МИНОБРНАУКИ РФ

ФГБОУ ВПО Тверской государственный технический университет

Кафедра “Программное обеспечение вычислительной техники”

**Курсовая работа**

**по дисциплине:** «Теория языков программирования и методов трансляции»

**по теме:** «Разработка транслятора для простого языка программирования»

Выполнил: студент группы

«ПИН-1406»

Савельев Александр Владимирович

Проверил:

Калабин Александр Леонидович

Тверь 2017

Содержание

[I. Цель работы 3](#_Toc500376209)

[II. Теоретическая часть 3](#_Toc500376210)

[III. Проектная часть 3](#_Toc500376211)

[1. БНФ реализуемого языка 3](#_Toc500376212)

[2. Список классов лексем реализуемого языка соответственно: 4](#_Toc500376213)

[3. Листинг программы 5](#_Toc500376214)

[4. Тестирование 7](#_Toc500376215)

[IV. Выводы 9](#_Toc500376216)

[V. Список использованной литературы 10](#_Toc500376217)

# Цель работы

Основной целью данной курсовой работы является получение практических навыков, позволяющих разрабатывать трансляторы языков программирования. Составить систему тестов, позволяющую выявить ошибки на этапе разработки и продемонстрировать работу готового приложения. Для разработки использовалась среда разработки Visual Studio 2017 Community Edition и язык программирования C#.

# Теоретическая часть

Транслятор – это программа, выполняющая трансляцию программы, то есть преобразование текста с исходного языка программирования на низкоуровневый язык машинных команд.

Лексический анализ – процесс разбора исходной программы на лексемы.

Синтаксический анализ – процесс сопоставления последовательности лексем с формальной грамматикой. Обычно применяется совместно с лексическим анализатором. Результатом работы будет новое представление исходной программы в постфиксной форме.

Генератор кода – последний этап трансляции. Результат работы – это файл с созданным объектным кодом программы.

В данной курсовой работе выходным языком транслятора является ассемблер. Особенностью его архитектуры является то, что все действия выполняются только над элементами в вершине стека, результаты операций также помещаются в вершину стека. Поэтому в арифметических и логических операциях нет необходимости в указании адреса операндов.

# Проектная часть

## БНФ реализуемого языка

Форма Бекуса-Наура – набор правил, последовательным применением которых можно построить любое предложение.

*Вариант 10. Савельев А.*

<Программа> ::= <Объявление переменных> <Описание вычислений> <Оператор печати>

<Описание вычислений> ::= Begin <Список присваиваний> End

<Объявление переменных> ::= Integer <Список переменных>

<Список переменных> ::= <Идент>; | <Идент> , <Список переменных> <Список присваиваний>::= <Присваивание> |

<Присваивание> <Список присваиваний> <Присваивание> ::= <Идент> := <Выражение> ;

<Выражение> ::= <Ун.оп.> <Подвыражение> | <Подвыражение> <Подвыражение> :: = ( <Выражение> ) | <Операнд> |

< Подвыражение > <Бин.оп.> <Подвыражение>

<Ун.оп.> ::= "-"

<Бин.оп.> ::= "-" | "+" | "\*" | "/"

<Операнд> ::= <Идент> | <Const>

<Идент> ::= <Буква> <Идент> | <Буква>

<Const> ::= <Цифра> <Const> | <Цифра>

<Оператор печати>::=Print <Идент>

<Буква> – буква латинского алфавита (a...z)

<Цифра> – цифра от 0 до 9.

Все переменные должны быть объявлены до начала вычислений. На одной строке может быть только объявление переменных или один оператор присваивания.

Позднее, для расширения возможностей языка были добавлены следующие конструкции:

* WHILE <Выражение> DO <Список операторов> ENDWHILE
* IF<Выражение>THEN<Список операторов>ENDIF

Список классов лексем реализуемого языка соответственно:

В качестве основных используемых абстракций выступают:  
\* «Expression» - Выражение типа “a=b+c”, где левая часть – переменная, а правая вычислимая конструкция.

