + 1;
}
for (i = 1; i <= c; i++)
{
arr[1][i][c] = c;
}
while (a == b)
{

На форме располагаются следующие элементы управления:

1) кнопка загрузки исходного файла – позволяет пользователю указать путь к файлу, содержащему исходную программу;
2) список выбора исходного языка – позволяет пользователю выбрать язык, на котором написана исходная программа; при загрузке программы из файла заполняется автоматически;
3) текстовое поле исходного кода программы – позволяет пользователю ввести исходный код программы; отображает исходный код при загрузке его из файла;
4) список выбора конечного языка – позволяет пользователю выбрать конечный язык;
5) кнопка начала миграции – позволяет пользователю начать миграцию кода;
6) текстовое поле конечного кода программы – позволяет пользователю увидеть и отредактировать результат работы мигратора;
7) кнопка сохранения конечного файла – позволяет пользователю сохранить конечный код в файл.

3 Структурные принципы обработки кода

3.1 Соглашения описания программы

Программа описывается с использованием синтаксиса языков C# 5.0 и XML и некоторых классов платформы .NET Framework 3.5. Ниже представлены некоторые блоки кода. Их разметка соответствует разметке среды программирования Microsoft Visual Studio 2012. Для удобства восприятия присутствует следующее форматирование:

- //это комментарий, /* тоже комментарий */ – пример выделения комментариев;
- class – черным цветом выделяются ключевые слова языка программирования;
- <Name> - в угловых скобках приводятся xml-теги;
- Lexem – светло-зеленым цветом выделены названия классов и структур;
- <...> - многоточие в белых угловых скобках обозначает пропуск части кода алгоритма, незначительной для понимания его работы.

3.2 Некоторые основные конструкции программы

Для понимания работы данной программы необходимо ознакомиться с некоторыми основными конструкциями данных, которыми она оперирует.

1. Структура Lexem является представлением лексемы. Она содержит ряд конструкторов и свойств:

```
struct Lexem
{
    public string Content; //сама лексема
    public LexemType Type; //ее тип
```

```
        public LexemSubType SubType; //подтип лексемы
        <...>
    }
```

2. Перечисления LexemType и LexemSubType содержат все возможные типы и подтипы лексем.

```
enum LexemType
{
    Divider, //разделители
    Keyword, //ключевые слова
    Variable, //переменные
    VarType //объявление типа данных
}
```

3. Функция поиска лексемы в коллекции по имени.

```
//функция поиска лексемы с именем name среди коллекции
лексем sourceLangLex
static Lexem FindLexemByName(string name, List<Lexem>
sourceLangLex)
{
        foreach (Lexem lex in sourceLangLex)
        {
                if (name == lex.Content)
                {
                        return lex;
                }
        }

        //если переданная строка является числом,
возвращаем константу
        if (char.IsDigit(name[0]))
```

```
        {
            return new Lexem(name, LexemType.Variable,
LexemSubType.constant);
        }

        return new Lexem(name, LexemType.Variable,
LexemSubType.variable);
}
```

4. Функция проверки нахождения лексемы в коллекции.

```
//функция проверки наличия лексемы с именем name
среди коллекции sourceLangLex
static bool CheckLexemContaining(string name, List<Lexem>
sourceLangLex)
{
    foreach (Lexem lex in sourceLangLex)
    {
        if (name == lex.Content)
        {
            return true;
        }
    }
    return false;
}
```

3.3 Входные данные

Для корректной работы программе необходимы следующие входные данные: текст программы на исходном языке программирования, словари лексем исходного и конечного языков.

Словарь лексем языка

Словари лексем исходного и конечного языков представлены объектами класса ConvertableLanguage. Этот класс принимает при инициализации аргумент – название исходного языка. Далее, в зависимости от аргумента, при помощи специальной процедуры происходит загрузка лексем из xml-файла, созданного по определенным правилам и содержащего описание основных значимых выражений и конструкций конкретного языка.

Каждый xml-файл разделен на три основных блока. Блок <Name> содержит название языка, который представляет данный файл. Блок <Lexems> содержит описание минимальных значащих частей языка, таких как разделители (подблок <Dividers>), ключевые слова (подблок <Keywords>) и типы данных (подблок <VarTypes>). В последнем блоке <Constructions> содержатся некоторые синтаксические конструкции, характерные для конкретного языка.

Пример содержимого xml-файла для языка Pascal:

```
<?xml version="1.0" encoding="ascii" ?>
<ConvertableLanguage>

 <Name>Pascal</Name>

 <Lexems>
  <Dividers>
   <EndOfLine>;</EndOfLine>
   <SpaceSymbol>&#32;</SpaceSymbol>
   <StampleOpenSymbol>(</StampleOpenSymbol>
```

```
    <StampleCloseSymbol>)</StampleCloseSymbol>
    <SquareStampleOpen>[</SquareStampleOpen>
    <SquareStampleClose>]</SquareStampleClose>
    <ColonSymbol ambiguous="single">:</ColonSymbol>
    <CommaSymbol>,</CommaSymbol>
    <MathMore>&gt;</MathMore>
    <MathLess>&lt;</MathLess>
    <ArrayRange ambiguous="double">..</ArrayRange>
    <ProgramEnd ambiguous="single">.</ProgramEnd>
    <LineComment>//</LineComment>
    <BlockCommentOpen>{</BlockCommentOpen>
    <BlockCommentClose>}</BlockCommentClose>
    <StringOpen>'</StringOpen>
    <CharOpen>'</CharOpen>
  </Dividers>

  <Keywords>
    <ProgramStart>program</ProgramStart>
    <ConstDef>const</ConstDef>
    <VarBlock>var</VarBlock>
    <BlockStart>begin</BlockStart>
    <BlockEnd>end</BlockEnd>
    <AlgIf>if</AlgIf>
    <AlgThen>then</AlgThen>
    <AlgElse>else</AlgElse>
    <AlgFor>for</AlgFor>
    <AlgTo>to</AlgTo>
    <AlgDo>do</AlgDo>
    <AlgOf>of</AlgOf>
    <AlgWhile>while</AlgWhile>
```

```
    <AlgRepeat>repeat</AlgRepeat>
    <AlgUntil>until</AlgUntil>
    <Procedure>procedure</Procedure>
    <Function>function</Function>
    <MathPlus>+</MathPlus>
    <MathMinus>-</MathMinus>
    <MathEqual>=</MathEqual>
    <MathUnequal>&lt;&gt;</MathUnequal>
    <LogicAssign ambiguous="double">:=</LogicAssign>
    <LogicAnd>and</LogicAnd>
    <ArrayType>array</ArrayType>
  </Keywords>

  <VarTypes>
    <Type32sInt>integer</Type32sInt>
    <TypeChar>char</TypeChar>
    <Type32Float>real</Type32Float>
    <TypeString>string</TypeString>
  </VarTypes>
</Lexems>

<Constructions>
  <VariableDefinition type="keyword"></VariableDefinition>
  <ArrayDefinition gap="true"></ArrayDefinition>


  <ForLoopConstruction>
    <AlgFor></AlgFor>
    <SpaceSymbol></SpaceSymbol>
    <Variable></Variable>
```

```
    <SpaceSymbol></SpaceSymbol>
    <LogicAssign></LogicAssign>
    <SpaceSymbol></SpaceSymbol>
    <Constant></Constant>
    <SpaceSymbol></SpaceSymbol>
    <AlgTo></AlgTo>
    <SpaceSymbol></SpaceSymbol>
    <Constant></Constant>
    <SpaceSymbol></SpaceSymbol>
    <AlgDo></AlgDo>
   </ForLoopConstruction>

   <RepeatLoopConstruction needBlock="optional"
reversed="true"></RepeatLoopConstruction>

   <OperatorClosers needStamples="optional"
needCloser="true"></OperatorClosers>
  </Constructions>
</ConvertableLanguage>
```

В процессе работы программы создается два объекта описанного класса, представляющие словари исходного и конечного языков.

Текст программы на исходном языке программирования

Текст программы на исходном языке программирования – это текст, который будет мигрирован в текст программы на конечном языке программирования. Предполагается, что данный текст корректен, то есть отвечает стандартам языка, на котором написан, может быть скомпилирован компилятором этого языка, а также, что этот язык

поддерживается мигратором. Исходный текст может быть введен как вручную, так и путем загрузки из файла исходного кода. Текст считывается в виде коллекции строк.

3.4 Синтаксический анализ

Синтаксический анализ (парсинг) — это процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой.

В данной программе, как уже было описано выше, формальная грамматика является частью словаря исходного языка.

На данном этапе происходит анализ коллекции исходных строк и разбиение всего текста на лексемы.

Происходит это с помощью посимвольного перебора и сравнения. Для каждой строки коллекции в цикле со счетчиком происходит копирование каждого символа этой строки в строку-буфер и сравнение с коллекцией разделителей соответствующего объекта класса ConvertableLanguage. Как только обнаруживается, что текущий символ является разделителем, происходит сравнение строки-буфера с коллекцией лексем исходного языка, определение типа данной лексемы и добавление его в коллекцию лексем кода исходной программы.

Пример работы синтаксического анализатора:

```
for (int i = 0; i < sourceFile.Count(); i++)
{
    //создаем переменные, хранящие номера символов
    начала и конца подстроки
    int startLexemIndex = 0;
    int endLexemIndex = 0;
```

```
    <...>
    for (int j = 1; j < sourceFile[i].Length; j++)
    {
        <...>
        //если данный символ содержится в списке разделителей
         if (CheckLexemContaining(sourceFile[i][j], dividers))
        {
                endLexemIndex = j; //конечному индексу лексемы
                присваиваем значение текущего символа
                <...>
                //создаем лексему-буфер на основе полученной
подстроки
                bufLexem =
                FindLexemByName(sourceFile[i].Substring(startLexemIn
                dex, endLexemIndex - startLexemIndex),
                sourceLangLex);
                <...>
                //добавляем лексему-буфер в список лексем
                sourceFileLexem.Add(bufLexem);
                <...>
                //присваиваем лексеме-буферу значение
                разделителя
                bufLexem = FindLexemByName(new
                string(sourceFile[i][j], 1), dividers);
                //добавляем разделитель в список лексем
                sourceFileLexem.Add(bufLexem);
                startLexemIndex = j + 1; //обновляем значение
                начального индекса
                <...>
        }
```

```
    }
}
```

3.5 Миграция

Основной этап работы программы. Он заключается в полексемном переборе полученной на предыдущем этапе коллекции и преобразовании ее членов в лексемы конечного языка. Также на этом этапе происходит конвертирование синтаксических конструкций исходного языка в конструкции конечного языка.

Программно это происходит с помощью цикла foreach, в теле которого выполняется алгоритм преобразования исходных выражений в конечные по определенным правилам, в зависимости от того, чтение какой конструкции происходит в данный момент. Так мигратор может находиться в процессе чтения следующих конструкций исходного языка программирования:

1. объявление и инициализация переменных и констант;
2. объявление и инициализация массивов данных;
3. объявление условных конструкций;
4. объявление циклов со счетчиком;
5. объявление циклов с предусловием и постусловием;
6. объявление подпрограмм, функций и процедур;
7. объявление специфических для конкретного языка конструкций (например, названия программы и блока переменных на Pascal).

Принципы миграции основных синтаксических конструкций

Очевидно, что миграция различных синтаксических конструкций требует алгоритмов различной сложности. В данном разделе приведено несколько алгоритмов, заслуживающих особого внимания.

1. Объявление переменных

Трансляция конструкции объявления переменных является одной из базовых задач любого транслятора и не представляет особой сложности.

Пример трансляции конструкции объявления переменных с языка Pascal:

```
bool bVariableReading = false; //флаг чтения переменной
LexemSubType typeOfCurrentLexem = LexemSubType.any; /*
переменная-буфер, содержащая тип переменной, чтение
которой идет в данный момент */
List<Lexem> bufVariableLexems = new List<Lexem>(); /*
коллекция буферных лексем, необходимая при трансляции
нескольких переменных объявленных в одной конструкции
*/
<...>
//для каждой лексемы в исходной коллекции
foreach (Lexem lex in sourceFileLexem)
{
    <...>
    //если данная лексема является объявлением блока
    переменных
    if (lex.SubType == LexemSubType.VarBlock)
    {
```

```
        bVariableReading = true;
    }

    <...>

    if (bVariableReading)
    {
        //если данная лексема является переменной
        if (lex.Type == LexemType.Variable)
        {
            bufVariableLexems.Add(lex);
        }
        //если данная лексема является объявлением типа
        данных
        else if (lex.SubType == LexemSubType.VarType)
        {
            typeOfCurrentLexem = lex.subType;
        }
        //если данная лексема является символом конца
строки
        else if (lex.SubType == LexemSubType.EndOfLine)
        {
            /*
              Добавление лексем из bufVariableLexems и типа
            переменной в коллекцию
              конечных лексем и очистка буферных
            переменных
            */
            <...>
        }
```

```
        //если данная лексема является объявлением начала
блока
        else if (lex.SubType == LexemSubType.blockStart)
        {
            bVariableReading = false;
            //добавляем лексему в коллекцию конечных
            лексем
            <...>
        }

        continue;
    }
    <...>
}
```

2. Циклы

Циклы - это составные синтаксические конструкции, состоящие из тела и заголовка, в котором содержатся условия выполнения операций, расположенных в теле. В каждом из рассмотренных языков циклы представлены тремя видами конструкций: циклом со счетчиком, циклом с предусловием и циклом с постусловием. Миграция двух последних является довольно тривиальной задачей и особого интереса не представляет. Реализация же конструкций первого вида довольно сильно различается в разных языках, поэтому алгоритм ее трансляции стоит рассмотреть на примере.

Пример трансляции цикла со счетчиком с языка C:

```
bool bForLoopReading = false; //флаг чтения цикла со
счетчиком
List<Lexem> bufForLoopLexems = new List<Lexem>(); /*
```

```
коллекция буферных лексем, необходимых для трансляции
цикла со счетчиком */
<...>
foreach (Lexem lex in sourceFileLexem)
{
    //если данная лексема является первой лексемой
    конструкции цикла со счетчиком
    if (lex.SubType == LexemSubType.algFor)
    {
        bForLoopReading = true;
        continue;
    }

    <...>

    if (bForLoopReading)
    {
        //если данная лексема является переменной
        if (lex.Type == LexemType.Variable)
        {
            Lexem bufLex = lex;
            //если в буферной коллекции уже содержится
лексема-переменная
            if (CheckLexemContaining(LexemSubType.variable,
bufForLoopLexems))
            {
                /* если лексема-переменная имеет одинаковое
                содержимое с данной, просто пропускаем ее (т.к.
                при добавлении лексем в коллекцию конечных
                лексем используется единственный экземпляр
```

```
                каждой лексемы) */
                if (FindLexemWithSubType(LexemSubType.variable,
bufForLoopLexems).Content
                == lex.Content)
                continue;

                /* иначе наша лексема является константой (т.к. по
                правилам языка С в заголовке цикла со счетчиком
                может содержаться только одна переменная) */
                bufLex.SubType = LexemSubType.constant;
            }

            bufForLoopLexems.Add(bufLex);
        }

        //если данная лексема равна последней в определении
        цикла в исходном языке
        if (lex.SubType ==
        sourceLang.ForLoopLexems[sourceLang.ForLoopLexems.Co
        unt - 1].SubType)
        {
            /* добавляем все лексемы из буфера в соответствии с
            необходимой последовательностью */
            foreach (Lexem forLex in requiredLang.ForLoopLexems)
            {
                //если нужна переменная, добавляем ее из
                коллекции-буфера
                if (forLex.SubType == LexemSubType.variable)
                {
                    for (int i = 0; i < bufForLoopLexems.Count; i++)
```

```
            {
                if (bufForLoopLexems[i].SubType ==
                LexemSubType.variable)
                {
                    outputFileLexem.Add(bufForLoopLexems[i]);
                }
                break;
            }
        }

        //аналогично для константы
        else if (forLex.SubType == LexemSubType.constant)
        {
            <...>
        }

        //иначе просто добавляем необходимую лексему
        из списка
        else
        {
            foreach (Lexem rlex in requiredLangLex)
            {
                if (forLex.SubType == rlex.SubType)
                {
                    outputFileLexem.Add(rlex);
                    break;
                }
            }
        }
    }
```

```
            //обнуляем коллекцию буферов и снимаем флаг
            bufForLoopLexems = new List<Lexem>();
            bForLoopReading = false;
        }

        continue;
    }

    <...>

}
```

Коллекция sourceLang.ForLoopLexems заполняется в конструкторе класса ConvertableLanguage лексемами блока <Constructions> исходного xml-файла.

