ДЕЛЕГАТЫ. СОБЫТИЯ. АНОНИМНЫЕ ФУНКЦИИ

C# - ЛЕКЦИЯ 10

НА ПРОШЛОМ ЗАНЯТИИ

- Интерфейсы
- IEnumerable
- yield

СЕГОДНЯ В ПРОГРАММЕ

- Делегаты
- События
- Анонимные функции

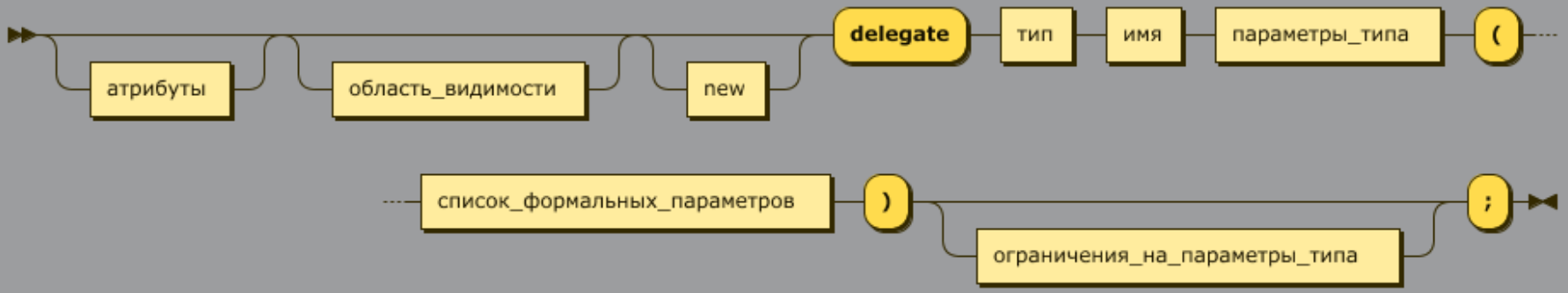
ДЕЛЕГАТЫ

Делегат - это тип, представляющий ссылки на методы с конкретной сигнатурой

Делегаты неявно наследуют от System.MulticastDelegate, который наследует от System.Delegate

```
объявление_делегата
    ::= атрибуты? область_видимости? new? 'delegate'
        тип имя параметры_типа '(' список_формальных_параметров ')'
        ограничения_на_параметры_типа? ';'
```

объявление_делегата:



```
public delegate int Compare(int a, int b);
```

```
public delegate int Sum(int a, int b);

class Program
{
    static void Main(string[] args)
    {
        Sum s1 = new Sum(SumNumbers);
        Sum s2 = SumNumbers;
        int r1 = s1(5, 7);
        int r2 = s2(10, 21);
    }

    private static int SumNumbers(int a, int b)
    {
        return a + b;
    }
}
```

ОПЕРАЦИИ НАД ДЕЛЕГАТАМИ

БИНАРНЫЙ "+"

- Объединение двух делегатов
- Результат - новый делегат
- При его вызове происходит последовательный вызов всех методов, на которые ссылаются исходные делегаты

```
public delegate void MyDelegate();

class Program
{
    static void Main(string[] args)
    {
        MyDelegate a = PrintWelcome;
        MyDelegate b = PrintGoodbye;
        MyDelegate c = a + b;

        c();
    }

    private static void PrintWelcome()
    {
        Console.WriteLine("Welcome!");
    }

    private static void PrintGoodbye()
    {
        Console.WriteLine("Goodbye");
    }
}
```

БИНАРНЫЙ "-"

- Удаление делегата из объединения
- Результат - новый делегат или null (при удалении последнего метода)
- При его вызове происходит последовательный вызов всех методов, на которые ссылается левый исходный делегат, но не ссылается правый

```
public delegate void MyDelegate();

class Program
{
    static void Main(string[] args)
    {
        MyDelegate a = PrintWelcome;
        MyDelegate b = PrintGoodbye;
        MyDelegate c = a + b;

        c();

        MyDelegate d = c - b;

        d();
    }

    private static void PrintWelcome()
    {
        Console.WriteLine("Welcome!");
    }

    private static void PrintGoodbye()
    {
        Console.WriteLine("Goodbye");
    }
}
```

СРАВНЕНИЕ: "==" И "!="

Делегаты равны, если

- они ссылаются на одни и те же методы...
- в том же порядке и количестве

```
public delegate void MyDelegate();

class Program
{
    static void Main()
    {
        MyDelegate a = PrintWelcome;
        MyDelegate b = PrintGoodbye;
        MyDelegate ab1 = a + b;
        MyDelegate ab2 = a + b;
        MyDelegate ba = b + a;

        Console.WriteLine(ab1 == ab2);  //  true
        Console.WriteLine(ab1 == ba);   //  false
    }

    private static void PrintWelcome()
    {
        Console.WriteLine("Welcome!");
    }

    private static void PrintGoodbye()
    {
        Console.WriteLine("Goodbye");
    }
}
```

ВЫЗОВ: "()"

- Последовательно вызывает все методы, на который ссылается делегат
- Результат — один из результатов (последний)

```
public delegate int MyDelegate();

class Program
{
    delegate int MyDelegate();

    static void Main()
    {
        MyDelegate a = GetNextNumber;
        MyDelegate b = GetNextNumber;
        MyDelegate c = GetNextNumber;
        MyDelegate d = a + b + c;

        Console.WriteLine(d());  //  3
    }

    private static int nextNumber = 1;

    private static int GetNextNumber()
    {
        return nextNumber++;
    }
}
```

ЧЛЕНЫ ДЕЛЕГАТА

```
public Delegate[] GetInvocationList();

public MethodInfo Method { get; }

public object Target { get; }

public T Invoke(...);
```

СТАНДАРТНЫЕ ТИПЫ ДЕЛЕГАТОВ

```
namespace System
{
    public delegate void Action();

    public delegate void Action<in T1>(T obj);

    public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);

    // ...
    //  Action с 16 параметрами
}
```

```
namespace System
{
    public delegate TResult Func<out TResult>();

    public delegate TResult Func<in T, out TResult>(T obj);

    public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);

    // ...
    //  Func с 16 параметрами
}
```

СОБЫТИЯ

Реализуют шаблон "Издатель-подписчик"

Издатель

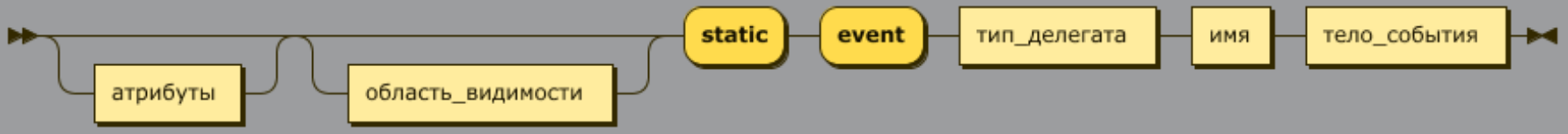
- Предоставляет возможность подписаться и отписаться от событий
- Генерирует события, уведомляя об этом всех подписчиков

Подписчик

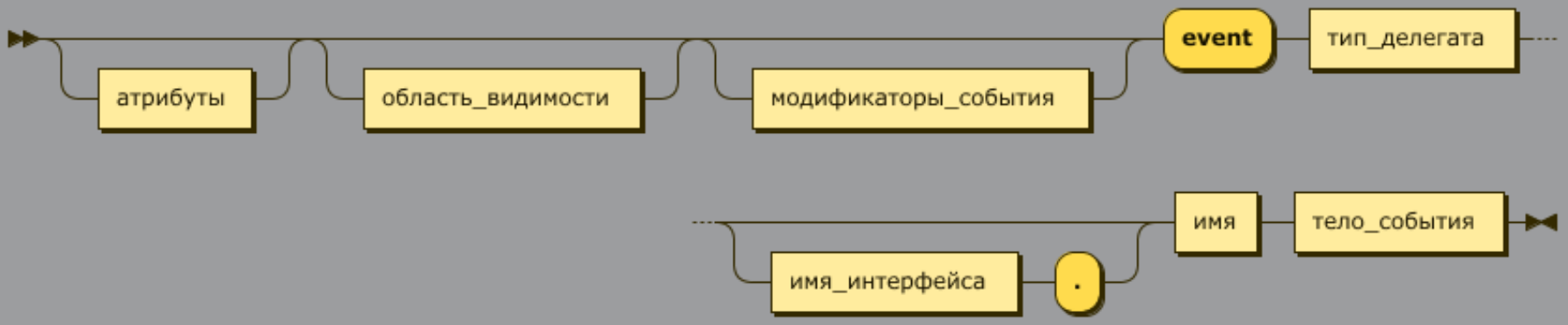
- Подписывается и отписывается от событий издателя
- Получает и обрабатывает события от издателя
- Ничего не знает о других подписчиках
- Не может влиять на издателя

```
объявление_статического_события
    ::= атрибуты? область_видимости? 'static' 'event' тип_делегата имя тел

объявление_события
    ::= атрибуты? область_видимости? модификаторы_события? 'event' тип_дел

модификаторы_события
    ::= 'abstract' 'override'? | 'virtual'? | 'sealed'? 'override' | 'new'

тело_события
    ::= ';' | '{' (объявление_метода_add объявление_метода_remove
        | объявление_метода_add | объявление_метода_remove) '}'

объявление_метода_add
    ::= атрибуты? область_видимости? 'add' (тело_метода_add | ';')
```

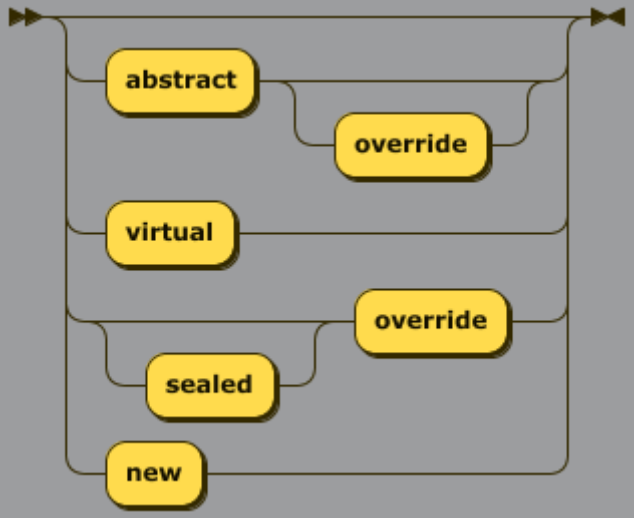
объявление_статического_события:



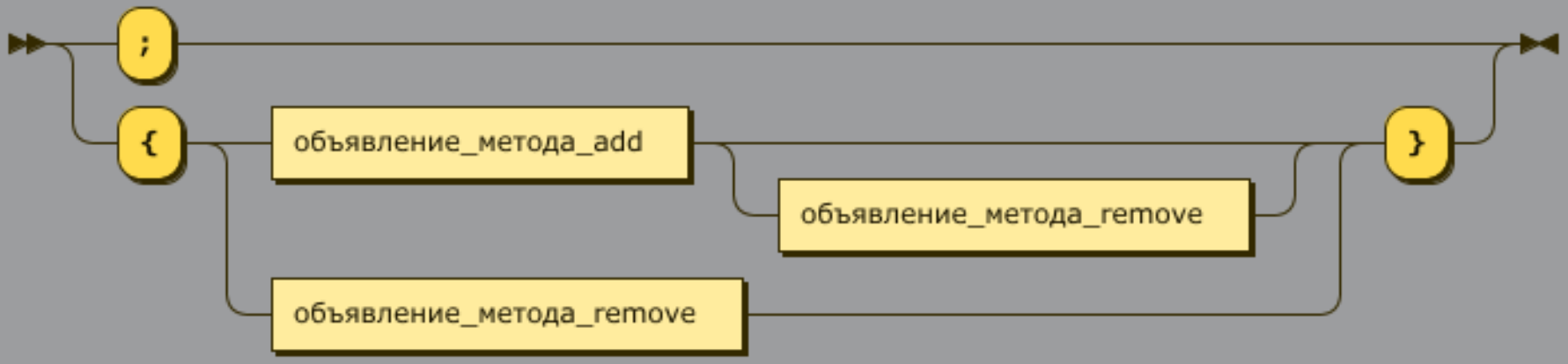
объявление_события:



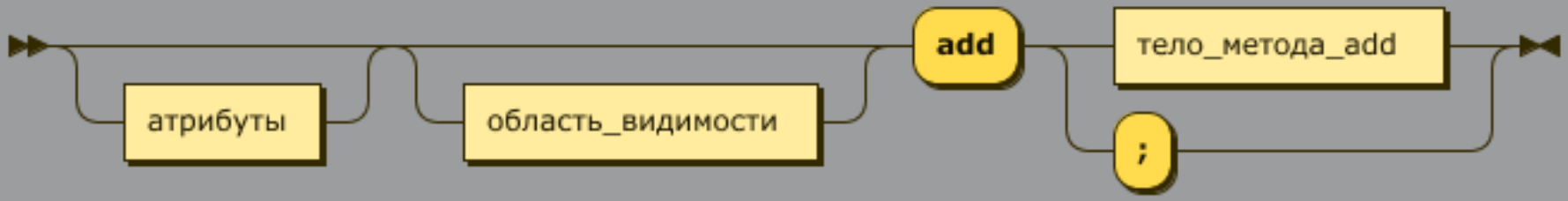
модификаторы_события:



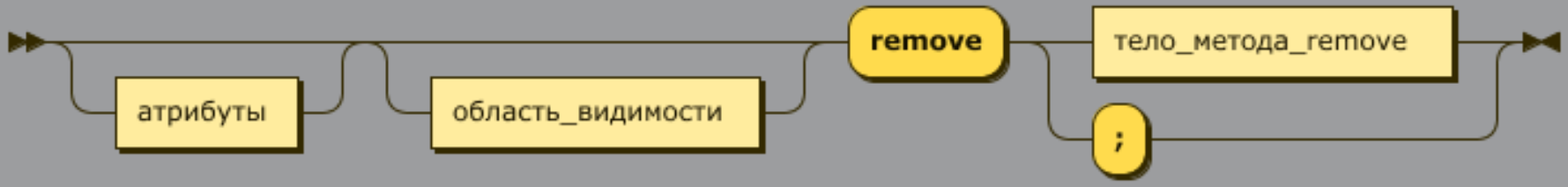
тело_события:



объявление_метода_add:



объявление_метода_remove:



```
class Newspaper
{
    //  ...
}

delegate void TakeNewspaper(Newspaper item);

class PublishingHouse
{
    public event TakeNewspaper Issue;
}
```

```
class Program
{
    static void Main()
    {
        var publisher = new PublishingHouse();

        //  подписка
        publisher.Issue += HandleNewIssue;
    }

    static void HandleNewIssue(Newspaper item)
    {
        //  ...
    }
}
```

```
class PublishingHouse
{
    public event TakeNewspaper Issue { add; remove; }
}
```

```
class PublishingHouse
{
    public event TakeNewspaper Issue
    {
        add { issue += value; }
        remove { issue -= value; }
    }

    private TakeNewspaper issue;
}
```

```
class PublishingHouse
{
    public event TakeNewspaper Issue;

    private void RaiseIssue(Newspaper item)
    {
        var handler = Issue;

        if (handler != null)
        {
            handler(item);
        }
    }
}
```

```
class PublishingHouse
{
    public event TakeNewspaper Issue;

    private void RaiseIssue(Newspaper item)
    {
        Issue?.Invoke(item);
    }
}
```

ОТЛИЧИЕ ОТ СВОЙСТВА ТИПА ДЕЛЕГАТА

Сгенерировать событие может только издатель

ВЫРАЖЕНИЯ АНОНИМНЫХ ФУНКЦИЙ

Анонимная функция - это выражение, представляющее собой подставляемое определение метода.

Анонимная функция не имеет значение или тип сама по себе, но может быть преобразована в совместимый тип делегата или дерева выражений.

Преобразование зависит от целевого типа:

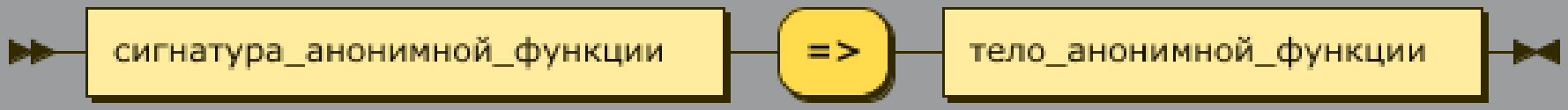
- Если это тип делегата, то результат - делегат, ссылающийся на метод, определяемый анонимной функцией
- Если это тип, приводимый к System.Linq.Expressions.Expression, то результат - дерево выражения

Разновидности анонимных функций:

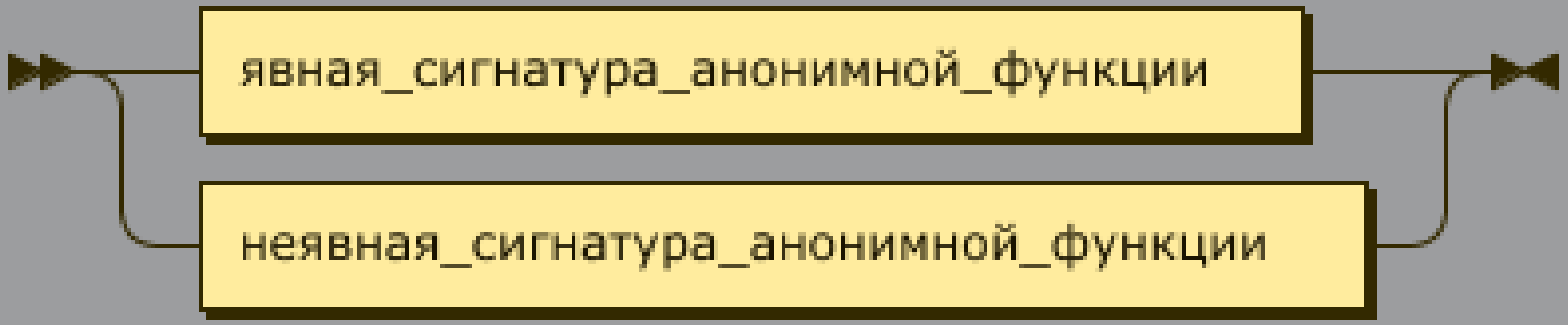
- Лямбда-выражения
- Выражения анонимных методов

```
лямбда_выражение
    ::= сигнатура_анонимной_функции '=>' тело_анонимной_функции

сигнатура_анонимной_функции
    ::= явная_сигнатура_анонимной_функции | неявная_сигнатура_анонимной_фу

явная_сигнатура_анонимной_функции
    ::= '(' список_формальных_параметров? ')'

неявная_сигнатура_анонимной_функции
    ::= имя | '(' имя1 (',' имя)? ')'

тело_анонимной_функции
    ::= '{' инструкция* '}' | выражение
```

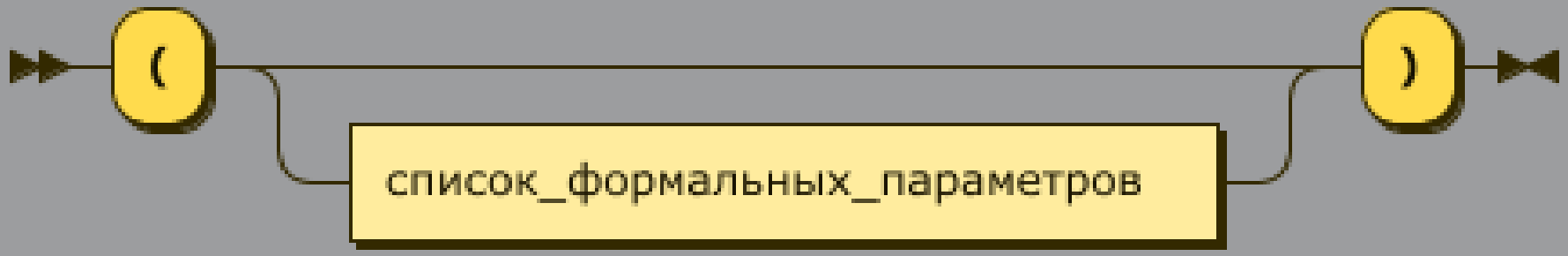
лямбда_выражение:



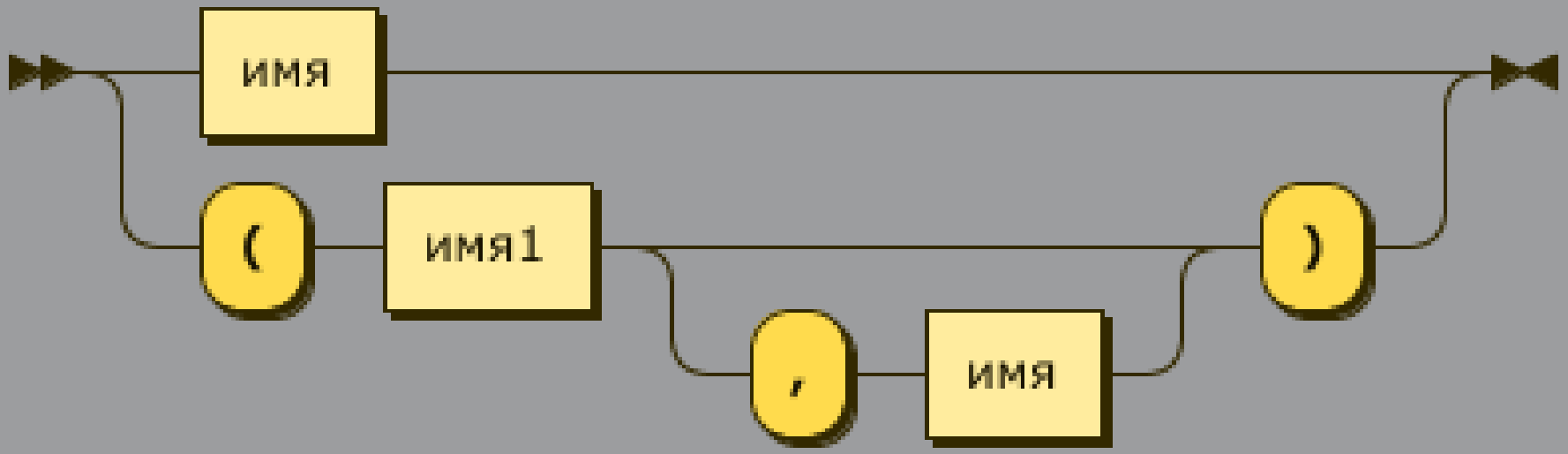
сигнатура_анонимной_функции:



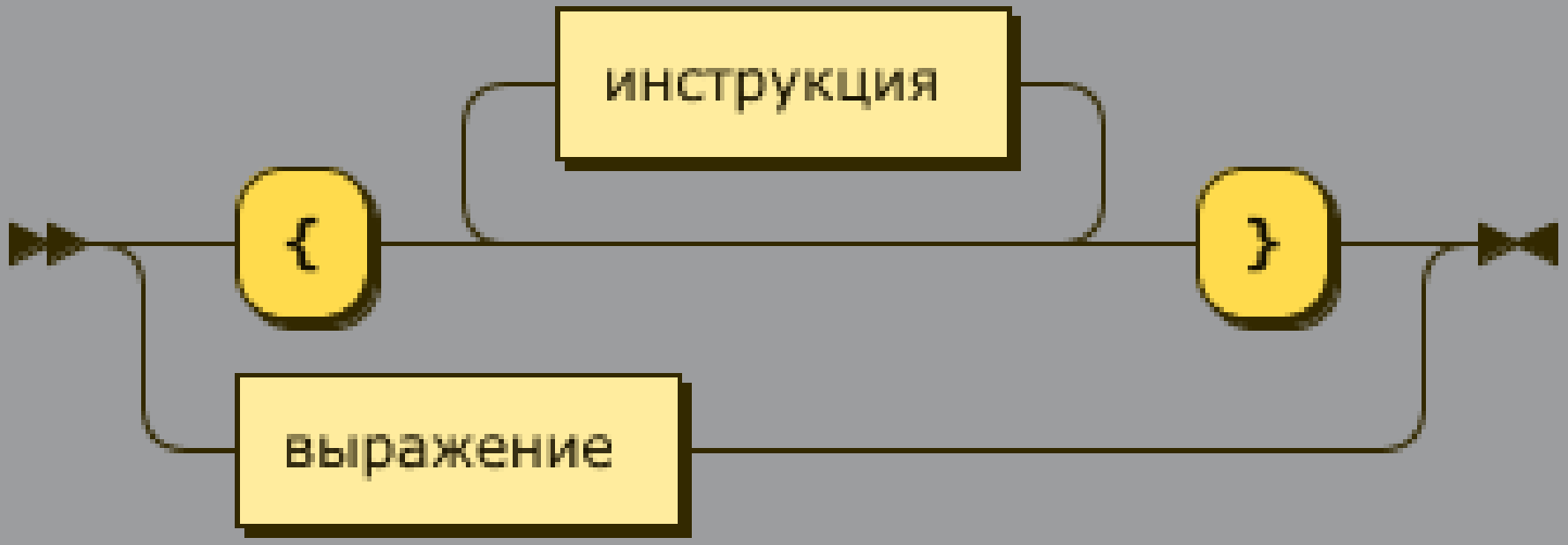
явная_сигнатура_анонимной_функции:



неявная_сигнатура_анонимной_функции:



тело_анонимной_функции:



```
//  неявная сигнатура, тело - выражение
x => x + 1
```

```
//  неявная сигнатура, тело - инструкция
x => { return x + 1; }
```

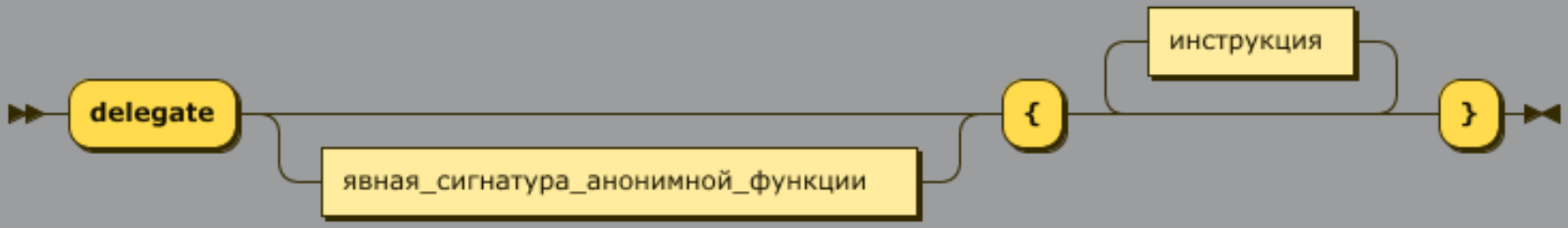
```
//  явная сигнатура, тело - выражение
(int x) => x + 1
```

```
//  неявная сигнатура, тело - выражение, несколько параметров
(x, y) => x * y
```

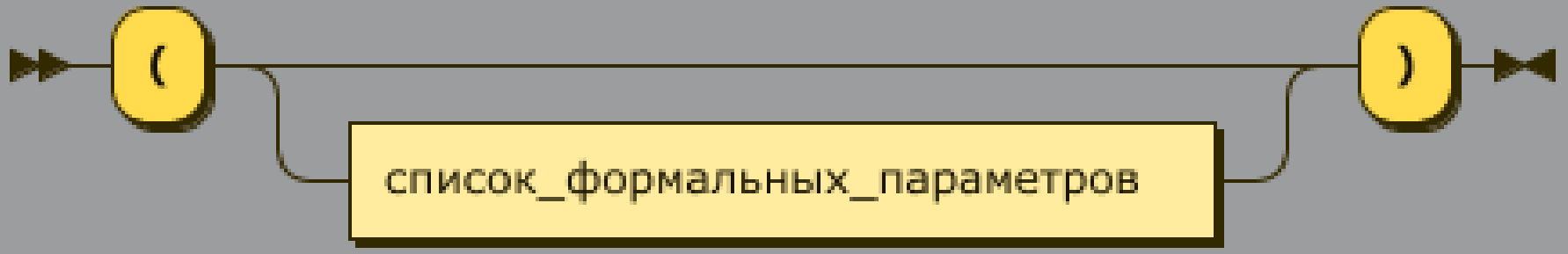
```
//  неявная сигнатура, тело - выражение, без параметров
() => Console.WriteLine();
```

```
выражение_анонимного_метода
    ::= 'delegate' явная_сигнатура_анонимной_функции? '{' инструкция*'}'

явная_сигнатура_анонимной_функции
    ::= '(' список_формальных_параметров? ')'
```

выражение_анонимного_метода:



явная_сигнатура_анонимной_функции:



```
//  нельзя опустить указание типа параметра
//  нельзя использовать выражение вместо тела
delegate (int x) { return x + 1; }
```

```
//  можно опустить указание сигнатуры
delegate { return x + 1; }
```

Различия

- Только лямбда-выражения могут быть преобразованы в деревья выражений
- Лямбда выражения рекомендуемы к использованию, а выражения анонимных методов - устарели

```
public delegate int Sum(int a, int b);

public delegate void Print(object obj);

class Program
{
    static void Main(string[] args)
    {
        Sum s = (a, b) => a + b;
        Print p = o => Console.WriteLine(o);
    }
}
```

ВНЕШНИЕ И ЗАХВАЧЕННЫЕ ПЕРЕМЕННЫЕ

Внешняя переменная - это любая локальная переменная или параметр, область действия которой включает анонимную функцию

В классе к этому относится и this

```
public delegate int Sum(int a, int b);

class Program
{
    static void Main()
    {
        //  i - внешняя переменная по отношению к лямбда-выражению ниже
        int i = 5;

        Sum s = (a, b) => a + b;

        //  j не является внешней переменной по отношению к лямбда-выражени
        int j = 3;
    }
}
```

Захваченная переменная - это такая внешняя, на которую ссылается анонимная функция

```
public delegate int Counter();

class Program
{
    static void Main()
    {
        //  i - захваченная лямбда-выражением ниже переменная
        int i = 5;

        Counter c = () => i++;
    }
}
```

Срок жизни захваченных переменных увеличивается до срока жизни анонимной функции

```
public delegate int Counter();

class Program
{
    static void Main()
    {
        Counter c = CreateCounter();
        Console.WriteLine(c()); //  0
        Console.WriteLine(c()); //  1
        Console.WriteLine(c()); //  2
    }

    static Counter CreateCounter()
    {
        int i = 0;
        Counter c = () => i++;
        return c;
    }
}
```

```
public delegate void D();

class Program
{
    static void Main()
    {
        foreach (D d in F())
            d();
    }

    static D[] F()
    {
        D[] result = new D[3];

        for (int i = 0; i < result.Length; i++)
        {
            int x = i * 2 + 1;

            result[i] = () => Console.WriteLine(x);
        }

        return result;
    }
}
```

```
Результат:
1
3
5
```

```
public delegate void D();

class Program
{
    static void Main()
    {
        foreach (D d in F())
            d();
    }

    static D[] F()
    {
        D[] result = new D[3];
        int x;

        for (int i = 0; i < result.Length; i++)
        {
            x = i * 2 + 1;
            result[i] = () => Console.WriteLine(x);
        }

        return result;
    }
}
```

```
Результат:
5
5
5
```

ВОПРОСЫ