|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Министерство науки и высшего образования РФ | | | | | | | | |
|  | | |  | | |  | | |
| Пермский государственный национальный  исследовательский университет | | | | | | | | |
|  | | |  | | |  | | |
|  | | ОТЧЁТ  по лабораторной работе «Разработка компилятора языка Pascal»  по дисциплине «Формальные грамматики и методы трансляции» | | | | |  | |
|  | | |  | | |  | | |
|  | Работу выполнил  студент гр. ПМИ-1,2  Валеев Р. Р. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись)  «\_\_» \_\_\_\_\_\_\_\_ 2021 | | |  | Проверил  асис.кафедры МОВС  Пономарев Ф.А. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись)  «\_\_» \_\_\_\_\_\_\_ 2021 | | |  |
|  |  | | |  |  | | |  |
| Пермь 2021 | | | | | | | | |

# Анализ

Для решения глобальной задачи – написание компилятора для подмножества языка Pascal, требуется решить несколько подзадач, а именно, спроектировать и разработать лексический, синтаксический и семантический анализаторы, а также генератор кода.

Последним этапом станет разработка генератора кода. Генерация кода – это процесс компиляции, конвертирующий синтаксически корректную программу в последовательность инструкций, которые могут выполняться на машине. В ходе данной работы предстоит написать транслятор, выполняющий перевод исходной программы в промежуточной язык Microsoft Intermediate Language (MSIL). Код, написанный на MSIL, будет преобразовываться в набор машинных команд во время выполнения программы.

Среда CLR (Common Language Runtime) – общеязыковая исполняющая среда, которая отвечает за выполнение кода, написанного на языке программирования С# или другом языке платформы Microsoft .NET Framework. Перед исполнением программы выполняется трансляция исходного кода в промежуточный язык MSIL. Язык MSIL – это независимый от процессора набор инструкций, который можно эффективно преобразовать в машинный код. Вместе с кодом на промежуточном языке создаются метаданные, которые позволяют сразу использовать созданный переносимый исполняемый файл (PE).

При проектировании и разработке генератора кода незаменимым для меня источником был сайт docs.microsoft.com. Также я познакомился с книгой Expert .NET 2.0 IL Assembler, Serge Lidin., написанной на английском языке. В ней достаточно подробно рассказывается о тонкостях программирования на языке MSIL.

# Проектирование

Перед началом разработки генератора кода было проведено начальное проектирование. На рисунке ниже представлена диаграмма классов, отражающая обновлённую структуру компилятора