3. Функции

Появление поддержки функционального программирования стало одной из отличительных черт языка Pascal по сравнению с его предшественниками (например, Fortran). Функции также являются неотъемлемой частью языка C, а Basic предоставляет множество возможностей работы с подпрограммами. Функциональное программирование, в общем – мощный инструмент, серьезно облегчающий работу программиста. Поэтому алгоритм трансляции подпрограмм заслуживает особого внимания, несмотря на относительную простоту его реализации.

Пример трансляции функции с языка Pascal на язык C:

```
bool bSubProgHeaderReading = false; //флаг чтения заголовка подпрограммы
```

```
bool bStampleOpened = false; //флаг открытия скобки
Lexem currSubProgName = new Lexem(); //лексема-буфер, содержащая тип данной функции
List<Lexem> bufArguments = new List<Lexem>(); /* коллекция буферных лексем, содержащая лексемы-аргументы */
LexemSubType typeOfCurrentLexem = LexemSubType.any; /* переменная-буфер, содержащая тип переменной, чтение которой идет в данный момент (для аргументов функции) */
int variableCursorPosition = 0; /* переменная, содержащая номер положения, на которое должна быть добавлена переменная из буфера */
<...>
foreach (Lexem lex in sourceFileLexem)
{
    if (lex.SubType == LexemSubType.procedure || lex.SubType
    == LexemSubType.function)
    {
        bSubProgHeaderReading = true;
        continue;
    }

    if (bSubProgHeaderReading)
    {
        //если данная лексема является переменной
        if (lex.Type == LexemType.Variable)
        {
            bufArguments.Add(lex);
            continue;
        }
```

```
//иначе если данная лексема является открывающей скобкой
else if (lex.SubType == LexemSubType.stampleOpenSymbol)
{
    /* добавление в коллекцию конечных лексем названия функции из коллекции bufArgumets (названия интерпретируются как переменные) и других лексем в зависимости от синтаксиса конечного языка */

    <...>

    bStampleOpened = true
    bufArguments.Clear(); //очистка буферной коллекции
    typeOfCurrentLexem = LexemSubType.any; //очистка буфера типов
}

//иначе если данная лексема является закрывающей скобкой
else if (lex.SubType == LexemSubType.stampleCloseSymbol)
{
    /* добавление в коллекцию конечных лексем из коллекции bufArgumets последнего считанного аргумента, его типа из переменной typeOfCurrentLexem и других лексем в зависимости
```

```
        от синтаксиса конечного языка */

        <...>

        bStampleOpened = false
        bufArguments.Clear();
        typeOfCurrentLexem = LexemSubType.any;
    }

    //иначе если данная лексема является типом данных
    else if (lex.Type == LexemType.VarType)
    {
        typeOfCurrentLexem = lex.SubType;
    }
    /* иначе если данная лексема является символом
    «точка с запятой» (разделяет аргументы функции и
    завершает объявление ее заголовка в Pascal */
    else if (lex.SubType == LexemSubType.Semicolon)
    {
        //если скобка была открыта
        if (bStampleOpened)
        {
            /* добавление в коллекцию конечных лексем из
            коллекции bufArgumets последнего считанного
            аргумента, его типа из переменной
            typeOfCurrentLexem и других лексем в
            зависимости от синтаксиса конечного языка */

            <...>
```

```
                bufArguments.Clear();
                typeOfCurrentLexem = LexemSubType.any;
            }
            else
            {
                /* вставка в коллекцию конечных лексем на
                позицию variableCursorPosition типа данной
                функции из переменной typeOfCurrentLexem и
                других лексем в зависимости от синтаксиса
                конечного языка */

                bSubProgHeaderReading = false;
                variableCursorPosition = 0;
            }
        }

        continue;
    }
}
```

Трансляция подпрограмм и процедур происходит по схожему алгоритму. Стоит отметить, что трансляция с языка С несколько сложнее, но в ее основе лежит тот же принцип.

3.6 Выходные данные

На финальном этапе работы программа формирует из полученной коллекции лексем коллекцию строк, которая отображается в специальном текстовом поле на форме пользователя. Также эта коллекция может сохраняться в файл исходного кода конечного языка программирования.

4 Возможности мигратора

Работа по созданию мигратора не закончена на сто процентов, поэтому некоторые конструкции языков программирования могут пока не поддерживаться. Также многие алгоритмы, используемые в программе, могут не являться наиболее оптимальными.

4.1 Что мигратор уже умеет

На данном этапе мигратор способен распознавать и транслировать следующие конструкции:

1. определение и инициализация переменных и массивов;
2. условный переход (алгоритмическое ветвление), полный (if...then…else) и неполный (if...then);
3. циклы со счетчиком, с предусловием и постусловием;
4. определение и вызов подпрограмм, функций и процедур, с аргументами и без них.

4.2 Что мигратор пока не умеет

В перспективе мигратор будет способен распознавать и транслировать:

1. Указатели на ячейки памяти. Это мощное средство управления памятью, занимаемой программой во время исполнения, позволяющее программисту ссылаться не на значение переменной, а на ее адрес. Алгоритм его миграции является одним из наиболее сложных, так как реализация указателей сильно различается в разных языках. Реализация этого алгоритма позволит реализовать алгоритм трансляции

и другого мощного механизма программирования – динамических массивов.

2. Пространства имен. Они позволяют задавать области видимости переменных для различных блоков программы. Для миграции этого механизма требуется реализация алгоритма контекстного распознавания переменных и их отнесения к определенным блокам программы. Также это откроет возможности для оптимизации кода исходной программы.
3. Типы данных, определяемые пользователем. Включают в себя структуры данных, перечисления и объединения (для C). Являются весьма удобным средством унификации данных, важным, в первую очередь, для абстрактного понимания работы программы программистом.
4. Импорт библиотек из внешних файлов. Важный механизм, служащий для создания объемных проектов. Позволяет объединять в единый исполнимый файл исходные коды, содержащиеся в разных файлах, например, динамических или статических библиотеках. Также необходим для работы со стандартными интерфейсами программирования операционных систем.
5. Некоторые команды препроцессоров. Позволяют управлять процессом компиляции. Необходим, прежде всего, для миграции с языка C. Миграция этих команд осложняется слабой их поддержкой в компиляторах языков Pascal и C, а также их различиями в разных версиях компиляторов.

4.3 Перспективы

Дальнейшее совершенствование мигратора может привести к созданию механизма, способного оптимизировать конструкции получаемого кода. «Научив» программу вариативно понимать логику исходного алгоритма мы получим возможность расширить функционал мигратора по применению его и к другим языкам программирования. Как перспективную идею можно рассматривать создание универсального мигратора, способного, обладая многими словарями, оперировать любыми алгоритмами, даже пока не созданными.

Приложение 1

Примеры миграции простых алгоритмических конструкций

1. Ввод-вывод

Данная программа является демонстрацией миграции простейших конструкций определения переменных, а также функций ввода-вывода. Она запрашивает у пользователя имя и выводит строку «Hello, введенное_имя».

Pascal	C	Basic

program hello_name;
var forename:**string**;
begin
writeln('Enter your name:');
readln(forename);
writeln('Hello, ', forename**);**
end.
	#include<stdio.h>
	void main()
	{
	char* forename;
	printf("Enter your name:");
	scanf("%s", forename);
	printf("Hello, %s", forename);
	}
		Dim As String forename
		Print "Enter your name:"
		Input forename
		Print "Hello " ; forename

2. Условный переход

Данный пример демонстрирует миграцию условной конструкции. Программа считывает число и возвращает его модуль.

Pascal	C	Basic

program program_1;
var
a:**integer**;
begin
write('Enter a number to get a module:');
read(a);
if a < 0 **then**
begin
write(-a);
end
else
begin
write(a);
end;
end.
	#include<stdio.h>
	void main()
	{
	int a;
	printf("Enter a number to get a module:");
	scanf("%i", &a);
	if (a < 0)
	{
	printf("%i", -a);
	}
	else
	{
	printf("%i", a);
	}
	}
		Dim a **As Integer**
		Print "Enter a number to get a module**:"**
		Input a
		If a **<** 0 **Then**
		Print -a
		Else
		Print a
		End **If**

3. Цикл со счетчиком

На примере этой программы показывается возможность миграции цикла For...to...do. Программа запрашивает число и выводит факториал этого числа.

Pascal	C	Basic

program program_1;
var
i:**integer**; n:**integer**; s:**integer**;
begin
s **:=** 1;
write('Enter a number to get a factorial:');
read(n);
for i **:=** 1 **to** n **do**
begin
s **:=** s*i;
end;
write('',s);
end.
	#include<stdio.h>
	void main()
	{
	int n,s;
	s **=** 1;
	printf("Enter a number to get a factorial:")**;**
	scanf(n)**;**
	int i**;**
	for (i = 1; i <= n; i++**)**
	{
	s = s*i**;**
	}
	printf("%d",**s**);
	}
		Dim As Integer i
		Dim As Integer n
		Dim As Integer s
		s **=** 1
		Print "Enter a number to get a factorial:"
		Input n
		For i **=** 1 **To** n
		s **=** s*i
		Next i
		Print "" ; s

4. Цикл с предусловием

Данная программа является примером миграции массива, цикла While...do, а также возможность миграции вложенных циклов. В ней происходит сортировка вставками массива из 5 элементов, вводимого с клавиатуры.

Pascal	C	Basic

program sort;
var arr:**array**[1..5] of **integer;**
i,j,key:**integer;**
begin
writeln('Enter 5 elements:');
for i := 1 **to** 5 **do**
begin
read(arr[i]);
end;
j := 0;
i := 0;
for i := 2 **to** 5 **do**
begin
key := arr[i];
	#include<stdio.h>
	void main()
	{
	int arr[6]; **int** i; **int** j; **int** key;
	printf("Enter 5 elements:");
	for (i = 1; i <= 5; i++)
	{
	scanf("%i", &arr[i]);
	}
	j = 0;
	i = 0;
	for (i = 2; i <= 5; i++)
	{
	key = arr[i];
		Dim As Integer i
		Dim As Integer j
		Dim As Integer key
		Dim arr(1 **To** 5) **As Integer**
		Print "Enter 5 elements:"
		For i = 1 **To** 5
		Input arr(i)
		Next i
		j = 0
		i = 0
		For i = 2 **To** 5
		key = arr(i)
		j = i - 1
		While (j > 0) **And** (arr(j) > key)

```
j := i - 1;
while (j > 0) and (arr[j] > key)
do
begin
arr[j + 1] := arr[j];
j := j - 1;
end;
arr[j + 1] := key;
end;
writeln('Sorted array:');
for i := 1 to 5 do
begin
write(arr[i]);
end;
end.
```

```
j = i - 1;
while ( (j > 0) && (arr[j] > key) )
{
arr[j + 1] = arr[j];
j = j - 1;
}
arr[j + 1] = key;
}
printf("Sorted :");
for (i = 1; i <= 5; i++)
{
printf("%i", arr[i]);
}
}
```

```
arr(j + 1) = arr(j)
j = j - 1
Wend
arr(j + 1) = key
Next i
Print "Sorted :"
For i = 1 To 5
Print arr(i)
Next i
```

5. Функции

Этот пример демонстрирует возможность миграции подпрограмм (функций). В ней происходит рекурсивное вычисление первых 20 членов последовательности Фибоначчи (последовательности, в которой каждый последующий член равен сумме предыдущих).

Pascal	C	Basic

program program_1;
var
i:**integer**;
function
fibonacci(n:**integer**):**integer**;
var
fib:**integer**;
begin
if (n > 1) **then**
begin
fib **:=** fibonacci(n-)+fibonacci(n-2);
fibonacci **:=** fib;
end
	#include<stdio.h>
	int fibonacci(**int** n)
	{
	if (n > 1)
	{
	int fib;
	fib **=** fibonacci(n-1)+fibonacci(n-2);
	return fib;
	}
	else
	{
	return 1;
	}
		Function fibonacci(n **As Integer**)
		As Integer
		Dim As Integer fib
		If (n > 1) **Then**
		fib **=** fibonacci(n-1)+fibonacci(n-2)
		Return fib
		Else
		Return 1
		End If
		End Function
		Dim As Integer i
		Print "First 20 numbers of Fibonacci sequence: "

```
else
begin
fibonacci := 1;
end;
end;
begin
write('First 20 numbers of
Fibonacci sequence: ');
for i := 1 to 21 do
begin
write(' ',fibonacci(i));
end;
end.
```

```
}
void main()
{
printf("First 20 numbers of
Fibonacci sequence: ");
int i;
for (i = 1; i < 21; i++)
{
printf("%d ",fibonacci(i));
}
}
```

```
For i = 1 To 21
Print " " ; fibonacci(i )
Next i
```

Приложение 2

Листинг программы

ConvertableLanguage.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml;
using System.IO;
using System.Windows.Forms;

namespace ProgrammingLanguagesConverter
{
   //перечисление типов лексем
   enum LexemType
   {
      Divider, //разделители
      Keyword, //ключевые слова
      Variable, //переменные
      VarType //объявление типа
   }

   //перечисление подтипов лексем
   enum LexemSubType
   {
      any, //любой

      variable, //переменная
            constant, //константа
```

```
    //***************указатели на
разделители*****************************//
    endOfLine, //указатель конца строки
    spaceSymbol, //указатель разделения слов
    stampleOpenSymbol, //указатель открытия скобок
    stampleCloseSymbol, //указатель закрытия скобок
    squareStampleOpen,
    squareStampleClose,
    colonSymbol, //двоеточие
        commaSymbol, //запятая
    arrayRange, //символ, указывающий границы массива
        lineComment, //строчный комментарий
        blockCommentOpen,
        blockCommentClose, //блочный комментарий
    stringOpen,
    charOpen,
        programEnd, //указатель конца программы
        mathPlus, //плюс
        mathMinus, //минус
        mathInc, //инкремент
        mathDec, //декремент
        mathMultiply, //умножение
        mathDivide, //деление
        mathEqual, //равно
        mathUnequal, //неравно
        mathMore, //больше
        mathLess, //меньше
        carriageReturn, //возврат каретки
```

```
	//***************указатели на блоки*************************************//
	programStart, //указатель начала программы
	varBlock, //указатель блока переменных
	constDef,
	blockStart, //указатель начала блока
	blockEnd, //указатель конца блока
	procedure,
	function,

