

Getting Started: Syntax Analysis

March 2014

Prerequisites

- Visual Studio 2013
- “Roslyn” End User Preview
- “Roslyn” SDK Project Templates
- “Roslyn” SDK Syntax Visualizer

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 that information is accurate and complete.

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

Understanding Syntax Trees

The **Syntax API** exposes the syntax trees the compilers use to understand Visual Basic and C# programs. They are produced by the same parser that runs when a project is built or a developer hits F5. The syntax trees have full-fidelity with the language; every bit of information in a code file is represented in the tree, including things like comments or whitespace. Writing a syntax tree to text will reproduce the exact original text that was parsed. The syntax trees are also immutable; once created a syntax tree can never be changed. This means consumers of the trees can analyze the trees on multiple threads, without locks or other concurrency measures, with the security that the data will never change under.

The four primary building blocks of syntax trees are:

- The **SyntaxTree** class, an instance of which represents an entire parse tree. **SyntaxTree** is an abstract class which has language-specific derivatives. To parse syntax in a particular language you will need to use the parse methods on the **CSharpSyntaxTree** (or **VisualBasicSyntaxTree**) class.
- The **SyntaxNode** class, instances of which represent syntactic constructs such as declarations, statements, clauses, and expressions.

- The **SyntaxToken** structure, which represents an individual keyword, identifier, operator, or punctuation.
- And lastly the **SyntaxTrivia** structure, which represents syntactically insignificant bits of information such as the whitespace between tokens, preprocessor directives, and comments.

SyntaxNodes are composed hierarchically to form a tree that completely represents everything in a fragment of Visual Basic or C# code. For example, were you to examine the following C# source file using the “Roslyn” Syntax Visualizer (In Visual Studio, choose View -> Other Windows -> Roslyn Syntax Visualizer) its tree view would look like this:

SyntaxNode: Blue
SyntaxToken: Green
SyntaxTrivia: Red

C# Code File

The image shows a Visual Studio window with a C# code file on the left and its corresponding syntax tree on the right. The code file is named 'Program' and contains the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;

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

The syntax tree on the right is titled 'Syntax Tree' and shows the hierarchical structure of the code. The root node is 'CompilationUnit [0..185]', which branches into 'UsingDirective [0..13]', 'UsingDirective [15..48]', and 'UsingDirective [50..68]'. The 'UsingDirective [50..68]' node branches into 'UsingKeyword [50..55]', 'QualifiedName [56..67]', and 'SemicolonToken [67..68]'. The 'QualifiedName [56..67]' node branches into 'IdentifierName [56..62]', 'DotToken [62..63]', and 'IdentifierName [63..67]'. The 'IdentifierName [63..67]' node branches into 'IdentifierToken [63..67]'. The 'SemicolonToken [67..68]' node branches into 'Trail: EndOfLineTrivia [68..70]'. The 'CompilationUnit [0..185]' node also branches into 'ClassDeclaration [72..185]', 'MethodDeclaration [94..182]', and 'EndOfFileToken [185..185]'. The 'ClassDeclaration [72..185]' node branches into 'ClassKeyword [72..77]', 'IdentifierToken [78..85]', 'OpenBraceToken [87..88]', and 'Block [131..182]'. The 'MethodDeclaration [94..182]' node branches into 'StaticKeyword [94..100]', 'Lead: WhitespaceTrivia [90..94]', 'Trail: WhitespaceTrivia [100..101]', 'PredefinedType [101..105]', 'ParameterList [110..125]', and 'Block [131..182]'. The 'PredefinedType [101..105]' node branches into 'VoidKeyword [101..105]' and 'IdentifierToken [106..110]'. The 'ParameterList [110..125]' node branches into 'OpenParenToken [110..111]', 'Parameter [111..124]', and 'CloseParenToken [124..125]'. The 'Parameter [111..124]' node branches into 'ArrayType [111..119]', 'PredefinedType [111..117]', 'ArrayRankSpecifier [117..119]', and 'IdentifierToken [120..124]'. The 'ArrayType [111..119]' node branches into 'PredefinedType [111..117]' and 'ArrayRankSpecifier [117..119]'. The 'PredefinedType [111..117]' node branches into 'IdentifierToken [120..124]'. The 'Block [131..182]' node branches into 'OpenBraceToken [131..132]', 'ExpressionStatement [142..175]', and 'CloseBraceToken [181..182]'. The 'ExpressionStatement [142..175]' node branches into 'InvocationExpression [142..174]', 'SemicolonToken [174..175]', and 'CloseBraceToken [181..182]'. The 'InvocationExpression [142..174]' node branches into 'SimpleMemberAccessExpression [142..159]', 'ArgumentList [159..174]', and 'CloseBraceToken [181..182]'. The 'SimpleMemberAccessExpression [142..159]' node branches into 'IdentifierName [142..149]', 'DotToken [149..150]', and 'IdentifierName [150..159]'. The 'ArgumentList [159..174]' node branches into 'SemicolonToken [174..175]' and 'CloseBraceToken [181..182]'. The 'CloseBraceToken [181..182]' node branches into 'CloseBraceToken [184..185]'. The 'EndOfFileToken [185..185]' node is the final node in the tree.

