Отчет

Формальные грамматики и методы трансляции

Юлия Кравчук, ПМИ-12-21

2023

Оглавление

[О компиляторе 2](#_Toc153711856)

[Модуль ввода-вывода 3](#_Toc153711857)

[Лексический анализатор 5](#_Toc153711858)

[Синтаксический анализатор 12](#_Toc153711859)

[Семантический анализатор 15](#_Toc153711860)

[Генератор 17](#_Toc153711861)

О компиляторе

Компилятор — это программа, которая переводит программу на языке высокого уровня в эквивалентную программу на другом (объектном) языке.

Структура компилятора:

* Модуль ввода-вывода
* Лексический анализатор
* Синтаксический анализатор
* Семантический анализатор

Анализатор – определение правильности исходной программы и формирование сообщений об ошибках, если они были найдены.

Также в компилятор входит генератор кода. Генератор кода – отображение машинно-независимого промежуточного представления исходной программы на реальную ЭВМ. Генератор является машинно-зависимой частью компилятора, так как преобразует исходную программу в объектный код для конкретной машины с её архитектурой и операционной системой.

Модуль ввода-вывода

Модуль ввода-вывода считывает поочередно литеры исходного кода, соединяет их в лексемы и передает их лексическому анализатору, который в свою очередь определяет их токены. Для удобства работы модуль ввода-вывода запоминает позицию первого символа каждой лексемы и также передает ее лексическому анализатору.

Модуль ввода-вывода считывает лексемы последовательно, обнаружив новую лексему, лексический анализатор сразу же определяет ее токен.

Если при чтении очередной лексемы каким-либо анализатором была обнаружена ошибка, то анализатор передает ее модулю ввода-вывода и прекращает работу программу при необходимости.

Чтение исходного файла происходит в классе Main. Определяя текущую позицию, переходим в класс File\_Work и через метод Get\_Lexeme получаем новую лексему. В данном методе считываем поочередно символы текущей строки, соединяем их в единую лексему, пока не встретится какой-либо разделяющий символ. Если разделяющий символ может являться частью лексемы, то рассматриваем эти случаи отдельно.

Реализация:

class File\_Work

{

string cur\_line;

Position cur\_position;

List<char> separate\_symbols = new List<char> { '<', '>', '=', ':', '+', '-', '\*', '(', ')', '{', '}', '[', ']', ';', ':', ',', '.', ' ', '/', '"', '\t'};

List<char> double\_symbols = new List<char> { '<', '>', '=', ':' };

public File\_Work(string line, Position position)

{

this.cur\_line = line;

this.cur\_position = new Position(position.Get\_Position().Item1, 0);

}

public Position Get\_Position() { return cur\_position; }

public string Get\_Lexeme(Position position)

{

bool flag\_string = false;

string lexeme = string.Empty;

for (int i = position.Get\_Position().Item2; i < cur\_line.Length; i++)

{

cur\_position = new Position(cur\_position.Get\_Position().Item1, cur\_position.Get\_Position().Item2 + 1);

if (!separate\_symbols.Contains(cur\_line[i]) || (flag\_string && cur\_line[i] != '"')) lexeme += cur\_line[i];

else

{

// token type => float

if (cur\_line[i] == '.' && int.TryParse(lexeme, out \_))

{

lexeme += cur\_line[i];

continue;

}

// token type => string

if (!flag\_string && cur\_line[i] == '"')

{

flag\_string = true;

lexeme += cur\_line[i];

continue;

}

if (flag\_string && cur\_line[i] == '"')

{

lexeme += cur\_line[i];

return lexeme;

}

// some lexeme

if (lexeme.Length > 0)

{

cur\_position = new Position(cur\_position.Get\_Position().Item1, cur\_position.Get\_Position().Item2 - 1);

if (Char.IsDigit(lexeme[0]) && Char.IsDigit(lexeme[lexeme.Length - 1]) && lexeme.Contains('.')) // float

lexeme = lexeme.Replace('.', ',');

return lexeme;

}

// move to next lexeme

if (cur\_line[i] == ' ' || cur\_line[i] == '\t') break;

