Технологии программирования

Лекция №5

ИС, весна 2022

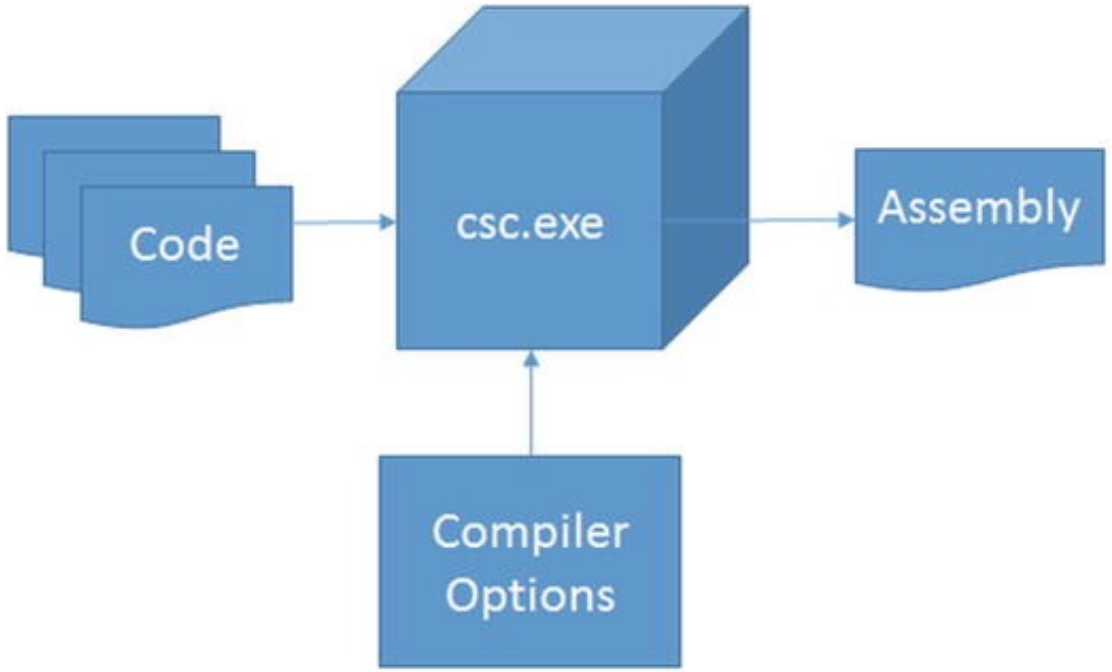
Компилятор

Назначение

- Деление на токены
- Определение, что токены значат
- Получение IL-кода из того, что получили

Компилятор = black box

- Сторонние решение должны изобретать велосипед
- Open source круто, комьюнити круто. Чёрная коробка мешает



