МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное
учреждение высшего образования
«Южный федеральный университет»

Институт математики, механики
и компьютерных наук им. И. И. Воровича

Кафедра алгебры и дискретной математики

Однороб Никита Сергеевич

Использование проекта Roslyn для создания компилятора модельного языка программирования

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
по направлению подготовки
02.03.02 – Фундаментальная информатика и информационные технологии

Научный руководитель –
доцент, к.ф.-м.н. Михалкович С.С.

Допущено к защите:
заведующий кафедрой _____ Штейнберг Б. Я.

Ростов-на-Дону – 2020

Оглавление

Введение

.NET Framework — программная платформа, включающая в себя общеязыковую среду исполнения программ (CLR). Ее главная особенность (и основная идея) заключается в совместимости программных частей, написанных на разных языках. Так, программа на C# может одновременно использовать библиотеки на Visual Basic.NET, C# и даже Visual C++, что дает гибкость при разработке.

Под платформу .NET Framework уже написано огромное количество ПО и библиотек, главным образом из-за популярности языка программирования C#. Но сейчас Microsoft начинает постепенно отказываться от нее. Заявлено, что .NET Framework 4.8 — последняя версия платформы. Ее поддержка (исправления ошибок и уязвимостей) будет продолжаться, но новых возможностей уже не добавится.

В качестве альтернативы Microsoft предлагает новую платформу .NET Core. Она также является общеязыковой, но одновременно с этим, в отличие от .NET Framework, еще и кроссплатформенной [13, с. 223]. Однако список языков программирования, на которых можно вести разработку под .NET Core, сильно ограничен: C#, Visual Basic и F# [3].

В данной работе будет продемонстрировано создание компилятора модельного языка программирования для платформы .NET Core с использованием компилятора C# Roslyn. Данная задача важна для создания полноценных языков программирования под .NET Core или портирования существующих, в частности, языка PascalABC.NET, который сейчас компилирует код для .NET Framework.

Постановка задачи

Целью данной работы является разработка компилятора для модельного языка программирования с использованием Roslyn под платформу .NET Core. Для достижения цели были сформулированы следующие задачи:

1. Изучить устройство компиляторов, их принципы работы и методы разработки.
2. Сформулировать и описать синтаксис и семантические правила для модельного языка программирования.
3. Реализовать построение синтаксического дерева программы.
4. Изучить компилятор с открытым исходным кодом Roslyn.
5. Реализовать изменения семантических проверок компилятора Roslyn.
6. Реализовать компилятор для модельного языка, используя инфраструктуру проекта Roslyn.

1. Общие сведения о компиляторах

Язык программирования — это средство описания компьютерной программы в виде, понятном для человека. Чтобы запустить программу, ее необходимо преобразовать в форму, пригодную для выполнения компьютером (машинный код) или виртуальной машиной (байт-код). Такой процесс называется *компиляцией*, а программа, ее выполняющая — *компилятором*.

Процесс компиляции можно разделить на фазы (рис. 1.1) [16, с. 6].

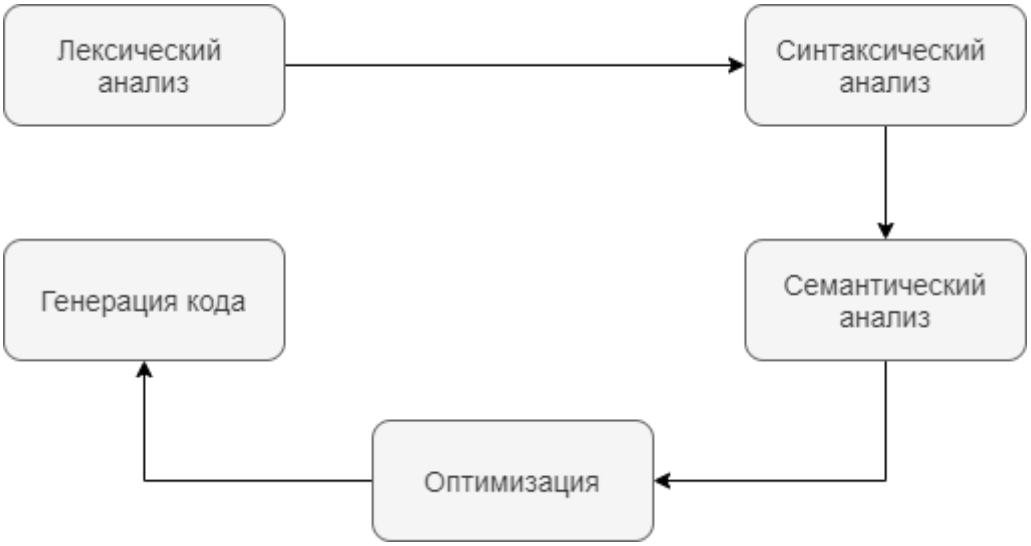

Рисунок 1.1. Фазы компиляции

1.1. Лексический анализ

Лексический анализ — первый этап компиляции и интерпретации, который выполняется лексическим анализатором (*лексером*). Лексический анализатор читает посимвольно исходный код программы и разбивает его на составные части языка (смысловые последовательности, называемые *лексемами*). Это могут быть идентификаторы, ключевые слова, числа, операторы [1, с. 33].

