Московский авиационный институт

(национальный исследовательский институт)

Институт «Компьютерные науки и прикладная математика»

**Курсовая работа**

**«Высокоуровневые виртуальные машины»**

**по курсу**

**«Системы программирования»**

**IV семестр**

*Студент:* Орусский В. Р.

*Группа: М8О-206Б-21*

*Руководитель:* *Киндинова В. В*.

*Оценка:*

*Дата:*

**Москва, 2023**

# **Common Intermediate Language**

Промежуточный язык Microsoft (Microsoft Intermediate Language, MSIL), также известный как Common Intermediate Language (CIL), является промежуточным языком, который используется в компиляции кода на языках программирования C# и Visual Basic .NET в исполняемый код Common Language Runtime (CLR). MSIL является промежуточным языком, потому что он не выполняется напрямую на процессоре, в отличие от машинного кода, который генерируется при компиляции непосредственно в целевой код.

Промежуточный язык Microsoft позволяет разработчикам писать приложения на .NET без привязки к конкретному языку программирования, поскольку он является общим для всех языков .NET и может быть компилирован с помощью любого компилятора .NET. Кроме того, MSIL обеспечивает безопасность кода, так как он подвергается строгой проверке во время компиляции, что уменьшает риск ошибок и повышает стабильность приложений.

Одним из важных преимуществ MSIL является его портируемость, что означает, что код, написанный на одной платформе, может быть скомпилирован и выполнен на другой платформе без изменений. Это обеспечивает гибкость и упрощает разработку исходного кода, который может работать на нескольких платформах, что особенно полезно для крупных организаций и разработчиков, которые хотят создавать приложения, которые могут выполняться на разных платформах.

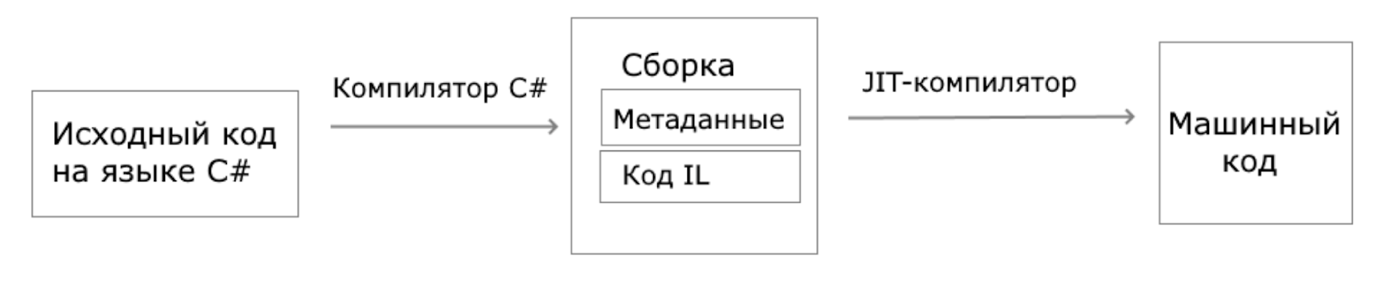


Рисунок 1 – процесс компиляции C# кода

Сравнение основных функций одной и той же программы на разных языках программирования. Для начала напишем самую популярную программу на любом языке программирования – вывод «Hello World!» в консоль. На рисунке №2 происходит вывод в консоль, прописанный внутри main функции.

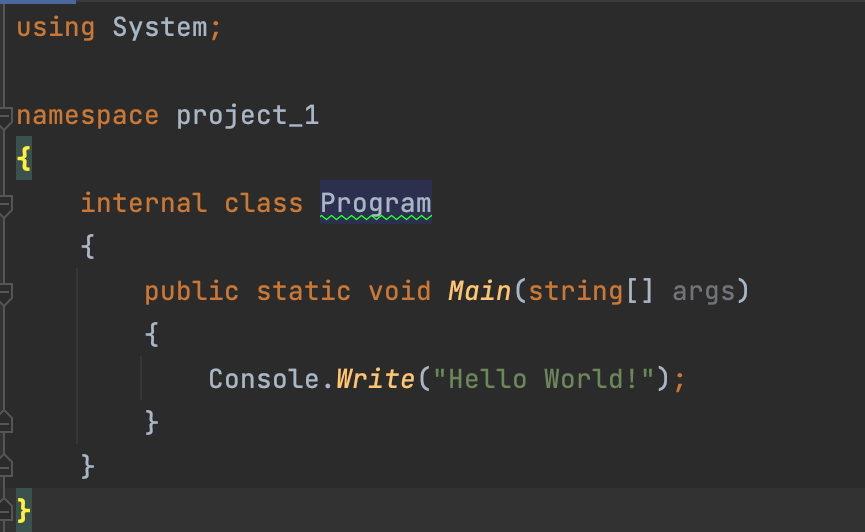
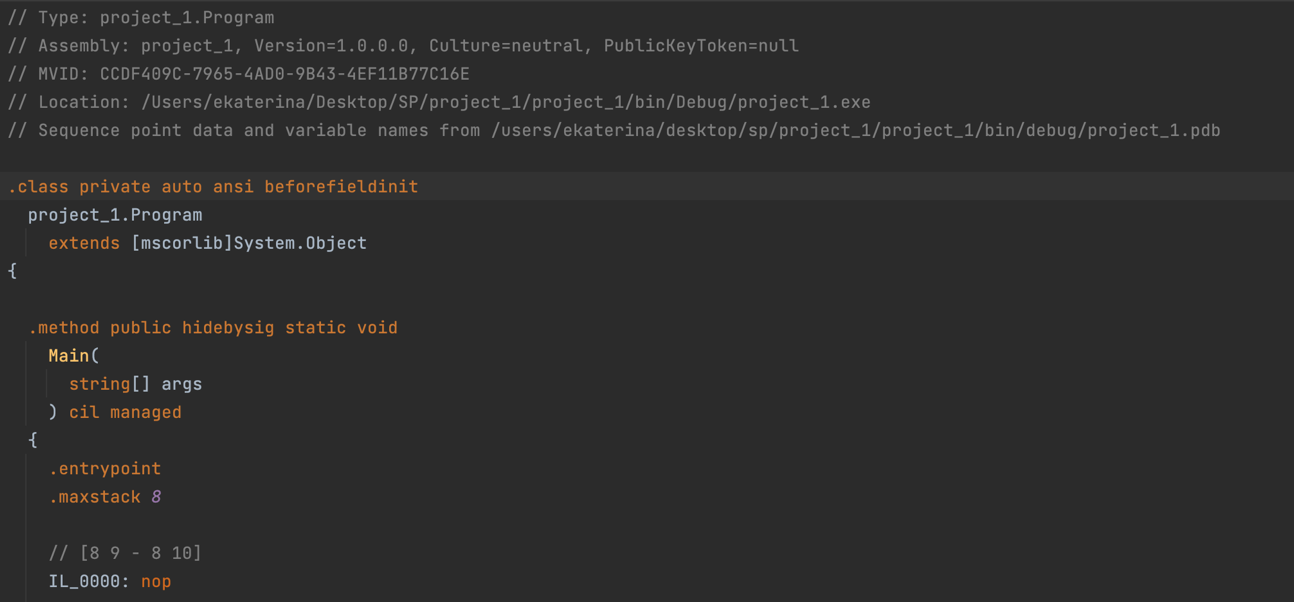


Рисунок 2 – код на языке C#

Чтобы получить IL код существует два инструмента: ReSharper от JetBrains, с помощью которого можно получить промежуточное представление C# кода и встроенный в Visual Studio дизассемблер. Используя второй инструмент, был получен следующий код:



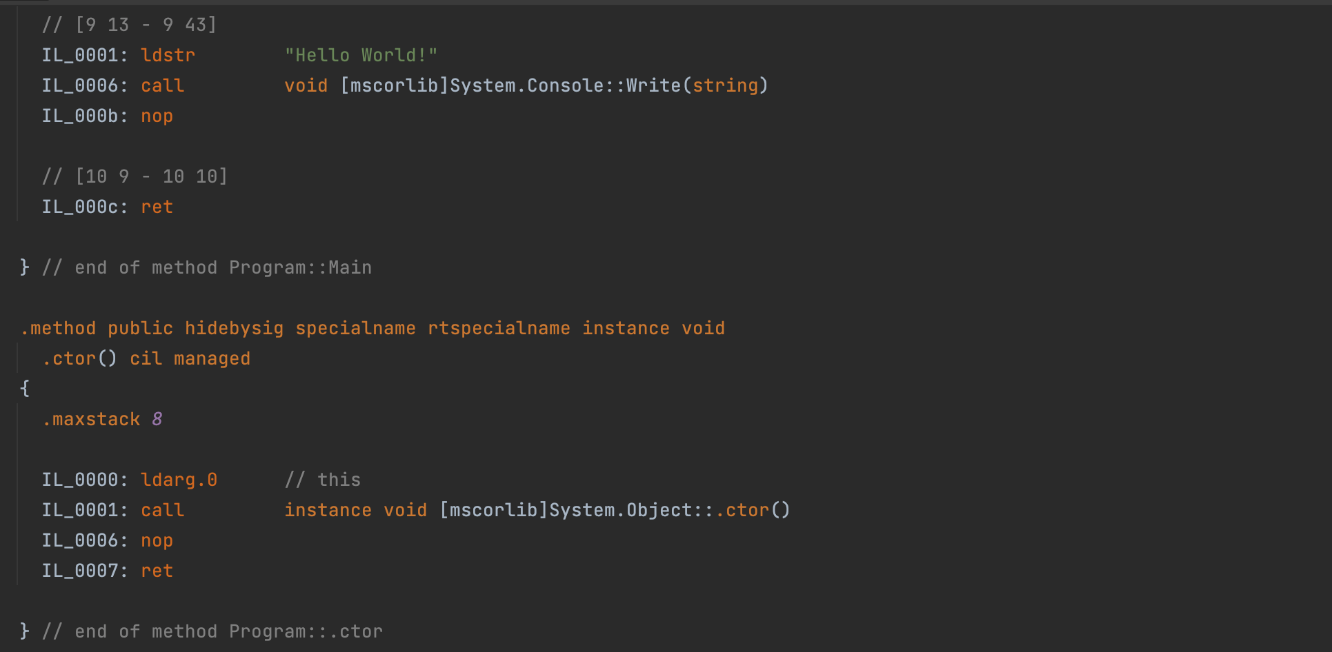


Рисунок 3 – код на языке IL

Сравнение C# и IL

|  |  |  |
| --- | --- | --- |
| Функция | C# | IL |
| 1. создание класса | internal class Program | .class private auto ansi beforefieldinit  project\_1.Program  extends [mscorlib]System.Object |
| 1. создание метода | public static void Main(string[] args) | .method public hidebysig static  void Main( string[] args)  cil managed |
| 1. вывод в консоль | Console.Write("Hello World!"); | IL\_0001: ldstr "Hello World!"  IL\_0006: call void [mscorlib]System.Console::Write(string)  IL\_000b: nop |

Создадим грамматику для перевода функций на язык C#:

*, где*

;

;

;

*.*

Правила регулярной грамматики:

Для более простого восприятия, некоторые нетерминалы будут заключаться в стрелочные скобки.

p1: ПРОГРАММА → using <<БИБЛИОТЕКА>>; namespace <<ПРОСТРАНСТВО\_ИМЕН>>

p2: БИБЛИОТЕКА → System

p3: ПРОСТРАНСТВО\_ИМЕН → namespace {<<АТРИБУТ КЛАСС>>}

p4: АТРИБУТ → interal | public | static | void

p5: КЛАСС → class <<name\_class>> {<<АТРИБУТ МЕТОД>>}

p6: МЕТОД → Main (<<arguments>>) {<<ФУНКЦИЯ>>}

p7: ФУНКЦИЯ → Console.Write(<<cout>>);

p8: namespace → project\_1

p9: name\_class → Program

p10: arguments → string[] args

p11: cout → Hello World!

, где

T = using, [:;{}], namespace, System, interal, public, static, void, class, Main, '(',')' Console.Write, project\_1, Program, string[┤]args, Hello World!, ‘.’, extends, [mscorlib]System.Object, [mscorlib]System.Console::Write, auto, ansi, beforefieldinit, hidebysig, cil managed, entrypoint, maxstack, 2, 4, 8, 16, nop, ldstr, call, ret, specialname, rtspecialname, instance, ldarg.0, IL\_0000, …, IL\_000z, ctor

;

;

*.*

*Правила регулярной грамматики:*

p1: ПРОГРАММА → <<КЛАСС>> <<ПРОСТРАНСТВО\_ИМЕН>> <<БИБЛИОТЕКА>> {<<МЕТОД>> <<СТАНДАРТНЫЙ\_МЕТОД>>}

p2: БИБЛИОТЕКА → extends <<lib>>

p3: lib → [mscorlib]System.Object | [mscorlib]System.Console::Write

p4: ПРОСТРАНСТВО\_ИМЕН → <<namespace>>.<<name\_class>>

p5: АТРИБУТ → privat | public | static | void | auto | ansi | beforefieldinit | hidebysig | cil managed | specialname | rtspecialname | instance

p6: КЛАСС → .class <<АТРИБУТ>>

p7: МЕТОД → .method <<АТРИБУТ>> Main(<<arguments>>) <<АТРИБУТ>> {<<in\_method>>}

p8: in\_method → .entrypoint.maxstack<<memory>> <<code\_name>> <<operation>> <<ФУНКЦИЯ>> <<code\_name>> <<operation>>

p9: memory → 2 | 4 | 8 | 16

p10: code\_name → IL\_0000|…|IL\_000z

p11: operation → nop | ldstr | call | ret | ldarg.0

p12: namespace → project\_1

p13: name\_class → Program

p14: arguments → string[] args

p15: cout → Hello World!

p16: ФУНКЦИЯ →

<<code\_name>> <<operation>> <<cout>>

<<code\_name>> <<operation>> <<АТРИБУТ>> <<lib>> (arguments)

<<code\_name>> <<operation>>

p17: СТАНДАРТНЫЙ\_МЕТОД → .method <<АТРИБУТ>>.ctor() <<АТРИБУТ>> {.maxstack <<memory>>

<<code\_name>> <<operation>>

<<code\_name>> <<operation>> <<АТРИБУТ>> <<lib>>::.ctor()

<<code\_name>> <<operation>>

<<code\_name>> <<operation>>}

Из полученных грамматик, можно составить таблицу для переводов из общих нетерминалов в терминалы C# и CIL. Подобные переводы производятся при запуске программ, написанных на C#. Сначала компилятор преобразует C# код в некоторые состояния (нетерминалы), после чего они переводятся в CIL.

Таблица 2. Таблица переводов

|  |  |  |
| --- | --- | --- |
| N | Правила грамматики | Элемент перевода |
| 1 | ПРОГРАММА →using <<БИБЛИОТЕКА>>;  namespace <<ПРОСТРАНСТВО\_ИМЕН>> | ПРОГРАММА →<<КЛАСС>> <<ПРОСТРАНСТВО\_ИМЕН>> <<БИБЛИОТЕКА>> {<<МЕТОД>> <<СТАНДАРТНЫЙ\_МЕТОД>>} |
| 2 | БИБЛИОТЕКА → System | БИБЛИОТЕКА → extends <<lib>> |
| 3 | ПРОСТРАНСТВО\_ИМЕН → namespace {<<АТРИБУТ КЛАСС>>} | ПРОСТРАНСТВО\_ИМЕН → <<namespace>>.<<name\_class>> |
| 4 | АТРИБУТ → interal | public | static | void | АТРИБУТ → privat | public | static | void | auto | ansi | beforefieldinit | hidebysig | cil managed | specialname | rtspecialname | instance |
| 5 | КЛАСС → class <<name\_class>> {<<АТРИБУТ МЕТОД>>} | КЛАСС → .class <<АТРИБУТ>> |
| 6 | МЕТОД → Main (<<arguments>>) {<<ФУНКЦИЯ>>} | МЕТОД → .method <<АТРИБУТ>> Main(<<arguments>>) <<АТРИБУТ>> {<<in\_method>>} |
| 7 | ФУНКЦИЯ → Console.Write(<<cout>>) | ФУНКЦИЯ →  <<code\_name>> <<operation>> <<cout>>  <<code\_name>> <<operation>> <<АТРИБУТ>> <<lib>> (arguments)  <<code\_name>> <<operation>> |

Так как язык IL является более низкоуровневым по сравнению с C#, то некоторые стандартно встроенные операции в C#, в IL необходимо вызывать явно в коде. Например – операция «ctor()».

Представление основной операции в IL

|  |  |  |  |
| --- | --- | --- | --- |
| Оператор | Код операции | Операнды | Семантика оператора |
| Вызывается для обеспечения базовой реализации в данном методе | CTOR | lib | Обращается к стандартной библиотеке System(lib) для обеспечения базовой реализации |

# **Вывод**

В данной работе продемонстрировано, как компиляторами (интерпретаторами) обрабатываются мультиплатформенные языки программирования. С помощью набора общих правил, код, написанный на исходном языке, приводится к более низкоуровневому языку (который отдаёт команды напрямую железу), а те в свою очередь выполняются вне зависимости от ОС, переменных окружения и прочих переменных, которые вызывают перебои в работе программ на разных платформах.