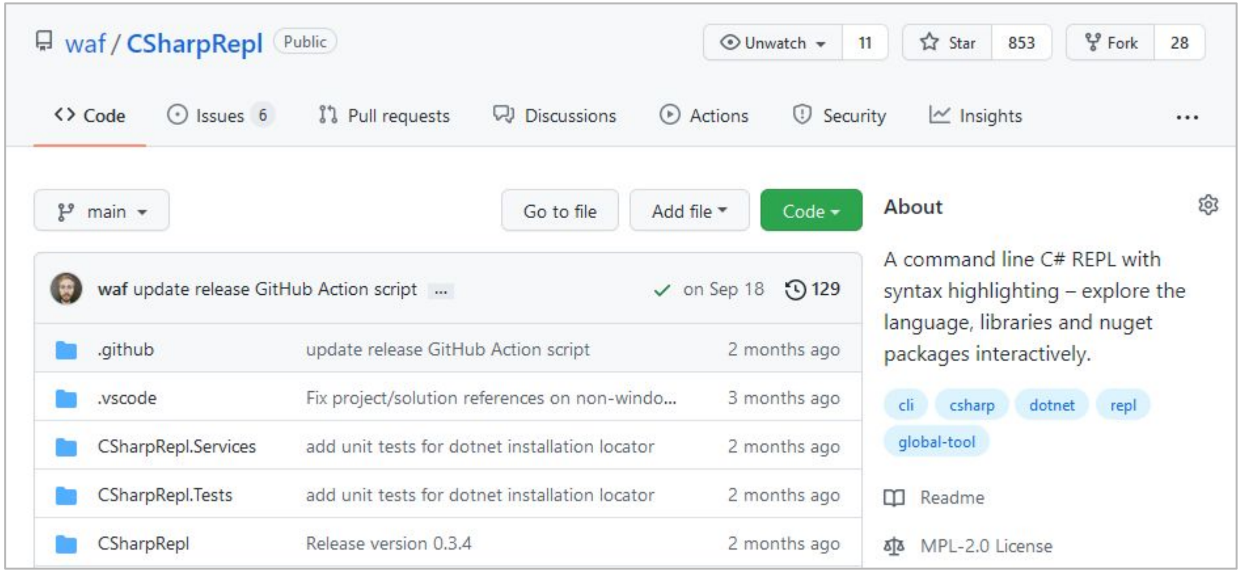
Building developer tooling with roslyn

DotNetDevs.at - Nov 2021

Will Fuqua
Head of Engineering at Jetabroad
Bangkok, Thailand
https://github.com/waf/



waf update release GitHub Action script ... on Sep 18 129

.github	update release GitHub Action script	2 months ago
.vscode	Fix project/solution references on non-windo...	3 months ago
CSharpRepl.Services	add unit tests for dotnet installation locator	2 months ago
CSharpRepl.Tests	add unit tests for dotnet installation locator	2 months ago
CSharpRepl	Release version 0.3.4	2 months ago

A command line C# REPL with syntax highlighting – explore the language, libraries and nuget packages interactively.

cli csharp dotnet repl global-tool

Readme

MPL-2.0 License

Roslyn

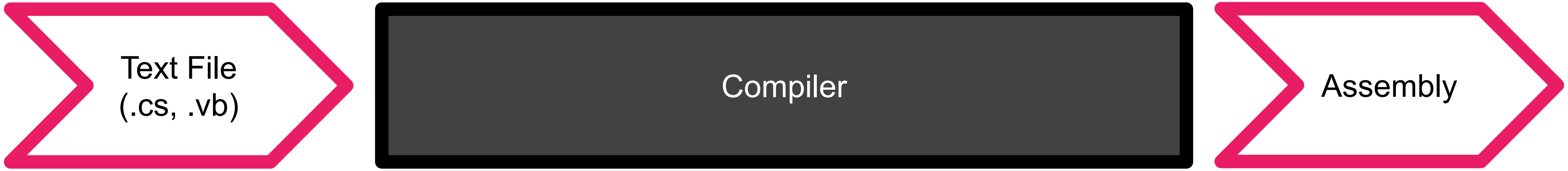
The .NET Compiler Platform

- Microsoft's compiler for C# and Visual Basic (.NET)
- Released in 2014
- https://github.com/dotnet/roslyn