Существуют инструменты для генерации кода лексических анализаторов. В данной работе используется GPLEX [9]. Он принимает на вход описание токенов языка и список ключевых слов и генерирует код лексера на языке C#.

Пример работы лексического анализатора приведен на рис. 1.2.

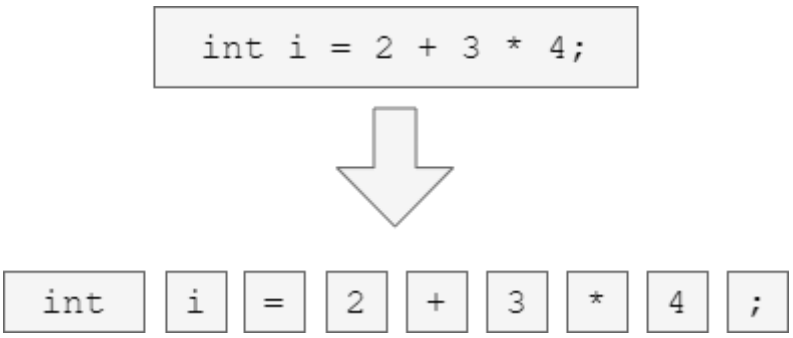

Рисунок 1.2. Пример работы лексического анализатора

1.2. Синтаксический анализ

Синтаксический анализ — проверка корректности программы с точки зрения *грамматики языка* программирования. Программа, выполняющая такую проверку, называется *парсером*.

Грамматика языка — набор правил, устанавливающий порядок следования токенов [6, с. 79]. Например, после знака арифметической операции не может следовать другой знак арифметической операции. После оператора присваивания должно следовать выражение.

Кроме того, в рамках синтаксического анализа строится абстрактное синтаксическое дерево программы (*AST — Abstract Syntax Tree*) [15, с. 13]. Это дерево, узлы которого — синтаксические конструкции программы (циклы, условия, идентификаторы, константы). Каждый внутренний узел представляет операцию, а дочерние узлы — аргументы этой операции. Пример AST приведен на рис. 1.3.

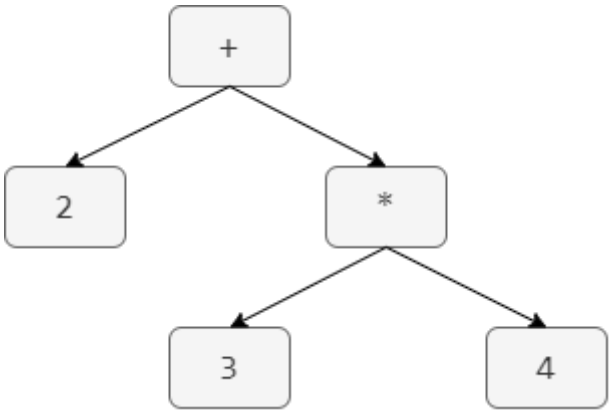

*Рисунок 1.3. AST выражения 2 + 3 * 4*

Существуют утилиты, генерирующие синтаксические анализаторы, получая на вход грамматические правила языка. В рамках данной работы используется генератор парсеров GPPG [10]. Он принимает на вход грамматику языка и генерирует код анализатора на языке C#.

1.3. Семантический анализ

Семантический анализ — проверка программы на соответствие *семантическим правилам* языка программирования. Важной ее частью является *проверка типов*. Так, выражение `int i = 3.1315;` верно с точки зрения грамматики, но неверно семантически — в большинстве языков переменной целого типа нельзя присвоить вещественное число без явного приведения.

Помимо этого, на этапе семантического анализа идет построение таблицы символов. Это структура данных, хранящая в себе информацию об объявленных в программе сущностях (переменных, функций, констант), их типов и областей видимости [8, с. 54] [12, с. 114]. Данная таблица используется при дальнейших семантических проверках (например, при проверках существования вызываемых методов) и генерации кода.

Семантический анализ является нетривиальной задачей ввиду огромного количества проверок. Для ее решения в данной работе был использован сторонний компилятор Roslyn.

2. Описание модельного языка

В данной работе рассматривается создание компилятора модельного языка программирования под платформу .NET Core. Чтобы показать возможность создания для нее компилятора произвольного языка программирования (например, PascalABC.NET), был реализован компилятор для модельного языка, удовлетворяющего следующим требованиям:

- наличие операторов ввода и вывода;
- встроенные типы данных, которые проецируются на типы .NET;
- возможность использования напрямую классов и методов .NET;
- наличие структур данных собственной реализации;
- наличие расширяемой стандартной библиотеки.

Описанные требования являются базисными для .NET-языков, поскольку позволяют реализовать любой функционал, предоставляемый платформой .NET Core.

В рамках данной работы был разработан модельный язык с C-подобным синтаксисом. В нем доступны стандартные типы из .NET (`int`, `double`, `string`). Для вывода на консоль используется оператор `print`. С помощью ключевого слова `netusing` можно подключить стандартные пространства имен .NET.