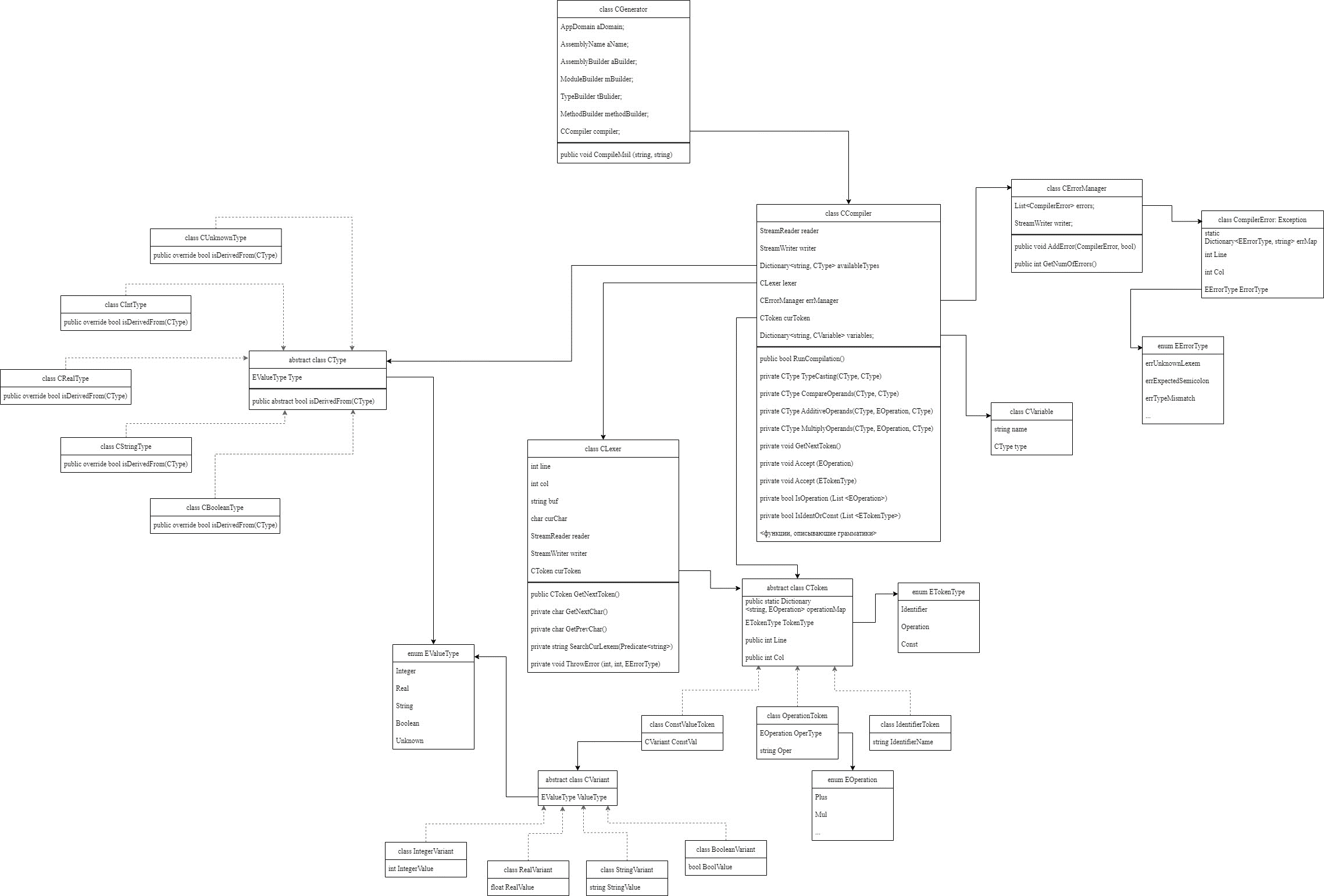


Рисунок 1 – Итоговая диаграмма классов

Будет добавлен новый класс CGenerator, который станет главным в иерархии. Он отвечает за запуск процесса компиляции и создание динамической сборки, которая будет сохранена на диске. Сборка создаётся в виде исполняемого файла (.exe) и содержит в себе модуль, в котором будет записан код на MSIL, и манифест (служебная информация о сборке).

Язык MSIL основан на использовании виртуального стека. Имеется набор кодов операций, называемых opcodes, при помощи которых осуществляются действия с верхними значениями стека. Например, можно поместить значение в стек, переместить значение из стека в память, удалить самое верхнее значение стека и т.д.

Действия по генерации будут происходить в классе компилятора параллельно с работой синтаксического и семантического анализаторов. Класс ILGenerator используется для генерации IL-кодов. Экземпляр этого класса передаётся в качестве одного из параметров классу CCompiler. Таким образом появляется возможность создавать инструкции языка MSIL внутри методов, осуществляющих разбор синтаксический конструкций.

# Разработка

На начальном этапе разработки возникла проблема, связанная с тем, что не все методы из System.Reflection.Emit доступны на платформе .NET Core. Поэтому было принято решение перевести код на .NET Framework.

Рассмотрим процесс создания динамической сборки:

aDomain = Thread.GetDomain();

aName = new AssemblyName("Compiler");

aName.Version = new Version("1.0");

aBuilder = aDomain.DefineDynamicAssembly(aName, AssemblyBuilderAccess.RunAndSave, Directory.GetCurrentDirectory());

mBuilder = aBuilder.DefineDynamicModule(aName.Name, aName.Name + ".exe");

tBulider = mBuilder.DefineType("Compiler", TypeAttributes.Public);

methodBuilder = tBulider.DefineMethod("Main", MethodAttributes.Public | MethodAttributes.Static, typeof(void), new Type[] { });

/\* создание ILGenerator \*/

il = methodBuilder.GetILGenerator();