	//***************указатели на типы переменных**************************//
		typeVoid,
	type8sInt, //указатель на целую 8-битную переменную со знаком
	type8usInt, //указатель на целую 8-битную переменную без знака
	type16sInt, //указатель на целую 16-битную переменную со знаком
	type16usInt, //указатель на целую 16-битную переменную без знака
	type32sInt, //указатель на целую 32-битную переменную со знаком
	type32usInt, //указатель на целую 32-битную переменную без знака
	type64sInt, //указатель на целую 64-битную переменную со знаком
	type64usInt, //указатель на целую 64-битную переменную без знака
	typeChar, //указатель на символьную переменную
```

```
	type32Float, //указатель на 32-битную с плавующей точкой
	type64Float, //указатель на 64-битную с плавующей точкой
		typeString,

	//***************указатели направления алгоритма**********************//
	algFor,
	algTo,
	algDo,
	algNext,
	algIf,
	algThen,
	algElse,
	algEnd,
	algWhile,
	algRepeat,
	algUntil,
	algLoop,
	algWend,
	algDownto,
	algSwitch,
	algCase,
	algOf,
	arrayType,

	//***************указатели на логические операции*******************//
	logicAnd, //и
```

```
        logicOr, //или
        logicNot, //не
                logicAssign, //присвоить
        logicAs,
        preProc,

        read,
        readln,
        write,
        writeln,

                returnWord //возвращение значения функции
    }

    //структура, описывающая лексему
    struct Lexem
    {
        public string Content; //сама лексема
        public LexemType Type; //ее тип
        public LexemSubType SubType; //подтип лексемы
        public string Ambiguousity; //однозначность

        public Lexem(string content, LexemType type,
LexemSubType subType)
        {
            this.Content = content;
            this.Type = type;
            this.SubType = subType;
                    this.Ambiguousity = null;
        }
```

```
            public Lexem(char content, LexemType type,
LexemSubType subType)
      {
         this.Content = "";
         this.Content += content;
         this.Type = type;
         this.SubType = subType;
                  this.Ambiguousity = null;
      }

      public void SetType(LexemType lt)
      {
         this.Type = lt;
      }

      public void SetSubType(LexemSubType lst)
      {
         this.SubType = lst;
      }
   }

       //класс конвертируемого языка
   class ConvertableLanguage
   {
          public string Name { get; private set; } //название

          public List<Lexem> MainLexems { get; private set; }
//основные лексемы
            public List<Lexem> ForLoopLexems { get; private set; }
```

```
//последовательность объявления цикла со счетчиком

            public bool BVarDefIsType { get; private set; }
        public bool BGapInArr { get; private set; }
            public bool BStrictlyStamples { get; private set; }
            public bool BNeedCloser { get; private set; }
        public bool BStrictlyBlockInRepeat { get; private set; }
        public bool BInversedRepeat { get; private set; }

            public ConvertableLanguage()
            {
                MainLexems = new List<Lexem>();
                ForLoopLexems = new List<Lexem>();
            }

            //функция загрузки лексем из файла
            public void LoadLexemsFromFile(string path)
            {
                var xmlTr = new XmlTextReader(new
FileStream(path, FileMode.Open));

                bool bReadingLexems = false; //флаг чтения
лексем
                bool bReadingForLoop = false; //флаг чтения
цикла со счетчиком
                bool bReadingName = false; //флаг чтения имени
                var bufLexem = new Lexem(); //лексема-буфер
                var readingLexemType = new LexemType();
//буфер чтения типа лексемы
```

```
        MainLexems.Clear();
        ForLoopLexems.Clear();

                //пока не кончился файл
                while (xmlTr.Read())
                {
                    //проверяем тип ноды
                    switch (xmlTr.NodeType)
                    {
                        //если элемент
                        case XmlNodeType.Element:
                            //проверяем имя
                            switch (xmlTr.Name)
                            {
                                //ставим соответствующий
флаг
                                case "Lexems":
                                    bReadingLexems = true;
                                    break;
                                case "VariableDefinition":
                    if (xmlTr.GetAttribute("type") == "type")
                        BVarDefIsType = true;
                    else
                        BVarDefIsType = false;
                                    break;
                     case "ArrayDefinition":
                        if (xmlTr.GetAttribute("gap") == "true")
                            BGapInArr = true;
                        else
                            BGapInArr = false;
```

```
                    break;
                                    case "OperatorClosers":
                    if (xmlTr.GetAttribute("needStamples") ==
"strictly")
                        BStrictlyStamples = true;
                    else
                        BStrictlyStamples = false;

                    if (xmlTr.GetAttribute("needCloser") == "true")
                        BNeedCloser = true;
                    else
                        BNeedCloser = false;
                                            break;
                case "RepeatLoopConstruction":
                    if (xmlTr.GetAttribute("needBlock") ==
"strictly")
                        BStrictlyBlockInRepeat = true;
                    else
                        BStrictlyBlockInRepeat = false;

                    if (xmlTr.GetAttribute("reversed") == "true")
                        BInversedRepeat = true;
                    else
                        BInversedRepeat = false;
                    break;
                                    case "ForLoopConstruction":
                                        bReadingForLoop = true;
                                        break;
                                    case "Dividers":
                                        readingLexemType =
```

```
LexemType.Divider;
                                        break;
                                    case "Keywords":
                                        readingLexemType =
LexemType.Keyword;
                                        break;
                                    case "VarTypes":
                                        readingLexemType =
LexemType.VarType;
                                        break;
                                    case "Name":
                                        bReadingName = true;
                                        break;
                                    default:
                                        //если уже идет чтение,
выполняем соответсвующие действия
                                        if (bReadingLexems)
                                        {
                                            bufLexem = new
Lexem {SubType = GetLexemSubTypeFromXml(xmlTr.Name),
Type = readingLexemType};
                                            if
(xmlTr.GetAttribute("ambiguous") != null)
                                            {

      bufLexem.Ambiguousity = xmlTr.GetAttribute("ambiguous");
                                            }
                                        }
                                        else if (bReadingForLoop)
                                        {
```

```
                                                bufLexem = new
Lexem(null, readingLexemType,
GetLexemSubTypeFromXml(xmlTr.Name));

        ForLoopLexems.Add(bufLexem);
                                                }
                                                break;
                                        }
                                        break;
                                case XmlNodeType.Text:
                                        if (bReadingLexems)
                                        {
                                                bufLexem.Content =
xmlTr.Value;
                                                MainLexems.Add(bufLexem);
                                        }
                                        else if (bReadingName)
                                        {
                                                Name = xmlTr.Value;
                                                bReadingName = false;
                                        }

                        if (xmlTr.Value == "\n")
                        {
                           MainLexems.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.endOfLine));
                        }
                                        break;
                                case XmlNodeType.EndElement:
                                        if (xmlTr.Name == "Lexems")
```

```
                                {
                                    bReadingLexems = false;
                                }
                                else if (xmlTr.Name ==
"ForLoopConstruction")
                                {
                                    bReadingForLoop = false;
                                }
                                break;
                            }
                        }

                        xmlTr.Close();
                    }

                    private LexemSubType
GetLexemSubTypeFromXml(string xmlName)
                    {
                        LexemSubType subtype = LexemSubType.any;

                        switch (xmlName)
                        {
                            case "Variable":
                                subtype = LexemSubType.variable;
                                break;
                            case "Constant":
                                subtype = LexemSubType.constant;
                                break;

                            //Dividers
```

```
                    case "EndOfLine":
                        subtype = LexemSubType.endOfLine;
                        break;
                    case "SpaceSymbol":
                        subtype = LexemSubType.spaceSymbol;
                        break;
                    case "StampleOpenSymbol":
                        subtype =
LexemSubType.stampleOpenSymbol;
                        break;
                    case "StampleCloseSymbol":
                        subtype =
LexemSubType.stampleCloseSymbol;
                        break;
            case "SquareStampleOpen":
               subtype = LexemSubType.squareStampleOpen;
               break;
            case "SquareStampleClose":
               subtype = LexemSubType.squareStampleClose;
               break;
                    case "ColonSymbol":
                        subtype = LexemSubType.colonSymbol;
                        break;
                    case "CommaSymbol":
                        subtype = LexemSubType.commaSymbol;
                        break;
            case "ArrayRange":
               subtype = LexemSubType.arrayRange;
               break;
                    case "MathPlus":
```

```
                    subtype = LexemSubType.mathPlus;
                    break;
            case "MathMinus":
                subtype = LexemSubType.mathMinus;
                break;
                    case "MathEqual":
                    subtype = LexemSubType.mathEqual;
                    break;
            case "MathUnequal":
                subtype = LexemSubType.mathUnequal;
                break;
                    case "LogicAssign":
                    subtype = LexemSubType.logicAssign;
                    break;
            case "LogicAs":
                subtype = LexemSubType.logicAs;
                break;
                    case "LineComment":
                    subtype = LexemSubType.lineComment;
                    break;
            case "BlockCommentOpen":
                subtype = LexemSubType.blockCommentOpen;
                break;
            case "BlockCommentClose":
                subtype = LexemSubType.blockCommentClose;
                break;
            case "StringOpen":
                subtype = LexemSubType.stringOpen;
                break;
            case "CharOpen":
```

```
            subtype = LexemSubType.charOpen;
            break;
                    case "MathLess":
                        subtype = LexemSubType.mathLess;
                        break;
                    case "MathMore":
                        subtype = LexemSubType.mathMore;
                        break;
                    case "MathInc":
                        subtype = LexemSubType.mathInc;
                        break;
        case "PreProc":
            subtype = LexemSubType.preProc;
            break;

                    //Keywords
                    case "ProgramStart":
                        subtype = LexemSubType.programStart;
                        break;
                    case "ProgramEnd":
                        subtype = LexemSubType.programEnd;
                        break;
                    case "VarBlock":
                        subtype = LexemSubType.varBlock;
                        break;
        case "ConstDef":
            subtype = LexemSubType.constDef;
            break;
                    case "BlockStart":
                        subtype = LexemSubType.blockStart;
```

```
                    break;
                case "BlockEnd":
                    subtype = LexemSubType.blockEnd;
                    break;
                case "AlgIf":
                    subtype = LexemSubType.algIf;
                    break;
                case "AlgThen":
                    subtype = LexemSubType.algThen;
                    break;
                case "AlgElse":
                    subtype = LexemSubType.algElse;
                    break;
            case "AlgEnd":
               subtype = LexemSubType.algEnd;
               break;
                case "AlgFor":
                    subtype = LexemSubType.algFor;
                    break;
                case "AlgTo":
                    subtype = LexemSubType.algTo;
                    break;
                case "AlgDo":
                    subtype = LexemSubType.algDo;
                    break;
            case "AlgNext":
               subtype = LexemSubType.algNext;
               break;
            case "AlgWhile":
               subtype = LexemSubType.algWhile;
```

```
            break;
        case "AlgRepeat":
            subtype = LexemSubType.algRepeat;
            break;
        case "AlgUntil":
            subtype = LexemSubType.algUntil;
            break;
        case "AlgLoop":
            subtype = LexemSubType.algLoop;
            break;
        case "AlgWend":
            subtype = LexemSubType.algWend;
            break;
        case "Procedure":
            subtype = LexemSubType.procedure;
            break;
        case "Function":
            subtype = LexemSubType.function;
            break;
        case "ReturnWord":
            subtype = LexemSubType.returnWord;
            break;
                    case "LogicAnd":
                        subtype = LexemSubType.logicAnd;
                        break;
        case "ArrayType":
            subtype = LexemSubType.arrayType;
            break;
        case "AlgOf":
            subtype = LexemSubType.algOf;
```

```
            break;
        case "Read":
            subtype = LexemSubType.read;
            break;
        case "ReadLine":
            subtype = LexemSubType.readln;
            break;
        case "Write":
            subtype = LexemSubType.write;
            break;
        case "WriteLine":
            subtype = LexemSubType.writeln;
            break;

                    //VarTypes
        case "TypeVoid":
            subtype = LexemSubType.typeVoid;
            break;
                    case "Type32sInt":
                        subtype = LexemSubType.type32sInt;
                        break;
                    case "TypeChar":
                        subtype = LexemSubType.typeChar;
                        break;
                    case "Type32Float":
                        subtype = LexemSubType.type32Float;
                        break;
                    case "TypeString":
                        subtype = LexemSubType.typeString;
                        break;
```

```
                }

                return subtype;
            }

            public void ShowLexemsInConsole()
            {
                string lexemInfo = null;

                lexemInfo += Name + "\n\n";
                lexemInfo += "******Main******\n";

                foreach (Lexem lex in MainLexems)
                {
                    lexemInfo += lex.Content + " -content " +
lex.SubType.ToString() + " -subType " + lex.Type + " -type " +
lex.Ambiguousity + " -amb\n";
                }

                lexemInfo += "\n******ForLoop******\n";

                foreach (Lexem lex in ForLoopLexems)
                {
                    lexemInfo += lex.Content + " -content " +
lex.SubType.ToString() + " -subType " + lex.Type + " -type " +
lex.Ambiguousity + " -amb\n";
                }