В языке доступны объявления и вызовы функций. Присутствуют массивы, базирующиеся на структуре данных собственной реализации. Переменные можно объявлять и инициализировать как обычным способом, так и с помощью кортежей. Доступны циклы `for` и `loop`.

Для модельного языка была реализована стандартная библиотека в виде некоторых математических функций, класса массивов и функций для замера времени выполнения кода.

3. Использование Roslyn

Roslyn — это компилятор C# с открытым исходным кодом, разрабатываемый Microsoft [14]. Он может компилировать программы на языке C# под платформы .NET Framework и .NET Core. Чтобы использовать Roslyn, необходимо свести программу на модельном языке программирования к программе на языке C#, а уже потом выполнять компиляцию.

Roslyn подходит для решения поставленных задач, поскольку C# предоставляет доступ ко всем возможностям платформы .NET. В данной работе язык C# рассматривается как проекция на весь .NET Core.

Roslyn принимает на вход код программы на языке C# или ее синтаксическое дерево [11, с. 39]. Лексический и синтаксический анализ необходимо выполнять самостоятельно, а полученное AST — преобразовать в AST программы на языке C#, после чего использовать Roslyn. Таким образом, Roslyn берет на себя все семантические проверки, оптимизацию и генерацию кода [5, с. 1976]. Схема компиляции с использованием Roslyn представлена на рис. 3.1.

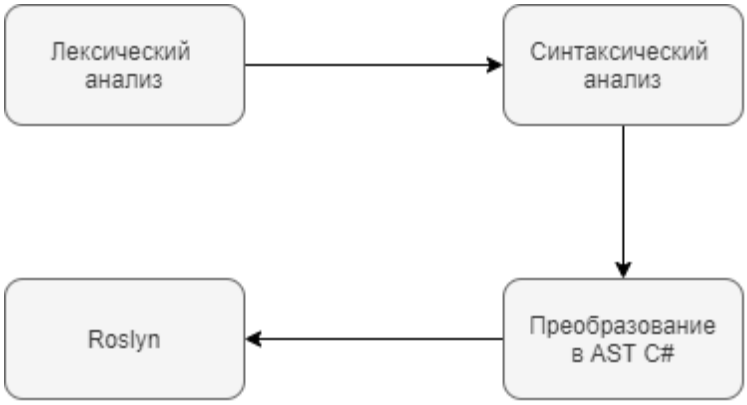

Рисунок 3.1. Схема компиляции с использованием Roslyn

В данной работе лексический и синтаксический анализаторы реализуются с помощью специальных генераторов GPLEX и GPPG. Далее рассматривается преобразование синтаксического дерева в дерево программы на C#.

3.1. Преобразование AST

Самый простой способ преобразовать одно синтаксическое дерево в другое — обойти начальное AST в глубину и в каждом узле генерировать новые узлы для целевого дерева. Это можно реализовать с помощью паттерна «визитор» [2].

Визитор — это класс, содержащий методы для обхода узлов исходного дерева. Пример такого класса приведен в листинге 3.1.

Листинг 3.1. Пример класса-визитора

```
class MyVisitor {
   public void VisitIntNumNode(IntNumNode node) { … }
   public void VisitPrintNode(PrintNode node) { … }
   …
}
```

Для генерации AST C# был реализован класс `RoslynTreeBuilderVisitor`, который обходит все узлы исходного синтаксического дерева и создает узлы дерева C#. Созданные узлы выражений (литералов, бинарных операторов, аргументов функций) сохраняются на стеке для получения к ним доступа из родительских узлов. Ниже представлен метод визитора для узла присваивания переменной значения `AssignVarNode` [11, с. 75]:

Листинг 3.2. Метод визитора для узла присваивания

```
public override void VisitAssignVarNode(AssignVarNode node) {
   node.Expression.Visit(this);
   node.ID.Visit(this);
   var kindAssigment = SyntaxKind.SimpleAssignmentExpression;
   var assignVar = AssignmentExpression(
      kindAssigment, expressions.Pop(), expressions.Pop()
   );
   AddStatementToCurrentBlock(ExpressionStatement(assignVar));
}
```

Сначала визитором обходятся дочерние узлы присваивания (идентификатор переменной и присваиваемое выражение), после чего

сгенерированные для них узлы оказываются на стеке `expressions`. Затем такие узлы снимаются со стека и используются для генерации родительских узлов.

Здесь важно отметить, что `AssignVarNode` является узлом синтаксического дерева модельного языка, а `AssignmentExpression` — статическим методом класса `SyntaxFactory` из Roslyn, генерирующим узел синтаксического дерева C#.

Полученное AST должно соответствовать семантическим правилам языка C#. Для этого при генерации учитываются следующие особенности:

- Если в модельном языке инструкции располагаются в самом внешнем блоке, в дереве C# эквивалентные узлы должны быть расположены в статическом методе `Main` класса `Program`;
- Объявлению переменной или функции в глобальной области видимости в модельном языке соответствует объявление статического члена класса `Program` в C#.