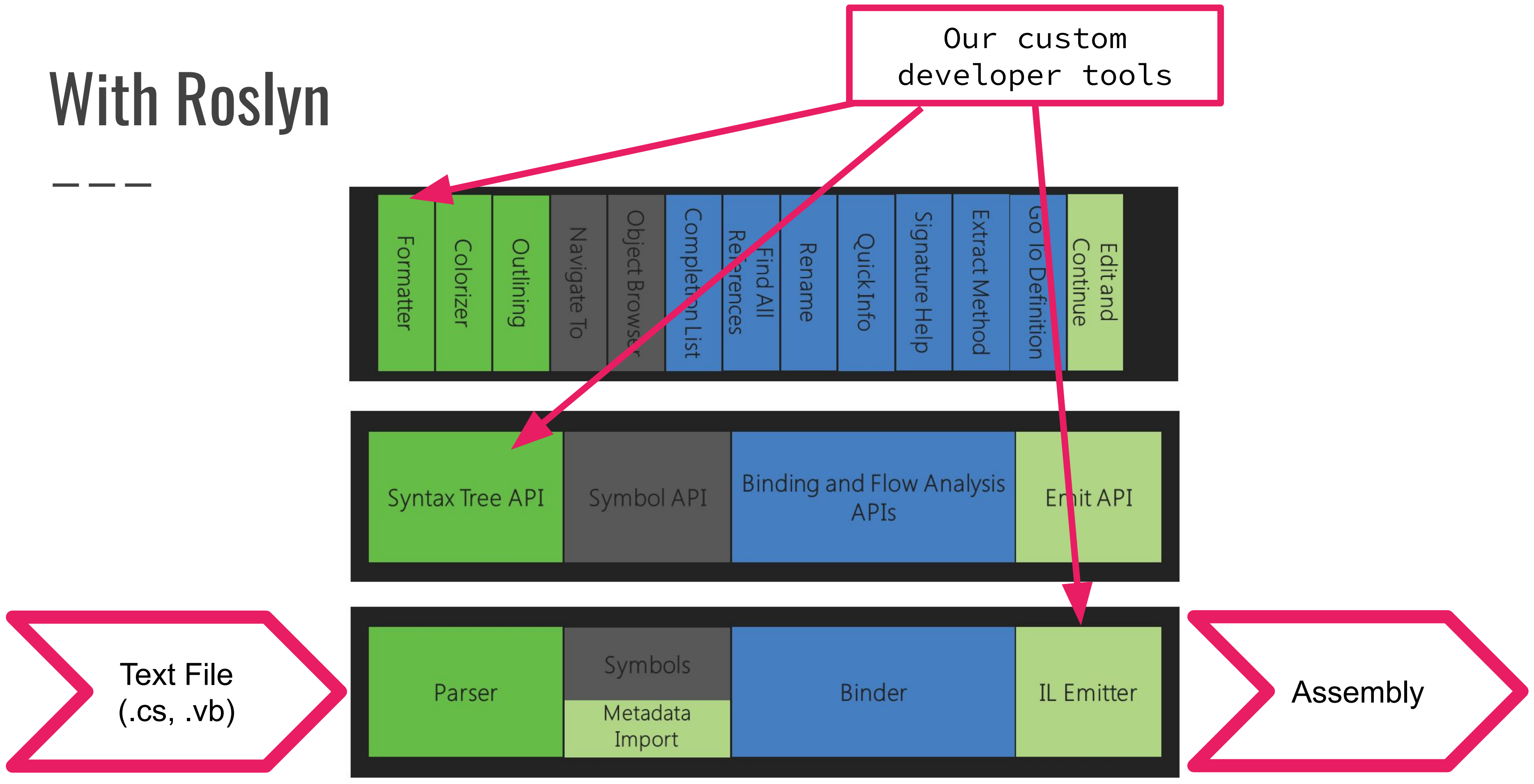
this presentation

- Overview of various Roslyn APIs
- How to use these APIs to create your own tooling

Before Roslyn



With Roslyn


Our custom developer tools
Formatter
Colorizer
Outlining
Navigate To
Object Browser
Completion List
Find All References
Rename
Quick Info
Signature Help
Extract Method
Go To Definition
Edit and Continue
Syntax Tree API
Symbol API
Binding and Flow Analysis APIs
Emit API
Text File (.cs, .vb)
Parser
Symbols
Metadata Import
Binder
IL Emitter
Assembly

With Roslyn

Compiler API

Analyze, modify, and compile code

Workspace API

Analyze, modify, and compile solutions

Diagnostic API

Create your own compiler warnings and suggested fixes

Scripting API

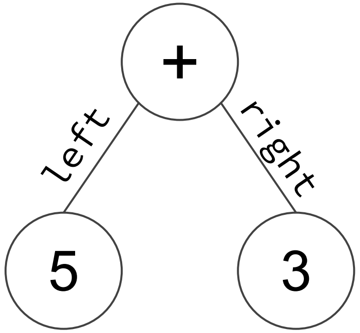
Run C# code as a script (e.g. C# Interactive)

Compiler API

We saw this in the previous Source Generator presentation!
(link)

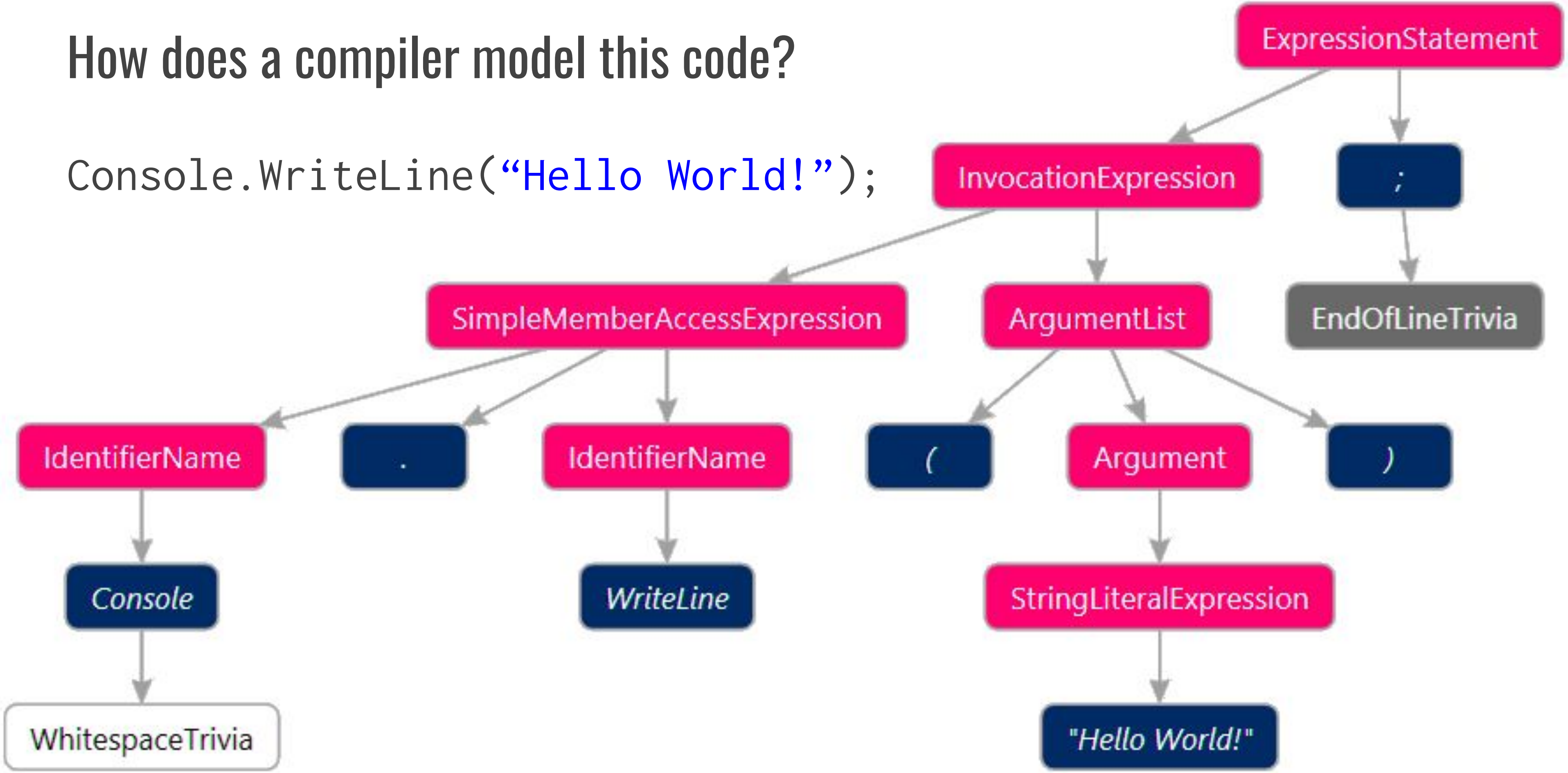
How does a compiler model this code?

5 + 3



How does a compiler model this code?

```
Console.WriteLine(“Hello World!”);
```



sharplab.io

We can use Roslyn's understanding of our code, along with LINQ, to query our program's structure.

A simple program analysis: What message are we printing?

```
SyntaxTree tree = CSharpSyntaxTree
    .ParseText(@"Console.WriteLine(""Hello World!"");");

SyntaxNode root = tree.GetRoot();

var message = root
    .DescendantNodes()
    .OfType<LiteralExpressionSyntax>()
    .Single();

message.Token.ValueText

“Hello World!”
```

```
Console.WriteLine("Hello World!");
```

⬇

```
Console.WriteLine("Hello Roslyn!");
```

```csharp
var message = ... // old message from previous slide

var newMessage = SyntaxFactory.LiteralExpression(
    SyntaxKind.StringLiteralExpression,
    SyntaxFactory.Literal("Hello Roslyn!")
);

SyntaxNode newProgram = root.ReplaceNode(message, newMessage);

string output = newProgram.ToFullString();

"Console.WriteLine(\"Hello Roslyn!\");"
```

How have we used the Compiler API?

```
public class FlightType
{
    public static readonly FlightType OneWay =
        new FlightType("OneWay", someConfigData);

    public static readonly FlightType Return =
        new FlightType("Return", otherConfigData);

    // other flight types defined, too.
}

// in some other class...
if (selectedFlightType == FlightType.OneWay)
{
    // do a one-way flight search
}
```

How have we used the Compiler API?

- Test data for unit tests used to be a bit of a mess

```
var flight = new Flight
{
    Airline = someAirline,
    FlightNumber = flightNumber,
    Price = somePrice
}
```

→

```
var flight =
    TestFlight.Build(...)
```

How have we used the Compiler API?

- https://github.com/Jetabroad/NUnitToXunit
- CSharpSyntaxRewriter

```
[TestFixture]
public class MyTest
{
    [SetUp]
    public void MySetupMethod()
    {
        // some setup code here
    }
}
```

➡

```
public class MyTest
{
    public MyTest()
    {
        // some setup code here
    }
}
```

The Pareto Principle is your friend when making changes across a large codebase.

The hardest 20% of the code takes 80% of the time

Automating only 80% of the work across a large code base is still a huge win.

Workspace API

Workspace API

```
string solutionPath = @"path\to\MySolution.sln";
var solution = await MSBuildWorkspace
    .Create()
    .OpenSolutionAsync(solutionPath);

var files =
    from project in solution.Projects
    from document in project.Documents
    select $"{project.Name}: {document.Name}";
```

“Find All References”

```
Document document = solution.Projects.First().Documents.First();

// get the first method declaration
MethodDeclarationSyntax method = document
      .GetSyntaxRoot().DescendentNodes()
      .OfType<MethodDeclarationSyntax>()
      .First();

// find all references to that method in the solution
var model = await document.GetSemanticModelAsync();
var methodSymbol = model.GetSymbolInfo(method);
var references = await SymbolFinder.FindReferencesAsync(methodSymbol, solution);
```

How have we used the Workspace API?

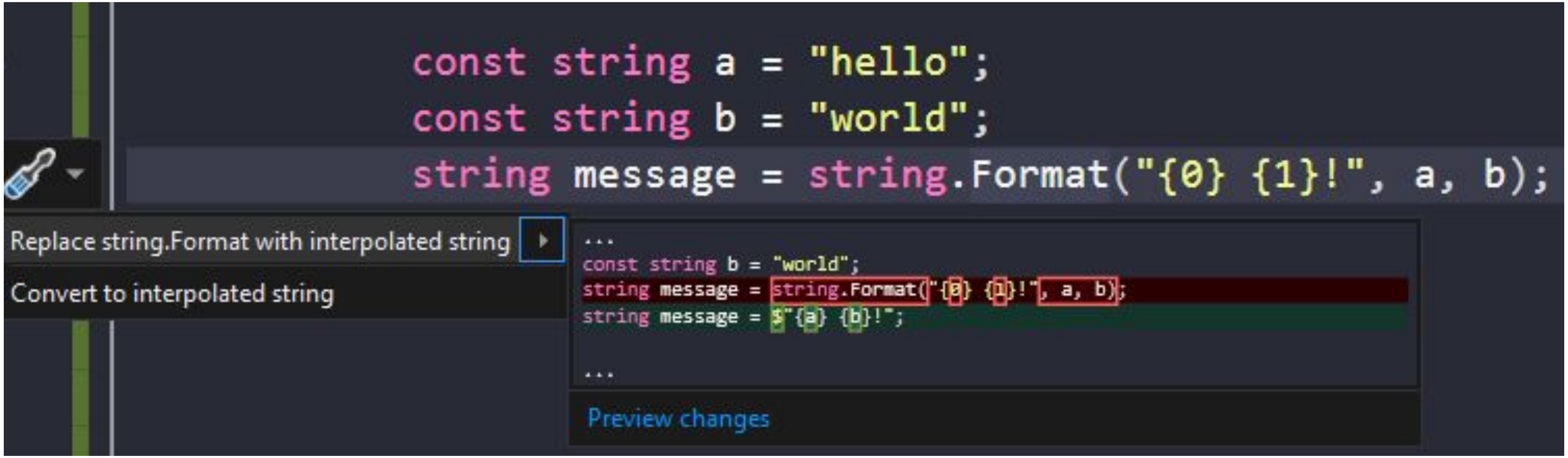
```
public class Serializer
{
    public byte[] Serialize(object obj) => ...
}

#if UNIT_TEST
public class SerializerTests
{
    [Fact]
    public void Serialize_WithNull_ThrowsException => ...
}
#endif
```

Syntax Highlighting

- Classifier API
- Demo

Diagnostic API



```
const string a = "hello";
const string b = "world";
string message = string.Format("{0} {1}!", a, b);
```

Replace string.Format with interpolated string
Convert to interpolated string

```
...
const string b = "world";
string message = string.Format("{0} {1}!", a, b);
string message = $"{a} {b}!";

...
```

How to use it?

- DiagnosticAnalyzer and CodeFixProvider
- Can be distributed as a NuGet package or a Visual Studio Extension
- Great documentation: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/tutorials/how-to-write-csharp-analyzer-code-fix

Scripting API

Demo: Build your own REPL

Enhancements for our REPL

- Add a nice set of default usings and assembly references.
- MetadataReferenceResolver: support referencing assemblies, nuget libraries, projects, solutions via "#r" syntax.
- SourceReferenceResolver: support importing files via "#load" syntax
- Synchronize our typed code with the Workspace API to provide:
  - Syntax Highlighting
  - Intellisense documentation
- Use the Semantic API to "power-up" our REPL's understanding of the code.

Interested? Check out these projects

- https://github.com/waf/CSharpRepl
  - C# command line REPL with syntax highlighting
- https://github.com/dotnet/interactive
  - Run C# code interactively (e.g. via notebooks). Powers https://github.com/jonsequitur/dotnet-repl
- https://github.com/filipw/dotnet-script
  - Run C# code as a script
- https://github.com/OmniSharp/omnisharp-roslyn
  - C# support for various editors (like vs code and vim) powered by roslyn