By navigating this tree structure you can find any statement, expression, token, or bit of whitespace in a code file!

Traversing Trees

Manual Traversal

The following steps use **Edit and Continue** to demonstrate how to parse C# source text and find a parameter declaration contained in the source.

Example – Manually traversing the tree

- 1) Create a new C# Roslyn Console Application project.
 - In Visual Studio, choose File -> New -> Project... to display the New Project dialog.
 - Under Visual C# -> Roslyn, choose “Console Application”.
 - Name your project “**GettingStartedCS**” and click OK.
- 2) Enter the following code into your **Main** method:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

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

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

- 3) 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.
- 4) Run the program.
 - In Visual Studio, choose Debug -> Start Debugging.
- 5) Inspect the root variable in the debugger by hovering over it and expanding the datatip.
 - Note that its **Usings** property is a collection with four elements; one for each using directive in the parsed text.
 - Note that the **Kind** of the root node (in ‘base’) is **SyntaxKind.CompilationUnit**.
 - Note that the **Members** collection of the **CompilationUnitSyntax** node has one element.

- 6) Insert the following statement at the end of the Main method to store the first member of the root **CompilationUnitSyntax** variable into a new variable:

```
var firstMember = root.Members[0];
```

- 7) 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.
- 8) Hover over the **firstMember** variable and expand the datatips to inspect it.
- Note that its **Kind** is **SyntaxKind.NamespaceDeclaration**.
 - Note that its run-time type is **NamespaceDeclarationSyntax**.
- 9) Cast this node to **NamespaceDeclarationSyntax** and store it in a new variable:

```
var helloWorldDeclaration = (NamespaceDeclarationSyntax)firstMember;
```

- 10) Execute this statement and examine the **helloWorldDeclaration** variable.
- Note that like the **CompilationUnitSyntax**, **NamespaceDeclarationSyntax** also has a **Members** collection.
- 11) Examine the **Members** collection.
- Note that it contains a single member. Examine it.
 - i. Note that its **Kind** is **SyntaxKind.ClassDeclaration**.
 - ii. Note that its run-time type is **ClassDeclarationSyntax**.
- 12) Cast this node to **ClassDeclarationSyntax** and store it in a new variable:

```
var programDeclaration = (ClassDeclarationSyntax)helloWorldDeclaration.Members[0];
```

- 13) Execute this statement.
- 14) Locate the **Main** declaration in the **programDeclaration.Members** collection and store it in a new variable:

```
var mainDeclaration = (MethodDeclarationSyntax)programDeclaration.Members[0];
```

- 15) Execute this statement and examine the members of the **MethodDeclarationSyntax** object.
- Note the **ReturnType**, and **Identifier** properties.
 - Note the **Body** property.
 - Note the **ParameterList** property; examine it.
 - i. Note that it contains both the open and close parentheses of the parameter list in addition to the list of parameters themselves.
 - ii. Note that the parameters are stored as a **SeparatedSyntaxList<ParameterSyntax>**.
- 16) Store the first parameter of the **Main** declaration in a variable.

```
var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

17) Execute this statement and examine the **argsParameter** variable.

- Examine the **Identifier** property; note that it is of the structure type **SyntaxToken**.
- Examine the properties of the **Identifier SyntaxToken**; note that the text of the identifier can be found in the **ValueText** property.

18) Stop the program.

- In Visual Studio, choose Debug -> Stop debugging.

19) Your program should look like this now:

```

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

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

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

            var firstMember = root.Members[0];

            var helloWorldDeclaration = (NamespaceDeclarationSyntax)firstMember;

            var programDeclaration = (ClassDeclarationSyntax)helloWorldDeclaration.Members[0];

            var mainDeclaration = (MethodDeclarationSyntax)programDeclaration.Members[0];

            var argsParameter = mainDeclaration.ParameterList.Parameters[0];

        }
    }
}

```

Query Methods

In addition to traversing trees using the properties of the **SyntaxNode** derived classes you can also explore the syntax tree using the query methods defined on **SyntaxNode**. These methods should be

immediately familiar to anyone familiar with XPath. You can use these methods with LINQ to quickly find things in a tree.

Example - Using query methods

- 1) Using IntelliSense, examine the members of the **SyntaxNode** class through the root variable.
 - Note query methods such as **DescendantNodes**, **AncestorsAndSelf**, and **ChildNodes**.
- 2) Add the following statements to the end of the Main method. The first statement uses a LINQ expression and the **DescendantNodes** method to locate the same parameter as in the previous example:

```
var firstParameters = from methodDeclaration in root.DescendantNodes()  
                      .OfType<MethodDeclarationSyntax>()  
                      where methodDeclaration.Identifier.ValueText == "Main"  
                      select methodDeclaration.ParameterList.Parameters.First();  
  
var argsParameter2 = firstParameters.Single();
```

- 3) Start debugging the program.
- 4) Open the Immediate Window.
 - In Visual Studio, choose Debug -> Windows -> Immediate.
- 5) Using the Immediate window, type the expression **argsParameter == argsParameter2** and press enter to evaluate it.
 - Note that the LINQ expression found the same parameter as manually navigating the tree.
- 6) Stop the program.

SyntaxWalkers

Often you'll want to find all nodes of a specific type in a syntax tree, for example, every property declaration in a file. By extending the **CSharpSyntaxWalker** class and overriding the **VisitPropertyDeclaration** method, you can process every property declaration in a syntax trees without knowing its structure beforehand. **CSharpSyntaxWalker** is a specific kind of **SyntaxVisitor** which recursively visits a node and each of its children.

Example - Implementing a SyntaxWalker

This example shows how to implement a **CSharpSyntaxWalker** which examines an entire syntax tree and collects any **using** directives it finds which aren't importing a **System** namespace.

- 1) Create a new C# Roslyn Console Application project; name it "**UsingCollectorCS**".
- 2) Enter the following code into your **Main** method:

```

        SyntaxTree tree = CSharpSyntaxTree.ParseText(
@"using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;

namespace TopLevel
{
    using Microsoft;
    using System.ComponentModel;

    namespace Child1
    {
        using Microsoft.Win32;
        using System.Runtime.InteropServices;

        class Foo { }
    }

    namespace Child2
    {
        using System.CodeDom;
        using Microsoft.CSharp;

        class Bar { }
    }
}");

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

```

- 3) Note that this source text contains **using** directives scattered across four different locations: the file-level, in the top-level namespace, and in the two nested namespaces.
- 4) Add a new class file to the project.
 - a. In Visual Studio, choose Project -> Add Class...
 - b. In the "Add New Item" dialog type **UsingCollector.cs** as the filename.
- 5) Add using directives for the **Microsoft.CodeAnalysis** namespace and the **Microsoft.CodeAnalysis.CSharp** namespace.

```

using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

```

- 6) Make the new **UsingCollector** class in this file extend the **CSharpSyntaxWalker** class:

```

class UsingCollector : CSharpSyntaxWalker

```

- 7) Declare a public read-only field in the **UsingCollector** class; we'll use this variable to store the **UsingDirectiveSyntax** nodes we find:

```
public readonly List<UsingDirectiveSyntax> Usings = new List<UsingDirectiveSyntax>();
```

8) Override the **VisitUsingDirective** method:

```
public override void VisitUsingDirective(UsingDirectiveSyntax node)
{
}
}
```

9) Using IntelliSense, examine the **UsingDirectiveSyntax** class through the **node** parameter of this method.

a. Note the **Name** property of type **NameSyntax**; this stores the name of the namespace being imported.

10) Replace the code in the **VisitUsingDirective** method with the following to conditionally add the found **node** to the **Usings** collection if **Name** doesn't refer to the **System** namespace or any of its descendant namespaces:

```
if (node.Name.ToString() != "System" &&
    !node.Name.ToString().StartsWith("System."))
{
    this.Usings.Add(node);
}
```

11) The **UsingCollector.cs** file should now look like this:

```
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 UsingCollectorCS
{
    class UsingCollector : CSharpSyntaxWalker
    {
        public readonly List<UsingDirectiveSyntax> Usings = new List<UsingDirectiveSyntax>();

        public override void VisitUsingDirective(UsingDirectiveSyntax node)
        {
            if (node.Name.ToString() != "System" &&
                !node.Name.ToString().StartsWith("System."))
            {
                this.Usings.Add(node);
            }
        }
    }
}
```

- 12) Return to the **Program.cs** file.
- 13) Add the following code to the end of the **Main** method to create an instance of the **UsingCollector**, use that instance to visit the root of the parsed tree, and iterate over the **UsingDirectiveSyntax** nodes collected and print their names to the **Console**:

```
var collector = new UsingCollector();
collector.Visit(root);

foreach (var directive in collector.Usings)
{
    Console.WriteLine(directive.Name);
}
```

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

```

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 UsingCollectorCS
{
    class Program
    {
        static void Main(string[] args)
        {
            SyntaxTree tree = CSharpSyntaxTree.ParseText(
@"using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;

namespace TopLevel
{
    using Microsoft;
    using System.ComponentModel;

    namespace Child1
    {
        using Microsoft.Win32;
        using System.Runtime.InteropServices;

        class Foo { }
    }

    namespace Child2
    {
        using System.CodeDom;
        using Microsoft.CSharp;

        class Bar { }
    }
}");

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

            var collector = new UsingCollector();
            collector.Visit(root);

            foreach (var directive in collector.Usings)
            {
                Console.WriteLine(directive.Name);
            }
        }
    }
}

```

```
}  
    }  
    }  
}
```

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

```
Microsoft.CodeAnalysis  
Microsoft.CodeAnalysis.CSharp  
Microsoft  
Microsoft.Win32  
Microsoft.CSharp  
Press any key to continue . . .
```

16) Observe that the walker has located all non-**System** namespace **using** directives in all four places.

17) Congratulations! You've just used the **Syntax API** to locate specific kinds of **C#** statements and declarations in **C#** source code.