// lexeme from double operator

if (i + 1 < cur\_line.Length && double\_symbols.Contains(cur\_line[i]) && double\_symbols.Contains(cur\_line[i + 1]))

{

cur\_position = new Position(cur\_position.Get\_Position().Item1, cur\_position.Get\_Position().Item2 + 1);

return cur\_line[i].ToString() + cur\_line[i + 1].ToString();

}

// single character lexeme

return cur\_line[i].ToString();

}

}

return lexeme; // some lexeme

}

}

Лексический анализатор

Лексический анализатор (класс Lexer) получает лексему, ее позицию и файл для записи в него возможной ошибки. В методе Get\_Token происходит определение токена данной лексемы.

Токен – это минимальная единица синтаксиса языка программирования. Код программы состоит из токенов, которые представляют ключевые слова (специальные символы, операторы и т.д.), идентификаторы (имена переменных, функций и т.д.), константы (значения переменных и их тип). Для каждого вида токена (в том числе, неопределенного типа) создан отдельный класс. При обнаружении неопределенного типа будет выведена ошибка.

Через методы Is\_KeyWord, Is\_Constant и Is\_Identifier определяем вид токена. Кроме основного вида токена, у таких токенов, как ключевые слова и константы, есть еще свои подтипы. Подтипы констант определяются тут же в Lexer, а подтипы ключевых слов в их классе в методе Get\_KeyWord.

Реализация:

internal class Lexer

{

string lexeme;

Position position;

StreamWriter writer;

public Lexer(string lexeme, Position position, StreamWriter writer)

{

this.lexeme = lexeme;

this.position = position;

this.writer = writer;

}

private bool Is\_Identifier()

{

return lexeme.All(x => (x >= 65 && x <= 90) || (x >= 97 && x <= 122) || (x >= 48 && x <= 57) || x == 95)

&& !(lexeme[0] >= 48 && lexeme[0] <= 57);

}

public bool Is\_Constant(ref Const\_type type)

{

if (lexeme.StartsWith("\"") && lexeme.EndsWith("\""))

{

type = Const\_type.STRING; return true;

}

if (lexeme == "True" || lexeme == "False" || lexeme == "true" || lexeme == "false")

{

type = Const\_type.BOOLEAN; return true;

}

if (lexeme.All(x => char.IsDigit(x)))

{

type = Const\_type.INTEGER; return true;

}

if (double.TryParse(lexeme, out \_))

{

type = Const\_type.FLOAT; return true;

}

return false;

}

public Token Get\_Token()

{

if (KeyWord.Is\_KeyWord(lexeme)) return new KeyWord(KeyWord.Get\_KeyWord(lexeme), position);

Const\_type const\_Type = Const\_type.STRING;

if (Is\_Constant(ref const\_Type)) return new Constant(const\_Type, lexeme, position);

if (Is\_Identifier()) return new Identifier(lexeme, position);

writer.WriteLine(new Error("Unrecoghized characters", position, lexeme, "Синтаксическая ошибка"));

return new Unknown(lexeme, position);

}

}

**Тестирование:**

1. Правильные входные данные:

Входной файл:

program name;

const pi := 3.14;

var a, b: float;

c, i: integer;

d: string;

e, f: boolean;

begin

a := 5.2;

b := 9.4 + 8.7 - 9.2;

c := 8 + 8 \* (7 + 7);

d := "stroka";

e := true;

f := a <= b;

if c mod 2 = 0 then

write("c - even number")

else write("c - odd number");

for i := 1 to 5 do

write("\*");

end.

Выходной файл:

lexeme: program

token: KEYWORD (PROGRAM) -> (1, 1)

lexeme: name

token: IDENTIFIER -> (1, 9)

lexeme: ;

token: KEYWORD (SEMICOLON) -> (1, 13)

lexeme: const

token: KEYWORD (CONST) -> (2, 1)

lexeme: pi

token: IDENTIFIER -> (2, 7)

lexeme: :=

token: KEYWORD (ASSIGN) -> (2, 10)

lexeme: 3,14

token: CONST (FLOAT) -> (2, 13)

lexeme: ;

token: KEYWORD (SEMICOLON) -> (2, 17)