В терминах Roslyn компилируемой единицей C# является модуль (unit), который включает в себя пространства имен или классы напрямую. В методах визитора `RoslynTreeBuilderVisitor` генерируется такой модуль, который доступен как вычислимое поле `UnitNode` класса визитора.

3.2. Компиляция под .NET

После того, как было получено AST C#, его можно скомпилировать. Roslyn предоставляет для этого класс `CSharpCompilation`, с помощью которого можно настроить параметры компиляции.

В экземпляре такого класса необходимо указать синтаксическое дерево — AST C#, полученное в результате работы визитора. Помимо этого, можно установить тип выходного приложения (в рамках данной работы — консольное) и уровень оптимизации.

Листинг 3.3. Установка параметров компиляции

```
var visitor = new RoslynTreeBuilderVisitor();
syntaxTree.Visit(visitor); //запуск визитора
var syntaxTrees = new[] { visitor.UnitNode.SyntaxTree };
var options = new CSharpCompilationOptions(
   outputKind: OutputKind.ConsoleApplication,
   optimizationLevel: OptimizationLevel.Release
);
```

Кроме параметров компиляции, Roslyn требует указание ссылок на библиотеки .NET (`references`). Самая важная из них — библиотека целевой платформы. Для компиляции кода под .NET Framework необходимо загрузить `mscorlib.dll`, под .NET Core — `System.Private.CoreLib.dll`. Таким образом, Roslyn предоставляет возможность выбора целевой платформы.

В данной работе для компиляции была выбрана платформа .NET Core ввиду ее кроссплатформенности. По этой же причине сам компилятор также был разработан под .NET Core. Для загрузки библиотеки использовалась особенность примитивных типов в языке C#: они хранят информацию о целевой платформе, под которую скомпилирована программа. Таким образом, путь к нужной сборке можно получить с помощью следующего вызова [7]:

```
string netCoreLib = typeof(object).Assembly.Location;
```

Запустить процесс компиляции позволяет метод `Emit` класса `CSharpCompilation`.

Листинг 3.4. Компиляция под .NET Core

```
var compilation = CSharpCompilation.Create(
   "assemblyName", syntaxTrees, references, options
);
compilation.Emit(outputExeFileName);
```

Приведенный код генерирует исполняемый файл EXE. Но для работы программ для .NET Core необходим файл конфигурации `.runtimeconfig.json`. После добавления его генерации компилятор будет работать. Для проверки был создан файл `test.mylang` со следующим содержимым:

```
print "Hello, world!";
```

Лог его компиляции и консольный вывод сгенерированного EXE-файла представлены в листинге 3.5.

Листинг 3.5. Запуск компиляции и сгенерированного EXE

```
> MyCompiler.exe "test.mylang" "test.exe"
> dotnet test.exe
Hello, world!
```

Таким образом, разрабатываемый компилятор модельного языка успешно генерирует исполняемые файлы под платформу .NET Core.

4. Модификация Roslyn

4.1. Модификация текста ошибок

Одна из главных задач компилятора — обнаруживать ошибки в программе и сообщать о них программисту. Разработанный компилятор из-за использования Roslyn проверяет ошибки с точки зрения семантики языка C#.

Для демонстрации этого разрабатываемому компилятору был передан код на модельном языке, содержащий семантические ошибки.

Листинг 4.1. Код на модельном языке с семантическими ошибками

```
int i = 5.0;
string lower = toLowerCase("CAPS");
int j = a;
```

Консольный вывод компилятора, запущенного для такого кода, приведен в листинге 4.2. Ошибки выводятся в терминах языка C#.

Листинг 4.2. Вывод компилятора для кода с ошибками

```
> MyCompiler.exe "test.mylang" "test.exe"
(1,73): error CS0266: Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)
(1,93): error CS0103: The name 'toLowerCase' does not exist in the current context
(1,124): error CS0103: The name 'a' does not exist in the current context
```

Кроме того, абсолютно неверно указана локализация ошибок. Это происходит по причине того, что у лексем Roslyn нет заранее установленной позиции, она вычисляется согласно построенному AST. Здесь есть две проблемы:

- При построении AST C# считается, что его элементы расположены все в одной строке друг за другом (без пробельных символов). Для решения проблемы Roslyn предоставляет для синтаксического дерева узлы переносов строк, пробелов и табуляций [4, с. 1022].

- Модельный язык и C# имеют разный синтаксис. C# требует наличия класса и метода `Main`, поэтому никакая инструкция не может иметь позицию `(1,1)` из-за объявления класса и метода, в то время как в модельном языке такое возможно. Помимо этого, в двух языках разные длины инструкций (например, `print` в модельном и эквивалентный ему `Console.WriteLine` в C#)

Позиция токенов в компиляторе Roslyn задается не привычной парой «строка-столбец», а парой смещений (индексы символов с начала файла). Эти значения пересчитываются автоматически после каждого изменения AST, поэтому нет смысла переопределять их вручную — они все равно изменятся после любого изменения дерева. В пары «строка-столбец» смещения преобразуются уже на этапе вывода ошибок, сами пары в узлах не хранятся.

Roslyn предоставляет возможность добавить для каждого узла AST комментарий (`NodeAnnotation` в терминах компилятора). В данной работе такие комментарии были использованы для хранения позиций токенов исходного (модельного) языка. Например, выражению `print a;` модельного языка соответствует дерево Roslyn, эквивалентное следующему коду на C#[1]:

```
class Program { static void Main() { Console.WriteLine(a); } }
```

Идентификатор `a` в коде модельного языка имеет позицию `(1,7) (1,8)`, а в дереве Roslyn — `48;49` (здесь уже не пары, а смещения). С помощью `NodeAnnotation` идентификатору устанавливается комментарий с текстом `"1,7;1,8"`. После того, как AST C# построено, идет построение таблицы соответствий позиций (`locationMap` в коде разработанного компилятора): получаются все узлы Roslyn, имеющие комментарии, текст комментариев разбирается и данные о позициях заносятся в словарь, где ключи являются позициями в дереве C#, а значения — в исходном коде модельного языка.

Сообщения об ошибках Roslyn представляются объектами класса `Diagnostic`. Исходная неформатированная позиция хранится в поле

[1] Для наглядности в коде расставлены пробелы, но в дереве Roslyn они отсутствуют вовсе, даже между токенами class и Program, static, void и Main.

`Diagnostic.Location.SourceSpan`. Таким образом, при обработке ошибок полученная позиция из дерева C# используется как ключ в построенном словаре для получения исходной локализации ошибки.

В классе `Diagnostic` есть поля `Code` и `Arguments`. Они хранят код ошибки и аргументы шаблонной строки ее описания. Изначально эти поля являются внутренними (`internal`), т. е. недоступными извне. Для их использования модификаторы доступа для этих полей были изменены на `public` в классе `Diagnostic` и всех его наследниках.

Изменить текст ошибок можно двумя способами:

1. На стороне компилятора модельного языка при обработке ошибки получать ее код и аргументы, после чего формировать собственную строку.
2. Изменить текст ошибки в самом Roslyn. В файле `ErrorCode.cs` описаны коды ошибок и их константные обозначения. Например, для ошибки преобразования типа используется константа `ERR_NoImplicitConvCast` (соответствует коду `CS0266`). В файле ресурсов `CSharpResources.resx` каждой такой константе сопоставляется текст ошибки, и его можно изменить.

После описанных изменений разрабатываемому компилятору снова был передан код из листинга 4.1. Вывод представлен в листинге 4.3.

Листинг 4.3. Исправленный вывод компилятора для кода с ошибками

```
(1:8,1:11) Error: Can't convert type 'double' to 'int'
(2:15,2:26) Error: Unknow identifier 'toLowerCase'
(3:8,3:9) Error: Unknow identifier 'a'
```

Таким образом, была исправлена локализация ошибок и были изменены их описания.

4.2. Удаление семантических проверок

В предыдущих разделах компиляция кода на модельном языке производилась в предположении, что его семантические правила совпадают с правилами языка C#. В данном разделе рассматривается общий случай, когда такие правила различаются и необходима коррекция семантических проверок

Roslyn. Для демонстрации возможности создания полноценного компилятора необходимо уметь изменять семантику Roslyn в обе стороны: разрешить то, что запрещено в C#, и наоборот.

Например, в C# запрещено объявление переменной, если переменная с таким именем была объявлена в родительской области. Код на модельном языке, приведенный в листинге 4.4, приведет к ошибке со стороны семантики Roslyn.

Листинг 4.4. Код с переменной во вложенной области

```
int i = 2;
{
   double i = 1; //error CS0136
   print i;
}
print i;
```

В модельном языке такое объявление было разрешено.

За проверку семантики C# отвечает проект `Microsoft.CodeAnalysis.CSharp` из проекта Roslyn. Этап семантических проверок в терминах Roslyn называется `Binding`. В частности, класс `BlockBinder` проверяет семантику внутри всех блоков.

Был создан класс `MyBlockBinder`, наследующегося от `BlockBinder`. Для этого у базового класса был удален модификатор `sealed`, запрещающий наследование.

Классе `BlockBinder` содержит метод `BindIdentifier`, проверяющий семантику для идентификаторов. Одна из таких проверок — приватный метод `ValidateNameConflictsInScope`. Он проверяет конфликты имен в области видимости. Этот метод был объявлен `protected` и виртуальным.

Исходный метод возвращает `false`, если конфликта имен нет, и `true` в противном случае. После нескольких семантических проверок в начале идет цикл `for` (листинг 4.5)

Листинг 4.5. Цикл в исходном методе `ValidateNameConflictsInScope`

```
for (Binder binder = this; binder != null; binder = binder.Next) {
   // ...
   var scope = binder as LocalScopeBinder;
   if (scope?.EnsureSingleDefinition(...) == true) return true;
   // ...
}
```

Метод `EnsureSingleDefinition` проверяет, есть ли еще переменная с таким именем в текущей области видимости. Цикл проходит по всем блокам, начиная от текущего и заканчивая глобальной областью видимости. Как только в одном из них обнаруживается дублирующее объявление, возвращается `true` (есть конфликт).

Необходимо изменить этот цикл таким образом, чтобы проверка `EnsureSingleDefinition` вызывалась для локальных переменных только один раз. Если в одном блоке объявляется две переменных с одинаковым именем, это должно остаться ошибкой.

В наследуемом классе `MyBlockBinder` был переопределен метод `ValidateNameConflictsInScope`. Поскольку изменения требовались внутри цикла, код был скопирован практически полностью. В начале метода была объявлена переменная `wasCheckedConflict` с начальным значением `false`. Условие вызова `EnsureSingleDefinition` было изменено таким образом, чтобы он происходил только при ложном значении введенного флага. А после такого вызова было добавлено условие:

```
if (!wasCheckedConflict && symbol.Kind == SymbolKind.Local) {
   wasCheckedConflict = true;
}
```

Таким образом, проверка на наличие конфликта будет выполнена один раз для исходного блока, а дальше, если проверяется имя локальной переменной, такая проверка будет деактивирована.

После сборки проекта `Microsoft.CodeAnalysis.CSharp` код на модельном языке, представленный в листинге 4.4, был передан компилятору в виде файла `vars.mylang`. Результат компиляции представлен в листинге 4.6.

Листинг 4.6. Результат компиляции после деактивации проверки

```
> MyCompiler.exe "vars.mylang" "vars.exe"
> dotnet vars.exe
1
2
```

Таким образом, была расширена семантика языка C#.

4.3. Добавление семантических проверок

Также в данной работе была продемонстрирована возможность изменения семантики C# в другую сторону путем добавления семантических правил.

В модельном языке был реализован цикл `loop` следующего синтаксиса:

```
loop (N) { ... }
```

Он повторяет действия внутри своего блока заданное число раз, где `N` должно быть выражением целого типа.

При генерации AST C# цикл `loop` преобразуется в следующий цикл `for`:

```
for (long #loopCounter = 0; #loopCounter < N; #loopCounter++) {
...
}
```

Имя счетчика цикла начинается с символа '`#`', чтобы избежать возможного конфликта с переменными, объявленными программистом. Такие идентификаторы запрещены в языке синтаксически, поэтому пользователь не сможет создать такой идентификатор.

Это уже будет работать. Но такая целевая конструкция изначально допускает для выражения `N` любые сравнимые с `long` типы, включая вещественные. Учитывая, что `N` — количество итераций, оно должно быть целым.

Для решения данной проблемы была добавлена семантическая проверка, проверяющая тип параметра `N`.

Параметр N является частью условия цикла for. В Roslyn есть класс-визитор LocalBinderFactory, содержащий методы для обхода инструкций, в том числе, VisitForStatement. Он запускает семантические проверки для цикла for. В листинге 4.7 приведена часть этого метода, проверяющая условие цикла for.

Листинг 4.7. Фрагмент VisitForStatement, проверяющий условие цикла

```
if (condition != null) {
   binder = new ExpressionVariableBinder(condition, binder);
   AddToMap(condition, binder);
   Visit(condition, binder);
}
```

ExpressionVariableBinder — это класс, проверяющий выражения, содержащие переменные. Было установлено, что проверка бинарного сравнения «на меньше» выполняется в методе BindSimpleBinaryOperator.

Был создан класс MyExpressionVariableBinder, являющийся наследником исходного ExpressionVariableBinder, и в нем был переопределен метод BindSimpleBinaryOperator. Затем внутри фрагмента метода, запускающего проверку условия цикла for (приведен в листинге 4.7), переменной binder был присвоен объект нового класса. Таким образом, внутри условия цикла for был обеспечен вызов переопределенного метода для бинарных выражений.

В этот метод были добавлены семантические проверки. Сначала проверяется, что левый операнд является идентификатором, начинающимся с символа '#'. Если это так, проверяется тип правого операнда. Если он не является целочисленным, генерируется ошибка ERR_LoopCounterNoInteger, которую перед этим была добавлена в перечисление всех ошибок ErrorCode. В конце вызываются стандартные семантические проверки.

Для того, чтобы выводился текст добавленной ошибки помимо ее кода, в файл ресурсов CSharpResources.resx было добавлено ее текстовое описание.

Теперь при попытке скомпилировать следующий код:

```
loop (5.5) { print 1; }
```

разрабатываемый компилятор выводит сообщение об ошибке (листинг 4.8).

Листинг 4.8. Вывод компилятора для вещественного параметра `loop`

```
> MyCompiler.exe "cycles.mylang" "cycles.exe"
(1:6,1:9) Error 8730: Параметр цикла LOOP должен быть выражением целого типа
```

Итак, была сужена грамматика языка C#, а именно, были запрещены бинарные выражения, в которых левый операнд является идентификатором, начинающимся с символа `'#'`, а правый — не целочисленным выражением.

Таким образом, в данной работе были продемонстрированы изменения семантики языка C# (и, соответственно, семантических проверок Roslyn) как в сторону добавления семантических правил, так и их удаления. Была показана генерация собственных ошибок, изначально отсутствующих в Roslyn.

5. Стандартная библиотека языка

Каждый язык программирования должен иметь свою стандартную библиотеку — набор заранее реализованных классов и функций, доступных для вызова из любой программы, написанной на этом языке.

В рамках данной работы был реализован механизм расширяемости стандартной библиотеки. После генерации AST C#, эквивалентного исходной программе на модельном языке, происходит перебор файлов в директории `MyCompilerLibrary`. Дальнейшие действия зависят от типа файла.

- Если файл имеет расширение `cs` (программа на C#), то вызывается метод из Roslyn `CSharpSyntaxTree.ParseText` для получения AST такого модуля. Для того, чтобы объявленные классы были доступны из программ на модельном языке, они должны находиться внутри пространства имен `MyCompilerLibrary`.
- Если файл имеет расширение `mylang` (программа на модельном языке), она преобразуется в AST C# так же, как и основная программа. Все функции, объявленные в таком модуле, преобразуются в статические методы класса, название которого совпадает с названием файла.

Все полученные AST добавляются к AST C# исходной программы на модельном языке, после чего Roslyn компилирует все синтаксические деревья.

Таким образом, была обеспечена расширяемость стандартной библиотеки. Для добавления в нее новой функциональности достаточно реализовать новый класс на C# или новые функции на модельном языке и поместить файл с кодом в директорию `MyCompilerLibrary`. По мере расширения функциональности языка модули стандартной библиотеки, написанные на C#, можно транслировать в модельный язык.

В рамках данной работы были реализованы 3 модуля стандартной библиотеки: `Array`, `Math` и `Performance`.

5.1. Класс для замера времени выполнения кода

Класс `Performance` — это модуль стандартной библиотеки для замера времени выполнения кода. Данный модуль реализован на языке C# и представляет собой статический класс со следующими членами:

- `start()` — запустить таймер;
- `restart()` — сбросить таймер и заново запустить его;
- `stop()` — остановить таймер;
- `reset()` — сбросить таймер;
- `milliseconds` — значение таймера в миллисекундах.

С его помощью можно показать, что программы на модельном языке работают так же быстро, как и программы на C#. Для этого на модельном языке был реализован подсчет суммы следующего ряда при $N = 10000$:

$$\sum_{i=1}^{N}\sum_{j=1}^{N}\frac{1}{i+j}$$

Листинг 5.1. Подсчет суммы ряда на модельном языке

```
double sum = 0;
Performance.start();

for int i in 1..10000
     for int j in 1..10000
          sum = sum + 1.0 / (i + j);

Performance.stop();
print sum;
print Performance.milliseconds;
```

Приведенная программа считает сумму ряда за 230–240 миллисекунд. Эквивалентная программа на C# дает такой же результат.

5.2. Математические функции

Модуль `Math` — это модуль, написанный на самом модельном языке и содержащий несколько математических функций. В листинге 5.2 приведен фрагмент этого модуля.

Листинг 5.2. Фрагмент файла Math.mylang

```
double sinDeg(double angle) {
      return System.Math.Sin(angle * System.Math.PI / 180);
}
double cosDeg(double angle) {
      return System.Math.Cos(angle * System.Math.PI / 180);
}
```

Эти функции доступны из любых программ на модельном языке как статические методы класса `Math`. Приведенный фрагмент демонстрирует, что из модельного языка есть полный доступ к классам платформы .NET.

5.3. Стандартный класс массивов

Класс `Array<T>` — это стандартные массивы модельного языка. Объявление `int[] arr = int {1,2,3}` в модельном языке преобразуется в создание объекта `Array<T>`.

Данный класс реализован на языке C# и предоставляет возможность получения элемента по индексу, а также следующие члены:

- `length` — длина массива;
- `indices` — массив индексов;
- `swapByIndex(i, j)` — меняет элементы с индексами *i* и *j* местами;
- `toString()` — преобразует массив в строку.

Пример использования массивов приведен в листинге 5.3.

Листинг 5.3. Массивы в модельном языке

```
int[] array = int {2,4,6,8,10};
print array;
print array.indices;
```

Данный код выводит следующий результат:

```
[2,4,6,8,10]
[0,1,2,3,4]
```

Также для класса `Array<T>` была реализована деструктуризация массива в кортеж. Пример использования деструктуризации приведен в листинге 5.4.

Листинг 5.4. Деструктуризация массива в модельном языке

```
int[] array = int { 150, 200 };
(int a, int b) = array;
```

Деструктуризация была реализована как группа методов `Deconstruct` класса `Array<T>`. В такие методы также были добавлены проверки согласованности размера кортежа и длины массива, генерирующие ошибки времени выполнения при несовпадении размеров.

Заключение

В рамках данной работы был разработан компилятор для модельного языка программирования, генерирующий код под платформу .NET Core. Был спроектирован модельный язык, позволяющий получить доступ ко всем ее возможностям. Компиляция выполняется с использованием open-source компилятора Roslyn. Были выполнены изменения семантических проверок Roslyn и генерация новых ошибок.