\* «Condition» - Условие в зависимости от выполнения которого будет или не будет выполняться условие или цикл.

\* «Variable» - Прямая аналогия с переменной из обычного языка. Хранит имя и значение .

\* «IfThenConstruction» - Прямая аналогия с if(){} из языка C#;

\* «WhileCycle» - Прямая аналогия с while(){} из языка C#;

## Листинг программы

В качестве примера приведён метод InsertIntExpression, реализующий запись в числовую переменную результат числового выражения, приведённого к обратной польской записи…

public static void InsertIntExpression(string leftVar, string reversePolishNotationString)

{

for (int i = 0; i < reversePolishNotationString.Length; i++) //Для каждого символа в строке

{

if (Char.IsLetter(reversePolishNotationString[i]))

{

string a = string.Empty;

while (!PolishNotationAnalyzer.IsDelimeter(reversePolishNotationString[i]) && !PolishNotationAnalyzer.IsIntOperator(reversePolishNotationString[i])) //Пока не разделитель

{

a += reversePolishNotationString[i]; //Добавляем

i++;

if (i == reversePolishNotationString.Length) break;

}

AddValueToStack(a);

i--;

}

if (Char.IsDigit(reversePolishNotationString[i]))

{

string a = string.Empty;

while (!PolishNotationAnalyzer.IsDelimeter(reversePolishNotationString[i]) && !PolishNotationAnalyzer.IsIntOperator(reversePolishNotationString[i])) //Пока не разделитель

{

a += reversePolishNotationString[i]; //Добавляем

i++;

if (i == reversePolishNotationString.Length) break;

}

AddValueToStack(a);

i--;

}

else if (PolishNotationAnalyzer.IsIntOperator(reversePolishNotationString[i])) //Если символ - оператор

{

AddNewInstruction("pop bx");

AddNewInstruction("pop ax");

switch (reversePolishNotationString[i])

{

case '+':

AddNewInstruction("add ax, bx");

AddNewInstruction("push ax");

break;

case '-':

AddNewInstruction("sub ax, bx");

AddNewInstruction("push ax");

break;

case '\*':

AddNewInstruction("mul bx");

AddNewInstruction("push ax");

break;

case '/':

AddNewInstruction("cwd");

AddNewInstruction("div bl");

AddNewInstruction("push ax");

break;

}

}

}

AddNewInstruction("pop ax");

AddNewInstruction("mov " + leftVar+ ", ax");

## }

## Тестирование

Контрольные примеры:

1. корректный

*int a,b,c*

*begin*

*a=10*

*b=0*

*c=30*

*if c/a >= 3 Then*

*b = b + 1*

*EndIf*

*if c/a <= 3 Then*

*b = b + 2*

*EndIf*

*if c/a == 3 Then*

*b = b + 5*

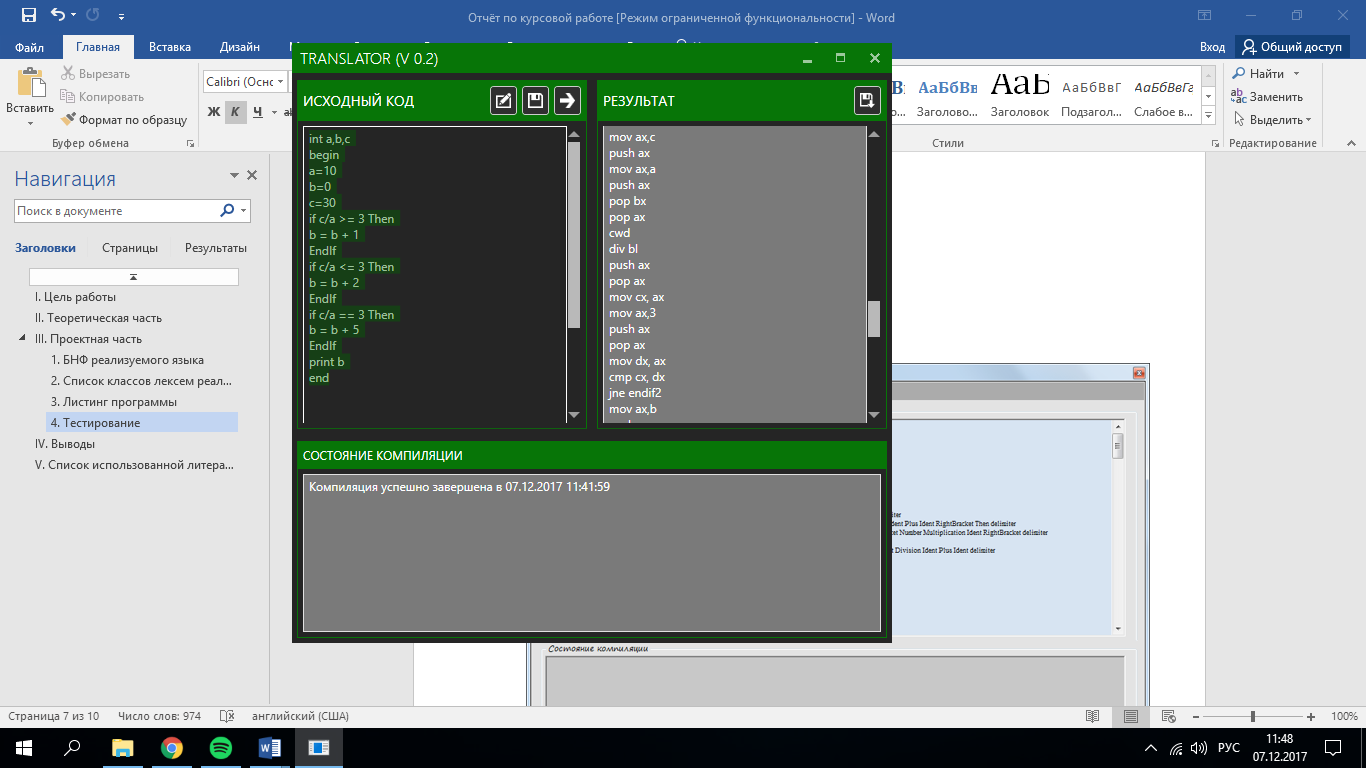
*EndIf*

*print b*

*end*

Пример работы транслятора:

Рис. 1. Вывод результата работы лексического анализатора на корректном примере.



1. некорректный

int a,b,c

begin

a=10

b=0

c=30

if c/a >= 3 Then

b = b + 1

EndIf

if c/a <= 3 Then

b = b + 2

EndIf

if c/a ! 3 Then

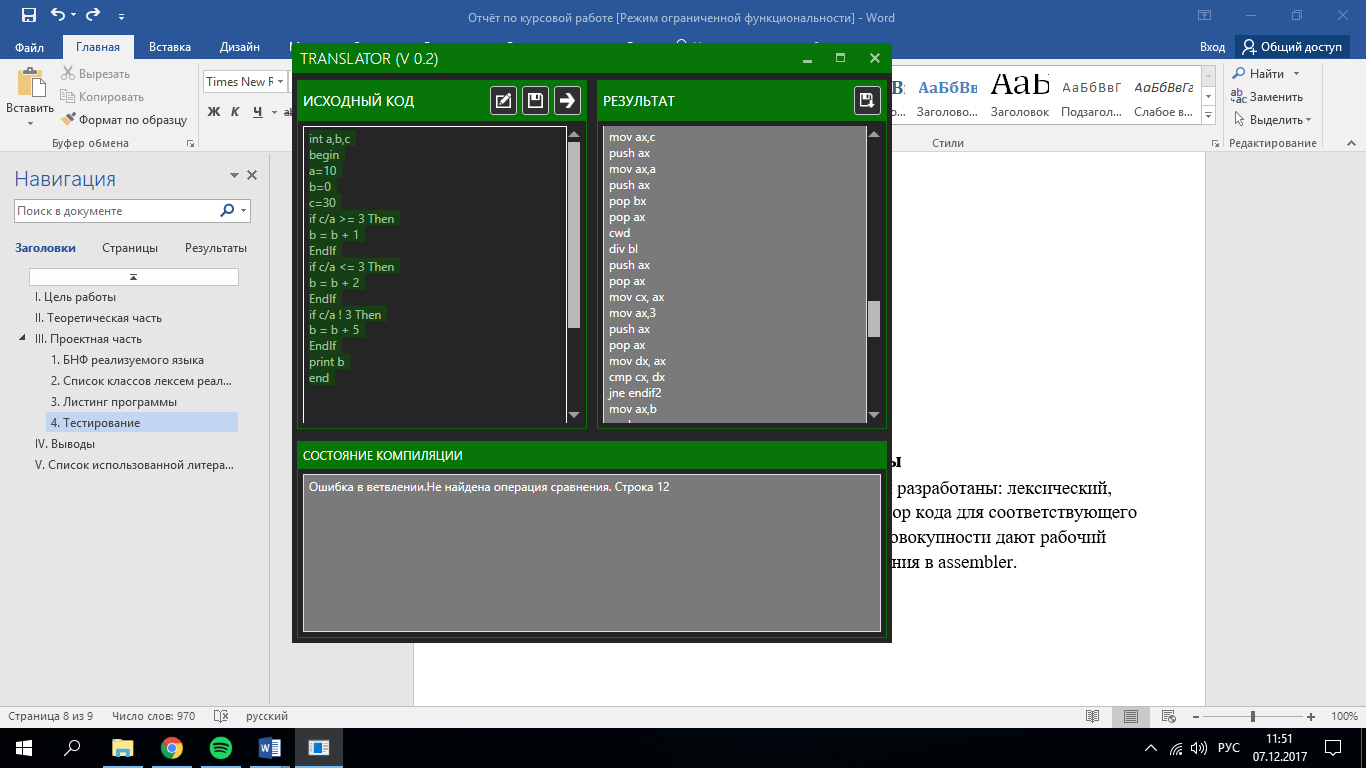
b = b + 5

EndIf

print b

end

Пример работы транслятора:



# Выводы

В ходе работы над курсовым проектом, мною было создано приложение, способное переводить код на простейшем языке программирования в набор команд для языка Assembler. В корневом хранилище проекта содержится набор тестов, иллюстрирующий работоспособность программы и корректную обработку ошибок в родительском коде.

# Список использованной литературы

1. Свердлов С.З. Языки программирования и методы трансляции: Учебное пособие. – СПб.: Питер, 2007. – 638 с.
2. Кревский И.Г., Селиверстов М.Н., Григорьева К.В. Формальные языки, грамматики и основы построения трансляторов: Учебное пособие. / Под ред. Бершадского А.М.. – Пенза: Изд-во Пенз. гос. ун-та, 2002. – 124 с.
3. Омельченко В.П., Демидова А.А. Информатика: учебник. – M.: ЭОТАР-Медиа, 2013. – 384 с.
4. <http://www.codenet.ru/progr/asm/newbee/lesson8.php>
5. <http://www.agpu.net/fakult/ipimif/fpiit/kafinf/umk/el_lib/calc_system/Assembler/guide/Text/Data.htm>
6. <http://natalia.appmat.ru/c&c++/assembler.html>
7. <http://www.tadviser.ru/index.php/%D0%A1%D1%82%D0%B0%D1%82%D1%8C%D1%8F:%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80>