lexeme: var

token: KEYWORD (VAR) -> (3, 1)

lexeme: a

token: IDENTIFIER -> (3, 5)

lexeme: ,

token: KEYWORD (COMMA) -> (3, 6)

lexeme: b

token: IDENTIFIER -> (3, 8)

lexeme: :

token: KEYWORD (COLON) -> (3, 9)

lexeme: float

token: KEYWORD (FLOAT) -> (3, 11)

lexeme: ;

token: KEYWORD (SEMICOLON) -> (3, 16)

lexeme: c

token: IDENTIFIER -> (4, 5)

lexeme: ,

token: KEYWORD (COMMA) -> (4, 6)

lexeme: i

token: IDENTIFIER -> (4, 8)

lexeme: :

token: KEYWORD (COLON) -> (4, 9)

lexeme: integer

token: KEYWORD (INTEGER) -> (4, 11)

lexeme: ;

token: KEYWORD (SEMICOLON) -> (4, 18)

lexeme: d

token: IDENTIFIER -> (5, 5)

lexeme: :

token: KEYWORD (COLON) -> (5, 6)

lexeme: string

token: KEYWORD (STRING) -> (5, 8)

lexeme: ;

token: KEYWORD (SEMICOLON) -> (5, 14)

lexeme: e

token: IDENTIFIER -> (6, 5)

lexeme: ,

token: KEYWORD (COMMA) -> (6, 6)

lexeme: f

token: IDENTIFIER -> (6, 8)

lexeme: :

token: KEYWORD (COLON) -> (6, 9)

lexeme: boolean

token: KEYWORD (BOOLEAN) -> (6, 11)

lexeme: ;

token: KEYWORD (SEMICOLON) -> (6, 18)

lexeme: begin

token: KEYWORD (BEGIN) -> (7, 1)

lexeme: a

token: IDENTIFIER -> (8, 5)

lexeme: :=

token: KEYWORD (ASSIGN) -> (8, 7)

lexeme: 5,2

token: CONST (FLOAT) -> (8, 10)

lexeme: ;

token: KEYWORD (SEMICOLON) -> (8, 13)

lexeme: b

token: IDENTIFIER -> (9, 5)

lexeme: :=

token: KEYWORD (ASSIGN) -> (9, 7)

lexeme: 9,4

token: CONST (FLOAT) -> (9, 10)

lexeme: +

token: KEYWORD (PLUS) -> (9, 14)

lexeme: 8,7

token: CONST (FLOAT) -> (9, 16)

lexeme: -

token: KEYWORD (MINUS) -> (9, 20)

lexeme: 9,2

token: CONST (FLOAT) -> (9, 22)

lexeme: ;

token: KEYWORD (SEMICOLON) -> (9, 25)

lexeme: c

token: IDENTIFIER -> (10, 5)

lexeme: :=

token: KEYWORD (ASSIGN) -> (10, 7)

lexeme: 8

token: CONST (INTEGER) -> (10, 10)

lexeme: +

token: KEYWORD (PLUS) -> (10, 12)

lexeme: 8

token: CONST (INTEGER) -> (10, 14)

lexeme: \*

token: KEYWORD (MULTI) -> (10, 16)

lexeme: (

token: KEYWORD (LPAR) -> (10, 18)

lexeme: 7

token: CONST (INTEGER) -> (10, 19)

lexeme: +

token: KEYWORD (PLUS) -> (10, 21)

lexeme: 7

token: CONST (INTEGER) -> (10, 23)

lexeme: )

token: KEYWORD (RPAR) -> (10, 24)

lexeme: ;

token: KEYWORD (SEMICOLON) -> (10, 25)

lexeme: d

token: IDENTIFIER -> (11, 5)

lexeme: :=

token: KEYWORD (ASSIGN) -> (11, 7)

lexeme: "stroka"

token: CONST (STRING) -> (11, 10)

lexeme: ;

token: KEYWORD (SEMICOLON) -> (11, 18)

lexeme: e

token: IDENTIFIER -> (12, 5)

lexeme: :=

token: KEYWORD (ASSIGN) -> (12, 7)