Для модельного языка была реализована стандартная библиотека. Был реализован механизм ее компиляции, позволяющий расширять библиотеку как на языке C#, так и на самом модельном языке.

Таким образом, было установлено, что создать компилятор произвольного языка программирования под .NET Core возможно с использованием Roslyn, но необходимо менять его семантические проверки. Стандартную библиотеку можно писать как на языке C#, так и на самом разрабатываемом языке. Такой язык будет работать так же быстро, как и C#.

Полученный результат позволяет утверждать, что перевод компилятора PascalABC.NET в инфраструктуру Roslyn также является возможным.

Исходный код разработанного компилятора размещен в GitHub-репозитории. Ссылка на него приведена в *Приложении А*.

Список литературы

1. Ахо А.В. Компиляторы: принципы, технологии и инструментарий / Альфед В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. — М.: ООО «И. Д. Вильямс», 2008. — 1178 с.
2. Визиторы по синтаксическому дереву — Вики ИТ мехмата ЮФУ. [Электронный ресурс] — URL: http://it.mmcs.sfedu.ru/wiki/Визиторы_по_синтаксическому_дереву (дата обращения 29.05.2020).
3. Обзор .NET Core. [Электронный ресурс] — URL: https://docs.microsoft.com/ru-ru/dotnet/core/about (дата обращения 25.05.2020).
4. Albahari J. C# 7.0 in a Nutshell: The Definitive Reference / J. Albahari, E. Johannsen. — Sebastopol: O'Reilly, 2018. — 1070 p.
5. Albahari J. C# 8.0 in a Nutshell: The Definitive Reference / J. Albahari, B. Albahari. — Sebastopol: O'Reilly, 2020. — 2228 p.
6. Bansal A. K. Introduction to programming languages / A. K. Bansal. — Boca Raton, Florida: CRC Press, 2014. — 566 p.
7. Compiling and Executing Code in a C# App. [Электронный ресурс] — URL: https://www.damirscorner.com/blog/posts/20190802-CompilingAndExecutingCodeInACsApp.html (дата обращения 29.05.2020).
8. Ezhil Selvi A. S. Compiler design concepts, Worked out Examples / A. S. Ezhil Selvi, J. J. Persis // ResearchGate. — 2017. [Электронный ресурс] — URL: https://www.researchgate.net/publication/316560026 (дата обращения 06.06.2020)
9. GPLEX – CodePLEX Archive. [Электронный ресурс] — URL: https://archive.codeplex.com/?p=gplex (дата обращения 26.05.2020).
10. GPPG – CodePLEX Archive. [Электронный ресурс] — URL: https://archive.codeplex.com/?p=gppg (дата обращения 26.05.2020).
11. Harrison N. Code Generation with Roslyn / N. Harrison — Lexington: Apress, 2017. — 104 p.

12. Mogensen T. Æ. Basics of Compiler Design / T. Æ. Mogensen — Anniversary edition. — Lulu.com, 2010. — 307 p.
13. Price M. J. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development. / M. J. Price — 4th edition. — Birmingham: Packt, 2019. — 784 p.
14. Roslyn — GitHub. [Электронный ресурс] — URL: https://github.com/dotnet/roslyn (дата обращения 27.05.2020).
15. Waite W. M. Compiler Construction / W. M. Waite, G. Goos — Karlsruhe, 1996. — 360 p.
16. Wirth N. Compiler Construction / N. Wirth. — Zürich: Addison-Wesley, 2017. — 44 p.

Приложение А. GitHub-репозиторий

Разработанный компилятор размещен в GitHub-репозитории по адресу https://github.com/nikitaodnorob/pl-compiler. Он содержит git-подмодуль Roslyn, нацеленный на репозиторий https://github.com/nikitaodnorob/roslyn-csharp/tree/diploma

`roslyn-csharp` — репозиторий Roslyn, из которого были удалены компилятор VB.NET и проекты, ненужные для сборки компилятора языка C#. В ветви `diploma` данного репозитория содержатся изменения семантики C#, которые были реализованы в данной работе:

1. Изменение модификаторов доступа членов `Code` и `Arguments` класса `Diagnostic` с `internal` на `public` (рассматривалось в 4.1) — коммит d1e4af07461a31d55cd159e3e12c79e641d8bcd8.
2. Разрешение объявление переменной, если переменная с таким именем была объявлена в родительской области (рассматривалось в 4.2) — коммит 02d8845ed628e527ba5e84953bac33ed70d7b702.
3. Запрет бинарных выражений, в которых левый операнд является идентификатором, начинающимся с символа `'#'`, а правый — не целочисленным выражением (рассматривалось в 4.3) — коммит d72fcd18f8dc6f886686f058dc5fc143e9be120d.

В самом репозитории `pl-compiler` присутствует файл `SYNTAX.MD`, в котором описан синтаксис модельного языка: https://github.com/nikitaodnorob/pl-compiler/blob/master/SYNTAX.MD.