                MessageBox.Show(lexemInfo);
            }
```

```
    }
}
```

LanguageConvertation.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Windows.Forms;
using System.Collections;

namespace ProgrammingLanguagesConverter
{
    //класс непосредственной конвертации
    static class LanguageConvertation
    {
        //функция, разбивающая входной текст на лексемы
        public static List<Lexem> SyntaxAnalyzer(List<Lexem>
sourceLangLex, List<string> sourceFile)
        {
            int startLexemIndex = 0;
            int endLexemIndex = 0; //индексаторы начала и конца
лексемы
                    string ambBufStr = null; //вспомогательная
строка для определения однозначности лексемы
            List<Lexem> dividers = new List<Lexem>(); //коллекция
разделителей
            List<Lexem> sourceFileLexem = new List<Lexem>();
//коллекция лексем входного файла
```

```
                //добавляем разделители из исходного языка в
коллекцию
        foreach (Lexem lex in sourceLangLex)
        {
            if (lex.Type == LexemType.Divider)
            {
                dividers.Add(lex);
            }
        }
                dividers.Add(new Lexem("\n", LexemType.Divider,
LexemSubType.carriageReturn));

                //проверяем каждую строку входного файла
        for (int i = 0; i < sourceFile.Count(); i++)
        {
            //добавляем символ перехода на новую строку
            sourceFile[i] += "\n";

            //в начале каждой строки присваиваем конечному и
начальному индексу нулевое значение
            if (sourceFile[i].Length > 0 && sourceFile[i][0] == ' ')
                startLexemIndex = 1;
            else
                startLexemIndex = 0;
            endLexemIndex = 0;

            //проверяем строку посимвольно, пока не встретим
разделитель
            for (int j = 0; j < sourceFile[i].Length; j++)
```

```
            {
                //избавляемся от лишних пробелов
                                if (j > 0 && sourceFile[i][j] == ' ' &&
sourceFile[i][j - 1] == ' ')
                {
                    startLexemIndex = j + 1;
                    continue;
                }

                                //если данный символ содержится в
списке разделителей
                                if (CheckLexemContaining(sourceFile[i][j],
dividers))
                {
                    Lexem bufLexem = new Lexem(); //создаем
лексему-буфер
                    endLexemIndex = j; //конечному индексу лексемы
присваиваем значение текущего символа

                    //если данный символ не равен начальному
индексу, добавляем новую лексему в коллекцию, иначе эта
лексема и является разделителем
                    if (j != startLexemIndex)
                    {
                                                //создаем лексему-буфер на
основе полученной подстроки
                                                bufLexem =
FindLexemByName(sourceFile[i].Substring(startLexemIndex,
endLexemIndex - startLexemIndex), sourceLangLex);
```

```
                                        //проверяем буфер
однозначности лексемы;
                                        //если не пуст, ищем в списке
лексем комбинированную,
                                        //иначе добавляем только
данную лексему
                                        if (ambBufStr != null)
                                        {
                                            Lexem ambBufLex =
FindLexemByName(ambBufStr + bufLexem.Content,
sourceLangLex);

                                            if
(ambBufLex.Ambiguousity == "double")
                                            {

    sourceFileLexem.Add(ambBufLex);
                                            }
                                            else
                                            {

    sourceFileLexem.Add(FindLexemByName(ambBufStr,
sourceLangLex));

    sourceFileLexem.Add(bufLexem);
                                            }

                                            ambBufStr = null;
                                        }
                                        else
```

```
                                    {

        sourceFileLexem.Add(bufLexem);
                                    }
                }

                //добавляем разделитель в список лексем
                                    bufLexem = FindLexemByName(new
string(sourceFile[i][j], 1), dividers);

                                    //проверяем однозначность
разделителя
                                    if (bufLexem.Ambiguousity ==
"single")
                                    {
                                        ambBufStr =
bufLexem.Content;
                                    }
                                    else if (ambBufStr != null)
                                    {
                                        Lexem ambBufLex =
FindLexemByName(ambBufStr + bufLexem.Content,
sourceLangLex);

                                        if (ambBufLex.Ambiguousity ==
"double")
                                        {

        sourceFileLexem.Add(ambBufLex);
                                        }
```

```
                                    else
                                    {

        sourceFileLexem.Add(FindLexemByName(ambBufStr,
sourceLangLex));

        sourceFileLexem.Add(bufLexem);
                                    }

                                        ambBufStr = null;
                                }
                                else
                                {
                    if (bufLexem.SubType ==
LexemSubType.stringOpen)
                    {
                       bufLexem.Content = "";
                    }

        sourceFileLexem.Add(bufLexem);
                                }

                    startLexemIndex = endLexemIndex = j + 1;
//перебрасываем индексаторы на следующий символ
                }
            }
        }

        return sourceFileLexem;
    }
```

```
            //функция непосредственного перевода коллекции лексем на исходном языке в коллекцию на необходимом
            public static List<Lexem> LexemTranslate(List<Lexem> sourceFileLexem, ConvertableLanguage sourceLang, ConvertableLanguage requiredLang)
            {
                List<Lexem> outputFileLexem = new List<Lexem>(); //коллекция выходных лексем

                List<Lexem> requiredLangLex = requiredLang.MainLexems; //коллекция лексем исходного языка

                bool bVariableReading = false; //флаг чтения переменной
                int variableCursorPosition = 0;
                LexemSubType typeOfCurrentLexem = LexemSubType.any;
                List<Lexem> bufVariableLexems = new List<Lexem>();
        List<Lexem> bufGlobalVariableLexems = new List<Lexem>();
        int cursorPositionInProc = 0;

        bool bConstReading = false;
            bool bConstValueReading = false;
        List<Lexem> bufConstValueLexems = new List<Lexem>();

        bool bNotTranslate = false;
```

```
        List<Lexem> bufNotTranslatedLexems = new
List<Lexem>();

        bool bArrayReading = false;
        bool bCommaInArrayRead = false;
        bool bVarSwitchedToArr = false;
        int arrayCursorPosition = 0;
        int squareStamplesCounter = 0;
            int stamplesOpenCounter = 0;
        int stamplesCloseCounter = 0;
        List<Lexem> bufArrayConstLexems = new List<Lexem>();

                bool bIfStateReading = false; //флаг чтения
условной конструкции
        int cursorIfElse = 0;

                bool bForLoopReading = false; //флаг чтения
цикла со счетчиком
                List<Lexem> bufForLoopLexems = new
List<Lexem>();
        Stack<Lexem> bufBasicForLoopLexems = new
Stack<Lexem>();
        int wasReadForLoopCount = 0;

                bool bWhileLoopReading = false;

        bool bRepeatLoopReading = false;
        bool bRepeatDefenitonReading = false;
        bool bCarriageReturnRead = false;
```

```
        bool bDoRead = false;

        string basicLoopRotation = "";

        bool bNameRead = false;

        LexemSubType subProgRead = LexemSubType.any;
        int blockOpenCloseCounter = 0;

        bool bSubProgHeaderReading = false;
              Lexem currSubProgName = new Lexem();

        bool bMainConstructionReading = false;
        bool bMainConstructionInserted = false;

        bool bInputReading = false;
        bool bOutputReading = false;
        bool bQuotesRead = false;
        bool bEscapeRead = false;
        int quoteInIOCursor = 0;

        bool bPreProcReading = false;

        List<Lexem> allVars = new List<Lexem>();

                    //проверяем каждую лексему в исходной
коллекции
                    foreach (Lexem lex in sourceFileLexem)
                    {
                        //если данная лексема является лексемой
```

```
перехода на следующую строку переходим на следующую
итерацию
                    if (lex.Content == "\n" && sourceLang.Name
!= "Basic")
            {
               if (bPreProcReading)
               {
                  bPreProcReading = false;
               }
                           outputFileLexem.Add(lex);
                           continue;
                    }

            if (cursorIfElse > 0)
            {
               if (lex.SubType == LexemSubType.algElse)
               {
                  outputFileLexem.Insert(cursorIfElse,
FindLexemWithSubType(LexemSubType.algElse,
requiredLangLex));
                  cursorIfElse = -1;
                  basicLoopRotation += "i";
                  continue;
               }
               else
               {
                  outputFileLexem.Insert(cursorIfElse,
FindLexemWithSubType(LexemSubType.algEnd,
requiredLangLex));
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.alg
If, requiredLangLex));
                cursorIfElse = 0;
            }
        }

                    //считаем количество открытых скобок
        if (lex.SubType == LexemSubType.squareStampleOpen)
                    {
                        squareStamplesCounter++;
                    }
        else if (lex.SubType ==
LexemSubType.stampleOpenSymbol)
        {
            stamplesOpenCounter++;
        }
        else if (lex.SubType ==
LexemSubType.stampleCloseSymbol)
        {
            stamplesCloseCounter++;
        }
        else if (lex.SubType == LexemSubType.blockStart)
        {
            blockOpenCloseCounter++;
        }
        else if (lex.SubType == LexemSubType.blockEnd)
        {
```

```
                blockOpenCloseCounter--;
            }

            if (bMainConstructionReading)
            {
                if (lex.SubType == LexemSubType.blockStart)
                {
                    bMainConstructionReading = false;
                    bVariableReading = false;

outputFileLexem.Add(FindLexemWithSubType(lex.SubType,
requiredLangLex));
                }
                else if (lex.SubType == LexemSubType.endOfLine)
                {
                    bMainConstructionReading = false;
                }

                continue;
            }
            else if (bPreProcReading)
            {
                continue;
            }

            if (sourceLang.Name == "Basic" &&
!bMainConstructionInserted)
            {
                if (requiredLang.Name == "C")
                {
```

```
                if (lex.SubType != LexemSubType.function &&
lex.SubType != LexemSubType.procedure)
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.ty
peVoid, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.pr
ogramStart, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleOpenSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleCloseSymbol, requiredLangLex));
                    outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockStart, requiredLangLex));
                    outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
                    if (bufGlobalVariableLexems.Count > 0)
                    {
                       foreach (Lexem blex in
bufGlobalVariableLexems)
                       {
```

```
outputFileLexem.Add(FindLexemWithSubType(blex.SubType,
requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                        outputFileLexem.Add(new
Lexem(blex.Content, LexemType.Variable,
LexemSubType.variable));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.en
dOfLine, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Insert(variableCursorPosition, new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
                    }
                    bufGlobalVariableLexems.Clear();
                }
                cursorPositionInProc = outputFileLexem.Count;
                bMainConstructionInserted = true;
            }
        }
        else if (requiredLang.Name == "Pascal")
        {
            if (lex.SubType != LexemSubType.varBlock &&
lex.SubType != LexemSubType.procedure)
            {
```

```
                    outputFileLexem.Insert(variableCursorPosition,
new Lexem("\n", LexemType.Divider,
LexemSubType.carriageReturn));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockStart, requiredLangLex));
                    outputFileLexem.Insert(variableCursorPosition,
new Lexem("\n", LexemType.Divider,
LexemSubType.carriageReturn));
                    bMainConstructionInserted = true;
                }
            }
        }

                    //если данная лексема является
переменной
                    if (lex.Type == LexemType.Variable)
                    {
            Lexem blex = lex;

            if (typeOfCurrentLexem != LexemSubType.any)
            {
                blex.SubType = typeOfCurrentLexem;
            }
            allVars.Add(blex);

                        if (bVariableReading)
                        {
                            if (!bConstValueReading)
                            {
```

```
                    if (subProgRead == LexemSubType.any &&
sourceLang.Name == "Pascal" && requiredLang.Name == "C")
                        bufGlobalVariableLexems.Add(lex);
                    else
                        bufVariableLexems.Add(lex);
                                }
                            }
                else if (bArrayReading)
                {
                    if (subProgRead == LexemSubType.any &&
sourceLang.Name == "Pascal" && requiredLang.Name == "C" &&
!bVarSwitchedToArr)
                    {
                        bVarSwitchedToArr = true;
                        for (int i = 0; i < bufGlobalVariableLexems.Count;
i++)
                        {
                            Lexem plex = bufGlobalVariableLexems[i];
                            plex.Content = "<" +
bufGlobalVariableLexems[i].Content;
                            bufGlobalVariableLexems.RemoveAt(i);
                            bufGlobalVariableLexems.Insert(i, plex);
                        }
                    }

                    if (lex.SubType == LexemSubType.constant)
                    {
                        if (bufGlobalVariableLexems.Count > 0)
                        {
                            for (int i = 0; i <
```

```
bufGlobalVariableLexems.Count; i++)
                    {
                        if (bufGlobalVariableLexems[i].Content[0] ==
'<')
                        {
                            if
(!Char.IsDigit(bufGlobalVariableLexems[i].Content[1]))
                            {
                                Lexem plex =
bufGlobalVariableLexems[i];
                                plex.Content = plex.Content.Insert(1,
lex.Content);
                                bufGlobalVariableLexems.RemoveAt(i);
                                bufGlobalVariableLexems.Insert(i, plex);
                            }
                            else if
(!Char.IsDigit(bufGlobalVariableLexems[i].Content[2]))
                            {
                                Lexem plex =
bufGlobalVariableLexems[i];
                                plex.Content = plex.Content.Insert(2,
lex.Content);
                                bufGlobalVariableLexems.RemoveAt(i);
                                bufGlobalVariableLexems.Insert(i, plex);
                            }
                        }
                    }
                }

                if (sourceLang.BGapInArr &&
```

```
!requiredLang.BGapInArr && bufArrayConstLexems.Count > 0)
                    {
                        int bufCont =
System.Convert.ToInt32(lex.Content) -
System.Convert.ToInt32(bufArrayConstLexems[0].Content) + 1;
                        bufArrayConstLexems.Clear();
                        bufArrayConstLexems.Add(new
Lexem(bufCont.ToString(), LexemType.Variable,
LexemSubType.constant));
                        continue;
                    }

                    bufArrayConstLexems.Add(lex);
                }
                else
                {
                    bufVariableLexems.Add(lex);
                }
            }
                            else if (bForLoopReading)
                            {
                                Lexem bufLex = lex;

                                if
(CheckLexemContaining(LexemSubType.variable,
bufForLoopLexems))
                                {
                                    if
(FindLexemWithSubType(LexemSubType.variable,
bufForLoopLexems).Content == lex.Content) continue;
```

```
                                    bufLex.SubType =
LexemSubType.constant;
                                }

                                bufForLoopLexems.Add(bufLex);
                            }
                else if (bSubProgHeaderReading)
                {
                    bufVariableLexems.Add(lex);
                }
                else if (wasReadForLoopCount == -2)
                {
                    wasReadForLoopCount = 0;
                    continue;
                }
                else if (bInputReading)
                {
                    if (requiredLang.Name == "C")
                    {
                        if (!bQuotesRead && squareStamplesCounter ==
0)
                        {
                            LexemSubType lsto = LexemSubType.any;

                            for (int i = 0; i < allVars.Count; i++)
                            {
                                if (allVars[i].Content == lex.Content)
                                {
                                    lsto = allVars[i].SubType;
```

```
                        break;
                    }
                }

                switch (lsto)
                {
                    case LexemSubType.typeString:
                        outputFileLexem.Add(new Lexem("\"%s\",
", LexemType.Variable, LexemSubType.any));
                        break;
                    case LexemSubType.type32sInt:
                        outputFileLexem.Add(new Lexem("\"%i\",
&", LexemType.Variable, LexemSubType.any));
                        break;
                    default:
                        MessageBox.Show("fd");
                        break;
                }
            }
        }

        outputFileLexem.Add(lex);
    }
    else if (bOutputReading)
    {
        if (requiredLang.Name == "C")
        {
            if (!bQuotesRead && squareStamplesCounter ==
0)
            {
```

```
                    LexemSubType lsto = LexemSubType.any;

                    for (int i = 0; i < allVars.Count; i++)
                    {
                        if (allVars[i].Content == lex.Content)
                        {
                            lsto = allVars[i].SubType;
                            break;
                        }
                    }

                    switch (lsto)
                    {
                        case LexemSubType.typeString:
                            outputFileLexem.Insert(quoteInIOCursor,
new Lexem("%s", LexemType.Variable, LexemSubType.any));
                            break;
                        case LexemSubType.type32sInt:
                            outputFileLexem.Add(new Lexem("\"%i\",
", LexemType.Variable, LexemSubType.any));
                            break;
                        default:
                            MessageBox.Show("fd");
                            break;
                    }
                }
            }

            if (sourceLang.Name == "C")
            {
```

```
                    if (lex.Content[0] == '%')
                    {
                        continue;
                    }
                }

                outputFileLexem.Add(lex);
            }
            else if (!bMainConstructionReading &&
!bInputReading)
            {
                if (!bNameRead)
                {
                    bNameRead = true;
                    squareStamplesCounter = 0;
                }
                outputFileLexem.Add(lex);
            }

                            continue;
                        }

            if (lex.Type == LexemType.VarType)
                        {

                typeOfCurrentLexem = lex.SubType;

                            if (sourceLang.Name == "C")
                            {
                                bVariableReading = true;
```

```
                variableCursorPosition = cursorPositionInProc;
                        }
            else
            {
                for (int i = 0; i < allVars.Count; i++)
                {
                    if (allVars[i].SubType == LexemSubType.variable)
                    {
                        Lexem bufl = allVars[i];
                        bufl.SubType = typeOfCurrentLexem;
                        allVars.RemoveAt(i);
                        allVars.Insert(i, bufl);
                    }
                }

                if (subProgRead == LexemSubType.any)
                {
                    for (int y = 0; y < bufGlobalVariableLexems.Count;
y++)
                    {
                        if (bufGlobalVariableLexems[y].Type !=
LexemType.VarType)
                        {
                            Lexem blx = bufGlobalVariableLexems[y];
                            bufGlobalVariableLexems.RemoveAt(y);
                            blx.SubType = lex.SubType;
                            blx.Type = LexemType.VarType;
                            bufGlobalVariableLexems.Insert(y, blx);
                        }
                    }
```

```
                }
            }
            continue;
                    }
                    //если данная лексема является началом
цикла со счетчиком, чекаем соответствующий флаг
                    else if (lex.SubType == LexemSubType.algFor)
                    {
                        bForLoopReading = true;
                        continue;
                    }
                        //аналогично для условной
конструкции
                    else if (lex.SubType == LexemSubType.algIf)
                    {
            if (bCommaInArrayRead) continue;

                        bIfStateReading = true;


      outputFileLexem.Add(FindLexemWithSubType(LexemSubTyp
e.algIf, requiredLangLex));


outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                        if (requiredLang.BStrictlyStamples)
                        {

      outputFileLexem.Add(FindLexemWithSubType(LexemSubTyp
```

```
e.stampleOpenSymbol, requiredLangLex));
                                }

                                continue;
                            }
                    else if (lex.SubType == LexemSubType.algEnd)
                    {
                        bCommaInArrayRead = true; //флажок прочтения
AlgEnd
                        outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockEnd, requiredLangLex));
                        outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
                        continue;
                    }
                    else if (lex.SubType == LexemSubType.algWhile)
                    {
                        if (bRepeatLoopReading)
                        {
                            if (requiredLang.BStrictlyBlockInRepeat)
                            {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockEnd, requiredLangLex));
                                outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.any));
                            }
```

```
                bRepeatDefenitonReading = true;

                if (requiredLang.Name == "Basic")
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.alg
Loop, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                }

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.alg
Until, requiredLangLex));

                if (requiredLang.BStrictlyStamples)
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleOpenSymbol, requiredLangLex));
                }

                continue;
            }

            bWhileLoopReading = true;
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.alg
While, requiredLangLex));

                if (requiredLang.BStrictlyStamples)
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleOpenSymbol, requiredLangLex));
                }

                if (sourceLang.Name == "Basic") bDoRead = false;

                continue;
            }
            else if (lex.SubType == LexemSubType.algRepeat)
            {
                bRepeatLoopReading = true;

                if (requiredLang.Name == "Basic")

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.alg
Do, requiredLangLex));
                else

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.alg
```

```
Repeat, requiredLangLex));

                if (requiredLang.BStrictlyBlockInRepeat)
                {
                    outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.any));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockStart, requiredLangLex));
                }

                continue;
            }
            else if (lex.SubType == LexemSubType.algUntil)
            {
                if (requiredLang.BStrictlyBlockInRepeat)
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockEnd, requiredLangLex));
                    outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.any));
                }

                if (requiredLang.Name == "Basic")
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.alg
Loop, requiredLangLex));
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                }

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.alg
Until, requiredLangLex));

                if (requiredLang.BStrictlyStamples)
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleOpenSymbol, requiredLangLex));
                }

                bRepeatDefenitonReading = true;

                continue;
            }
            else if (lex.SubType == LexemSubType.algDo)
            {
                if (sourceLang.Name == "Basic")
                {
                    bDoRead = true;
                    continue;
                }
            }
            else if (lex.SubType == LexemSubType.varBlock)
```

```
                    {
                bConstReading = false;
                        bVariableReading = true;
                variableCursorPosition = cursorPositionInProc;
                        continue;
                    }
            else if (lex.SubType == LexemSubType.constDef)
            {
                bConstReading = true;
                bVariableReading = true;
                variableCursorPosition = cursorPositionInProc;
                continue;
            }
            else if (lex.SubType == LexemSubType.endOfLine)
            {
                bInputReading = false;
                bOutputReading = false;
                quoteInIOCursor = 0;

                if (bVariableReading)
                {
                    if (sourceLang.Name == "C")
                    {
                        if (requiredLang.Name == "Pascal")
                        {
                            foreach (Lexem bufLex in bufVariableLexems)
                            {

outputFileLexem.Insert(variableCursorPosition++, bufLex);
```

```
outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.colonSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                    }
                }
                else if (requiredLang.Name == "Basic")
                {
                    foreach (Lexem bufLex in bufVariableLexems)
                    {

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.varBlock,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
```

```
outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.logicAs,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++, bufLex);

outputFileLexem.Insert(variableCursorPosition++, new
Lexem("\n", LexemType.Divider, LexemSubType.any));

                    }
                }

                typeOfCurrentLexem = LexemSubType.any;
                bVariableReading = false;
                bufVariableLexems.Clear();
                continue;
            }
            else if (sourceLang.Name == "Pascal")
```

```
                {
                    if (requiredLang.Name == "C" && subProgRead != LexemSubType.any)
                    {
                        int i = variableCursorPosition;
                        if (bConstReading)
                        {
                            outputFileLexem.Insert(i++, FindLexemWithSubType(LexemSubType.constDef, requiredLangLex));
                            outputFileLexem.Insert(i++, FindLexemWithSubType(LexemSubType.spaceSymbol, requiredLangLex));
                        }
                        outputFileLexem.Insert(i++, FindLexemWithSubType(typeOfCurrentLexem, requiredLangLex));
                        outputFileLexem.Insert(i++, FindLexemWithSubType(LexemSubType.spaceSymbol, requiredLangLex));
                        for (int j = 0; j < bufVariableLexems.Count; j++)
                        {
                            outputFileLexem.Insert(i, bufVariableLexems[j]);
                            i++;

                            if (j == bufVariableLexems.Count - 1) continue;

                            outputFileLexem.Insert(i,
```

```
FindLexemWithSubType(LexemSubType.commaSymbol,
requiredLangLex));
                    i++;
                    outputFileLexem.Insert(i,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                    i++;
                }
                if (bConstReading)
                {
                    outputFileLexem.Insert(i++,
FindLexemWithSubType(LexemSubType.logicAssign,
requiredLangLex));
                    outputFileLexem.Insert(i++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                    bufConstValueLexems.Reverse();
                    foreach (Lexem bflex in
bufConstValueLexems)
                    {
                        outputFileLexem.Insert(i++, bflex);
                    }
                }

                outputFileLexem.Insert(i++,
FindLexemWithSubType(lex.SubType, requiredLangLex));
                outputFileLexem.Insert(i,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
            }
```

```
                else if (requiredLang.Name == "Basic")
                {
                    foreach (Lexem bufLex in bufVariableLexems)
                    {

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.varBlock,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.logicAs,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++, bufLex);
```

```
outputFileLexem.Insert(variableCursorPosition++, new
Lexem("\n", LexemType.Divider, LexemSubType.any));

                    }
                }

                bConstValueReading = false;
                bufConstValueLexems.Clear();
                bufVariableLexems.Clear();
                continue;
            }

            variableCursorPosition = cursorPositionInProc;
        }
        else if (bArrayReading)
        {
            if (!bCommaInArrayRead)
            {
                foreach (Lexem bufLex in bufVariableLexems)
                {
                    switch (requiredLang.Name)
                    {
                        case "C":
                            {

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));
```

```
outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++, bufLex);

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.squareStampleOpen,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
bufArrayConstLexems[0]);

outputFileLexem.Insert(variableCursorPosition++,
bufArrayConstLexems[1]);

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.squareStampleClose,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                        }
                        break;
```

```
                    case "Pascal":
                      {

outputFileLexem.Insert(variableCursorPosition++, bufLex);

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.colonSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.arrayType,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.squareStampleOpen,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++, new
Lexem("1", LexemType.Variable, LexemSubType.constant));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.arrayRange,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
bufArrayConstLexems[0]);

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.squareStampleClose,
requiredLangLex));
```

```
outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.algOf, requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                        }
                        break;

                    case "Basic":
                        {

outputFileLexem.Insert(variableCursorPosition++,
```

```
FindLexemWithSubType(LexemSubType.varBlock,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++, bufLex);

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.stampleOpenSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++, new
Lexem("1", LexemType.Variable, LexemSubType.constant));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.algTo,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
bufArrayConstLexems[bufArrayConstLexems.Count - 1]);
```

```
outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.stampleCloseSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.logicAs,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++, new
Lexem("\n", LexemType.Divider, LexemSubType.carriageReturn));
                        }
                        break;
                    }
                }
            }
            else
            {
```

```
                switch (requiredLang.Name)
                {
                   case "C":
                      {

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.squareStampleOpen,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
bufArrayConstLexems[1]);

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.squareStampleClose,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(cursorPositionInProc + 1,
FindLexemWithSubType(LexemSubType.spaceSymbol,
```

```
requiredLangLex));
                        }
                        break;

                    case "Pascal":
                        {

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.commaSymbol,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++, new Lexem("1",
LexemType.Variable, LexemSubType.constant));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.arrayRange,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
bufArrayConstLexems[0]);

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.squareStampleClose,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
```

```
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.algOf, requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                        }
                        break;

                    case "Basic":
                        {

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.commaSymbol,
requiredLangLex));
```

```
outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++, new Lexem("1",
LexemType.Variable, LexemSubType.constant));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.algTo,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
bufArrayConstLexems[bufArrayConstLexems.Count - 1]);

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.stampleCloseSymbol,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++, new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
```

```
outputFileLexem.Insert(cursorPositionInProc + 4,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(cursorPositionInProc + 5,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                        }
                        break;
                    }

                    squareStamplesCounter = 0;
                    arrayCursorPosition = cursorPositionInProc;
                    bCommaInArrayRead = false;
                }

                bArrayReading = false;
                if (sourceLang.Name == "Pascal")
                {
                    bVariableReading = true;
                    variableCursorPosition = cursorPositionInProc;
                }
                bufVariableLexems.Clear();
                bufArrayConstLexems.Clear();
                continue;
            }
            else if (bRepeatDefenitonReading)
            {
                if (requiredLang.BStrictlyStamples)
                {
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleCloseSymbol, requiredLangLex));
                }

                bRepeatLoopReading = false;
                bCarriageReturnRead = false;
                bRepeatDefenitonReading = false;
            }
            else if (bMainConstructionReading)
            {
                bMainConstructionReading = false;
                continue;
            }

            bNameRead = false;
        }
        else if (lex.SubType == LexemSubType.carriageReturn)
        {
            if (bInputReading || bOutputReading)
            {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleCloseSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.en
dOfLine, requiredLangLex));
                bInputReading = false;
                bOutputReading = false;
                quoteInIOCursor = 0;
```

```
                }

                if (bVariableReading)
                {
                    if (requiredLang.Name == "Pascal")
                    {
                        foreach (Lexem bufLex in bufVariableLexems)
                        {

outputFileLexem.Insert(variableCursorPosition++, bufLex);

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.colonSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                        }
                    }
                    else if (requiredLang.Name == "C")
                    {
```

```
                int i = variableCursorPosition;
                if (bConstReading)
                {
                    outputFileLexem.Insert(i++,
FindLexemWithSubType(LexemSubType.constDef,
requiredLangLex));
                    outputFileLexem.Insert(i++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                }
                outputFileLexem.Insert(i++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));
                outputFileLexem.Insert(i++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                for (int j = 0; j < bufVariableLexems.Count; j++)
                {
                    outputFileLexem.Insert(i,
bufVariableLexems[j]);
                    i++;

                    if (j == bufVariableLexems.Count - 1) continue;

                    outputFileLexem.Insert(i,
FindLexemWithSubType(LexemSubType.commaSymbol,
requiredLangLex));
                    i++;
                    outputFileLexem.Insert(i,
FindLexemWithSubType(LexemSubType.spaceSymbol,
```

```
requiredLangLex));
                    i++;
                }
                if (bConstReading)
                {
                    outputFileLexem.Insert(i++,
FindLexemWithSubType(LexemSubType.logicAssign,
requiredLangLex));
                    outputFileLexem.Insert(i++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                    bufConstValueLexems.Reverse();
                    foreach (Lexem bflex in bufConstValueLexems)
                    {
                        outputFileLexem.Insert(i++, bflex);
                    }
                }

                outputFileLexem.Insert(i++,
FindLexemWithSubType(lex.SubType, requiredLangLex));
                outputFileLexem.Insert(i++,
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));
            }

            bVariableReading = false;
            bufVariableLexems.Clear();
            outputFileLexem.Add(lex);
            continue;
        }
```

```
            else if (bArrayReading)
            {
               if (!bCommaInArrayRead)
               {
                  foreach (Lexem bufLex in bufVariableLexems)
                  {
                     switch (requiredLang.Name)
                     {
                        case "C":
                           {

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++, bufLex);

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.squareStampleOpen,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
bufArrayConstLexems[0]);

outputFileLexem.Insert(arrayCursorPosition++,
bufArrayConstLexems[1]);
```

```
outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.squareStampleClose,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                        }
                        break;

                    case "Pascal":
                        {

outputFileLexem.Insert(arrayCursorPosition++, bufLex);

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.colonSymbol,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.arrayType,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.squareStampleOpen,
```

```
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++, new Lexem("1",
LexemType.Variable, LexemSubType.constant));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.arrayRange,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
bufArrayConstLexems[0]);

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.squareStampleClose,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.algOf, requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));
```

```
outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                        }
                        break;
                    }
                }
            }
            else
            {
                switch (requiredLang.Name)
                {
                    case "C":
                        {

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.squareStampleOpen,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
bufArrayConstLexems[1]);

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.squareStampleClose,
requiredLangLex));
```

```
outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition + 1,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                        }
                        break;

                    case "Pascal":
                        {

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.commaSymbol,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
```

```
outputFileLexem.Insert(arrayCursorPosition++, new Lexem("1",
LexemType.Variable, LexemSubType.constant));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.arrayRange,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
bufArrayConstLexems[bufArrayConstLexems.Count - 1]);

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.squareStampleClose,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.algOf, requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
```

```
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                        }
                        break;
                    }

                    squareStamplesCounter = 0;
                    arrayCursorPosition = cursorPositionInProc;
                    bCommaInArrayRead = false;
                }

                bArrayReading = false;
                bufVariableLexems.Clear();
                bufArrayConstLexems.Clear();
                outputFileLexem.Add(lex);
                continue;
            }
            else if (bWhileLoopReading)
            {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleCloseSymbol, requiredLangLex));
                outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
```

```
ockStart, requiredLangLex));
                outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
                bWhileLoopReading = false;
            }
            else if (bDoRead)
            {
                if (outputFileLexem.Count > 0 &&
                    (outputFileLexem[outputFileLexem.Count -
1].Type != LexemType.Divider &&
                    outputFileLexem[outputFileLexem.Count -
1].SubType != LexemSubType.spaceSymbol &&
                    outputFileLexem[outputFileLexem.Count -
1].Content != "\n" &&
                    outputFileLexem[outputFileLexem.Count -
1].SubType != LexemSubType.blockStart &&
                    outputFileLexem[outputFileLexem.Count -
1].SubType != LexemSubType.blockEnd &&
                    outputFileLexem[outputFileLexem.Count -
1].SubType != LexemSubType.algElse)
                   )
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.en
dOfLine, requiredLangLex));
                }

                outputFileLexem.Add(lex);
                if (!bCarriageReturnRead &&
requiredLang.BStrictlyBlockInRepeat)
```

```
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.alg
Repeat, requiredLangLex));
                    outputFileLexem.Add(lex);

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockStart, requiredLangLex));
                    outputFileLexem.Add(lex);
                    bCarriageReturnRead = true;
                }

                continue;
            }
            else if (bRepeatDefenitonReading)
            {
                if (requiredLang.BStrictlyStamples)
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleCloseSymbol, requiredLangLex));
                }

                bRepeatLoopReading = false;
                bCarriageReturnRead = false;
                bRepeatDefenitonReading = false;
            }

            if (outputFileLexem.Count > 0 &&
                (outputFileLexem[outputFileLexem.Count - 1].Type
```

```
!= LexemType.Divider &&
                    outputFileLexem[outputFileLexem.Count -
1].SubType != LexemSubType.spaceSymbol &&
                    outputFileLexem[outputFileLexem.Count -
1].Content != "\n" &&
                    outputFileLexem[outputFileLexem.Count -
1].SubType != LexemSubType.blockStart &&
                    outputFileLexem[outputFileLexem.Count -
1].SubType != LexemSubType.blockEnd &&
                    outputFileLexem[outputFileLexem.Count -
1].SubType != LexemSubType.algElse
                    )
                   )
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.en
dOfLine, requiredLangLex));
                }

                outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
            }
            else if (lex.SubType == LexemSubType.blockStart)
            {
                if (sourceLang.Name == "Pascal")
                {
                    bVariableReading = false;
                    bConstReading = false;

                    if (requiredLang.Name == "C" &&
```

```
blockOpenCloseCounter == 1)
                {
                    if (subProgRead == LexemSubType.any)
                    {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.ty
peVoid, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.pr
ogramStart, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleOpenSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleCloseSymbol, requiredLangLex));
                        outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));

outputFileLexem.Add(FindLexemWithSubType(lex.SubType,
requiredLangLex));
                        outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
                        if (bufGlobalVariableLexems.Count > 0)
                        {
                            foreach (Lexem blex in
bufGlobalVariableLexems)
```

```
                    {
                        if (blex.Content[0] == '<')
                        {
                            Lexem qlex = new Lexem();
                            qlex.Content =
blex.Content.Substring(3);

outputFileLexem.Add(FindLexemWithSubType(blex.SubType,
requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                            outputFileLexem.Add(new
Lexem(qlex.Content, LexemType.Variable,
LexemSubType.variable));
                            qlex = new Lexem
                            {
                                Content =
(Convert.ToInt32(blex.Content[1].ToString()) +
Convert.ToInt32(blex.Content[2].ToString())).ToString(),
                                Type = LexemType.Variable,
                                SubType = LexemSubType.constant,
                                Ambiguousity = "single"
                            };

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sq
uareStampleOpen, requiredLangLex));
                            outputFileLexem.Add(qlex);

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sq
```

```
uareStampleClose, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.endOfLine, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.spaceSymbol, requiredLangLex));

outputFileLexem.Insert(variableCursorPosition, new Lexem("\n", LexemType.Divider, LexemSubType.carriageReturn));
                        }
                        else
                        {

outputFileLexem.Add(FindLexemWithSubType(blex.SubType, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.spaceSymbol, requiredLangLex));
                            outputFileLexem.Add(new Lexem(blex.Content, LexemType.Variable, LexemSubType.variable));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.endOfLine, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.spaceSymbol, requiredLangLex));

outputFileLexem.Insert(variableCursorPosition, new Lexem("\n",
```

```
LexemType.Divider, LexemSubType.carriageReturn));
                        }
                    }
                    bufGlobalVariableLexems.Clear();
                }
                cursorPositionInProc = outputFileLexem.Count
+ 1;
                continue;
            }
            else
            {

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(lex.SubType, requiredLangLex));
                outputFileLexem.Insert(variableCursorPosition,
new Lexem("\n", LexemType.Divider,
LexemSubType.carriageReturn));
                continue;
            }
        }
    }
}
else if (lex.SubType == LexemSubType.blockEnd)
{
    if (subProgRead != LexemSubType.any &&
blockOpenCloseCounter == 0)
    {
        if (requiredLang.Name == "Basic")
        {
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.alg
End, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                    if (subProgRead == LexemSubType.typeVoid)

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.pr
ocedure, requiredLangLex));
                    else

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.fu
nction, requiredLangLex));
                    outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));

                    cursorPositionInProc = outputFileLexem.Count +
1;
                }
                else if (requiredLang.Name == "Pascal")
                {
                    cursorPositionInProc = 0;
                }

                currSubProgName = new Lexem();
                subProgRead = LexemSubType.any;
            }

            if (basicLoopRotation.Length > 0)
            {
```

```
                switch
(basicLoopRotation[basicLoopRotation.Length - 1])
                {
                    case 'i':
                        if (cursorIfElse == -1)
                        {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.alg
End, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.alg
If, requiredLangLex));
                            cursorIfElse = 0;
                        }
                        else
                        {
                            cursorIfElse = outputFileLexem.Count;
                        }
                        basicLoopRotation =
basicLoopRotation.Substring(0, basicLoopRotation.Length - 1);
                        continue;
                        break;

                    case 'f':

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.alg
Next, requiredLangLex));
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(bufBasicForLoopLexems.Pop());
                    basicLoopRotation =
basicLoopRotation.Substring(0, basicLoopRotation.Length - 1);
                    continue;
                    break;

                case 'w':

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.alg
Wend, requiredLangLex));
                    outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
                    basicLoopRotation =
basicLoopRotation.Substring(0, basicLoopRotation.Length - 1);
                    continue;
                    break;
                }
            }
        }
        else if (lex.SubType == LexemSubType.mathEqual)
        {
            if (requiredLang.Name == "Basic")
            {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.log
icAssign, requiredLangLex));
```

```
                continue;
            }
        }
        else if (lex.SubType == LexemSubType.algNext)
        {
            if (wasReadForLoopCount == -1)
            {
                wasReadForLoopCount = -2;
            }

            outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockEnd, requiredLangLex));
            outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
            continue;
        }
        else if (lex.SubType == LexemSubType.algLoop)
        {
            if (bDoRead)
            {
                bDoRead = false;
                bRepeatLoopReading = true;
            }
            else
            {
                outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockEnd, requiredLangLex));
                outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
                continue;
            }
        }
        else if (lex.SubType == LexemSubType.procedure ||
lex.SubType == LexemSubType.function)
        {
            if (bCommaInArrayRead)
            {
                if (requiredLang.Name == "C")
                    cursorPositionInProc = outputFileLexem.Count;
                else
                    cursorPositionInProc = 0;

                blockOpenCloseCounter = 0;
                continue;
            }

            if (requiredLang.Name == "Pascal")
            {
                variableCursorPosition = cursorPositionInProc;
                cursorPositionInProc = 0;
                outputFileLexem.Insert(variableCursorPosition++,
new Lexem("\n", LexemType.Divider,
LexemSubType.carriageReturn));
                outputFileLexem.Insert(variableCursorPosition++,
```

```
FindLexemWithSubType(lex.SubType, requiredLangLex));
                }

                bVariableReading = false;
                bSubProgHeaderReading = true;
                continue;
            }
            else if (lex.SubType == LexemSubType.programStart)
            {
                bMainConstructionReading = true;
                continue;
            }
            else if (lex.SubType == LexemSubType.returnWord)
            {
                if (requiredLang.Name == "Pascal")
                {
                    outputFileLexem.Add(currSubProgName);

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.log
icAssign, requiredLangLex));
                    continue;
                }
            }
            else if (lex.SubType == LexemSubType.logicAssign)
            {
                if (requiredLang.Name == "C")
                {
```

```
                if (outputFileLexem[outputFileLexem.Count -
1].Content == currSubProgName.Content)
                {

outputFileLexem.RemoveAt(outputFileLexem.Count - 1);

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.ret
urnWord, requiredLangLex));
                    currSubProgName = new Lexem();
                    continue;
                }
                else if (outputFileLexem[outputFileLexem.Count -
2].Content == currSubProgName.Content)
                {

outputFileLexem.RemoveAt(outputFileLexem.Count - 2);

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.ret
urnWord, requiredLangLex));
                    currSubProgName = new Lexem();
                    continue;
                }
            }
        }
        else if (lex.SubType == LexemSubType.algElse)
        {
            if (sourceLang.Name == "Basic")
            {
                outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockEnd, requiredLangLex));
                outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));

outputFileLexem.Add(FindLexemWithSubType(lex.SubType,
requiredLangLex));
                outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockStart, requiredLangLex));
                outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
                continue;
            }
        }
        else if (lex.SubType == LexemSubType.write ||
lex.SubType == LexemSubType.writeln)
        {
            squareStamplesCounter = 0;

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.wr
ite, requiredLangLex));
            bOutputReading = true;
            stamplesOpenCounter = 0;
            stamplesCloseCounter = 0;
            if (sourceLang.Name == "Basic")
            {
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleOpenSymbol, requiredLangLex));
                }
                continue;
            }
            else if (lex.SubType == LexemSubType.read ||
lex.SubType == LexemSubType.readln)
            {
                squareStamplesCounter = 0;

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.re
ad, requiredLangLex));
                bInputReading = true;
                stamplesOpenCounter = 0;
                stamplesCloseCounter = 0;
                if (sourceLang.Name == "Basic")
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleOpenSymbol, requiredLangLex));
                }
                continue;
            }
            else if (lex.SubType == LexemSubType.preProc)
            {
                bPreProcReading = true;
                continue;
            }
            else if (lex.SubType == LexemSubType.programEnd)
```

```
            {
                continue;
            }

            //если идет чтение цикла со счетчиком
                        if (bForLoopReading)
                        {
                                //если данная лексема равна
последней в определении цикла в исходном языке
                    if (lex.SubType ==
sourceLang.ForLoopLexems[sourceLang.ForLoopLexems.Count -
1].SubType || (lex.Content == "\n" && sourceLang.Name ==
"Basic"))
                                {
                                        //добавляем все лексемы из
буфера в соответствии с необходимой последовательностью
                                        foreach (Lexem forLex in
requiredLang.ForLoopLexems)
                                        {
                                                //если нужна переменная,
добавляем из буфера
                                                if (forLex.SubType ==
LexemSubType.variable)
                                                {
                                                        for (int i = 0; i <
bufForLoopLexems.Count; i++)
                                                        {
                                                                if
(bufForLoopLexems[i].SubType == LexemSubType.variable)
                                                                {
```

```
                    if (requiredLang.Name == "Basic")
                    {

bufBasicForLoopLexems.Push(bufForLoopLexems[i]);
                    }

      outputFileLexem.Add(bufForLoopLexems[i]);
                                        }
                                        break;
                                    }
                                }

                                //аналогично для константы
                                else if (forLex.SubType ==
LexemSubType.constant)
                                {
                                    for (int i = 0; i <
bufForLoopLexems.Count; i++)
                                    {
                                        if
(bufForLoopLexems[i].SubType == LexemSubType.constant)
                                        {

      outputFileLexem.Add(bufForLoopLexems[i]);

      bufForLoopLexems.RemoveAt(i);
                                            break;
                                        }
                                    }
                                }
```

```
                                    //иначе просто добавляем
необходимую лексему из списка
                                    else
                                    {
                                        foreach (Lexem rlex in
requiredLangLex)
                                        {
                                            if (forLex.SubType ==
rlex.SubType)
                                            {

        outputFileLexem.Add(rlex);
                                                break;
                                            }
                                        }
                                    }
                                }

                if (requiredLang.Name == "Basic")
                {
                   basicLoopRotation += 'f';
                }
                else
                {
                   wasReadForLoopCount = -1;
                }

                if (sourceLang.Name == "Basic")
                {
```

```
                    outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockStart, requiredLangLex));
                    outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
                }

                bufForLoopLexems = new List<Lexem>();

                        bForLoopReading = false;
                    }

                    continue;
                }
                //если идет чтение условного перехода
                else if (bIfStateReading)
                {
                    if (lex.SubType ==
LexemSubType.blockStart)
                    {
                        if (requiredLang.BStrictlyStamples)
                        {

    outputFileLexem.Insert(outputFileLexem.Count - 2,
FindLexemWithSubType(LexemSubType.stampleCloseSymbol,
requiredLangLex));
                        }
```

```
                                if (requiredLang.BNeedCloser &&
!sourceLang.BNeedCloser)
                                {

      outputFileLexem.Insert(outputFileLexem.Count - 1,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

      outputFileLexem.Insert(outputFileLexem.Count - 1,
FindLexemWithSubType(LexemSubType.algThen,
requiredLangLex));
                                }

                  if (requiredLang.Name == "Basic")
                  {
                     basicLoopRotation += 'i';
                  }
                                    bIfStateReading = false;
                           }
               else if (sourceLang.Name == "Basic")
               {
                  if (lex.SubType == LexemSubType.algThen)
                  {
                     if (requiredLang.BStrictlyStamples)
                     {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleCloseSymbol, requiredLangLex));
                     }
```

```
                    if (requiredLang.BNeedCloser)
                    {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.alg
Then, requiredLangLex));
                    }

                    outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockStart, requiredLangLex));
                    outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
                    bIfStateReading = false;
                    continue;
                }
                else if (lex.SubType == LexemSubType.logicAssign)
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.m
athEqual, requiredLangLex));
                    continue;
                }
            }
                        }
        else if (bWhileLoopReading)
```

```
            {
                if (lex.SubType == LexemSubType.blockStart)
                {
                    if (requiredLang.BStrictlyStamples)
                    {
                        outputFileLexem.Insert(outputFileLexem.Count -
2, FindLexemWithSubType(LexemSubType.stampleCloseSymbol,
requiredLangLex));
                    }

                    if (requiredLang.BNeedCloser &&
!sourceLang.BNeedCloser)
                    {
                        outputFileLexem.Insert(outputFileLexem.Count -
1, FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                        outputFileLexem.Insert(outputFileLexem.Count -
1, FindLexemWithSubType(LexemSubType.algDo,
requiredLangLex));
                    }

                    if (requiredLang.Name == "Basic")
                    {
                        basicLoopRotation += "w";
                    }

                    bWhileLoopReading = false;
                }
                else if (lex.SubType == LexemSubType.logicAssign &&
sourceLang.Name == "Basic")
```

```
            {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.m
athEqual, requiredLangLex));
               continue;
            }
            else if (lex.SubType == LexemSubType.algDo &&
requiredLang.Name == "Basic")
            {
               continue;
            }

            if (squareStamplesCounter > 0)
            {
               switch (lex.SubType)
               {
                  case LexemSubType.stampleOpenSymbol:
                     if (sourceLang.Name == "Pascal" ||
sourceLang.Name == "C")
                     {
                        bNameRead = false;
                        continue;
                     }
                     break;

                  case LexemSubType.squareStampleOpen:
                     if (squareStamplesCounter <= 1)
                     {
                        if (requiredLang.Name == "Basic")
                        {
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleOpenSymbol, requiredLangLex));
                        continue;
                    }
                }
                else
                {
                    if (requiredLang.Name == "Basic" ||
requiredLang.Name == "Pascal")
                    {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.co
mmaSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                        continue;
                    }
                }
                break;

            case LexemSubType.squareStampleClose:
                if (sourceLang.Name != "C")
                {
                    bNameRead = false;
                    squareStamplesCounter = 0;
                    if (requiredLang.Name == "Basic")
                    {
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleCloseSymbol, requiredLangLex));
                        continue;
                    }
                }
                else continue;
                break;

            case LexemSubType.commaSymbol:
                if (requiredLang.Name == "C")
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sq
uareStampleClose, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sq
uareStampleOpen, requiredLangLex));
                    continue;
                }
                break;

            case LexemSubType.logicAssign:
                bNameRead = false;
                if (requiredLang.Name == "Pascal")
                {

outputFileLexem.Insert(outputFileLexem.Count - 1,
FindLexemWithSubType(LexemSubType.squareStampleClose,
requiredLangLex));
                }
```

```
                    else
                    {

outputFileLexem.Insert(outputFileLexem.Count - 1,
FindLexemWithSubType(LexemSubType.stampleCloseSymbol,
requiredLangLex));
                    }
                    break;
                }
            }
        }
        else if (bRepeatLoopReading)
        {
            if (bRepeatDefenitonReading &&
sourceLang.BInversedRepeat != requiredLang.BInversedRepeat)
            {
                switch (lex.SubType)
                {
                    case LexemSubType.mathEqual:

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.m
athUnequal, requiredLangLex));
                    break;
                case LexemSubType.mathUnequal:
                    if (requiredLang.Name == "Basic")

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.log
icAssign, requiredLangLex));
                    else
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.m
athEqual, requiredLangLex));
                        break;
                    case LexemSubType.mathMore:
                        if (requiredLang.Name == "Basic")

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.log
icAssign, requiredLangLex));
                        else

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.m
athEqual, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.m
athLess, requiredLangLex));
                        break;
                    case LexemSubType.mathLess:
                        if (requiredLang.Name == "Basic")

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.log
icAssign, requiredLangLex));
                        else

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.m
athEqual, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.m
athMore, requiredLangLex));
                        break;
                    case LexemSubType.logicAnd:
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.log
icOr, requiredLangLex));
                        break;
                    case LexemSubType.logicOr:

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.log
icAnd, requiredLangLex));
                        break;
                    case LexemSubType.logicNot:
                        continue;
                    default:

outputFileLexem.Add(FindLexemWithSubType(lex.SubType,
requiredLangLex));
                        break;
                }

                continue;
            }

            if ((!requiredLang.BStrictlyBlockInRepeat) &&
(lex.SubType == LexemSubType.blockEnd || lex.SubType ==
LexemSubType.blockStart))
            {
                continue;
            }
        }
        else if (bSubProgHeaderReading)
        {
```

```
            switch (sourceLang.Name)
            {
               case "C":
                  if (requiredLang.Name == "Pascal")
                  {
                     if (lex.SubType ==
LexemSubType.commaSymbol)
                     {

outputFileLexem.Insert(variableCursorPosition++,
bufVariableLexems[0]);

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.colonSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                        cursorPositionInProc =
variableCursorPosition;
                        bVariableReading = false;
```

```
                    bufVariableLexems.Clear();
                }
                else if (lex.SubType ==
LexemSubType.stampleCloseSymbol)
                {
                    if (bVariableReading)
                    {

outputFileLexem.Insert(variableCursorPosition++,
bufVariableLexems[0]);

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.colonSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));
                        bVariableReading = false;
                        bufVariableLexems.Clear();
                    }


outputFileLexem.Insert(variableCursorPosition, lex);
                    bSubProgHeaderReading = false;
                    cursorPositionInProc =
outputFileLexem.Count + 1;
                }
            }
            else if (requiredLang.Name == "Basic")
```

```
                    {
                        if (lex.SubType ==
LexemSubType.commaSymbol)
                        {

outputFileLexem.Insert(variableCursorPosition++,
bufVariableLexems[0]);

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.logicAs,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.commaSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
```

```
requiredLangLex));
                        bVariableReading = false;
                        cursorPositionInProc =
variableCursorPosition;
                        bufVariableLexems.Clear();
                    }
                    else if (lex.SubType ==
LexemSubType.stampleCloseSymbol)
                    {
                        if (bVariableReading)
                        {

outputFileLexem.Insert(variableCursorPosition++,
bufVariableLexems[0]);

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.logicAs,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));
```

```
                        bVariableReading = false;
                        bufVariableLexems.Clear();
                    }


outputFileLexem.Insert(variableCursorPosition, lex);
                        bSubProgHeaderReading = false;
                        cursorPositionInProc =
outputFileLexem.Count + 1;
                    }
                }
                break;

            case "Pascal":
                if (requiredLang.Name == "C")
                {
                    if (lex.SubType ==
LexemSubType.stampleOpenSymbol)
                    {
                        variableCursorPosition =
outputFileLexem.Count;
                        outputFileLexem.Add(bufVariableLexems[0]);

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleOpenSymbol, requiredLangLex));
                        currSubProgName = bufVariableLexems[0];
                        bufVariableLexems.Clear();
```

```
                    typeOfCurrentLexem = LexemSubType.any;
                }
                else if (lex.SubType ==
LexemSubType.stampleCloseSymbol)
                {
                    if (bufVariableLexems.Count > 0)
                    {

outputFileLexem.Add(FindLexemWithSubType(typeOfCurrentLex
em, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(bufVariableLexems[0]);
                        bufVariableLexems.Clear();
                    }
                    outputFileLexem.Add(lex);
                    stamplesOpenCounter = 0;
                    typeOfCurrentLexem = LexemSubType.any;
                }
                else if (lex.SubType ==
LexemSubType.endOfLine)
                {
                    if (stamplesOpenCounter == 0)
                    {
                        if (typeOfCurrentLexem !=
LexemSubType.any)
                        {
```

```
outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                        subProgRead = typeOfCurrentLexem;
                    }
                    else
                    {

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.typeVoid,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                        subProgRead = LexemSubType.typeVoid;
                    }
                    bSubProgHeaderReading = false;
                    cursorPositionInProc =
outputFileLexem.Count + 1;
                }
                else
                {
                    if (bufVariableLexems.Count > 0)
                    {
```

```
outputFileLexem.Add(FindLexemWithSubType(typeOfCurrentLex
em, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(bufVariableLexems[0]);

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.co
mmaSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                        bufVariableLexems.Clear();
                    }

                    typeOfCurrentLexem = LexemSubType.any;
                }
            }
        }
        else if (requiredLang.Name == "Basic")
        {
            if (lex.SubType ==
LexemSubType.stampleOpenSymbol)
            {
                variableCursorPosition =
outputFileLexem.Count;
                outputFileLexem.Add(bufVariableLexems[0]);

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
```

```
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleOpenSymbol, requiredLangLex));
                    currSubProgName = bufVariableLexems[0];
                    bufVariableLexems.Clear();
                    typeOfCurrentLexem = LexemSubType.any;
                }
                else if (lex.SubType ==
LexemSubType.stampleCloseSymbol)
                {
                    if (bufVariableLexems.Count > 0)
                    {

outputFileLexem.Add(bufVariableLexems[0]);

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.log
icAs, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(typeOfCurrentLex
em, requiredLangLex));
                        bufVariableLexems.Clear();
                    }
                    outputFileLexem.Add(lex);
```

```
                    stamplesOpenCounter = 0;
                    typeOfCurrentLexem = LexemSubType.any;
                }
                else if (lex.SubType ==
LexemSubType.endOfLine)
                {
                    if (stamplesOpenCounter == 0)
                    {
                        if (typeOfCurrentLexem !=
LexemSubType.any)
                        {

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.function,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.log
icAs, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(typeOfCurrentLex
```

```
em, requiredLangLex));
                        subProgRead = typeOfCurrentLexem;
                    }
                    else
                    {

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.procedure,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                        subProgRead = LexemSubType.typeVoid;
                    }

                    bSubProgHeaderReading = false;
                    cursorPositionInProc =
outputFileLexem.Count + 1;
                }
                else
                {
                    if (bufVariableLexems.Count > 0)
                    {

outputFileLexem.Add(bufVariableLexems[0]);

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.log
icAs, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(typeOfCurrentLex
em, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.co
mmaSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                        bufVariableLexems.Clear();
                    }
                    typeOfCurrentLexem = LexemSubType.any;
                }
            }
        }
        break;

    case "Basic":
        if (requiredLang.Name == "Pascal")
        {
            if (lex.SubType ==
LexemSubType.commaSymbol)
            {

outputFileLexem.Insert(variableCursorPosition++,
```

```
bufVariableLexems[0]);

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.colonSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                    cursorPositionInProc =
variableCursorPosition;
                    bufVariableLexems.Clear();
                }
                else if (lex.SubType ==
LexemSubType.stampleOpenSymbol)
                {

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
```

```
bufVariableLexems[0]);

outputFileLexem.Insert(variableCursorPosition++, lex);
                        bVariableReading = false;
                        bufVariableLexems.Clear();
                    }
                    else if (lex.SubType ==
LexemSubType.stampleCloseSymbol)
                    {
                        if (bufVariableLexems.Count > 0)
                        {

outputFileLexem.Insert(variableCursorPosition++,
bufVariableLexems[0]);

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.colonSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));
                            bVariableReading = false;
                            bufVariableLexems.Clear();
                        }


outputFileLexem.Insert(variableCursorPosition++, lex);
                    }
                    else if (lex.SubType ==
```

```
LexemSubType.carriageReturn)
                        {

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.colonSymbol,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++, lex);

outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.blockStart,
requiredLangLex));

outputFileLexem.Insert(variableCursorPosition++, lex);
                            cursorPositionInProc =
outputFileLexem.Count + 1;
                            bSubProgHeaderReading = false;
                            subProgRead = typeOfCurrentLexem;
                        }
                    }
                    else if (requiredLang.Name == "C")
                    {
```

```
                    if (lex.SubType ==
LexemSubType.commaSymbol)
                    {

outputFileLexem.Add(FindLexemWithSubType(typeOfCurrentLex
em, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                        outputFileLexem.Add(bufVariableLexems[0]);

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.co
mmaSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                        bufVariableLexems.Clear();

                        typeOfCurrentLexem = LexemSubType.any;
                        bufVariableLexems.Clear();
                    }
                    else if (lex.SubType ==
LexemSubType.stampleOpenSymbol)
                    {
                        cursorPositionInProc =
outputFileLexem.Count;

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                        outputFileLexem.Add(bufVariableLexems[0]);
```

```
                    outputFileLexem.Add(lex);
                    bVariableReading = false;
                    bufVariableLexems.Clear();

                    typeOfCurrentLexem = LexemSubType.any;
                }
                else if (lex.SubType ==
LexemSubType.stampleCloseSymbol)
                {
                    if (bufVariableLexems.Count > 0)
                    {

outputFileLexem.Add(FindLexemWithSubType(typeOfCurrentLex
em, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(bufVariableLexems[0]);
                        typeOfCurrentLexem = LexemSubType.any;
                        bufVariableLexems.Clear();
                    }
                    outputFileLexem.Add(lex);
                }
                else if (lex.SubType ==
LexemSubType.carriageReturn)
                {
                    if (typeOfCurrentLexem !=
LexemSubType.any)
                    {
```

```
                        outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));

outputFileLexem.Insert(cursorPositionInProc,
FindLexemWithSubType(typeOfCurrentLexem,
requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                        typeOfCurrentLexem = LexemSubType.any;
                    }
                    else
                    {
                        outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));

outputFileLexem.Insert(cursorPositionInProc,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                        typeOfCurrentLexem = LexemSubType.any;
                    }
                    outputFileLexem.Add(new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockStart, requiredLangLex));
                    outputFileLexem.Add(new Lexem("\n",
```

```
LexemType.Divider, LexemSubType.carriageReturn));
                        bSubProgHeaderReading = false;
                        if (typeOfCurrentLexem ==
LexemSubType.any)
                           subProgRead = LexemSubType.typeVoid;
                        else
                           subProgRead = typeOfCurrentLexem;
                     }
                  }
                  break;
               }

               continue;
            }
            else if (bNameRead)
            {
               if (squareStamplesCounter > 0)
               {
                  switch (lex.SubType)
                  {
                     case LexemSubType.stampleOpenSymbol:
                        if (sourceLang.Name == "Pascal" ||
sourceLang.Name == "C")
                        {
                           bNameRead = false;
                           continue;
                        }
                        break;

                     case LexemSubType.squareStampleOpen:
```

```
                    if (squareStamplesCounter <= 1)
                    {
                        if (requiredLang.Name == "Basic")
                        {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleOpenSymbol, requiredLangLex));
                            continue;
                        }
                    }
                    else
                    {
                        if (requiredLang.Name == "Basic" ||
requiredLang.Name == "Pascal")
                        {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.co
mmaSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                            continue;
                        }
                    }
                    break;

                case LexemSubType.squareStampleClose:
                    if (sourceLang.Name != "C")
                    {
                        bNameRead = false;
```

```
                    squareStamplesCounter = 0;
                    if (requiredLang.Name == "Basic")
                    {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleCloseSymbol, requiredLangLex));
                        continue;
                    }
                }
                else continue;
                break;

            case LexemSubType.commaSymbol:
                if (requiredLang.Name == "C")
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sq
uareStampleClose, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sq
uareStampleOpen, requiredLangLex));
                    continue;
                }
                break;

            case LexemSubType.logicAssign:
                bNameRead = false;
                if (requiredLang.Name == "Pascal")
                {
```

```
outputFileLexem.Insert(outputFileLexem.Count - 1,
FindLexemWithSubType(LexemSubType.squareStampleClose,
requiredLangLex));
                    }
                    else
                    {

outputFileLexem.Insert(outputFileLexem.Count - 1,
FindLexemWithSubType(LexemSubType.stampleCloseSymbol,
requiredLangLex));
                    }
                    break;
                }
            }
            else
            {
                switch (lex.SubType)
                {
                    case LexemSubType.stampleOpenSymbol:
                        if (sourceLang.Name == "Basic")
                        {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sq
uareStampleOpen, requiredLangLex));
                            continue;
                        }
                        break;

                    case LexemSubType.stampleCloseSymbol:
                        bNameRead = false;
```

```
                    squareStamplesCounter = 0;

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleCloseSymbol, requiredLangLex));
                    continue;
                }
            }
        }
                    else if (bVariableReading)
                    {
            if (bConstValueReading)
            {
                if (lex.SubType == LexemSubType.blockStart)
                {

bufConstValueLexems.Add(FindLexemWithSubType(LexemSubTy
pe.stampleOpenSymbol, requiredLangLex));
                }
                else if (lex.SubType == LexemSubType.blockEnd)
                {

bufConstValueLexems.Add(FindLexemWithSubType(LexemSubTy
pe.stampleCloseSymbol, requiredLangLex));
                }
                else if (lex.SubType ==
LexemSubType.stampleOpenSymbol)
                {

bufConstValueLexems.Add(FindLexemWithSubType(LexemSubTy
pe.blockStart, requiredLangLex));
```

```
                }
                else if (lex.SubType ==
LexemSubType.stampleCloseSymbol)
                {

bufConstValueLexems.Add(FindLexemWithSubType(LexemSubTy
pe.blockEnd, requiredLangLex));
                }
                else
                {

bufConstValueLexems.Add(FindLexemWithSubType(lex.SubType,
requiredLangLex));
                }

                continue;
            }

                        if (sourceLang.Name == "C")
                        {
                            if (lex.SubType ==
LexemSubType.logicAssign)
                            {
                                bConstValueReading = true;
                            }
                else if (lex.SubType ==
LexemSubType.squareStampleOpen)
                {
                    bArrayReading = true;
                    bVariableReading = false;
```

```
                }
                else if (lex.SubType ==
LexemSubType.stampleOpenSymbol)
                {
                   if (!bMainConstructionReading)
                   {
                      bVariableReading = false;
                      bSubProgHeaderReading = true;

                      subProgRead = typeOfCurrentLexem;

                      if (typeOfCurrentLexem ==
LexemSubType.typeVoid)
                      {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.pr
ocedure, requiredLangLex));
                         variableCursorPosition =
outputFileLexem.Count;

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.en
dOfLine, requiredLangLex));
                      }
                      else
                      {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.fu
nction, requiredLangLex));
                         variableCursorPosition =
outputFileLexem.Count;
```

```
                    if (requiredLang.Name == "Pascal")

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.col
onSymbol, requiredLangLex));
                    else if (requiredLang.Name == "Basic")
                    {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.log
icAs, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                    }


outputFileLexem.Add(FindLexemWithSubType(typeOfCurrentLex
em, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.en
dOfLine, requiredLangLex));
                }


outputFileLexem.Insert(variableCursorPosition++,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
```

```
outputFileLexem.Insert(variableCursorPosition++,
bufVariableLexems[0]);

outputFileLexem.Insert(variableCursorPosition++, lex);
                    cursorPositionInProc = variableCursorPosition;
                    currSubProgName = bufVariableLexems[0];
                    bufVariableLexems.Clear();
                }
            }
                        }
            else if (sourceLang.Name == "Pascal")
            {
                if (lex.SubType == LexemSubType.blockStart)
                {
                    bVariableReading = false;
                    bConstReading = false;

outputFileLexem.Add(FindLexemWithSubType(lex.SubType,
requiredLangLex));
                }
                else if (lex.SubType == LexemSubType.mathEqual)
                {
                    bConstValueReading = true;
                }
                else if (lex.SubType ==
LexemSubType.squareStampleOpen)
                {
                    bArrayReading = true;
                    bVariableReading = false;
                }
```

```
            }
            else if (sourceLang.Name == "Basic")
            {
               if (lex.SubType ==
LexemSubType.stampleOpenSymbol)
               {
                  bArrayReading = true;
                  bVariableReading = false;
               }
            }

            continue;
                     }
         else if (bArrayReading)
         {
            if (lex.SubType == LexemSubType.commaSymbol ||
(lex.SubType == LexemSubType.squareStampleOpen &&
squareStamplesCounter > 1))
            {
               if (!bCommaInArrayRead)
               {
                  switch (requiredLang.Name)
                  {
                     case "C":
                        {
                           for (int i = 0; i < bufVariableLexems.Count;
i++)
                           {

outputFileLexem.Insert(arrayCursorPosition++,
```

```
bufVariableLexems[i]);
                        if (i != bufVariableLexems.Count - 1)
                        {

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.commaSymbol,

requiredLangLex));
                        }
                    }

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.squareStampleOpen,
                                                requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
bufArrayConstLexems[1]);

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.squareStampleClose,
                                                requiredLangLex));
                }
                break;

            case "Pascal":
                {
                    for (int i = 0; i < bufVariableLexems.Count;
```

```
i++)
                        {

outputFileLexem.Insert(arrayCursorPosition++,
bufVariableLexems[i]);
                            if (i != bufVariableLexems.Count - 1)
                            {

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.commaSymbol,

requiredLangLex));
                            }
                        }

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.colonSymbol,
                                                requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.arrayType,
                                                requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.squareStampleOpen,
                                                requiredLangLex));
```

```
outputFileLexem.Insert(arrayCursorPosition++,
                                        new Lexem("1",
LexemType.Variable, LexemSubType.constant));

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.arrayRange,
                                                requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
bufArrayConstLexems[bufArrayConstLexems.Count-1]);
                    }
                    break;

                case "Basic":
                    {

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.varBlock,

requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.spaceSymbol,
                                                requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
```

```
FindLexemWithSubType(LexemSubType.logicAs,
                                                    requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.spaceSymbol,
                                                    requiredLangLex));
                        for (int i = 0; i < bufVariableLexems.Count;
i++)
                        {

outputFileLexem.Insert(arrayCursorPosition++,
bufVariableLexems[i]);
                            if (i != bufVariableLexems.Count - 1)
                            {

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.commaSymbol,

requiredLangLex));
                            }
                        }

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.spaceSymbol,
                                                    requiredLangLex));
```

```
outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.stampleOpenSymbol,
                                                    requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
                                            new Lexem("1",
LexemType.Variable, LexemSubType.constant));

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.spaceSymbol,
                                                    requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.algTo,
                                                    requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.spaceSymbol,
                                                    requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
bufArrayConstLexems[0]);
                        }
                        break;
                    }
```

```
                    bCommaInArrayRead = true;
                }
                else
                {
                    switch (requiredLang.Name)
                    {
                        case "C":
                            {

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.squareStampleOpen,
                                                    requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
bufArrayConstLexems[1]);

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.squareStampleClose,
                                                    requiredLangLex));
                            }
                            break;

                        case "Pascal":
                            {

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.commaSymbol,
```

```
                                                    requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.spaceSymbol,
                                                    requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
                                            new Lexem("1",
LexemType.Variable, LexemSubType.constant));

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.arrayRange,
                                                    requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
bufArrayConstLexems[bufArrayConstLexems.Count - 1]);
                        }
                        break;

                    case "Basic":
                        {

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.commaSymbol,
                                                    requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
```

```
FindLexemWithSubType(LexemSubType.spaceSymbol,
                                                    requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.spaceSymbol,
                                                    requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
                                            new Lexem("1",
LexemType.Variable, LexemSubType.constant));

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.spaceSymbol,
                                                    requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.algTo,
                                                    requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,

FindLexemWithSubType(LexemSubType.spaceSymbol,
                                                    requiredLangLex));

outputFileLexem.Insert(arrayCursorPosition++,
bufArrayConstLexems[0]);
```

```
                    }
                    break;
                }
            }

            bufArrayConstLexems.Clear();
        }

        continue;
    }
    else if (bInputReading)
    {
        if (lex.SubType == LexemSubType.stringOpen)
        {
            if (bQuotesRead)
            {
                if (requiredLang.Name == "C")
                    quoteInIOCursor = outputFileLexem.Count;
                bQuotesRead = false;
            }
            else
                bQuotesRead = true;
        }

        if (requiredLang.Name == "Basic")
        {
            if (lex.SubType ==
LexemSubType.stampleOpenSymbol && stamplesOpenCounter <
2)
            {
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.spaceSymbol, requiredLangLex));
                    continue;
                }
                else if (lex.SubType == LexemSubType.stampleCloseSymbol && stamplesCloseCounter < 2)
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.spaceSymbol, requiredLangLex));
                    continue;
                }
            }

            if (sourceLang.Name == "C")
            {
                if (bQuotesRead)
                    continue;
            }

            if (squareStamplesCounter > 0)
            {
                switch (lex.SubType)
                {
                    case LexemSubType.stampleOpenSymbol:
                        if (sourceLang.Name == "Pascal" || sourceLang.Name == "C")
                        {
```

```
                        bNameRead = false;
                        continue;
                    }
                    break;

                case LexemSubType.squareStampleOpen:
                    if (squareStamplesCounter <= 1)
                    {
                        if (requiredLang.Name == "Basic")
                        {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleOpenSymbol, requiredLangLex));
                            continue;
                        }
                    }
                    else
                    {
                        if (requiredLang.Name == "Basic" ||
requiredLang.Name == "Pascal")
                        {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.co
mmaSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                            continue;
                        }
                    }
```

```
                    break;

                case LexemSubType.squareStampleClose:
                    if (sourceLang.Name != "C")
                    {
                        bNameRead = false;
                        squareStamplesCounter = 0;
                        if (requiredLang.Name == "Basic")
                        {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleCloseSymbol, requiredLangLex));
                            continue;
                        }
                    }
                    else continue;
                    break;

                case LexemSubType.commaSymbol:
                    if (requiredLang.Name == "C")
                    {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sq
uareStampleClose, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sq
uareStampleOpen, requiredLangLex));
                        continue;
                    }
                    break;
```

```
                    case LexemSubType.logicAssign:
                        bNameRead = false;
                        if (requiredLang.Name == "Pascal")
                        {

outputFileLexem.Insert(outputFileLexem.Count - 1,
FindLexemWithSubType(LexemSubType.squareStampleClose,
requiredLangLex));
                        }
                        else
                        {

outputFileLexem.Insert(outputFileLexem.Count - 1,
FindLexemWithSubType(LexemSubType.stampleCloseSymbol,
requiredLangLex));
                        }
                        break;
                    }
                }
            }
            else if (bOutputReading)
            {
                if (lex.SubType == LexemSubType.stringOpen)
                {
                    if (bQuotesRead)
                    {
                        if (requiredLang.Name == "C")
                            quoteInIOCursor = outputFileLexem.Count;
                        bQuotesRead = false;
```

```
                }
                else
                    bQuotesRead = true;
            }

            if (requiredLang.Name == "Basic")
            {
                if (lex.SubType ==
LexemSubType.stampleOpenSymbol && stamplesOpenCounter <
2)
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                    continue;
                }
                else if (lex.SubType ==
LexemSubType.stampleCloseSymbol && stamplesCloseCounter <
2)
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                    continue;
                }
                else if (lex.SubType ==
LexemSubType.commaSymbol)
                {
                    if (!bQuotesRead)
                    {
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.spaceSymbol, requiredLangLex));
                        outputFileLexem.Add(new Lexem(";", LexemType.Divider, LexemSubType.any));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.spaceSymbol, requiredLangLex));
                    }
                    continue;
                }
            }

            if (squareStamplesCounter > 0)
            {
                switch (lex.SubType)
                {
                    case LexemSubType.stampleOpenSymbol:
                        if (sourceLang.Name == "Pascal" || sourceLang.Name == "C")
                        {
                            bNameRead = false;
                            continue;
                        }
                        break;

                    case LexemSubType.squareStampleOpen:
                        if (squareStamplesCounter <= 1)
                        {
                            if (requiredLang.Name == "Basic")
```

```
                    {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleOpenSymbol, requiredLangLex));
                        continue;
                    }
                }
                else
                {
                    if (requiredLang.Name == "Basic" ||
requiredLang.Name == "Pascal")
                    {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.co
mmaSymbol, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sp
aceSymbol, requiredLangLex));
                        continue;
                    }
                }
                break;

            case LexemSubType.squareStampleClose:
                if (sourceLang.Name != "C")
                {
                    bNameRead = false;
                    squareStamplesCounter = 0;
                    if (requiredLang.Name == "Basic")
                    {
```

```
outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sta
mpleCloseSymbol, requiredLangLex));
                        continue;
                    }
                }
                else continue;
                break;

            case LexemSubType.commaSymbol:
                if (requiredLang.Name == "C")
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sq
uareStampleClose, requiredLangLex));

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.sq
uareStampleOpen, requiredLangLex));
                    continue;
                }
                break;

            case LexemSubType.logicAssign:
                bNameRead = false;
                if (requiredLang.Name == "Pascal")
                {

outputFileLexem.Insert(outputFileLexem.Count - 1,
FindLexemWithSubType(LexemSubType.squareStampleClose,
requiredLangLex));
```

```
                    }
                    else
                    {

outputFileLexem.Insert(outputFileLexem.Count - 1,
FindLexemWithSubType(LexemSubType.stampleCloseSymbol,
requiredLangLex));
                    }
                    break;
                }
            }
        }


outputFileLexem.Add(FindLexemWithSubType(lex.SubType,
requiredLangLex));
                }

        switch (requiredLang.Name)
        {
            case "Pascal":
                outputFileLexem.Insert(0,
FindLexemWithSubType(LexemSubType.programStart,
requiredLangLex));
                outputFileLexem.Insert(1,
FindLexemWithSubType(LexemSubType.spaceSymbol,
requiredLangLex));
                outputFileLexem.Insert(2, new Lexem("program_1",
LexemType.Variable, LexemSubType.any));
                outputFileLexem.Insert(3,
```

```
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));
                outputFileLexem.Insert(6, new Lexem("\n",
LexemType.Divider, LexemSubType.carriageReturn));
                if (sourceLang.Name == "Basic")
                {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockEnd, requiredLangLex));
                }

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.pr
ogramEnd, requiredLangLex));

                int bufEndIndexer = 0;
                bool bNeedInsertEndOfLine = false;
                bool bVarInserted = false;
                bool bFunctionRead = false;
                int cursorToVar = 0;
                for (int i = 0; i < outputFileLexem.Count; i++)
                {
                   if (outputFileLexem[i].SubType ==
LexemSubType.blockEnd)
                   {
                      if (bNeedInsertEndOfLine)
                      {
                         outputFileLexem.Insert(bufEndIndexer,
FindLexemWithSubType(LexemSubType.endOfLine,
requiredLangLex));
                         i++;
```

```
                }
                else
                    bNeedInsertEndOfLine = true;

                bufEndIndexer = i + 1;
            }
            else if (outputFileLexem[i].SubType ==
LexemSubType.algElse)
            {
                bNeedInsertEndOfLine = false;
            }
            else if (outputFileLexem[i].SubType ==
LexemSubType.procedure || outputFileLexem[i].SubType ==
LexemSubType.function)
            {
                bFunctionRead = true;
            }
            else if (outputFileLexem[i].SubType ==
LexemSubType.endOfLine)
            {
                if (bFunctionRead)
                {
                    bFunctionRead = false;
                    bVarInserted = false;
                }
            }

            if (!bVarInserted)
            {
                if (outputFileLexem[i].SubType ==
```

```
LexemSubType.variable)
                    {
                        cursorToVar = i;
                        if (outputFileLexem[i + 1].SubType ==
LexemSubType.colonSymbol)
                        {
                            outputFileLexem.Insert(cursorToVar++, new
Lexem("\n", LexemType.Divider, LexemSubType.carriageReturn));
                            outputFileLexem.Insert(cursorToVar++,
FindLexemWithSubType(LexemSubType.varBlock,
requiredLangLex));
                            outputFileLexem.Insert(cursorToVar++, new
Lexem("\n", LexemType.Divider, LexemSubType.carriageReturn));
                            bVarInserted = true;
                        }
                    }
                }
            }
            break;
        case "C":
            for (int i = 0; i < outputFileLexem.Count; i++)
            {
                if (outputFileLexem[i].SubType ==
LexemSubType.blockEnd && outputFileLexem[i + 1].SubType ==
LexemSubType.endOfLine)
                {
                    outputFileLexem.RemoveAt(i + 1);
                }
            }
```

```
            if (sourceLang.Name == "Basic")
            {

outputFileLexem.Add(FindLexemWithSubType(LexemSubType.bl
ockEnd, requiredLangLex));
            }
            break;
        }

        int op = 1;
        for (int i = 1; i < outputFileLexem.Count; i++)
        {
          if (outputFileLexem[op].Content == "" ||
outputFileLexem[op].Content == null)
          {
            outputFileLexem.RemoveAt(op);
            continue;
          }

          if ((outputFileLexem[op].Content == "\n" ||
outputFileLexem[op].Content == " ") && outputFileLexem[op -
1].Content == "\n")
          {
            outputFileLexem.RemoveAt(op);
            continue;
          }

          op++;
        }
```

```
            return outputFileLexem;
        }

        //функция поиска лексемы с именем name среди
коллекции лексем sourceLangLex
        static Lexem FindLexemByName(string name,
List<Lexem> sourceLangLex)
        {
            foreach (Lexem lex in sourceLangLex)
            {
                if (name.ToLower() == lex.Content.ToLower())
                {
                    return lex;
                }
            }

            //если переданная строка является числом,
возвращаем константу
            if (char.IsDigit(name[0]))
            {
                return new Lexem(name,
LexemType.Variable, LexemSubType.constant);
            }

            return new Lexem(name, LexemType.Variable,
LexemSubType.variable);
        }

        //вторая версия той же функции
        static Lexem FindLexemByName(char name,
```

```
List<Lexem> sourceLangLex)
        {
            string strName = null;
            strName += name;

            return FindLexemByName(strName,
sourceLangLex);
        }

        //функция проверки наличия лексемы с именем
name среди коллекции sourceLangLex
        static bool CheckLexemContaining(char name,
List<Lexem> sourceLangLex)
        {
            string str = null;
            str += name;

            foreach (Lexem lex in sourceLangLex)
            {
                if (str == lex.Content)
                {
                    return true;
                }
            }
            return false;
        }

        //вторая версия той же функции
        static bool CheckLexemContaining(string name,
List<Lexem> sourceLangLex)
```

```
            {
                foreach (Lexem lex in sourceLangLex)
                {
                    if (name == lex.Content)
                    {
                        return true;
                    }
                }
                return false;
            }

            static bool CheckLexemContaining(LexemSubType lst,
List<Lexem> sourceLangLex)
            {
                foreach (Lexem lex in sourceLangLex)
                {
                    if (lst == lex.SubType)
                    {
                        return true;
                    }
                }
                return false;
            }

            //функция поиска лексемы в коллекции по подтипу
        static Lexem FindLexemWithSubType(LexemSubType lst,
List<Lexem> inputLexem)
        {
            foreach (Lexem lex in inputLexem)
            {
```

```
            if (lst == lex.SubType)
                return lex;
        }
        return new Lexem("", LexemType.Variable,
LexemSubType.any);
    }
  }
}
```