lexeme: true

token: CONST (BOOLEAN) -> (12, 10)

lexeme: ;

token: KEYWORD (SEMICOLON) -> (12, 14)

lexeme: f

token: IDENTIFIER -> (13, 5)

lexeme: :=

token: KEYWORD (ASSIGN) -> (13, 7)

lexeme: a

token: IDENTIFIER -> (13, 10)

lexeme: <=

token: KEYWORD (LESS\_EQUAL) -> (13, 12)

lexeme: b

token: IDENTIFIER -> (13, 15)

lexeme: ;

token: KEYWORD (SEMICOLON) -> (13, 16)

lexeme: if

token: KEYWORD (IF) -> (15, 5)

lexeme: c

token: IDENTIFIER -> (15, 8)

lexeme: mod

token: KEYWORD (MOD) -> (15, 10)

lexeme: 2

token: CONST (INTEGER) -> (15, 14)

lexeme: =

token: KEYWORD (EQUAL) -> (15, 16)

lexeme: 0

token: CONST (INTEGER) -> (15, 18)

lexeme: then

token: KEYWORD (THEN) -> (15, 20)

lexeme: write

token: IDENTIFIER -> (16, 9)

lexeme: (

token: KEYWORD (LPAR) -> (16, 14)

lexeme: "c - even number"

token: CONST (STRING) -> (16, 15)

lexeme: )

token: KEYWORD (RPAR) -> (16, 32)

lexeme: else

token: KEYWORD (ELSE) -> (17, 5)

lexeme: write

token: IDENTIFIER -> (17, 10)

lexeme: (

token: KEYWORD (LPAR) -> (17, 15)

lexeme: "c - odd number"

token: CONST (STRING) -> (17, 16)

lexeme: )

token: KEYWORD (RPAR) -> (17, 32)

lexeme: ;

token: KEYWORD (SEMICOLON) -> (17, 33)

lexeme: for

token: KEYWORD (FOR) -> (19, 5)

lexeme: i

token: IDENTIFIER -> (19, 9)

lexeme: :=

token: KEYWORD (ASSIGN) -> (19, 11)

lexeme: 1

token: CONST (INTEGER) -> (19, 14)

lexeme: to

token: KEYWORD (TO) -> (19, 16)

lexeme: 5

token: CONST (INTEGER) -> (19, 19)

lexeme: do

token: KEYWORD (DO) -> (19, 21)

lexeme: write

token: IDENTIFIER -> (20, 9)

lexeme: (

token: KEYWORD (LPAR) -> (20, 14)

lexeme: "\*"

token: CONST (STRING) -> (20, 15)

lexeme: )

token: KEYWORD (RPAR) -> (20, 18)

lexeme: ;

token: KEYWORD (SEMICOLON) -> (20, 19)

lexeme: end

token: KEYWORD (END) -> (22, 1)

lexeme: .

token: KEYWORD (POINT) -> (22, 4)

1. Неправильные входные данные (добавим несколько ошибок):

Входной файл:

program name;

const 1pi := 3.14; {неправильное имя переменной}

var a, b: float;

c, i: integer;

d: string;

e, f: boolean;

begin

a := 5.2;

b := 9.4 + 8.7 - 9.2;

c := 8 + 8 \* (7 + 7);

d := "stroka; {нет ожидаемого символа}

e := true!; {неверный лишний символ}

f := a <= b;

if c mod 2 = 0 then

write("c - even number")

else write("c - odd number");

for i := 1 to 5 do

write("\*");

end.

Выходной файл:

!!! Error: Unrecoghized characters (Синтаксическая ошибка) -> (2, 7) -> 1pi

lexeme: 1pi

token: UNKNOWN -> (2, 7)

!!! Error: Unrecoghized characters (Синтаксическая ошибка) -> (11, 10) -> "stroka;

lexeme: "stroka;

token: UNKNOWN -> (11, 10)

!!! Error: Unrecoghized characters (Синтаксическая ошибка) -> (12, 10) -> true!

lexeme: true!

token: UNKNOWN -> (12, 10)

Синтаксический анализатор

Синтаксический анализатор (Syntexer) — это программа или подпрограмма, производящая сопоставление линейной последовательности лексем формального (языка программирования, регулярных выражений и т.д.) или естественного языка его формальной грамматике.