/\* установить точка входа \*/

aBuilder.SetEntryPoint(methodBuilder, PEFileKinds.ConsoleApplication);

сompiler = new CCompiler(InFilePath, OutFilePath, il);

/\* запуск процесса компиляции \*/

bool compIsOver = сompiler.RunCompilation();

il.Emit(OpCodes.Ret);

/\* сохранение сборки, если во время компиляции не было ошибок \*/

if (compIsOver)

{

tBulider.CreateType();

aBuilder.Save(aName.Name + ".exe");

}

В этом фрагменте кода создаётся однофайловая сборка, включающая в себя модуль, содержащий тип (класс) Compiler, в котором объявлен метод Main. В теле этого метода представлены инструкции на языке MSIL, сгенерированные после анализа исходного кода программы. Для создания инструкций в методе Main создаётся объект il класса ILGenerator. Команды помещаются на стек с помощью вызова метода Emit для генератора MSIL. Чтобы создать исполняемый файл, вызывается метод SetEntryPoint, который определяет точку входа в сборку и запрашивает создание консольного приложения. Динамические модули в сборке сохраняются при сохранении сборки с помощью метода Save. Используя утилиту ildasm.exe, можно посмотреть содержимое сборки:

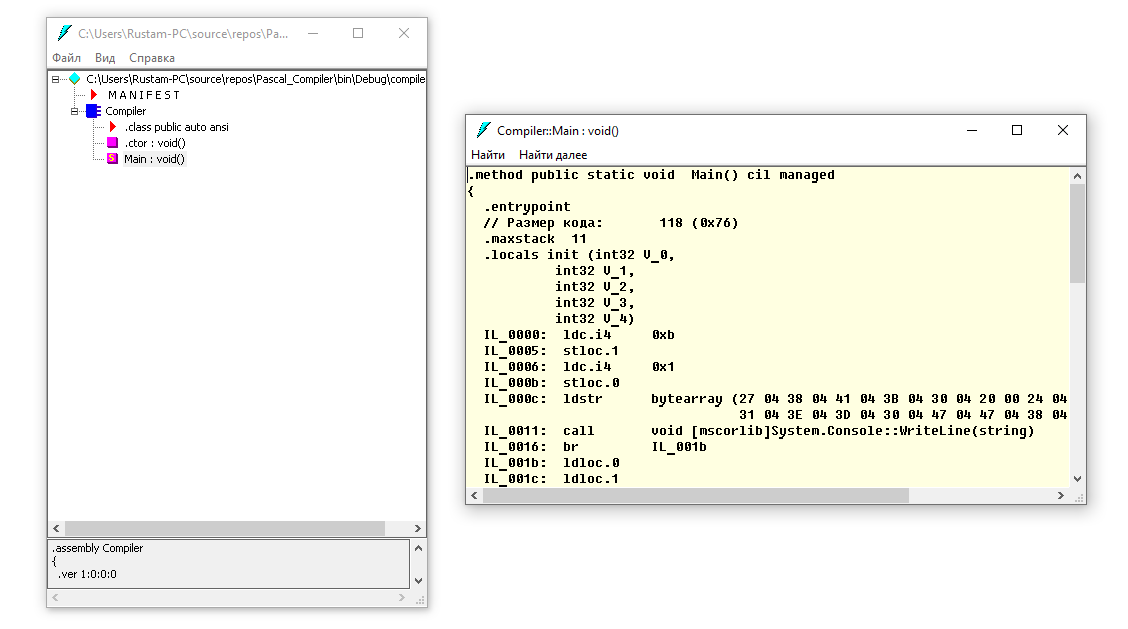


Рисунок – Содержимое сборки

Рассмотрим действия по генерации кода, связанные с объявлением локальных переменных. Сначала требуется задать начало и конец лексической области. Это осуществляется путём вызова методов BeginScope() и EndScope() класса ILGenerator. Начало и конец лексической области совпадают с началом и завершением анализа конструкции *<раздел переменных>*. Объявление локальной переменной происходит следующим образом:

variables[varName].Lb = il.DeclareLocal(typeof(int));

Для хранения информации, связанной с генерацией кода, в класс Variables добавлено поле Lb, в котором храниться объект созданной переменной. В качестве примера приведено создание целочисленной переменной.

Перейдём к описанию действий по генерации кода, позволяющих вычислять выражения. В метод Factor, осуществляющий разбор множителя, были добавлены инструкции, осуществляющие загрузку на вершину стека значений целочисленных, вещественных, строковых и логических констант (в виде целочисленных констант 0 и 1). Для получения значения переменной используется инструкция с кодом OpCodes.Ldloc, которая позволяет поместить значение локальной переменной в стек. Также метод Factor теперь принимает два входных параметра: CType prevType – тип предыдущего множителя и bool floatDivision – флаг, указывающий, что операцией является деление (‘/’). Эти параметры определяют необходимость преобразования верхнего значения в стеке в вещественный тип. Рассмотрим случаи, когда требуется приведение одного типа к другому:

Таблица 1 – Описание ситуаций, требующих неявного приведения типов

|  |  |  |  |
| --- | --- | --- | --- |
| Тип предыдущего операнда | Тип рассматриваемого операнда | Операция деление (‘/’) | Действие |
| int32 | int32 | true | привести предыдущий операнд к типу float64; поместить рассматриваемый операнд в стек; привести к типу float64 |
| int32 | float64 | true/false | привести предыдущий операнд к типу float64 и поместить рассматриваемый операнд в стек |
| float64 | int32 | true/false | поместить рассматриваемый операнд в стек и привести к типу float64 |

В методе Term выполняется генерация инструкций для осуществения одной из мультипликативных операций над двумя верхними значениями стека. Например, действия по генерации кода для операции ‘\*’ выглядят следующим образом:

if (operation == EOperation.Mul)

il.Emit(OpCodes.Mul);

Аналогично в метод SimpleExpression добавляются команды для генерации инструкций, позволяющих выполнить аддитивную операцию над элементами стека. Отдельно стоит отметить случай, когда выполняется сложение двух строк. Код операции OpCodes.Add используется для создания инструкции сложения двух числовых значений, но не строк. Поэтому конкатенация строк осуществляется с помощью вызова метода Concat класса String:

il.Emit(OpCodes.Call, typeof(string).GetMethod("Concat", new Type[] { typeof(string), typeof(string) }));

Используется перегрузка, принимающая два строковых значения и возвращающая объединённую строку, которая будет помещена на вершину стека.

В методе Expression может осуществляться сравнение двух значений. Операции сравнения «меньше», «больше» и «равно» имеют свои коды операций, однако кодов для операций «больше или равно», «меньше или равно» и «не равно» в языке MSIL нет. Эти операции можно реализовать через отрицание результата одной из имеющихся операций сравнения.

Метод AssignmentStatement анализирует конструкцию *<оператор присваивания>*. В операторе присваивания значение правой части выражения присваивается левой части выражения. Инструкция OpCodes.Stloc извлекает из стека верхнее значение и помещает его в локальную переменную. При необходимости выполняется конвертация типа, а именно, если переменная имеет тип real, а правая часть выражения – тип int. В таком случае необходимо конвертировать значение на вершине стека из целочисленного в вещественное.

В конструкции *<условный оператор>* для реализации ветвления используются метки. Метки объявляются следующим образом:

Label falseLabel = il.DefineLabel();

Label trueLabel = il.DefineLabel();

Label endLabel = il.DefineLabel();

В языке MSIL есть команды условного и безусловного перехода к следующей инструкции. Команды условного перехода отличаются от безусловных в одном аспекте: они передают управление инструкции, если значение на вершине стека равно true (1) или false (0). Рассмотрим фрагмент метода IfStatement, в котором используется метки для реализации ветвления:

il.Emit(OpCodes.Brtrue, trueLabel); // if true

il.Emit(OpCodes.Br, falseLabel); // if false

il.MarkLabel(trueLabel);

Statement();

il.Emit(OpCodes.Br, endLabel);

il.MarkLabel(falseLabel);

if (IsOperation(new List<EOperation>() { EOperation.Else }))

{

GetNextToken();

Statement();

}

il.Emit(OpCodes.Br, endLabel);

il.MarkLabel(endLabel);

Если значение на вершине стека равно true (1), то мы «прыгаем» на метку trueLabel и вызываем метод Statement для ветки true. Иначе мы становимся на метку falseLabel. При наличии альтернативной ветки мы анализируем конструкцию *<оператор>* в ветке false. Далее во всех случаях осуществляется безусловный переход к метке endLabel. Разбор условного оператора заканчивается.

Действия по генерации кода в методе LoopStatement похожи на те, которые были использованы при анализе конструкции *<условный оператор>.* Ниже приведён фрагмент кода, в котором продемонстрированы инструкции для организации цикла (не включая проверку синтаксиса и семантики):

Label loopCondition = il.DefineLabel();

Label loopBody = il.DefineLabel();

Label loopEnd = il.DefineLabel();

il.Emit(OpCodes.Br, loopCondition);

il.MarkLabel(loopCondition);

// здесь вызывается Expression(), т.е. формируется значение в стеке

il.Emit(OpCodes.Brtrue, loopBody); // выражение true

il.Emit(OpCodes.Br, loopEnd); // выражение false

il.MarkLabel(loopBody);

Statement();

il.Emit(OpCodes.Br, loopCondition);

il.MarkLabel(loopEnd);

Чтобы оценить правильность работы генератора кода, был реализован консольный вывод. Появился метод OutputStatement, осуществляющий разбор следующей БНФ:

<оператор вывода> ::= **writeln** (<выражение> | <пусто>) | **write** (<выражение> | <пусто>)

В этот метод также были добавлены команды для генерации кода. На стек помещается инструкция, которая вызывает метод с названием methodName (WriteLine/Write) класса Console:

il.Emit(OpCodes.Call, typeof(Console).GetMethod(methodName, new Type[] { <тип> }));

Вместо *<тип>* указывается одно из следующих значений: typeof(int), typeof(double), typeof(string), typeof(bool) или typeof(void). Это значение определяет тип передаваемого в метод аргумента.

Если в процессе анализа программы будет обнаружена ошибка, то дальнейшие действия по генерации кода не выполняются. В таком случае сборка не сохраняется на диск, а информация о найденных ошибках выводится в текстовый файл output.txt.

# Тестирование

В этом разделе протестирована работа компилятора на различных входных примерах

1. **Вычисление факториала числа**

program factorial;

var i, num, fact: integer;

begin

num := 8;

fact := 1;

i := 2;

while (i <= num) do

begin

fact := fact \* i;

i := i + 1;

end;

write ('Факториал числа ');

writeln (num);

writeln (fact);

end.

Ожидаемый результат:

Факториал числа 8

40320

Фактический результат:



1. **Вывод чисел Фибоначчи**

program fib;

var

i, n, s, a, a1: integer;

begin

n := 11;

i := 1;

writeln('Числа Фибоначчи');

while i <= n do

begin

if (i = 1) or (i = 2) then

s := 1

else

s := a + a1;

a := a1;

a1 := s;

writeln(s);

i := i + 1;

end;

end.

Ожидаемый результат:

Числа Фибоначчи

1

1

2

3

5

8

13

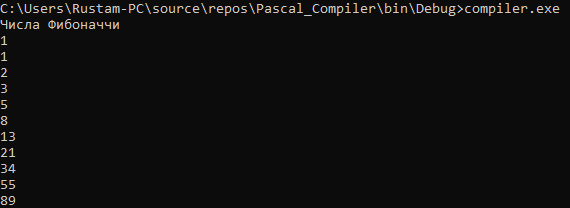
21

34

55

89

Фактический результат:



1. **Перестановка первой и последней цифр заданного числа**

program firstlastdigit;

var n, n1, p, a, i : integer;

begin

n := 25615;

a := n;

i := 1;

p := n mod 10; { последняя цифра введенного числа }

while n >= 10 do

begin

i := i \* 10;

n := n div 10;

end;

n1 := a - n\*i - p + n + p \* i;

write ('Число после перестановки первой и последней цифр числа = ');

writeln(n1);

end.

Ожидаемый результат:

Число после перестановки первой и последней цифр числа = 55612

Фактический результат:



1. **Операции над строками: конкатенация и сравнение**

program operations\_str;

var s1, s2: string;

begin

s1 := 'Кот';

s2 := 'Пёс';

if s1 < s2 then

s1 := s1 + 'o' + s2

else

s2 := s2 + 'o' + s1;

writeln (s1);

writeln (s2);

end.

Ожидаемый результат:

КотoПёс

Пёс

Фактический результат:



1. **Нахождение наибольшего числа из трёх**

programcompthreedig;

var

a,b,c:integer;

begin

a := 1;

b := 4;

c := 2;

if (a > b) and (a > c) then

write(a)

else

if (b > a) and (b > c) then

write(b)

else

write(c);

end.

Ожидаемый результат:

4

Фактический результат:



1. **Деление (/) 0 на 0, 1 на 0 и целочисленное деление (div) 0 на 0**

program divByZero;

begin

write ('0 / 0 = ');

writeln (0 / 0);

write ('1 / 0 = ');

writeln(1 / 0);

write ('0 div 0 = ');

writeln (0 div 0);

end.

Ожидаемый результат:

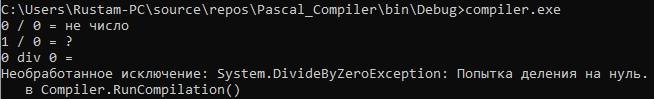
0 / 0 = NaN

1 / 0 = ∞

0 div 0 =

Исключение: System.DivideByZeroException: Попытка деления на нуль.

Фактический результат:



Примечание: символ «?» выводится, так как в таблице ASCII нет символа бесконечности.

1. **Логические выражения**

program logic;

var x, y: real;

b1, b2, b3, b4, b5, b6: boolean;

begin

x := 7.5;

y := 4.0;

b1 := x < y;

b2 := x <= y;

b3 := x > y;

b4 := x >= y;

b5 := x = y;

b6 := x <> y;

write ('Первое меньше второго: ');

writeln (b1);

write ('Первое меньше или равно второму: ');

writeln (b2);

write ('Первое больше второго: ');

writeln (b3);

write ('Первое больше или равно второму: ');

writeln (b4);

write ('Первое равно второму: ');

writeln (b5);

write ('Первое не равно второму: ');

writeln (b6);

end.

Ожидаемый результат:

Первое меньше второго: False

Первое меньше или равно второму: False

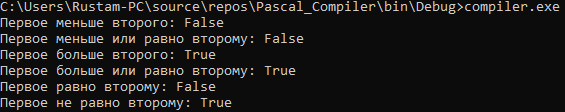
Первое больше второго: True

Первое больше или равно второму: True

Первое равно второму: False

Первое не равно второму: True

Фактический результат:



1. **Найти значение логической функции**

program logicfunc;

var a, b, c, F:boolean;

begin

a:=false; b:=true; c:=true;

F := (not a) and (b or c);

writeln (f);

a:=true; b:=true; c:= false;

F := (not a) and (b or c);

writeln (F);

a:=false; b:=false; c:= false;

F := (not a) and (b or c);

writeln (F);

end.

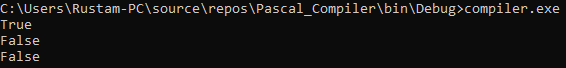
Ожидаемый результат:

True

False

False

Фактический результат:



1. **Решение системы трёх линейных уравнений с тремя неизвестными**

program equationsystem;

var a1,a2,a3,b1,b2,b3,c1,c2,c3,d1,d2,d3,x,y,z,e,ex,ey,ez:real;

begin

a1 := 1; b1:=1; c1:=2; d1:=-1;

a2 := 2; b2:=-1; c2:=2; d2:=-4;

a3 := 4; b3:=1; c3:=4; d3:=-2;

e := (a1\*b2\*c3+b1\*c2\*a3+c1\*a2\*b3-a3\*b2\*c1-b3\*c2\*a1-c3\*a2\*b1);

ex := (d1\*b2\*c3+b1\*c2\*d3+c1\*d2\*b3-d3\*b2\*c1-b3\*c2\*d1-c3\*d2\*b1);

ey := (a1\*d2\*c3+d1\*c2\*a3+c1\*a2\*d3-a3\*d2\*c1-d3\*c2\*a1-c3\*a2\*d1);

ez := (a1\*b2\*d3+b1\*d2\*a3+d1\*a2\*b3-a3\*b2\*d1-b3\*d2\*a1-d3\*a2\*b1);

if ( e=0 ) and ( (ex=0) or (ey=0) or (ez=0) ) then

writeln('бесконечное множество решений')

else if ( e<>0 ) and ( (ex=0) or (ey=0) or (ez=0) ) then

writeln('нет решений')

else begin

x:=ex/e; y:=ey/e; z:=ez/e;

write('x = ');

writeln (x);

write('y = ');

writeln (y);

write('z = ');

writeln (z);

end;

end.

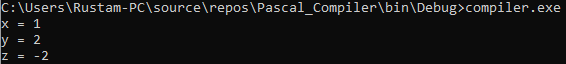
Ожидаемый результат:

x = 1

y = 2

z = -2

Фактический результат:



1. **Вывести таблицу умножения**

program tablemult;

var i, j: integer;

begin

i := 1;

while (i <= 10) do

begin

j := 1;

while j <= 10 do

begin

write (i);

write ('\*');

write (j);

write ('=');

write(i\*j);

write(' ');

j:=j+1;

end;

writeln();

i:=i+1;

end;

end.

Ожидаемый результат:

1\*1=1 1\*2=2 1\*3=3 1\*4=4 1\*5=5 1\*6=6 1\*7=7 1\*8=8 1\*9=9 1\*10=10

2\*1=2 2\*2=4 2\*3=6 2\*4=8 2\*5=10 2\*6=12 2\*7=14 2\*8=16 2\*9=18 2\*10=20

3\*1=3 3\*2=6 3\*3=9 3\*4=12 3\*5=15 3\*6=18 3\*7=21 3\*8=24 3\*9=27 3\*10=30

4\*1=4 4\*2=8 4\*3=12 4\*4=16 4\*5=20 4\*6=24 4\*7=28 4\*8=32 4\*9=36 4\*10=40

5\*1=5 5\*2=10 5\*3=15 5\*4=20 5\*5=25 5\*6=30 5\*7=35 5\*8=40 5\*9=45 5\*10=50

6\*1=6 6\*2=12 6\*3=18 6\*4=24 6\*5=30 6\*6=36 6\*7=42 6\*8=48 6\*9=54 6\*10=60

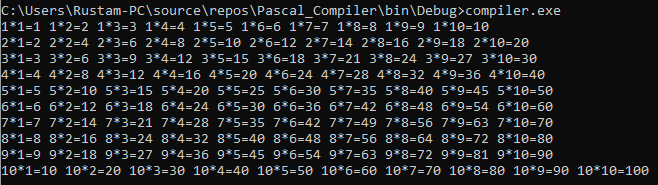
7\*1=7 7\*2=14 7\*3=21 7\*4=28 7\*5=35 7\*6=42 7\*7=49 7\*8=56 7\*9=63 7\*10=70

8\*1=8 8\*2=16 8\*3=24 8\*4=32 8\*5=40 8\*6=48 8\*7=56 8\*8=64 8\*9=72 8\*10=80

9\*1=9 9\*2=18 9\*3=27 9\*4=36 9\*5=45 9\*6=54 9\*7=63 9\*8=72 9\*9=81 9\*10=90

10\*1=10 10\*2=20 10\*3=30 10\*4=40 10\*5=50 10\*6=60 10\*7=70 10\*8=80 10\*9=90 10\*10=100

Фактический результат:



1. **Входные данные содержат лексические/синтаксические/семантические ошибки**

program errprog1;

var 1m, b,, n:integer;

s string;

flag boolean

, Y: real;

5: integer;

begin

m = 15;

n := (52+652;

Y := (m + n) / 12;

while m <> n do

begin

if m := n then

m := m - n;

else

n :5= n - m;

end;

if Y / 2 > m then

flag := true;

if flag

begin

Y = Y / 2;

flag := not flag;

end

else

Y := m div 2;

end.

Ожидаемый результат:

Действия по генерации кода не выполняются. Выходной файл программы не создаётся. В output.txt будет занесена информация о найденных ошибках во время компиляции.

Фактический результат:

