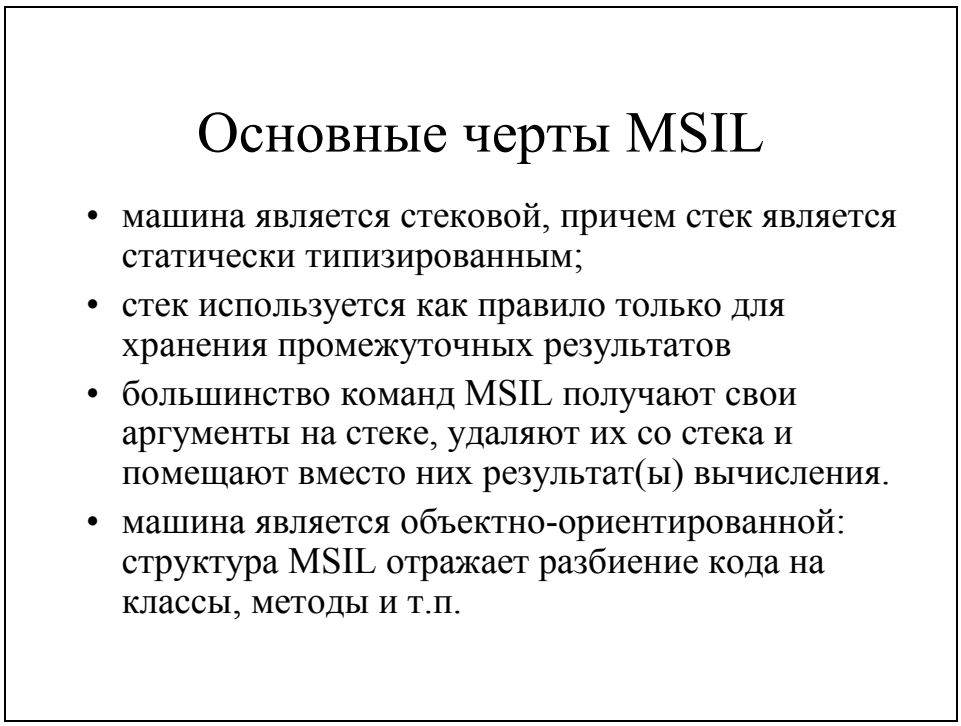
14. Генерация MSIL

- Основные свойства MSIL
- Пример генерации MSIL по программе на Си-бемоль
- Механизм рефлексии в .NET (на примере)
- Процесс генерации MSIL

Лекция 14. Генерация MSIL

В этой лекции рассматриваются следующие вопросы:

- Основные свойства MSIL, целевой платформы нашего компилятора
- Пример генерации MSIL по программе на C♭
- Механизм рефлексии в .NET (на примере)Процесс генерации MSIL по дереву разбора исходной программы



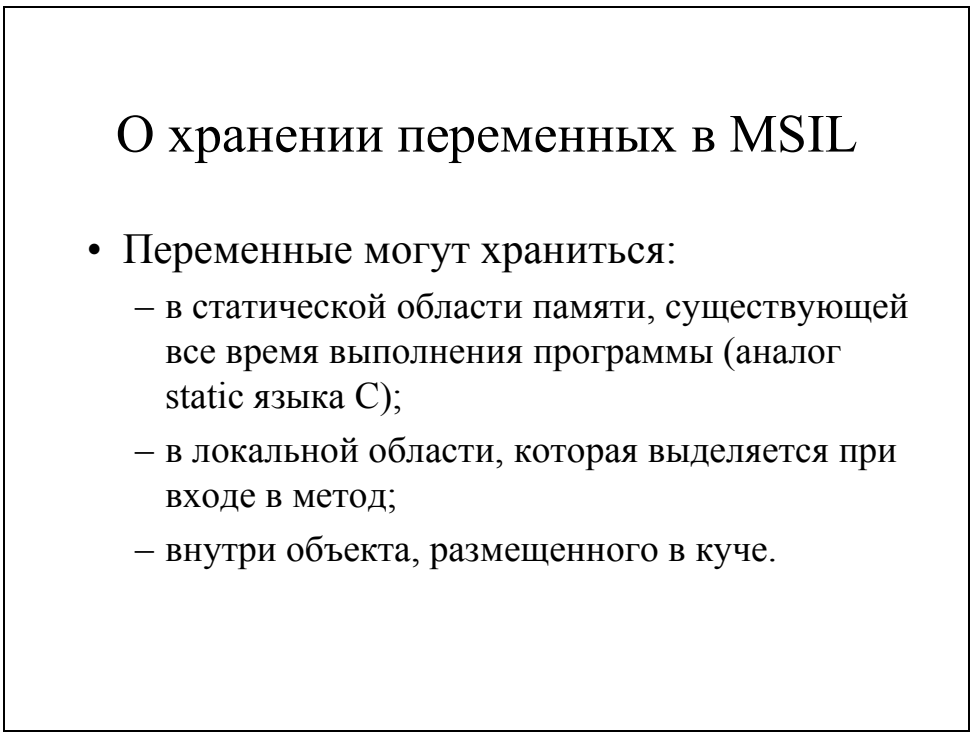
Основные черты MSIL

Трансляторы, ориентированные на платформу .NET, должны генерировать код на MSIL, являющемся языком ассемблера некоторой виртуальной машины. Поэтому мы начнем с рассмотрения основных свойств целевой платформы нашего компилятора. В данной лекции мы не пытаемся дать полного и детального описания всех команд платформы; задача данного обзора – дать представление об основных идеях MSIL.

Программы на MSIL переводятся в исполняемый код реального процессора лишь непосредственно перед исполнением с помощью так называемой *компиляции времени исполнения* (которую также называют Just-In-Time compilation или динамической компиляцией). При этом выполняется довольно сложный типовой анализ программы и проверки условий корректности кода.

Основные черты архитекутры виртуальной машины MSIL таковы:

- машина является стековой, причем стек является статически типизированным; это означает, что в каждой точке программы JIT-compiler должен иметь возможность статически определить типы содержимого всех ячеек стека. На практике это означает, например, невозможность написать код, кладущий в цикле значения на стек
- ячейки стека представлены как 4-байтовые или 8-байтовые знаковые целовые (обозначаемые как I4 и I8, соответстсвенно; более короткие значения представляются как 4-байтовые);
- стек используется как правило только для хранения промежуточных результатов вычисления (т.е. не рекомендуется хранить на стеке временные переменные, такие, как счетчик цикла и т.п.)
- большинство команд MSIL получают свои аргументы на стеке, удаляют их со стека и помещают вместо них результат(ы) вычисления.
- машина является объектно-ориентированной: структура MSIL отражает разбиение кода на классы, методы и т.п.



О хранении переменных в MSIL

Существует несколько вариантов хранения переменных в MSIL:

- в статической области памяти, существующей все время выполнения программы
- в локальной области, которая выделяется при входе в метод;
- внутри объекта, размещенного в куче.

Статическая область памяти предназначена, например, для статических полей класса, глобальных переменных, констант и т.п. Например, в этой области памяти должны находиться переменные, описанные с модификатором static в языке C.

Локальная память выделяется для автоматических и временных переменных, параметров и т.п.

Куча предназначена для хранения динамических объектов. Кроме того, при реализации языков со вложенными процедурами и процедурными значениями, необходимо обеспечить доступ из вложенной процедуры к переменным из объемлющей среды. При этом, если язык допускает использование вложенных процедур в качестве переменных процедурного типа, то единственным способом для реализации такого механизма в безопасном режиме является построение в куче так называемого *замыкания процедуры (closure)*, которое будет содержать переменные из объемляющих сред, необходимые для работы данной процедуры.

Работа с указателями в MSIL

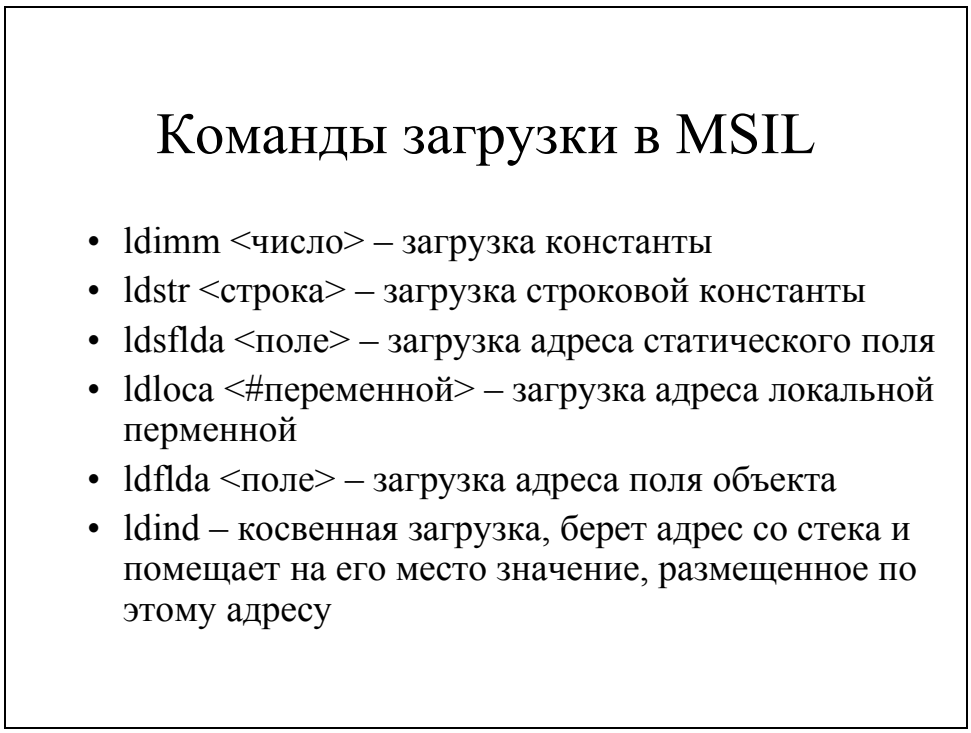
- Правила манипуляции с адресами переменных и полей запрещают хранение адреса в тех ситуациях, когда транслятор не может гарантировать существование объекта, на который ссылается адрес.
- В .Net SDK входит программа PEVerify, которая осуществляет проверку типовой корректности исполняемого модуля

Работа с указателями в MSIL

Машина MSIL ориентирована на безопасность работы с указателями. Например, правила манипуляции с адресами переменных и полей запрещают хранение адреса в тех ситуациях, когда транслятор не может гарантировать существование объекта, на который ссылается адрес.

На практике это означает, что в безопасном режиме указатели на локальные переменные могут быть только переданы параметрами в другие функции; присваивание их в другие переменные запрещено.

Кроме того, в .NET SDK входит программа PEVerify, которая осуществляет проверку типовой корректности исполняемого модуля.

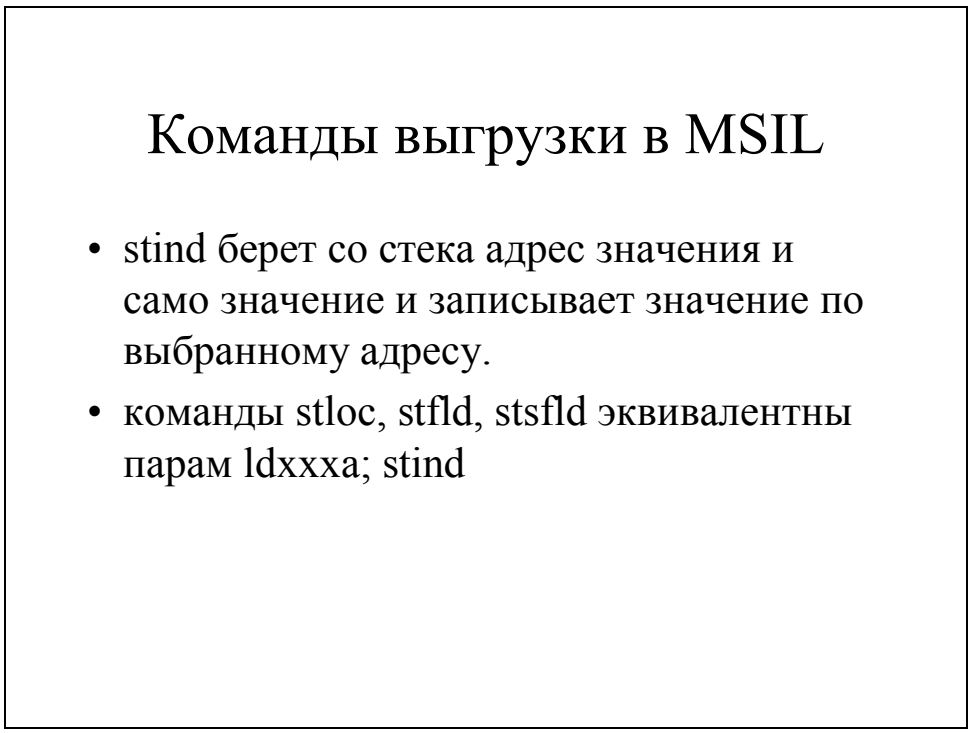


Команды загрузки в MSIL

Перейдем к рассмотрению основных команд MSIL. Начнем с команд загрузки:

- ldimm <число> – загрузка константы
- ldstr <строка> – загрузка строковой константы
- ldsflda <поле> – загрузка адреса статического поля
- ldloca <# переменной> – загрузка адреса локальной перменной
- ldflda <поле> – загрузка адреса поля объекта
- ldind – косвенная загрузка, берет адрес со стека и помещает на его место значение, размещенное по этому адресу

Поскольку, как правило, нам необходим не адрес переменной, а ее значение, то существуют команды загрузки значения на стек: ldsfld, ldloc, ldfld. Каждая из этих команд эквивалентна паре команд `ldxxxa; ldind`.



Команды выгрузки в MSIL

Команды выгрузки в основном построены так же, как и команды загрузки (только с противоположным результатом работы), и потому не особо нуждаются в комментариях.

Отметим команду stind, которая берет со стека адрес значения вместе с самим значением и записывает значение по выбранному адресу.

Кроме того, упомянем, что как и в командах загрузки, команды

```
stloc, stfld, stsfld
```

эквивалентны следующим парам команд:

```
ldxxxxa; stind
```

Арифметические команды MSIL

- Арифметические команды MSIL выполняют вычисления на стеке
- Вычислительные команды (ADD, MUL, SUB…) существуют в знаковом и беззнаковом (.u) вариантах, а также могут выполнять контроль за переполнением (.ovf)
- Кроме того, есть логические команды AND, OR, XOR (только знаковые, без контроля переполнения) и операции сравнения

Арифметические команды MSIL

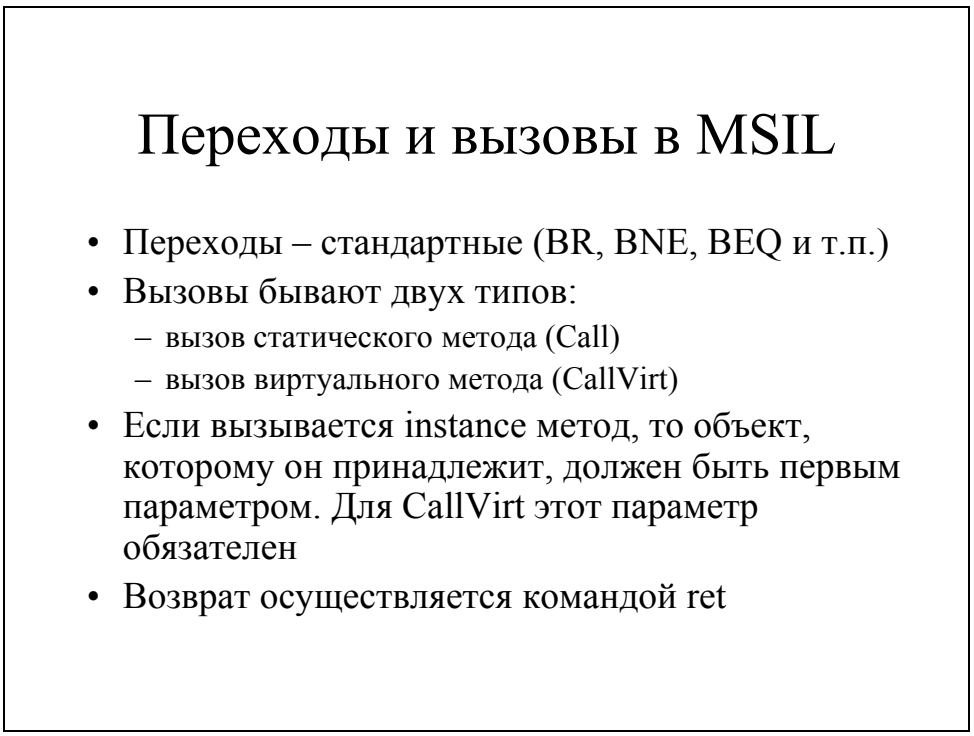
Арифметические команды позволяют выполнять вычисления. Все они берут аргументы со стека и кладут на их место результат.
Команды целочисленной арифметики существуют в знаковом и беззнаковом (с суффиксом .u) вариантах и могут быть записаны с суффиксом обработки переполнения (.ovf), который порождает исключение при возникновении переполнения. К этим командам относятся: ADD, SUB, MUL, DIV, MOD.

Также есть логические операции, которые не могут быть знаковыми или иметь суффикс контроля за переполнением. Логические операции делятся на бинарные (AND, OR, XOR) и унарные (NOT, NEG).
Наконец, в MSIL есть некоторый набор операций сравнения. Эти операции снимают со стека операнды и помещают на их место результат 0/1. Они могут быть беззнаковыми или знаковыми (с суффиксом .s). Кроме того, существуют специальные варианты сравнения, учитывающие возможность сравнения чисел с плавающей запятой различного порядка (такие операции имеют суффикс .un).

Интересно отметить, что при наличии полного комплекта операций перехода, создатели MSIL не включили в систему команд операций сравнения "<=" и ">=". Это приводит к тому, что для целочисленных значений операцию "<=" приходится эмулировать с помощью следующего набора команд:

```
cgt; ldc.i4.0; ceq
```

Соответственно, для вещественных значений операцию "<=" необходимо представлять аналогично, только первая команда должна быть заменена на cgt.un. Тем не менее, с точки зрения конечной программы в машинных кодах это, видимо, несущественно, так как такой набор операций легко соптимизировать в одну ассемблерную команду целевой архитектуры.



Переходы и вызовы в MSIL

Переходы в .NET мало чем отличаются от используемых в обычных ассемблерах. Сразу отметим, что все команды переходов существуют в стандартном и коротком виде (для записи коротких переходов используется суффикс .s). Помимо обычного безусловного перехода (`BR`), в MSIL существует целый ряд условных переходов (`BEQ`, `BNE`, `BGT`, `BRFALSE` – переход по false, null или нулю на верхушке стека – и все прочие переходы, включая беззнаковые и неупорядоченные варианты).
Существует две основных команды вызова:

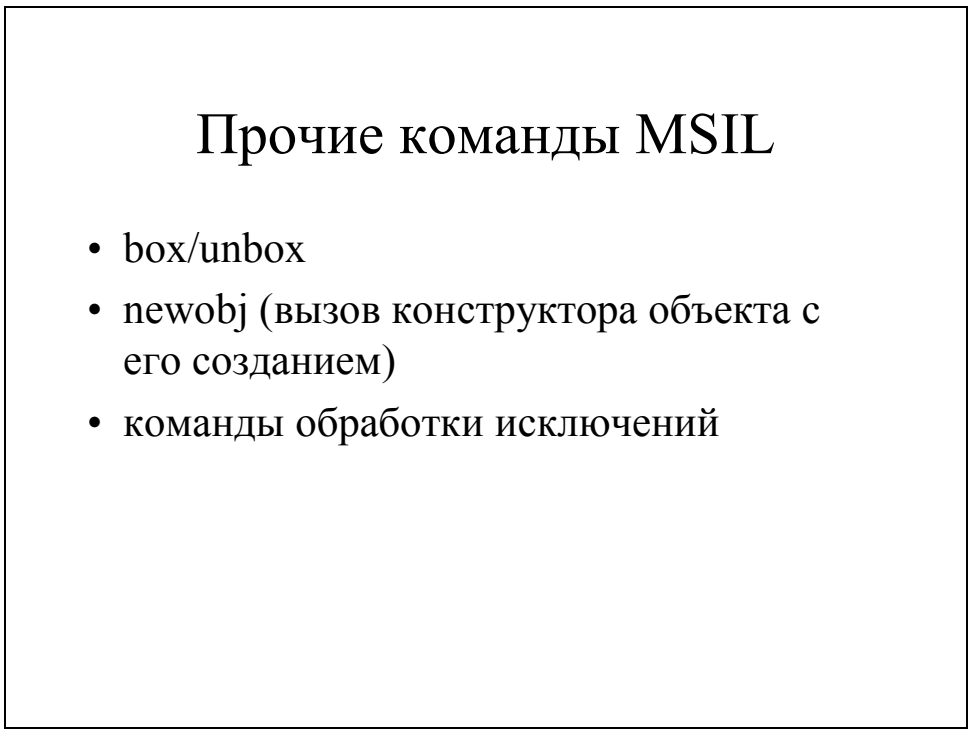
- вызов статического метода (`CALL`)
- вызов виртуального метода (`CALLVIRT`)

Если вызывается метод экземпляра объекта, то объект, которому он принадлежит, должен быть первым параметром; для `CALLVIRT` этот параметр обязателен, поскольку виртуальных статических методов в .NET не бывает.

Команда вызова может снабжена префиксом tail. Это означает, что значение, возвращаемое вызываемой процедурой, является также возвращаемым значением и для вызывающей процедуры. В таком случае можно превратить вызов процедуры с последующим возвратом значения в одну команду безусловного перехода на вызываемую процедуру; для этого также необходимо удалить текущую рамку стека. Эта оптимизация позволяет избежать разрастания стека во многих рекурсивных алгоритмах. Недостатком такой оптимизации являются трудности отслеживания стека вызовов при отладке.

Помимо основных методов вызова, .NET предоставляет команду `CALLI`, выполняющую небезопасный вызов процедуры по адресу точки входа.

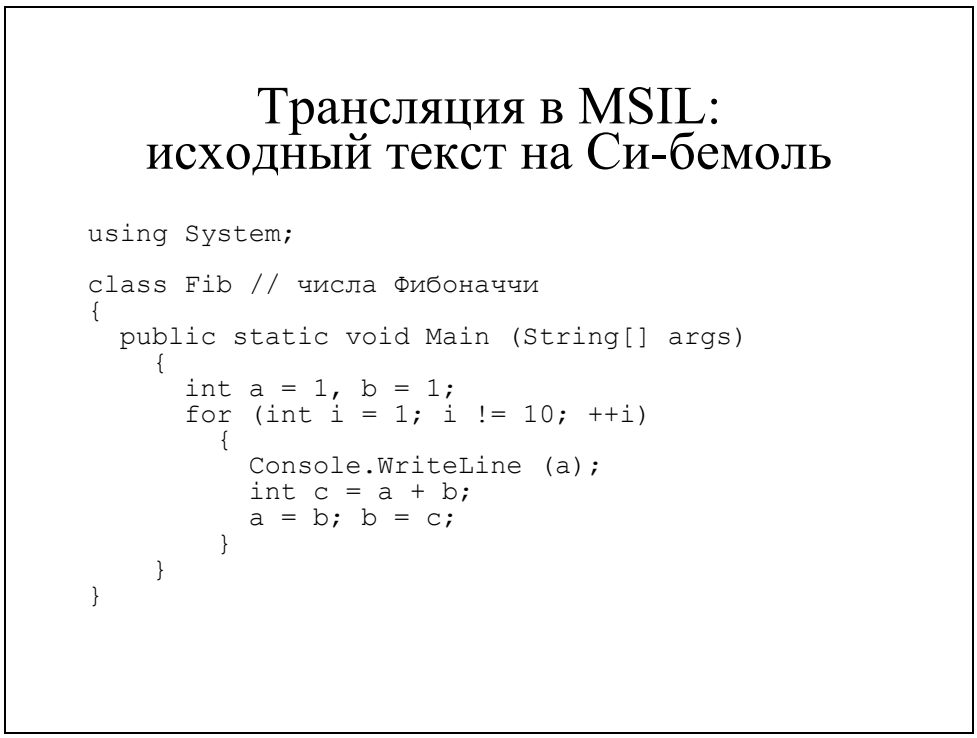
Возврат осуществляется командой `RET`, которая для методов, не возвращающих результат, не имеет параметров. Для всех прочих методов эта команда ожидает параметр – возвращаемое значение на верхушке стека.



Прочие команды MSIL

Упомянем вкратце некоторые интересные команды MSIL, которые редко встречаются в традиционных ассемблерах.

Во-первых, внимания заслуживают специальные команды `BOX` и `UNBOX`, реализующие функциональность упаковки и распаковки значений (см. лекцию 2).
Во-вторых, в MSIL предусмотрена специальная команда для создания нового объекта – `NEWOBJ`. Семантика этой команды такова: создается новый объект и для него вызывается конструктор. Эта операция является критичной для обеспечения целостности данных, так как при ее выполнении гарантируется инициализация объекта (а иначе появляется потенциальная возможность использования "мусорных" ссылочных значений).
В-третьих, отметим, что MSIL содержит специальные команды для обработки исключений (`THROW`,`RETHROW`,`ENDFINALLY`,`ENDFILTER`,`LEAVE`), что не очень традиционно для низкоуровневых языков.Общая идея реализации исключений заключается в следующем: транслятором создается специальная таблица обработчиков исключений в данном try-блоке; затем при возникновении исключения виртуальная машина .NET просматривает эти таблицы и вызывает соответствующие обработчики. На самом деле, детали реализации исключений не очень существенны, так как при генерации MSIL можно воспользоваться существующими примитивами более высокого уровня (см. ниже про Reflection.Emit).



Трансляция в MSIL: исходный текст на C♭

Продемонстрируем трансляцию в MSIL на примере следующей программы, написанной на C♭ и вычисляющей числа Фибоначчи:

```
using System;

class Fib
{
   public static void Main (String[] args)
   {
      int a = 1, b = 1;
      for (int i = 1; i != 10; ++i)
      {
         Console.WriteLine (a);
         int c = a + b;
         a = b; b = c;
      }
   }
}
```

На следующих слайдах мы покажем результаты трансляции этой программы в MSIL.

Трансляция в MSIL: пример

```
// объявление имени assembly
.assembly fib as "fib" { /* здесь могут быть параметры */ }
.class public Fib {
  .method public static void Main () {
    .entrypoint // означает начало assembly
       .locals (int32 a, int32 b)
       ldc.i4.1  // загрузка константы 1
       stloc   a // сохранение 1 в a (a = 1)
       ldc.i4.1
       stloc   b // аналогично: b = 1
       ldc.i4.1  // загрузка 1 на стек (счетчик цикла)
Loop:
        ldloc   a
        call    void System.Console::WriteLine(int32)
        ldloc   a // stack: 1 a
        ldloc   b // stack: 1 a b
        add       // stack: 1 (a+b)
        ldloc   b
        stloc   a // a = b
        stloc   b // b = (a+b)
        ldc.i4.1
        add     // инкремент счетчика
        dup
        ldc.i4.s   10
        bne.un.s   Loop // сравнение и переход на новую итерацию
        pop  // удаление счетчика цикла со стека
        ret
  }
}
```

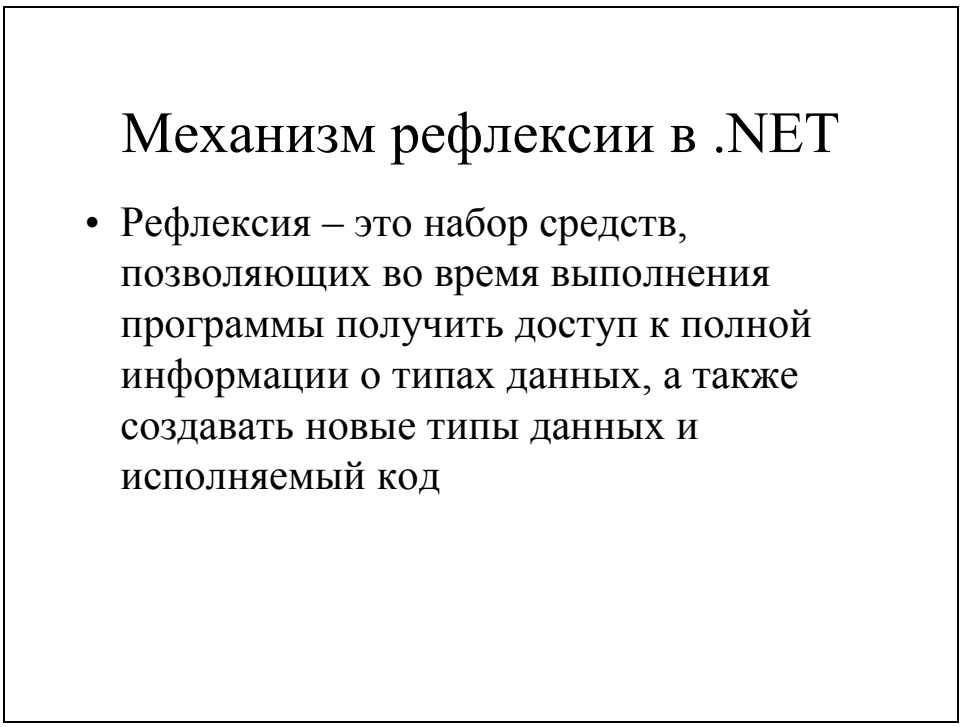
Трансляция в MSIL: пример

Программа на MSIL начинается с объявления имени сборки, в которую входит данная программа. Слушателям рекомендуется ознакомиться с более подробным описанием сборки самостоятельно.

Затем объявляется класс Fib, в котором производятся вычисления. Здесь же находится основная точка входа в сборку (.entrypoint внутри Main). Затем объявляются локальные переменные; отметим, что в процессе реальной трансляции имена этих переменных будут утеряны.

Наконец, происходит инициализация переменных, подготовка к началу цикла (загрузка счетчика цикла на стек) и выполнение основных вычислений программы: печать очередного числа Фибоначчи, загрузка рабочих переменных на стек, их сложение, присваивание результатов и увеличение счетчика.

Затем происходит сравнение счетчика цикла с максимальным значением цикла и в случае выполнения неравенства "счетчик не равен 10" происходит переход на начало цикла. По окончании цикла происходит удаление счетчика цикла со стека и выход из метода.



Механизм рефлексии в .NET

В .NET предусмотрен специальный механизм доступа к метаданным приложения, который называется *рефлексия*. С помощью рефлексии мы можем получить доступ к полной информации о типах данных приложения во время исполнения, а также можем создавать новые типы данных и исполняемый код. Сначала мы изучим, каким образом можно получить доступ к уже существующим метаданным.

С помощью методов класса Reflection по данному объекту или его имени можно получить значение типа Type, которое содержит практически исчерпывающую информацию об этом классе – в частности, можно получить его список полей, методов, непосредственного предка данного класса, интерфейсы, которые реализованы в этом объекте и т.д. Всю эту информацию можно хранить и обрабатывать с помощью специальных классов, таких, как MethodInfo, FieldInfo и т.п. Таким образом, для данной сборки можно получить список ее модулей, у которых можно получить список типов и так далее до любых подробностей устройства объекта (вплоть до конкретного бинарного кода его методов, разумеется, только при наличии соответствующих прав доступа).

Отметим, что не все классы, имеющие отношение к рефлексии находятся в пространстве имен Reflection, например, сам класс Type находится в пространстве имен System. Такая "нелогичность" объясняется тем, что класс Type используется не только в ситуациях, связанных с рефлексией. Например, параметром метода ToArray класса ArrayList является значение типа Type, задающее тип элемента массива.
Поскольку рефлексия является сложной и объемной темой, мы продемонстрируем этот механизм на небольшом примере, в котором реализована функция, позволяющая распечатать объект практически любого типа в виде, пригодном для чтения человеком. Схожий пример (MetaInfo) можно также найти в примерах, входящих в состав .NET SDK.

Пример на рефлексию

```
// Пример, демонстрирующий использование рефлексии
// Автор: Антон Москаль,
// Санкт-Петербургский государственный университет, 2001

using System;
using System.Text;
using System.Reflection;

class Sample
{
   public struct Struct { public String name; public int[] vec; }
   public static void Main (String[] args)
   {
       Struct s;
       s.name = "Name";
       s.vec = new int [2];
       s.vec [0] = 1;
       s.vec [1] = 4;
       Console.WriteLine (Refl.Repr(s));
   }
}
```

Пример на рефлексию

На данном слайде приведена программа, вызывающая упомянутую выше функцию распечатки объекта. Результатом работы данного примера будет следующая строка:

```
STRUCT{name:Name, vec:[2]{1, 4}}
```

Рассмотрим реализацию функции класса Refl:

```
class Refl
{
   public static string Repr (Object o)
   {
       // получаем дескриптор типа o
       Type t = o.GetType ();

       // если o - массив:
       if (t.IsArray)
          return ReprArray ((Array) o);
       // если o - структура (value type, не являющийся встроенным типом):
       else if (t.IsValueType && !t.IsPrimitive)
          return ReprStruct (t, o);
       // в противном случае используем стандартную фукнцию ToString:
       else
          return o.ToString ();
   }

   public static string ReprArray (Array a)
   {
       string res = "["+a.Length+"]{";
       // и дальше в цикле добавляем в res
       // список представлений его элементов:
       String sep = "";
       for (int i = 0; i != a.Length; ++i, sep = ", ")
          // для бестипового массива индексацию почему-то использовать
          // нельзя, поэтому приходится пользоваться функцией GetValue:
          res += sep + Repr (a.GetValue (i));

       return res + "}";
   }
```

```
    public static string ReprStruct (Type t, Object o)
    {
        // получаем массив дескрипторов полей:
        FieldInfo [] flds = t.GetFields ();
        string res = "STRUCT{";
        String sep = "";
        // в цикле добавляем представления полей в виде
        // <имя поля>: <значение поля>
        foreach (FieldInfo fld in flds)
        {
           // метод GetValue в дескриторе поля
           // позволяет выбрать это поле из объекта:
           res += sep + fld.Name + ":" + Repr (fld.GetValue (o));
           sep = ", ";
        }
        return res + "}";
    }
}
```

Генерация MSIL с использованием Reflection.Emit

Практически для каждой сущности в MSIL, которая может быть получена с помощью механизма рефлексии, в Reflection.Emit существует специальный класс с суффиксом Builder, который может быть использован для генерации этой сущности (например, AssemblyBuilder или TypeBuilder). Отметим, что соглашение об образовании имен не всегда строгое – например, классу FieldInfo соответствует FieldBuilder, а не FieldInfoBuilder, как можно было бы подумать.

Классы с суффиксом Builder предназначены для генерации описываемых сущностей. В каждом таком классе есть методы, позволяющие добавлять поля, методы и прочее содержание, например:

```
TypeBuilder tb;
MethodBuilder mb = tb.DefineMethod("Call");
ILGenerator ilg = mb.GetILGenerator();
```

Класс ILGenerator, использованный в приведенном примере, уже может генерировать собственно код виртуальной машины путем использования метода Emit и различных дополнительных методов (DefineLabel, DeclareLocal, BeginScope-EndScope и т.п.).

О последовательности генерации MSIL

Структура MSIL задает определенную последовательность генерации кода, в которой код генерируется в несколько проходов:

- вначале создается объект AssemblyBuilder, который будет использоваться для создания сборки
- затем при помощи метода DefineDynamicModule создаются объекты типа ModuleBuilder, которые будут использоваться для порождения входящих в сборку модулей (отметим, что в случае создания модуля, не входящего в сборку, первый шаг опускается)
- далее необходимо создать TypeBuilders для классов всех уровней (в том числе, и для вложенных классов); при этом конкретный порядок создания классов может быть весьма нетривиальным из-за возможности ссылок классов друг на друга в иерархии наследования
- затем создаются билдеры для методов и полей классов
- и вот только в этот радостный момент можно генерировать собственно код методов

Такая последовательность шагов при генерации является практически единственно возможной, поскольку на каждом следующем этапе нам может понадобиться сослаться на сущности, созданные на предыдущем этапе. Например, для создания поля нам нужно иметь возможность сослаться на его тип, а для генерации кода – иметь возможность сослаться на используемые поля и методы.

Генерация кода виртуальной машины

Рассмотрим процесс генерации следующего кода:

```
class LowLevelSample {
    public static void Run() {
       int i;
       i = 0;
       try {
          Start:
             if (i == 10) throw new Exception();
             Console.WriteLine (i);
             i = i + 1;
             goto Start;
       }
       catch (Exception) {
          Console.WriteLine ("Finished");
       }
    }
}
```

Генерация кода виртуальной машины

В качестве примера генерации кода для виртуальной машины попробуем породить код на MSIL для следующей программы на C#:

```
class LowLevelSample {
    public static void Run() {
       int i;
       i = 0;
       try {
          Start:
             if (i == 10) throw new Exception();
             Console.WriteLine (i);
             i = i + 1;
             goto Start;
       }
       catch (Exception) {
          Console.WriteLine ("Finished");
       }
    }
}
```

На примере этой программы мы подробно рассмотрим процесс генерации кода, в том числе и такие интересные аспекты, как генерация меток и обработка исключительных ситуаций. Данная программа носит чисто демонстрационный характер – разумеется, использовать исключения для выхода из цикла в нормальной программе не следует.

Создание сборки и класса

Вначале мы создаем сборку, в которой будет находиться модуль, содержащий наш класс:

```
AppDomain ad = System.Threading.Thread.GetDomain();
AssemblyName an = new AssemblyName();
an.Name=System.IO.Path.GetFileNameWithoutExtension("numbers.exe");
AssemblyBuilder ab = ad.DefineDynamicAssembly
               (an,  AssemblyBuilderAccess.RunAndSave);
ModuleBuilder modb = ab.DefineDynamicModule
               ("numbers.exe", "numbers.exe");
modb.CreateGlobalFunctions();
TypeBuilder classb = modb.DefineType ("Sample");
MethodBuilder mb = classb.DefineMethod
               ("Run",
                MethodAttributes.Public|MethodAttributes.Static,
                typeof (void), null);
```

Создание сборки и класса

Вначале необходимо создать домен приложения, в котором мы будем создавать класс:

```
AppDomain ad = System.Threading.Thread.GetDomain();
```

Далее мы создаем имя сборки:

```
AssemblyName an = new AssemblyName();
an.Name = System.IO.Path.GetFileNameWithoutExtension ("numbers.exe");
```

Теперь создаем класс, генерирующий сборку, и используем его для создания модуля, который будет содержать наш класс (отметим, что атрибут RunAndSave означает, что данная сборка может быть исполнена или сохранена на диске):

```
AssemblyBuilder ab = ad.DefineDynamicAssembly
             (an,  AssemblyBuilderAccess.RunAndSave);
ModuleBuilder modb = ab.DefineDynamicModule ("numbers.exe", "numbers.exe");
modb.CreateGlobalFunctions ();
```

Далее мы создаем билдер класса Sample и создаем в этом классе метод Run. Первый параметр DefineMethod задает имя метода, второй – атрибуты метода (в нашем случае – public и static), затем задается тип результата метода и массив типов параметров (если параметры у метода отсутствуют, как в нашем примере, то можно передать пустой указатель).

```
TypeBuilder classb = modb.DefineType ("LowLevelSample");
MethodBuilder mb = classb.DefineMethod
             ("Run",
              MethodAttributes.Public|MethodAttributes.Static,
              typeof (void), null);
```

Генерация кода: исключения, метки

Теперь мы можем перейти к собственно генерации кода; генерируется присваивание, начинается блок try, определяется и ставится метка

```
ILGenerator il = mb.GetILGenerator();
LocalBuilder var_i = il.DeclareLocal (typeof(int));
il.Emit (OpCodes.Ldc_I4_0);
il.Emit (OpCodes.Stloc, var_i);
Label EndTry = il.BeginExceptionBlock();
Label Start = il.DefineLabel();
il.MarkLabel (Start);
```

Генерация кода: исключения, метки

Теперь мы можем создать генератор кода и приступить к порождению машинного кода для метода Run. Класс ILGenerator содержит в себе методы, необходимые для порождения команд MSIL. Далее мы создаем локальную переменную, на которую можем дальше ссылаться с использованием ее билдера var_I. В случае наличия вложенных блоков их рекомендуется ограничивать вызовами BeginScope() и EndScope().

```
ILGenerator il = mb.GetILGenerator();
LocalBuilder var_i = il.DeclareLocal (typeof (int));
```

Теперь мы генерируем присваивание значения 0 в эту переменную. Метод Emit является основным методом для генерации кода MSIL и в качестве своего первого параметра принимает код команды. OpCodes содержит константы, соответствующие всем командам MSIL. Отметим, что метод Emit перегружен и может иметь различные параметры, тип которых зависит от генерируемой команды. Например, в последующем операторе мы используем параметр типа LocalBuilder для ссылки на локальную переменную:

```
il.Emit (OpCodes.Ldc_I4_0);
il.Emit (OpCodes.Stloc, var_i);
```

Для оформления try-catch блока используется метода BeginExceptionBlock, который "открывает" блок обработки исключений и создает метку, на которую впоследствии можно будет сослаться.

```
Label EndTry = il.BeginExceptionBlock();
```

Теперь мы заводим метку, которая понадобится для организации цикла. Интересно, что метод DefineLabel не генерирует никакого кода, а просто создает значение типа Label, которое в дальнейшем может быть использовано для генерации переходов. Рано или поздно метка должна быть привязана к позиции в MSIL при помощи вызова MarkLabel. Таким образом, DefineLabel соответствует объявлению идентификатора метки (в явном виде такая операция есть только в Pascal), в то время как MarkLabel соответствует определению (установке) метки.

```
Label Start = il.DefineLabel();
il.MarkLabel (Start);
```

Генерация кода: условный оператор

Далее мы генерируем условный оператор, исключение и завершаем цикл:

```
Label Next = il.DefineLabel();
il.Emit (OpCodes.Ldloc, var_i);
il.Emit (OpCodes.Ldc_I4, 10);
il.Emit (OpCodes.Bne_Un_S, Next);

Type[] no_types = new Type [0];
il.Emit (OpCodes.Newobj, typeof
(System.Exception).GetConstructor (no_types));
il.ThrowException (typeof (System.Exception));
il.MarkLabel (Next);

il.Emit (OpCodes.Ldloc, var_i);
Type [] arg = new Type [1];
arg [0] = typeof (int);
MethodInfo writeLine =typeof(System.Console).GetMethod
   ("WriteLine", arg);
il.EmitCall (OpCodes.Call, writeLine, null);
…
```

Генерация кода: условный оператор

Теперь нам необходимо оттранслировать условный оператор. Так как никаких if'ов в ассемблере MSIL нет (кроме условных переходов), то необходимо предварительно преобразовать этот оператор к следующей форме: `if (i!=10) goto Next`, где Next – это специальная метка, расположенная после ветки then условного оператора.

```
Label Next = il.DefineLabel();
il.Emit (OpCodes.Ldloc, var_i);
il.Emit (OpCodes.Ldc_I4, 10);
il.Emit (OpCodes.Bne_Un_S, Next);
```

Затем мы создаем новый экземпляр System.Exception: метод GetConstructor по массиву, содержащему типы параметров конструктора, находит в типе соответствующий конструктор. В нашем случае аргументов у конструктора нет, поэтому массив пуст. После этих действий можно спокойно поставить метку Next:

```
Type[] no_types = new Type [0];
il.Emit (OpCodes.Newobj, typeof (System.Exception).GetConstructor (no_types));
il.ThrowException (typeof (System.Exception));
il.MarkLabel (Next);
```

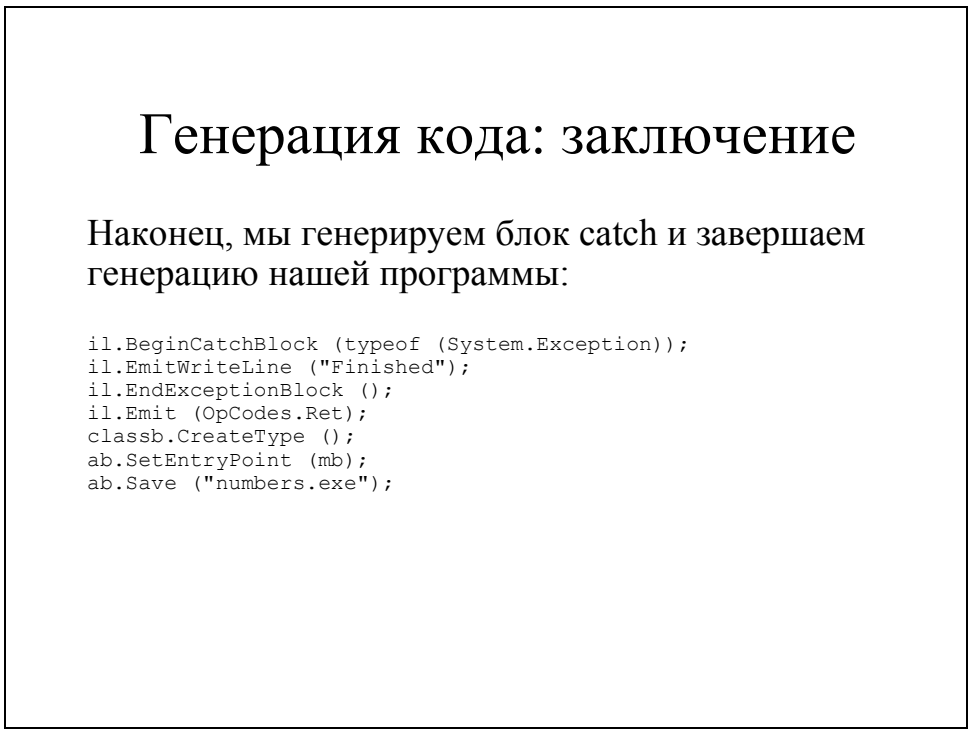
Далее генерируется вывод значения переменной *i* на консоль. Процесс мало отличается от генерации создания нового объекта, за исключением того, что мы вызываем не конструктор, а метод, и, следовательно, должны задать его имя:

```
il.Emit (OpCodes.Ldloc, var_i);
Type[] arg = new Type [1];
arg [0] = typeof (int);
MethodInfo writeLine = typeof (System.Console).GetMethod ("WriteLine", arg);
il.EmitCall (OpCodes.Call, writeLine, null);

il.Emit (OpCodes.Ldloc, var_i); // i = i + 1;
il.Emit (OpCodes.Ldc_I4_1);
il.Emit (OpCodes.Add);
il.Emit (OpCodes.Stloc, var_i);
```

Наконец, генерируется переход к следующей итерации цикла `(goto Start)`:

```
il.Emit (OpCodes.Br, Start);
```



Генерация кода: заключение

Теперь мы перейдем к генерации catch-блока. Стоит отметить следующую деталь: если бы *try* не заканчивался переходом к следующей итерации, то метод BeginCatchBlock автоматически сгенерировал бы команду для завершения блока.

```
il.BeginCatchBlock (typeof (System.Exception));
```

Метод EmitWriteLine используется для генерации вызова System.Console.WriteLine от строковой константы. Тот же эффект можно достичь и более прямым путем (так же, как мы выше выводили значение переменной *var_i*). Результат будет идентичен, но использованный нами только что способ более краток и удобен, например, для вставки в код вывода отладочной информации.

```
il.EmitWriteLine ("Finished");
```

Метод EndExceptionBlock завершает генерацию try-catch блока. Он одновременно выполняет определение метки EndTry, которую создал BeginExceptionBlock. После окончания этого блока необходимо вставить генерацию оператора return.

```
il.EndExceptionBlock ();
il.Emit (OpCodes.Ret);
```

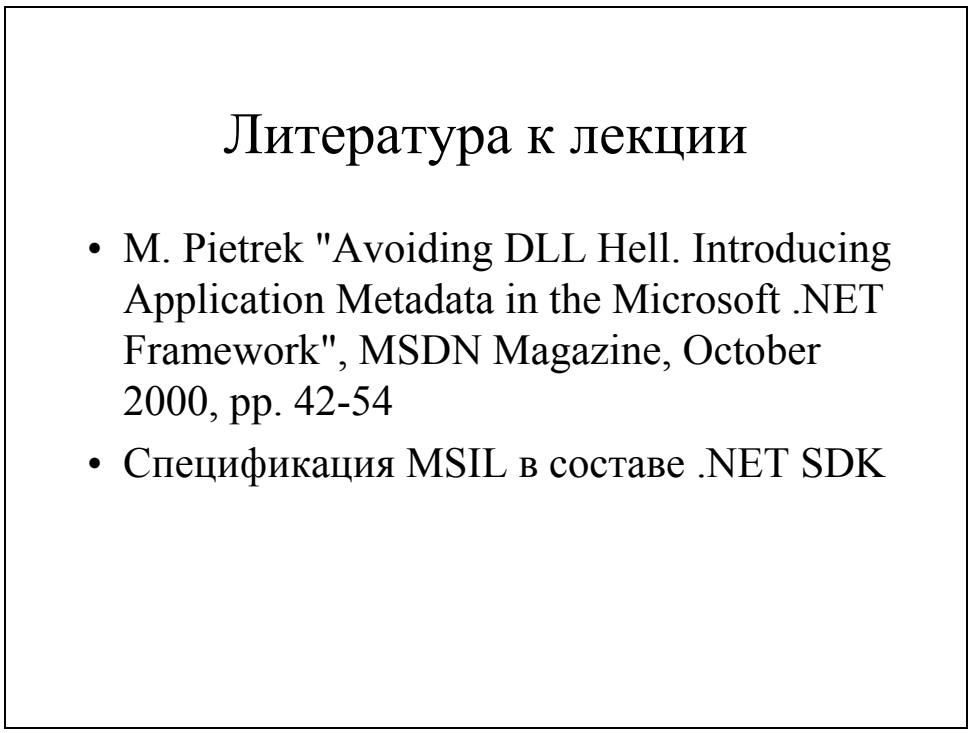
Теперь нам осталось выполнить только несколько завершающих действий: операция CreateType завершает создание типа. После этого мы уже не можем внести никаких изменений и дополнений в класс LowLevelSample.

```
classb.CreateType();
```

Метод SetEntryPoint позволяет установить точку входа для сборки (на уровне генерации MSIL нет никаких неявных соглашений на эту тему, как, скажем, в C#, где точкой входа является метод Main):

```
ab.SetEntryPoint (mb);
ab.Save ("numbers.exe");
```

И вот выполнено последнее действие: после операции Save сборка записана в исполняемый файл на диске. Процесс генерации кода завершен.



Литература к лекции

- M. Pietrek "Avoiding DLL Hell. Introducing Application Metadata in the Microsoft .NET Framework", MSDN Magazine, October 2000, pp. 42-54
- Спецификация MSIL в составе .NET SDK