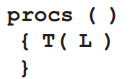
В информатике форма Бэкуса–Наура или нормальная форма Бэкуса (BNF) — это метасинтаксическая нотация для контекстно-свободных грамматик, часто используемая для описания синтаксиса языков, используемых в вычислительной технике, таких как языки программирования, форматы документов, наборы команд и протоколы связи.

Все правила, описывающие синтаксис языка, имеют вид:

,

где S — обозначение синтаксической конструкции, а L может состоять из символов языка, синтаксических конструкций и метасимволов «|» и «{ }».

Основная идея написания синтаксического анализатора заключается в следующем: для каждого правила необходимо описать функцию, тело которой является результатом некоторого преобразования правой части этого правила:



Используемые БНФ в моем коде:

<программа>::= program <имя> <блок>;

<имя>::=<буква>{<буква>|<цифра>}

<блок>::=<раздел переменных><раздел операторов>

<раздел переменных>::= var{<описание однотипных переменных>};

<описание однотипных переменных>::=<имя>{,<имя>}:<тип>

<раздел операторов>::=<составной оператор>

<составной оператор>::= begin <оператор>{;<оператор>} end

<оператор>::=<простой оператор>| <сложный оператор>

<сложный оператор>::=<составной оператор>

<простой оператор>::=<оператор присваивания>|<пустой оператор>

<оператор присваивания>::=<переменная>:=<выражение>

<выражение>::=<простое выражение>

<простое выражение>::=<знак><слагаемое>{<аддитивная операция><слагаемое>}

<аддитивная операция>::=+ | -

<слагаемое>::=<множитель>{<мультипликативная операция><множитель>}

<мультипликативная операция>::=\* | DIV | MOD

<множитель>::= <переменная>|<константа без знака>(<выражение>)

< условный оператор > ::= if < логическое выражение > then < составной оператор >

| if < логическое выражение > then < составной оператор >

else < составной оператор >

< логическое выражение > ::= | < отношение>

< простое логическое выражение> ::= < логическое слагаемое >

{ or < логическое слагаемое >}

< логическое слагаемое >::= < логический множитель > | < логическое слагаемое > |

and < логический множитель >

< логический множитель >:=< константа логического типа > |   
< переменная логического типа >

< отношение > ::= < скалярное отношение > | < строковое отношение >

< скалярное отношение > ::= < арифметическое выражение >< операция сравнения >

< арифметическое выражение > | < простое логическое

выражение >< операция сравнения >

< простое логическое выражение >

< строковое отношение > ::= < строковая константа >< операция сравнения >

< строковая константа > | < переменная >< операция

сравнения > < строковая константа > | < строковая

константа >< операция сравнения >< переменная > |

< переменная >< операция сравнения >< переменная >

< операция сравнения > ::= = | <> | <= | < | >= | >

< оператор цикла с предусловием > ::= while < логическое выражение > do

<составной оператор>

Пример кода для БНФ <программа>:

public void Program()

{

if (!Accept(KeyWords.PROGRAM)) Print\_Error("Uncorrect program start", "Syntax error");

else

{

Next\_Token();

if (!Accept(Token\_type.IDENTIFIER)) Print\_Error("Uncorrect program name", "Syntax error");

Next\_Token();

if (!Accept(KeyWords.SEMICOLON)) Print\_Error("Not found semicolon after program name", "Syntax error");

Next\_Token();

if (Accept(KeyWords.VAR)) Var();

Block();

Next\_Token();

if (!Accept(KeyWords.POINT)) Print\_Error("Not found point after block program", "Syntax error");

}

}

О реализации:

Через метод Accept (в трех перегрузках) проверяю, является ли текущий токен ожидаемым.

private bool Accept(List<KeyWords> keys)

{

return token is KeyWord key && keys.Contains(key.Get\_Type\_KeyWord());

}

private bool Accept(Token\_type token\_type)

{

return token.Get\_Type() == token\_type;

}

private bool Accept(KeyWords key)

