# Компиляция для платформы .NET

Лекция №4

Теория языков программирования

# Вопросы этой лекции

Допустим, целевая платформа — .NET:

1. Как написать компилятор?
2. Как покрыть компилятор тестами?

# Вопросы этой лекции

Допустим, целевая платформа — .NET:

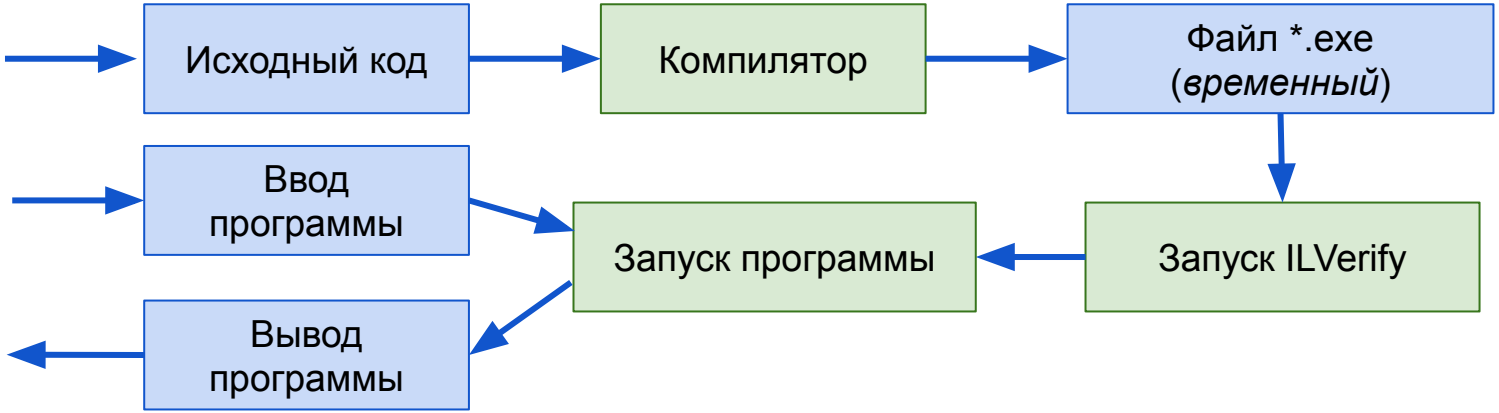
1. Как написать компилятор?
2. Как покрыть компилятор тестами?

Подход ATDD (Acceptance Test Driven Development):

1. Как покрыть компилятор тестами?
2. Как написать компилятор?

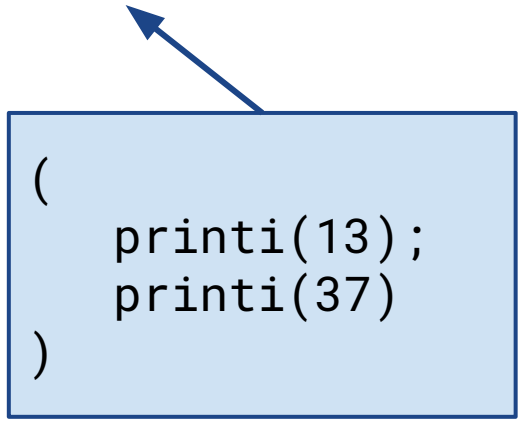
# Тесты компилятора

# Первый тест на Gherkin



# Первый тест на Gherkin

```
#language: ru

Функциональность: ввод-вывод и завершение программы

   Сценарий: вывод чисел функцией printi

      Пусть я скомпилировал программу "print_int.tig"

      Когда я выполняю программу

      Тогда я увижу вывод:

      """

      1337

      """
```



# Шаг компиляции

*InputOutput.feature*

```
Пусть я скомпилировал программу "print_int.tig"
```

*CompilerStepDefinitions.cs*

```
[Given(@"^я скомпилировал программу ""(.*)""$")]

public async Task ПустьЯСкомпилировалПрограмму(string name) {


}
```

# Шаг компиляции

```
[Given(@"^я скомпилировал программу ""(.*)""$")]
public async Task ПустьЯСкомпилировалПрограмму(string name) {
    string path = Samples.GetSampleProgramPath(name);
    Assert.True(File.Exists(path), $"File {path} does not exist");




}
```

# Шаг компиляции

```
[Given(@"^я скомпилировал программу ""(.*)""$")]
public async Task ПустьЯСкомпилировалПрограмму(string name) {
    string path = Samples.GetSampleProgramPath(name);
    Assert.True(File.Exists(path), $"File {path} does not exist");
    _compiledProgram ??= TempFile.CreateEmpty("program-", ".exe");
    _compilerTestDriver.RunCompiler(path, _compiledProgram.Path);


}
```

# Шаг компиляции

```
[Given(@"^я скомпилировал программу ""(.*)""$")]
public async Task ПустьЯСкомпилировалПрограмму(string name) {
    string path = Samples.GetSampleProgramPath(name);
    Assert.True(File.Exists(path), $"File {path} does not exist");
    _compiledProgram ??= TempFile.CreateEmpty("program-", ".exe");
    _compilerTestDriver.RunCompiler(path, _compiledProgram.Path);
    await DotnetIlVerifyRunner.Run(_compiledProgram.Path);
}
```

```
ilverify {exe_path} -r {dotnet_runtime_path}
```

# Шаг компиляции

*InputOutput.feature*

```
Когда я выполняю программу
```

*CompilerStepDefinitions.cs*

```
[When("^(?:я )?выполняю программу$")]

public async Task КогдаВыполняюПрограмму() {


}
```

# Шаг выполнения

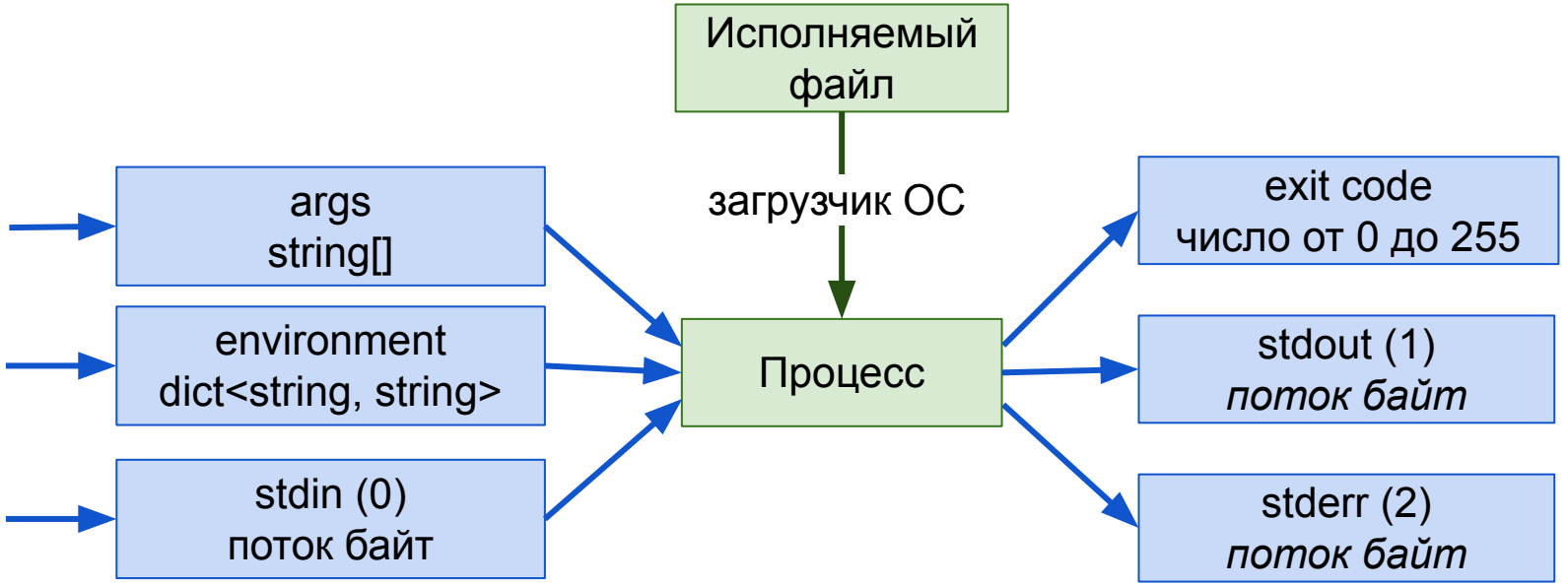
```
[When("^(?:я )?выполняю программу$")]
public async Task КогдаВыполняюПрограмму() {
    Assert.NotNull(_compiledProgram);
    Assert.True(File.Exists(_compiledProgram.Path), $"...");
}
```

# Шаг выполнения

```
[When("^(?:я )?выполняю программу$")]
public async Task КогдаВыполняюПрограмму() {
    Assert.NotNull(_compiledProgram);
    Assert.True(File.Exists(_compiledProgram.Path), $"...");
    CancellationTokenSource cts = new(RunTimeout);
    _output = await DotnetConsoleProgramRunner.Run(
        _compiledProgram.Path, _input.ToString(), cts.Token
    );
}
```

dotnet exec {exe_path}

# Модель запуска консольной программы



# Особенности .NET и MSIL

# Особенности .NET

1. Всё есть объект
    - Типы данных — тоже объекты
    - Нет функций — есть методы
    - Точка входа — метод Program.Main

# Особенности .NET

1. Всё есть объект
    - Типы данных — тоже объекты
    - Нет функций — есть методы
    - Точка входа — метод Program.Main
2. MSIL — промежуточный код среды .NET
    - Стековая машина
    - Стек — для значений (числа, ссылки на объекты)
    - Куча и сборка мусора — для объектов

# Особенности языка MSIL

1. Использует стек для вычислений

# Особенности языка MSIL

1. Использует стек для вычислений
2. Весь код — в методах

# Особенности языка MSIL

1. Использует стек для вычислений
2. Весь код — в методах
3. У метода есть локальные переменные и аргументы
    - Вне стека вычислений
    - Идентифицируются по индексам

# Особенности языка MSIL

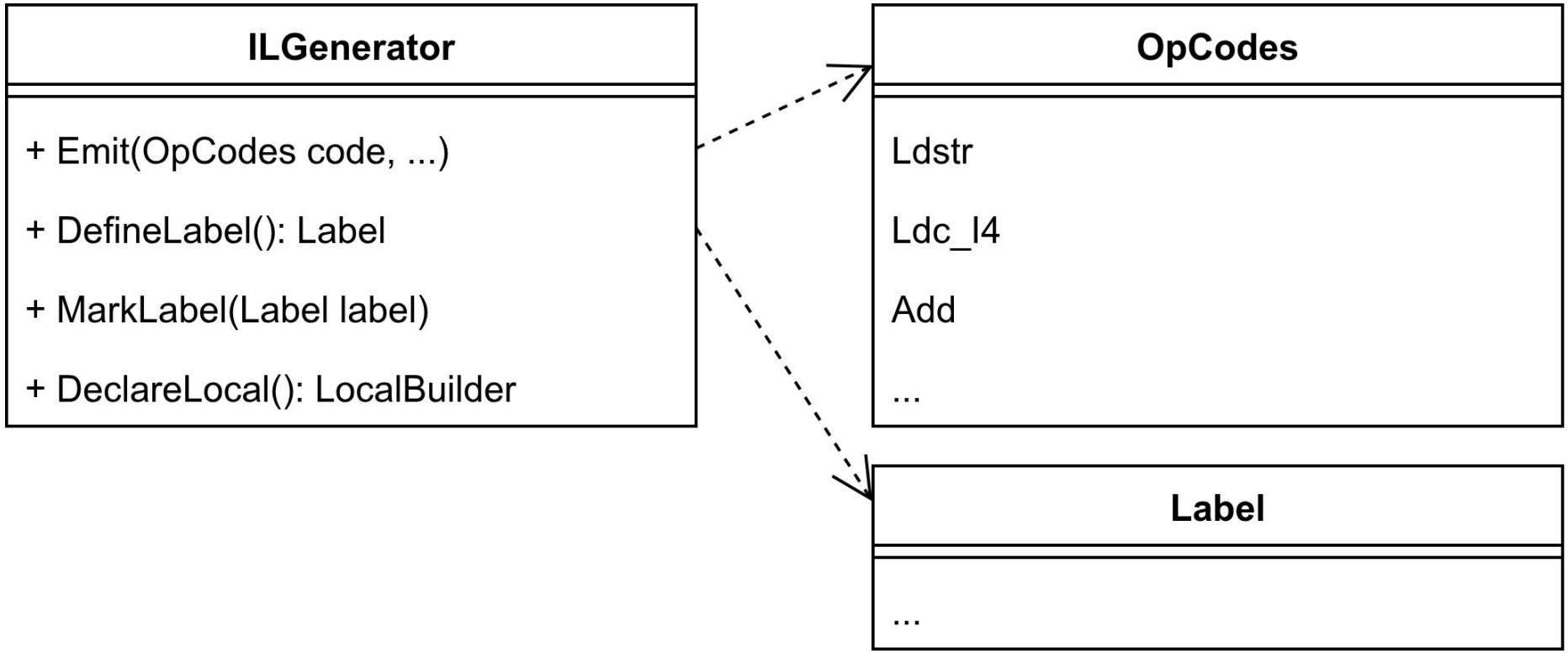
1. Использует стек для вычислений
2. Весь код — в методах
3. У метода есть локальные переменные и аргументы
    - Вне стека вычислений
    - Идентифицируются по индексам
4. Есть метки (labels) и переходы к ним

# Особенности языка MSIL

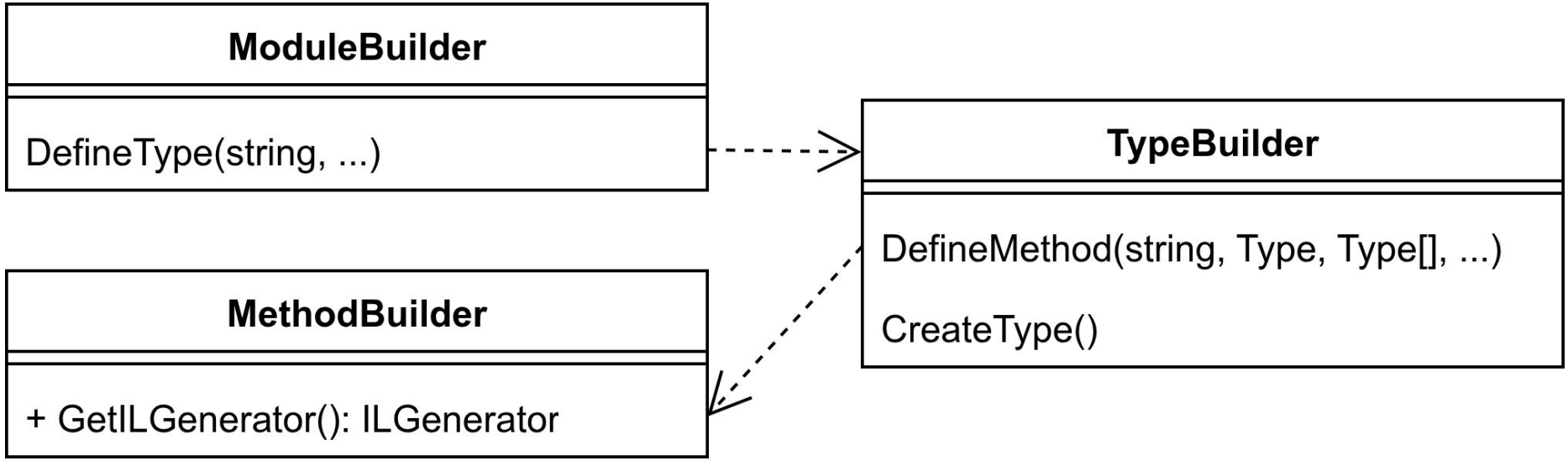
1. Использует стек для вычислений
2. Весь код — в методах
3. У метода есть локальные переменные и аргументы
    - Вне стека вычислений
    - Идентифицируются по индексам
4. Есть метки (labels) и переходы к ним
5. Поддерживает ООП
    - `OpCodes.Call` вызывает метод
    - `OpCodes.Callvirt` вызывает виртуальный метод
    - `OpCodes.Calli` вызывает указатель на метод

# Генерация MSIL

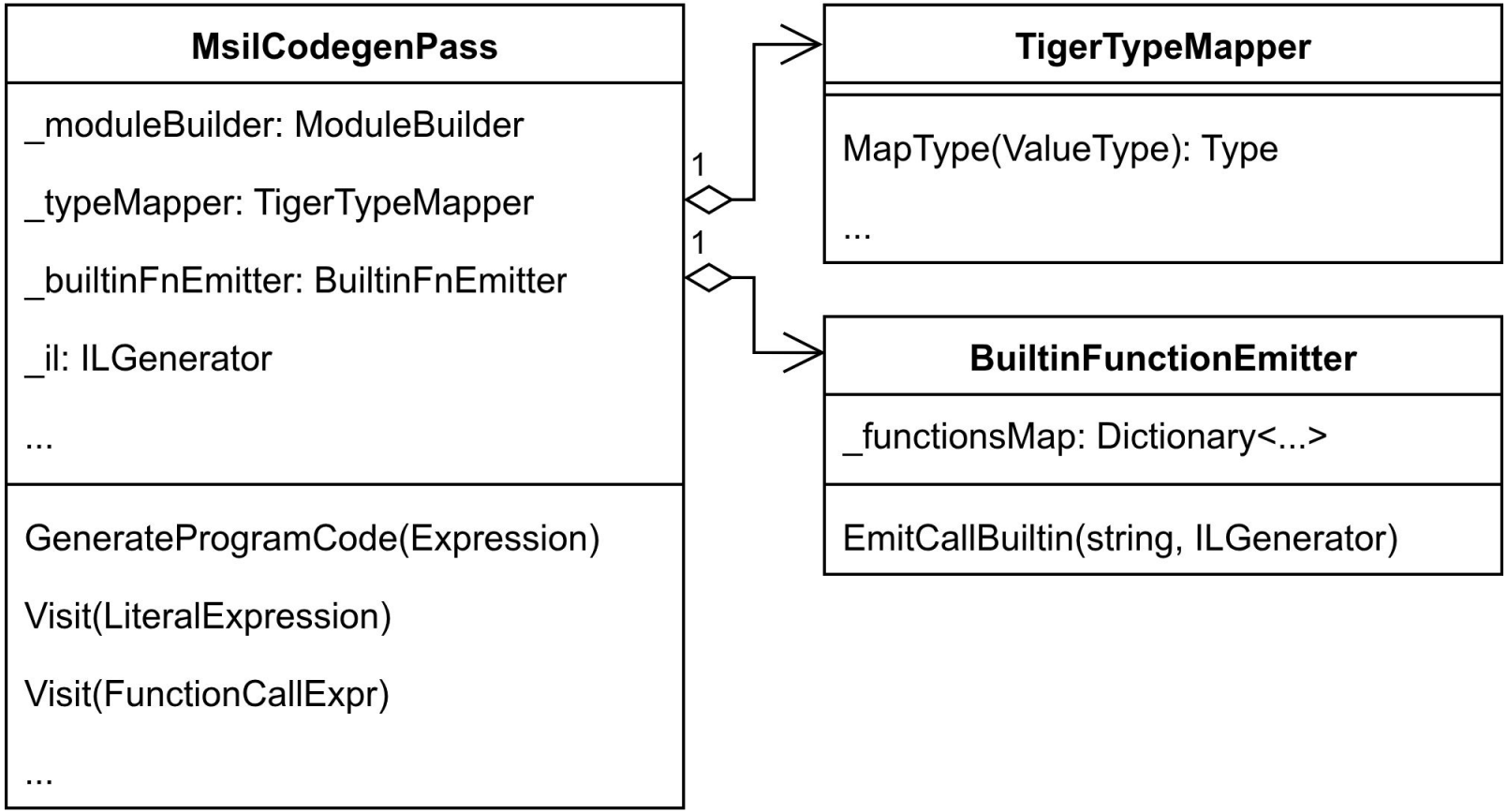
# Пространство имён System.Reflection.Emit



# Как получить ILGenerator?



# Шаблон Visitor в новом модуле MsilCodegen



# Новые проекты и модули

- Compiler — драйвер компилятора (консольная программа)
- MsilCodegen — генерация MSIL для модуля (ModuleBuilder)
- MsilBackend — сохранение MSIL в исполняемый файл

# Семантика MSIL

# Правила семантики MSIL

1. Каждый метод должен завершаться OpCodes.Ret

# Правила семантики MSIL

1. Каждый метод должен завершаться OpCodes.Ret
2. В конце метода на стеке должно быть:
    - 0 значений, если метод имеет тип void
    - 1 значение в остальных случаях

# Правила семантики MSIL

1. Каждый метод должен завершаться OpCodes.Ret
2. В конце метода на стеке должно быть:
   - 0 значений, если метод имеет тип void
   - 1 значение в остальных случаях
3. Не допускается stack underflow

# Правила семантики MSIL

1. Каждый метод должен завершаться OpCodes.Ret
2. В конце метода на стеке должно быть:
   - 0 значений, если метод имеет тип void
   - 1 значение в остальных случаях
3. Не допускается stack underflow
4. Разные ветки должны завершаться одним размером стека

# Правила семантики MSIL

1. Каждый метод должен завершаться OpCodes.Ret
2. В конце метода на стеке должно быть:
   - 0 значений, если метод имеет тип void
   - 1 значение в остальных случаях
3. Не допускается stack underflow
4. Разные ветки должны завершаться одним размером стека

```
ilverify {exe_path} -r {dotnet_runtime_path}
```

# Инструкции MSIL

# Инструкции для работы со стеком

| | |
|---|---|
| ldc.i4 X | добавление числа X на стек |
| ldc.i4.m1 | добавление числа -1 на стек |
| ldstr | добавление интернированной строки в стек |
| dup | добавление в стек копии вершины |
| pop | удаление вершины стека |

# Генерация чтения константы

```
public void Visit(LiteralExpression e) {
    if (e.Type == ValueType.Int) {
        _il.Emit(OpCodes.Ldc_I4, e.Value.AsInt());
    } else if (e.Type == ValueType.String) {
        _il.Emit(OpCodes.Ldstr, e.Value.AsString());
    } else {
        throw new NotImplementedException("...");
    }
}
```

# Инструкции для арифметических операций

| | |
|---|---|
| add | сложение 2 чисел со стека |
| sub | вычитание 2 чисел со стека |
| mul | умножение 2 чисел со стека |
| div | деление 2 чисел со стека |
| neg | унарный минус для вершины стека |

# Генерация операции унарного минуса

```
public void Visit(UnaryMinusExpression e) {
    // Генерируем код для вычисления операнда.
    e.Operand.Accept(this);
    // Применяем унарный минус.
    _il.Emit(OpCodes.Neg);
}
```

# Инструкции для переменных и аргументов

| | |
|---|---|
| stloc name | сохранение вершины стека в переменную name |
| ldloc name | добавление в стек значение переменной name |
| ldarg i | добавление в стек значения i-го аргумента |

# Генерация чтения переменной

```
public void Visit(VariableAccessExpression e) {
    // Добавляем чтение переменной.
    LocalBuilder local = FindVariable(e.Name);
    _il.Emit(OpCodes.Ldloc, local);
}
```

# Генерация присваивания переменной

```
public void Visit(AssignmentExpression e) {
    e.Right.Accept(this);
    if (e.Left is VariableAccessExpression va) {
        LocalBuilder local = FindVariable(va.Variable.Name);
        _il.Emit(OpCodes.Stloc, local);
    } else {
        throw new NotImplementedException("...");
    }
}
```

# Инструкции для сравнений

| | |
|---|---|
| ceq | сравнение 2 чисел со стека<br>1 если равны, иначе 0 |
| clt | сравнение 2 чисел со стека<br>1 если вершина больше или равна, иначе 0 |
| cgt | сравнение 2 чисел со стека<br>1 если вершина меньше или равна, иначе 0 |

# Генерация логического отрицания

```
/// <summary>
/// Генерирует вызов встроенной функции not(b : int) : int.
/// </summary>
private void EmitNot(ILGenerator il) {
    il.Emit(OpCodes.Ldc_I4_0);
    il.Emit(OpCodes.Ceq);
}
```

# Инструкции перехода и вызова

| | |
|---|---|
| br label | безусловный переход к метке label |
| brtrue label | переход к метке label, если вершина стека не 0 |
| brfalse label | переход к метке label, если вершина стека 0 |

# Генерация кода для цикла while

```
public void Visit(WhileLoopExpression e) {
    Label loopStart = _il.DefineLabel(); // Метка проверки условия.
    Label loopEnd = _il.DefineLabel(); // Метка конца цикла.
    _il.MarkLabel(loopStart);
    e.Condition.Accept(this);
    _il.Emit(OpCodes.Brfalse, loopEnd);
    e.LoopBody.Accept(this);
    _il.Emit(OpCodes.Br, loopStart);
    _il.MarkLabel(loopEnd);
}
```

# Инструкции вызова и возврата

| | |
|---|---|
| call | вызов метода |
| callvirt | вызов виртуального метода |
| ret | возврат из метода |

# Генерация вызова пользовательской функции

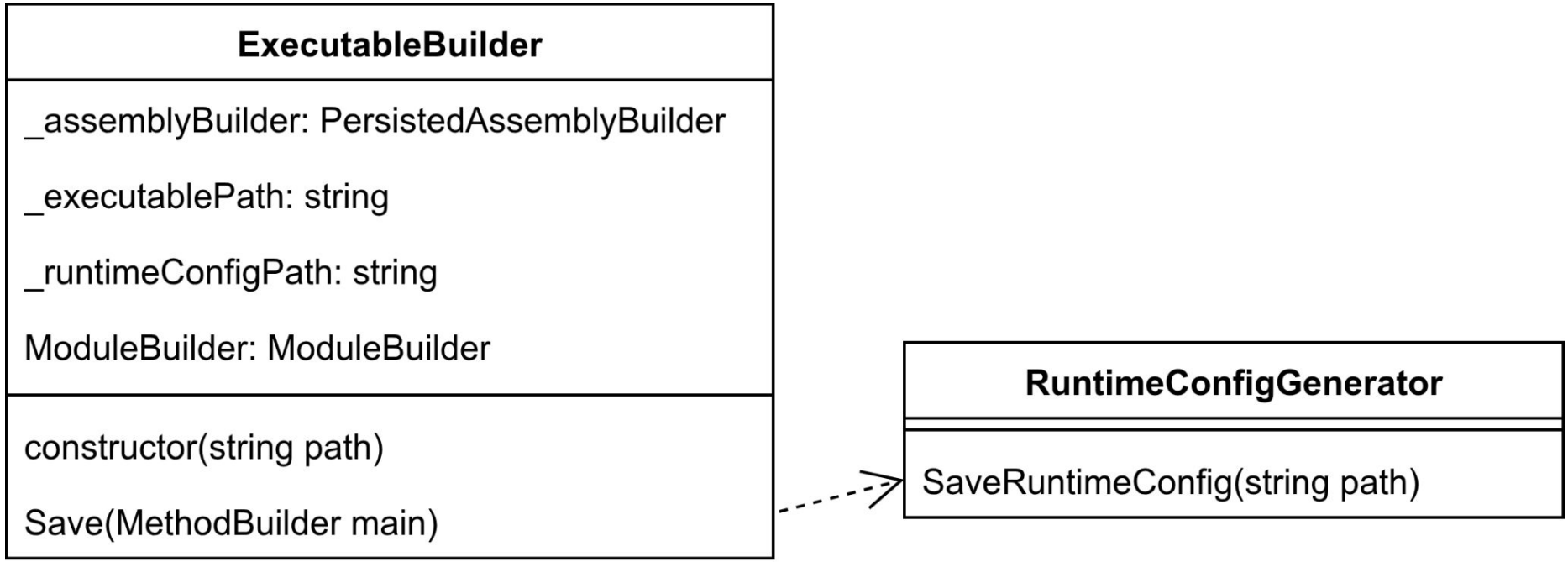
```
public void Visit(FunctionCallExpression e) {
    foreach (Expression argument in e.Arguments) {
        argument.Accept(this);
    }
    // Генерируем вызов пользовательской функции.
    if (!_methodsMap.TryGetValue(e.Name, out MethodBuilder? method)) {
        throw new InvalidOperationException("...");
    }
    _il.Emit(OpCodes.Call, method);
}
```

# Сохранение исполняемого файла

# Сохранение MSIL в исполняемый файл

```
ExecutableBuilder builder = new(outputPath);

MsilCodegenPass pass = new(builder.ModuleBuilder);

MethodBuilder main = pass.GenerateProgramCode(programAst);

builder.Save(main);
```

# Класс ExecutableBuilder



# Подытожим

# Вопросы этой лекции

1. Как покрыть компилятор тестами?
    - Запускаем компиляцию
    - Проверяем MSIL с помощью ILVerify
    - Запускаем программу
    - Проверяем stdout и exit code

# Вопросы этой лекции

1. Как покрыть компилятор тестами?
    - Запускаем компиляцию
    - Проверяем MSIL с помощью ILVerify
    - Запускаем программу
    - Проверяем stdout и exit code
2. Как написать компилятор?
    - Реализуем Visitor для обхода AST
    - Используем ILGenerator.Emit()
    - Используем метки и переходы для ветвлений и циклов
    - Используем PersistedAssemblyBuilder для сохранения dll / exe

Конец!