Компилятор = white box

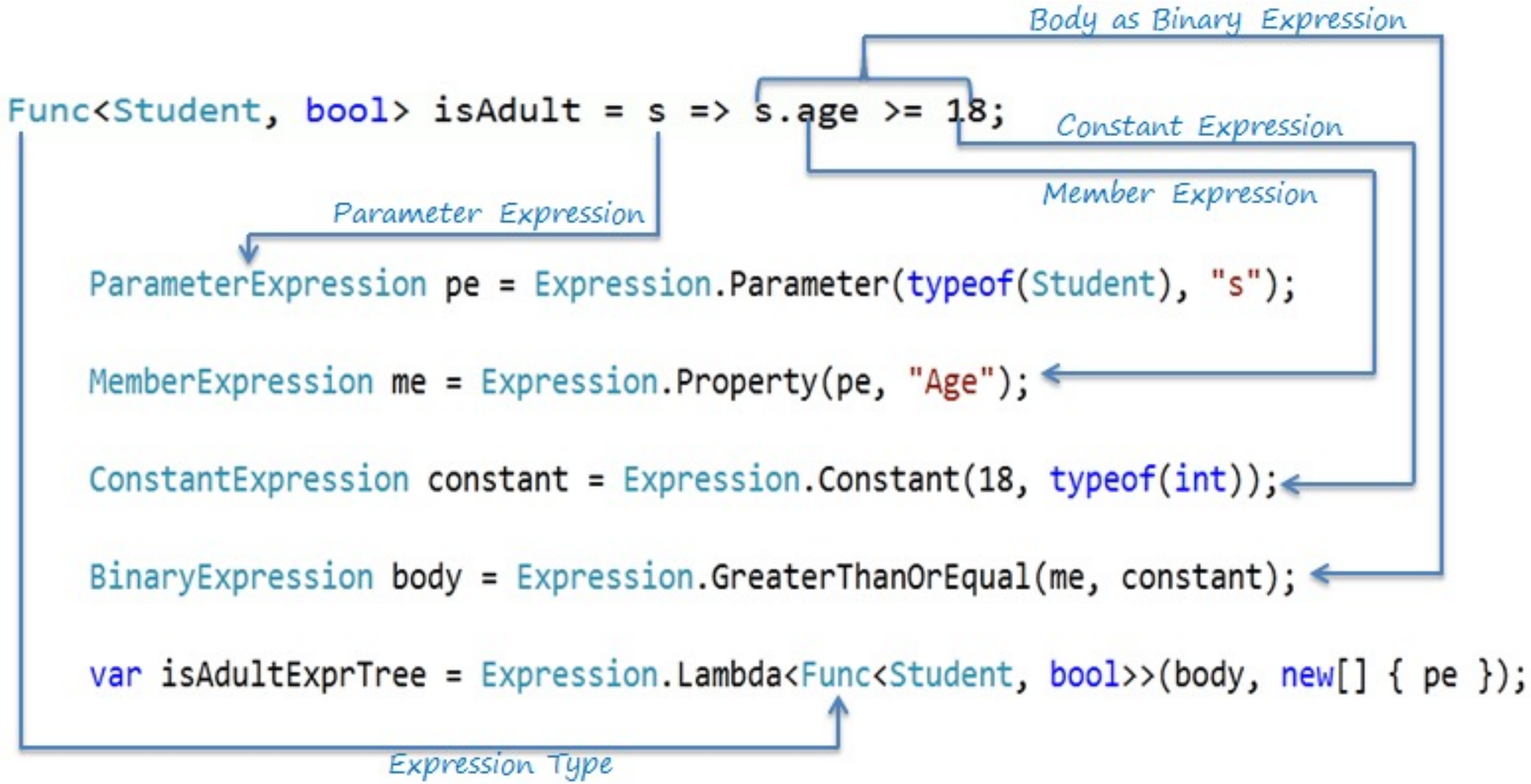
- Открытый код
- Предоставление API для интеграции с компилятором

Деревья выражений

Определения

- Выражение - это базовая иерархическая структура, которая описывает операцию, которая возвращает результат.
- Дерево выражения - это модель, которая представляет структуру выражения. Деревья представляются в виде типов производных от Expression.

Пример дерева



Сложение с дженериками

- Проблема: нельзя сделать дженерик метод, который складывает аргументы через плюс.
- Решение: написать на экспрешенах

```
var parameterType :ParameterExpression  = Expression.Parameter(typeof(T));
BinaryExpression add = Expression.Add( left: parameterType,  right: parameterType);
Expression<Func<T, T, T>> lambda = Expression
    .Lambda<Func<T, T, T>>(
        add,
        params parameters: parameterType,
        parameterType);
return lambda.Compile()(a, b);
```

Обход дерева

- ExpressionVisitor - базовый абстрактный класс, который содержит логику обхода

```
public class CustomVisitor : ExpressionVisitor
{
    protected override Expression VisitParameter(ParameterExpression node)
    {
        Console.WriteLine(node.Type);
        return base.VisitParameter(node);
    }
}
```

Деревья выражений для конвертации в другой язык

- Если мы можем обойти дерево, то мы можем проходя по нему генерировать другое дерево, даже на другом языке.

```
IQueryable<Student> query = GetDbQuery();
query = query.Where(s :Student => s.Id > 200);
```

```
SELECT *
FROM [dbo].[Students]
WHERE Id > 200
```

Синтаксические деревья

Компиляция изнутри C#

```
var tree = SyntaxFactory.ParseSyntaxTree(code);

var compilation = CSharpCompilation.Create(
    assemblyName: "HelloWorldCompiled.exe",
    options: new CSharpCompilationOptions(
        OutputKind.ConsoleApplication),
    syntaxTrees: new[] { tree },
    references: new[]
    {
        MetadataReference.CreateFromFile(
            typeof(object).Assembly.Location)
    });
```

```
using var stream = new MemoryStream();
var compileResult = compilation.Emit(stream);
var assembly = Assembly.Load(stream.GetBuffer());
assembly.EntryPoint.Invoke(obj: null,
    invokeAttr: BindingFlags.NonPublic
                | BindingFlags.Static,
    binder: null,
    parameters: new object[] { null },
    culture: null);
```

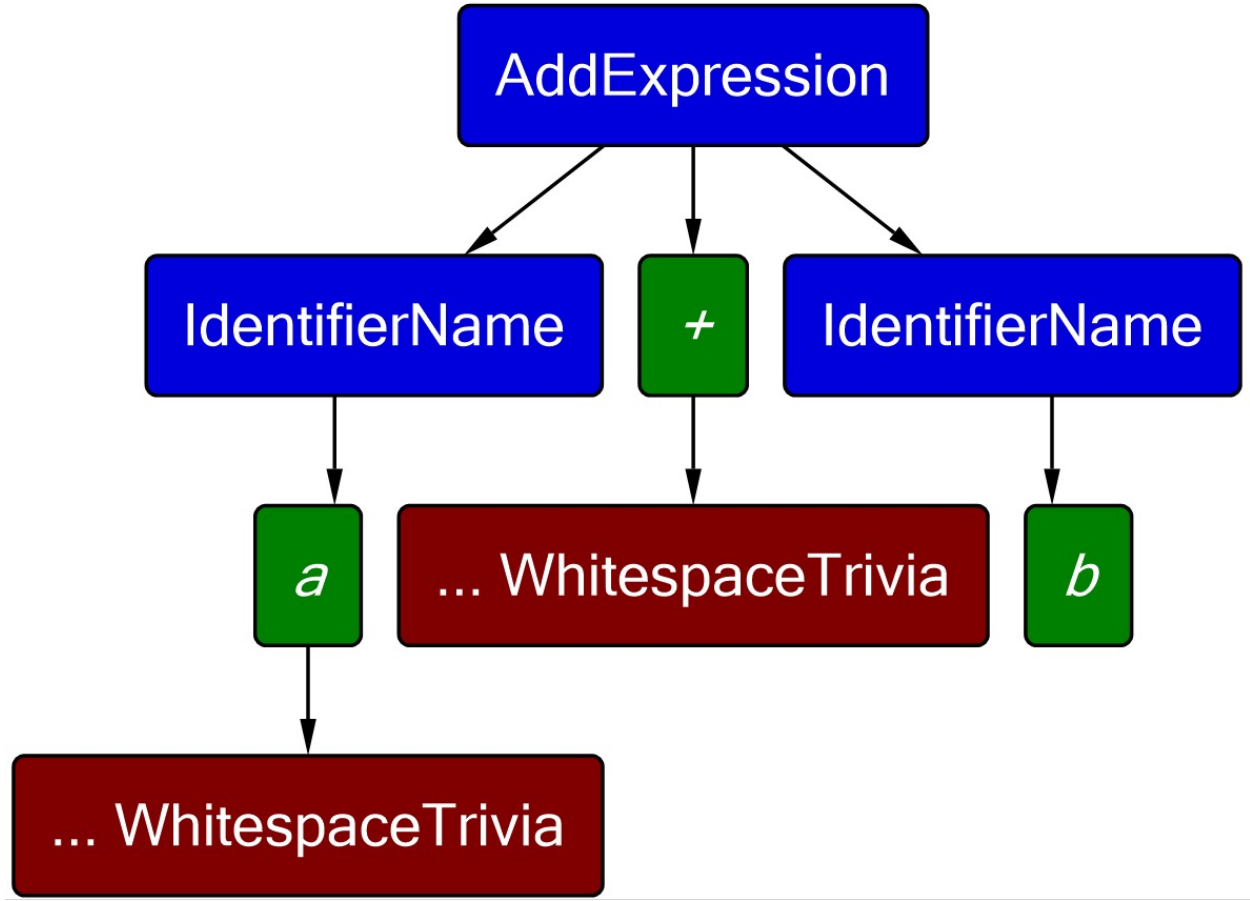
Те самые три этапа

- SyntaxFactory.ParseSyntaxTree - парсинг строки, где описан код
- CSharpCompilation.Create позволяет скомпилировать дерево
- compilation.Emit позволяет сгенерировать ассембли и загрузить его в стрим

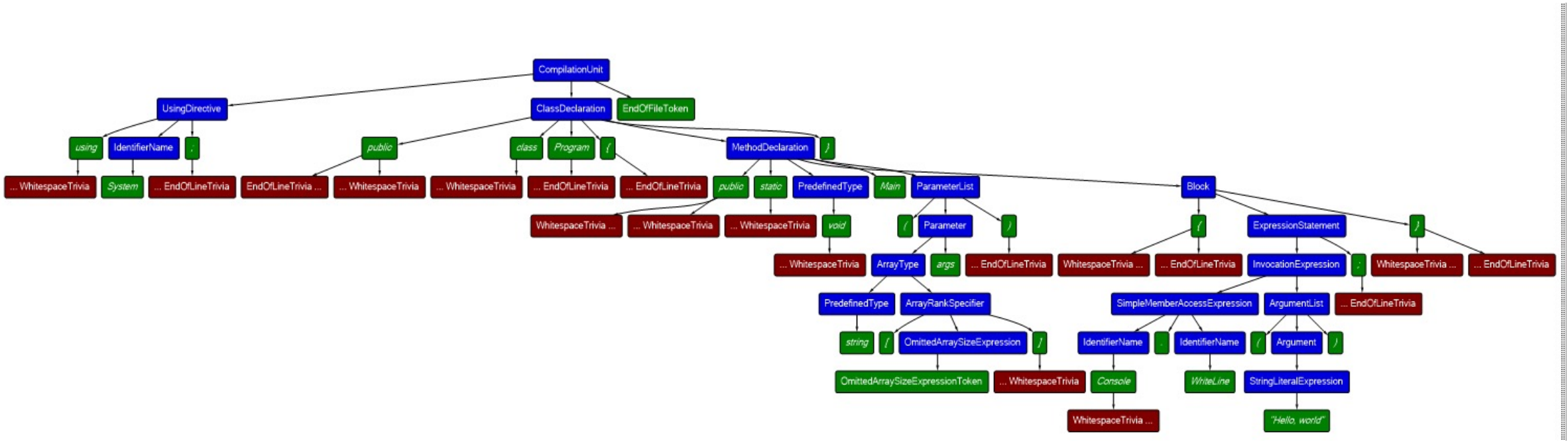
Деревья, элементы дерева

Есть три основных типа нод:

- SyntaxNode
- SyntaxToken
- SyntaxTrivia



Почему сложно визуализировать деревья



Генерируем дерево из C#

```
public static class Doubler
{
    public static int Double(int a)
    {
        return 2 * a;
    }
}
```

SyntaxFactory

- SyntaxFactory - класс с большим количеством методов для генерации различного рода токенов, код:
  - NamespaceDeclaration
  - IdentifierName
  - ClassDeclaration
  - MethodDeclaration

Код генерации дерева

```
var treeNamespace = SyntaxFactory.NamespaceDeclaration(
    SyntaxFactory.IdentifierName("BuildingTrees"));

var doublerClass = SyntaxFactory.ClassDeclaration(identifier: "Doubler");

var doubleMethod :ClassDeclarationSyntax  = doublerClass.WithMembers(
    SyntaxFactory.SingletonList<MemberDeclarationSyntax>(
        node: SyntaxFactory.MethodDeclaration(
            SyntaxFactory.PredefinedType(
                keyword: SyntaxFactory.Token(
                    SyntaxKind.IntKeyword)),
            SyntaxFactory.Identifier(text: "Double"))));
```

Поиск в дереве

```
        var code = @"
using System;

public class ContainsMethods
{
  public void Method1() { }
  public void Method2(int a, Guid b) { }
  public void Method3(string a) { }
  public void Method4(ref string a) { }
}";

        var tree = SyntaxFactory.ParseSyntaxTree(code);
        int count = tree // SyntaxTree
            .GetRoot() // SyntaxNode
            .DescendantNodesAndTokensAndSelf(
                descendIntoChildren: _ => true, descendIntoTrivia: true)
            .Count();

        Console.WriteLine(count);
```

Syntax walker

- Для обхода дереве и поиска в нём элементов можно использовать волкер-классы. Например, CSharpSyntaxWalker - это визитор, который начинает обход с определённой ноды и проходит по всем чилдовым.

```
public sealed class MethodWalker
    : CSharpSyntaxWalker
{
    public MethodWalker(SyntaxWalkerDepth depth = SyntaxWalkerDepth.Node)
        : base(depth)
    { }

    public override void VisitMethodDeclaration(MethodDeclarationSyntax node)
    {
```

Семантическая модель

- Проблема: Синтаксис не описывает достаточного количества информации, которая нам может понадобиться.
- Семантическая модель предоставляет слой поверх дерева, который даёт возможность получить информацию, которую сложно доставать из синтаксиса.

Семантическая модель из метода

```
var compilation = CSharpCompilation.Create(
    assemblyName: "MethodContent",
    syntaxTrees: new[] { tree },
    references: new[]
    {
        MetadataReference.CreateFromFile(typeof(object).Assembly.Location)
    });

var model = compilation.GetSemanticModel(tree, ignoreAccessibility: true);

var methods :IEnumerable<MethodDeclarationSyntax> = tree.GetRoot() // SyntaxNode
    .DescendantNodes(descendIntoChildren: _ => true) // IEnumerable<SyntaxNode>
    .OfType<MethodDeclarationSyntax>();

foreach (var method in methods)
{
    var methodInfo :IMethodSymbol? = model.GetDeclaredSymbol(method);
    var parameters = new List<string>();

    foreach (var parameter in methodInfo.Parameters)
        parameters.Add(item: $"{parameter.Type.Name} {parameter.Name}");

    Console.WriteLine($"{methodInfo.Name}({string.Join(", ", parameters)})");
}
```

Изменение дерева - постановка задачи

- Постановка задачи: есть дерево класса, где все модификаторы - это internal. Хочется получить вместо internal везде public

- Решение - использовать механизмы работы с деревьями, находить internal и заменять.

Изменение дерева - ReplaceNodes

```
Console.WriteLine(tree);
var methods :IEnumerable<MethodDeclarationSyntax>  = tree.GetRoot() // SyntaxNode
    .DescendantNodes(descendIntoChildren: _ ⇒ true) // IEnumerable<SyntaxNode>
    .OfType<MethodDeclarationSyntax>();

var newTree :SyntaxNode  = tree.GetRoot().ReplaceNodes(methods, ComputeReplacementNode);
Console.WriteLine(newTree);
```

Изменение дерева - ReplaceNodes

```
var visibilityTokens :List<SyntaxToken> = method.DescendantTokens(descendIntoChildren: _ => true) // IEnum
    .Where(_ => _.IsKind(SyntaxKind.PublicKeyword)
                || _.IsKind(SyntaxKind.PrivateKeyword)
                || _.IsKind(SyntaxKind.ProtectedKeyword)
                || _.IsKind(SyntaxKind.InternalKeyword)).ToList();

if (!visibilityTokens.Any(_ => _.IsKind(SyntaxKind.PublicKeyword)))
{
    var tokenPosition = 0;

    var newMethod = method.ReplaceTokens(visibilityTokens, computeReplacementToken: (_, __) =>
    {
        tokenPosition++;

        return tokenPosition == 1
            ? SyntaxFactory.Token(_.LeadingTrivia, SyntaxKind.PublicKeyword, _.TrailingTrivia)
            : new SyntaxToken();
    });
    return newMethod;
}
else
{
    return method;
}
```

Изменение дерева - Rewriter

```
public class Rewriter : CSharpSyntaxRewriter
{
    public override SyntaxNode VisitMethodDeclaration(MethodDeclarationSyntax node)
    {
        var visibilityTokens :ImmutableList<SyntaxToken> = node.DescendantTokens( descendIntoChildren
            .Where(_ => _.IsKind(SyntaxKind.PublicKeyword) ||
                        _.IsKind(SyntaxKind.PrivateKeyword) ||
                        _.IsKind(SyntaxKind.ProtectedKeyword) ||
                        _.IsKind(SyntaxKind.InternalKeyword)).ToImmutableList();

        if (!visibilityTokens.Any(_ => _.IsKind(SyntaxKind.PublicKeyword)))
        {
```

Как жить с иммутабельными структурами?

- Иммутабельные структуры не всегда очевидно реализованы
- В дотнете у многих структур есть пара Immutable* (например, ImmutableStack)

Иммутабельный массив

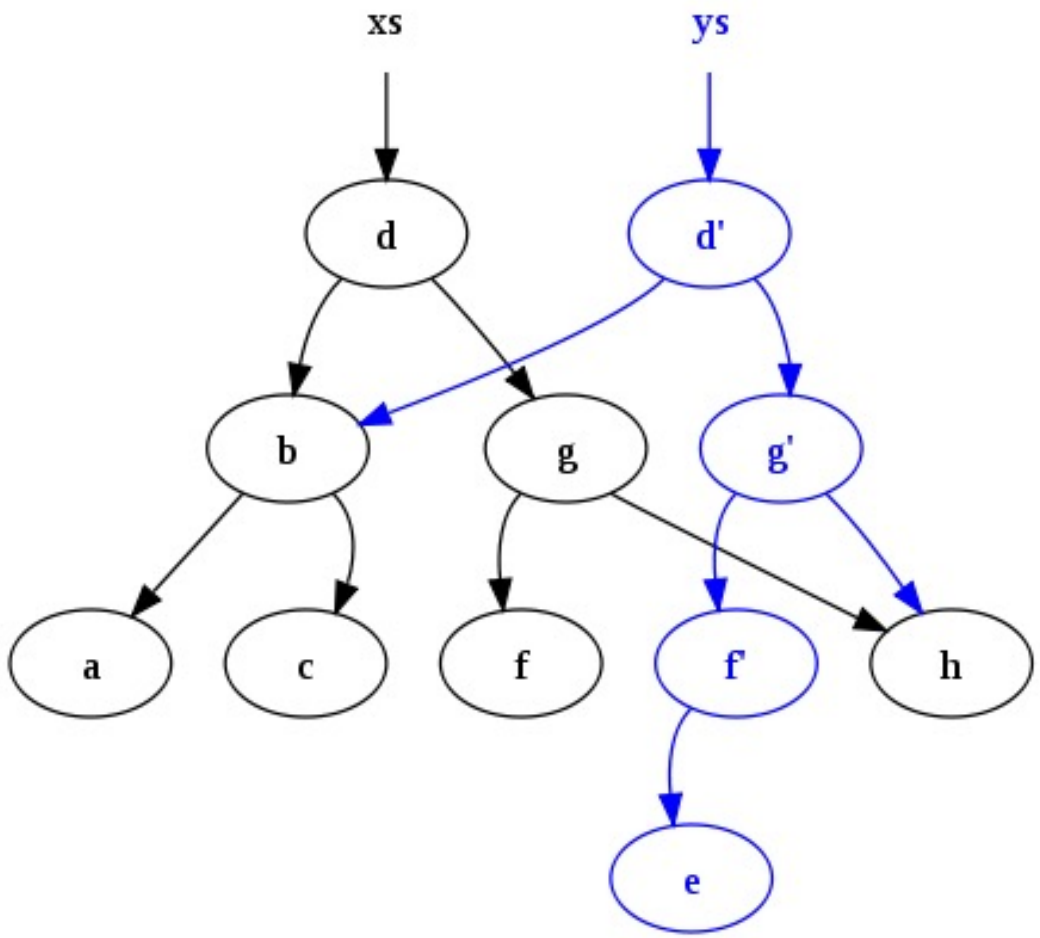
- Read методы ведут себя точно также
- Write методы создают модифицированную копию и возвращают
- ...если мы говорим про CopyOnWriteArray
- Несмотря на очевидно большие расходы CopyOnWriteArray могут работать быстро из-за поддержки со стороны процессоров
- CopyOnWriteArray много поглощают памяти, тут ничего не сделать

Иммутабельные стек и очередь

- Стек можно реализовать на линкед листе
- Иммутабельный стек ничем не отличается от линкед листа
- Иммутабельный стек не даёт гарантии линериализуемости
- Очередь можно написать на двух стеках. Иммутабельную очередь можно написать на двух иммутабельных стеках

Иммутабельный словарь

- Вспомним, что словарь можно реализовать на хеш-таблице и красно-чёрном дереве
- Вспомним, что есть алгоритмы работы с персистентными деревьями
- Получаем иммутабельный словарь
- Работает хорошо пока гарантируется сбалансированность



Форматирование деревьев

- Одним из важных требований к генерации - соблюдение форматирования
- Можно явно задавать trivia
- Можно воспользоваться NormalizeWhitespace

Source generator

Идея

- Основная идея Source generator'a - это встраивание генерации в процесс компиляции