{

return token is KeyWord word && word.Get\_Type\_KeyWord() == key;

}

Через метод Const\_type определяю тип созданной ранее переменной.

private Const\_type Const\_Type()

{

return Constant.Get\_Const\_Type(((KeyWord)token).Get\_Type\_KeyWord());

}

Нейтрализация – действия, которые позволяют продолжить анализ после обнаружения ошибки, пропустив несколько символов. Для этого при обнаружении простой ошибки (например, пропуск точки с запятой, несовпадение типов, использование неинициализированной переменной), выводим соответствующее сообщение и продолжаем анализ кода, будто данный ошибки не было. Если же ошибка является более сложной (например, неправильное слово program или начало составного блока), то компилятор прекращает свою работу.

Тестирование проведем вместе с работой семантического анализатора.

Семантический анализатор

Семантический анализатор (Semanter) проверяет программу на соответствие неформальным правилам языка. Неформальные правила задаются с помощью естественного языка и называются контекстными условиями. Многие контекстные условия связаны с идентификаторами и их использованием.

Этот анализатор проверяет уникальность идентификаторов и соответствие их типов, то есть нельзя записать в строковую переменную число и т.д.

Таблица идентификаторов – структура, в которой хранится имя идентификатора и его тип.

Когда синтаксический анализатор обрабатывает раздел переменных и констант семантический записывает имена этих констант и их тип в таблицу идентификаторов, если имя уже существует выдается сообщение об ошибке.

При анализе основной программы семантический анализатор проверяет, чтобы все переменные уже существовали и каждой переменной присваивалось значение ее типа.

Тестирование:

1. Входной файл (с синтаксическими ошибками):

program name;

var a, b: float {нет точки с запятой}

c, b: integer; {объявление уже существующей переменной}

d: string;

e, f: boolean;

begin

a := 5.2;

b := 9.4 + 8.7 - 9;

c := 8 + a \* (7.4 + 7; {нет закрывающейся скобки}

d := "stroka" + "str2";

f := a <= b and e; {использование неинициализированной переменной}

if c mod 2 = 0 then

begin

a := a \* a;

b := b + "stroka"; {несоответствие типов в выражении}

end

else

begin

e := e = f; {использование неинициализированной переменной}

f := 6; {несоответствие типов в выражении}

end;

while i := c do{использование необъявленной переменной и неправильная запись условия}

begin

d := 5; {несоответствие типов в выражении}

end;

end.

1. Выходной файл (найденные ошибки):

!!! Error: Not found semicolon after variable type (Syntax error) -> (3, 6) -> c

!!! Error: Variable is already defined (Semantic error) -> (3, 9) -> b

!!! Error: Not found closing bracket (Syntax error) -> (9, 27) -> ;

!!! Error: Using a variable without a value (Semantic error) -> (11, 22) -> e

!!! Error: Type mismatch in the expression (Semantic error) -> (15, 26) -> "stroka"

!!! Error: Using a variable without a value (Semantic error) -> (19, 15) -> e

!!! Error: Type mismatch in the expression (Semantic error) -> (20, 15) -> 6

!!! Error: Not found variable definition (Syntax error) -> (22, 12) -> i

!!! Error: Not found do after while expression (Syntax Error) -> (22, 15) -> :=

!!! Error: Type mismatch in the expression (Semantic error) -> (24, 15) -> 5

Генератор

Генератор (Generator) – это инструмент, который автоматически создает исходный код компилятора по описанию грамматики языка.

Основной принцип работы генераторов компиляторов основан на использовании грамматики языка программирования. Грамматика определяет правила синтаксиса языка и позволяет задать все возможные комбинации лексем и их связи. Генератор компиляторов на основе грамматики создает код, который может анализировать и преобразовывать входную программу в соответствии с заданными правилами.

Другими словами, происходит преобразование исходного кода на языке Pascal в выходной файл с кодом на языке C#. После успешного перевода одного языка в другой создается файл в формате exe для выполнения написанной программы в исходном коде.

Реализация (часть кода из класса Generated):

В данном примере представлена реализация перевода блока VAR, и функций ввода-вывода.

public static void Traslate\_VAR()

{

foreach (var variable in Semanter.Get\_Variables())

{

string lexeme = variable.Value.ToString();

string new\_lexeme = string.Empty;

switch (lexeme)

{

case "STRING": new\_lexeme = lexeme.ToLower(); break;

case "INTEGER": new\_lexeme = "int"; break;

case "BOOLEAN": new\_lexeme = "bool"; break;

case "FLOAT": new\_lexeme = "double"; break;

}

cur\_csharp\_code += new\_lexeme + " " + variable.Key + ";\n\t\t\t"; // value - type, key - name

}

}

// input variable value

public static void Add\_Readline(string lexeme, Const\_type const\_)

{

cur\_csharp\_code += $"\n\t\t\t{lexeme} = ";

switch (const\_)

{

case Const\_type.STRING: cur\_csharp\_code += $"Console.ReadLine();\n"; break;

case Const\_type.INTEGER: cur\_csharp\_code += $"int.Parse(Console.ReadLine());\n"; break;

case Const\_type.FLOAT: cur\_csharp\_code += $"double.Parse(Console.ReadLine());\n"; break;

}

Semanter.New\_Assignment(lexeme, "");

}

// output variable value

public static void Add\_Writeline(string lexeme)

{

cur\_csharp\_code += $"\n\t\t\tConsole.WriteLine({lexeme}" + (lexeme.EndsWith(")") ? ";" : ");");

}:

Реализация (часть кода из класса Main):

В данном примере представлено создание файла с расширением cs (код на языке C#) и компиляция выходного файла с расширением exe.

try

{

if (Syntaxer.Count\_Errors > 0) throw new Exception("Errors found. Read output txt file!");

Generator.Translate\_Pascal\_To\_CSharp();

Console.WriteLine("\nCreating cs file...");

File.WriteAllLines(csharp\_path, new[] { Generator.CSharp\_code });

Console.WriteLine("\nCompiling...");

CSharpCodeProvider provider = new CSharpCodeProvider();

CompilerResults results = provider.CompileAssemblyFromSource(new CompilerParameters(new string[0], exe\_path) { GenerateExecutable = true }, Generator.CSharp\_code);

if (results.Errors.Count > 0) throw new Exception("Error in the created cs file!");

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("\nDone!");

}

catch (Exception ex)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine("\n" + ex.Message);

}

finally

{

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine("\nPress any key");

Console.ReadKey();

}

При обнаруженных ошибках в исходном коде после выполнения работы синтаксического и семантического анализаторов, генератор даже не начнет свою работу. И в случае неправильного перевода программы также выдаст ошибку.

Тестирование:

1. Входной файл:

program name;

var a, b: float;

c, i: integer;

d: string;

e, f: boolean;

begin

writeln("Write a:"); {вывод стринговой константы на экран}

readln(a); {ввод значения переменной с клавиатуры}

b := 8 + a \* (7.4 + 7); {присваивание переменной значение мат. выражения}

writeln("Write d:"); {вывод стринговой константы на экран}

readln(d); {ввод значения переменной с клавиатуры}

d := "stroka" + d; {присваивание переменной значение текст. выражения}

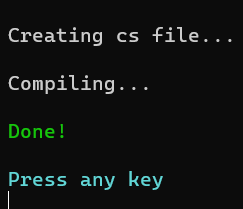
writeln("\nResult:"); {вывод стринговой константы на экран}

writeln(b); {вывод значения идентификатора на экран}

writeln(d); {вывод значения идентификатора на экран}

end.

1. Выходные файлы:
2. Вывод в консоли после выполнения работы всего компилятора:



1. Файл с расширением cs:

using System;

namespace Compiler

{

class Program

{

static void Main()

{

double a;

double b;

int c;

int i;

string d;

bool e;

bool f;

Console.WriteLine("Write a:");

a = double.Parse(Console.ReadLine());

b = 8+a\*(7.4+7);

Console.WriteLine("Write d:");

d = Console.ReadLine();

d = "stroka"+d;

Console.WriteLine("\nResult:");

Console.WriteLine(b);

Console.WriteLine(d);

Console.ReadKey();

}

}

}

1. Файл с расширением exe:

