The .NET Compiler Platform (“Roslyn”)

Getting Started: Semantic Analysis

March 2014

Prerequisites

- Getting Started: Syntax Analysis
- Visual Studio 2013
- “Roslyn” End User Preview
- “Roslyn” SDK Project Templates

Introduction

Today, the Visual Basic and C# compilers are black boxes – text goes in and bytes come out – with no transparency into the intermediate phases of the compilation pipeline. With the **.NET Compiler Platform** (formerly known as “Roslyn”), tools and developers can leverage the exact same data structures and algorithms the compiler uses to analyze and understand code with confidence that information is accurate and complete.

In this walkthrough we’ll explore the **Symbol** and **Binding APIs**. The **Syntax API** exposes the parsers, the syntax trees themselves, and utilities for reasoning about and constructing them.

Understanding Compilations and Symbols

The **Syntax API** allows you to look at the *structure* of a program. However, often you’ll want richer information about the semantics or *meaning* of a program. And while a loose code file or snippet of VB or C# code can be syntactically analyzed in isolation it’s not very meaningful to ask questions such as “what’s the type of this variable” in a vacuum. The meaning of a type name may be dependent on assembly references, namespace imports, or other code files. That’s where the **Compilation** class comes in.

A **Compilation** is analogous to a single project as seen by the compiler and represents everything needed to compile a Visual Basic or C# program such as assembly references, compiler options, and the set of source files to be compiled. With this context you can reason about the meaning of code. **Compilations** allow you to find **Symbols** – entities such as types, namespaces, members, and variables which names and other expressions refer to. The process of associating names and expressions with **Symbols** is called **Binding**.

Like **SyntaxTree**, **Compilation** is an abstract class with language-specific derivatives. When creating an instance of Compilation you must invoke a factory method on the **CSharpCompilation** (or **VisualBasicCompilation**) class.

Example – Creating a compilation

This example shows how to create a **Compilation** by adding assembly references and source files. Like the syntax trees, everything in the Symbols API and the Binding API is immutable.

1) Create a new C# Roslyn Console Application project.
   a. In Visual Studio, choose File -> New -> Project… to display the New Project dialog.
   b. Under Visual C# -> **Roslyn**, choose “Console Application”.
   c. Name your project “**SemanticsCS**” and click OK.
2) Replace the contents of your **Program.cs** with the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace SemanticsCS
{
    class Program
    {
        static void Main(string[] args)
        {
            SyntaxTree tree = CSharpSyntaxTree.ParseText(
@"using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}");

            var root = (CompilationUnitSyntax)tree.GetRoot();
         }
    }
}
```

3) Next, add this code to the end of your **Main** method to construct a **CSharpCompilation** object:

```
            var compilation = CSharpCompilation.Create("HelloWorld")
                                               .AddReferences(
                                                    new MetadataFileReference(
                                                        typeof(object).Assembly.Location))
                                               .AddSyntaxTrees(tree);
```

1) Move your cursor to the line containing the **closing brace** of your **Main** method and set a breakpoint there.
   - In Visual Studio, choose Debug -> Toggle Breakpoint.
2) Run the program.
   - In Visual Studio, choose Debug -> Start Debugging.
3) Inspect the root variable in the debugger by hovering over it and expanding the datatip.

The SemanticModel

Once you have a **Compilation** you can ask it for a **SemanticModel** for any **SyntaxTree** contained in that **Compilation**. **SemanticModels** can be queried to answer questions like “What names are in scope at this location?” “What members are accessible from this method?” “What variables are used in this block of text?” and “What does this name/expression refer to?”

Example – Binding a name

This example shows how to obtain a **SemanticModel** object for our HelloWorld **SyntaxTree**. Once the model is obtained, the name in the first **using** directive is bound to retrieve a **Symbol** for the **System** namespace.

1) Add this code to the end of your Main method. The code gets a **SemanticModel** for the HelloWorld **SyntaxTree** and stores it in a new variable:

```
            var model = compilation.GetSemanticModel(tree);
```

2) Set this statement as the next statement to be executed and execute it.
   - Right-click this line and choose Set Next Statement.
   - In Visual Studio, choose Debug -> Step Over, to execute this statement and initialize the new variable.
   - You will need to repeat this process for each of the following steps as we introduce new variables and inspect them with the debugger.
3) Now add this code to bind the **Name** of the “**using System;**” directive using the **SemanticModel.GetSymbolInfo** method:

```
            var nameInfo = model.GetSymbolInfo(root.Usings[0].Name);
```

4) Execute this statement and hover over the **nameInfo** variable and expand the datatip to inspect the **SymbolInfo** object returned.

        - Note the **Symbol** property. This property returns the **Symbol** this expression refers to. For expressions which don't refer to anything (such as numeric literals) this property will be null.
            - Note that the **Symbol.Kind** property returns the value **SymbolKind.Namespace**.

5) Cast the symbol to a **NamespaceSymbol** instance and store it in a new variable:

```
            var systemSymbol = (INamespaceSymbol)nameInfo.Symbol;
```

6) Execute this statement and examine the **systemSymbol** variable using the debugger datatips.
7) Stop the program.
    - In Visual Studio, choose Debug -> Stop debugging.
8) Add the following code to enumerate the sub-namespaces of the **System** namespace and print their names to the **Console**:

```
            foreach (var ns in systemSymbol.GetNamespaceMembers())
            {
                Console.WriteLine(ns.Name);
            }
```

9) Press Ctrl+F5 to run the program. You should see the following output:

```
Collections
Configuration
Deployment
Diagnostics
Globalization
IO
Reflection
Resources
Runtime
Security
StubHelpers
Text
Threading
Press any key to continue . . .
```

Example – Binding an expression

The previous example showed how to bind name to find a **Symbol**. However, there are other expressions in a C# program that can be bound that aren't names. This example shows how binding works with other expression types - in this case a simple string literal.

1) Add the following code to locate the "**Hello, World!**" string literal in the **SyntaxTree** and store it in a variable (it should be the only **LiteralExpressionSyntax** in this example):

```
            var helloWorldString = root.DescendantNodes()
                                       .OfType<LiteralExpressionSyntax>()
                                       .First();
```

2) Start debugging the program.

3) Add the following code to get the **TypeInfo** for this expression:

```
var literalInfo = model.GetTypeInfo(helloWorldString);
```

4) Execute this statement and examine the **literalInfo**.
   - Note that its **Type** property is not null and returns the **INamedTypeSymbol** for the **System.String** type because the string literal expression has a compile-time type of **System.String**
5) Stop the program.
6) Add the following code to enumerate the public methods of the **System.String** class which return strings and print their names to the **Console**:

```
var stringTypeSymbol = (INamedTypeSymbol)literalInfo.Type;

Console.Clear();
foreach (var name in (from method in stringTypeSymbol.GetMembers()
                                                  .OfType<IMethodSymbol>()
                   where method.ReturnType.Equals(stringTypeSymbol) &&
                         method.DeclaredAccessibility ==
                                    Accessibility.Public
                   select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

7) Press Ctrl+F5 to run to run the program without debugging it. You should see the following output:

```
Join
Substring
Trim
TrimStart
TrimEnd
Normalize
PadLeft
PadRight
ToLower
ToLowerInvariant
ToUpper
ToUpperInvariant
ToString
Insert
Replace
Remove
Format
Copy
Concat
Intern
IsInterned
Press any key to continue . . .
```

8) Your **Program.cs** file should now look like this:

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace SemanticsCS
{
    class Program
    {
        static void Main(string[] args)
        {
            SyntaxTree tree = CSharpSyntaxTree.ParseText(
@" using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}");

            var root = (CompilationUnitSyntax)tree.GetRoot();

            var compilation = CSharpCompilation.Create("HelloWorld")
                                               .AddReferences(
                                                    new MetadataFileReference(
                                                        typeof(object).Assembly.Location))
                                               .AddSyntaxTrees(tree);

            var model = compilation.GetSemanticModel(tree);

            var nameInfo = model.GetSymbolInfo(root.Usings[0].Name);

            var systemSymbol = (INamespaceSymbol)nameInfo.Symbol;

            foreach (var ns in systemSymbol.GetNamespaceMembers())
            {
                Console.WriteLine(ns.Name);
            }

            var helloWorldString = root.DescendantNodes()
                                       .OfType<LiteralExpressionSyntax>()
                                       .First();
            var literalInfo = model.GetTypeInfo(helloWorldString);

            var stringTypeSymbol = (INamedTypeSymbol)literalInfo.Type;

            Console.Clear();
            foreach (var name in (from method in stringTypeSymbol.GetMembers()
                                                              .OfType<IMethodSymbol>()
                                  where method.ReturnType.Equals(stringTypeSymbol) &&
                                        method.DeclaredAccessibility ==
                                                   Accessibility.Public
                                  select method.Name).Distinct())
            {
                Console.WriteLine(name);
            }
        }
    }
}
```

9) Congratulations! You've just used the **Symbol** and **Binding APIs** to analyze the meaning of names and expressions in a C# program.