# General

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.

# Semantics

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.

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.

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?”

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);  
 }  
 }  
 }  
}

# Syntax Analysis

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.

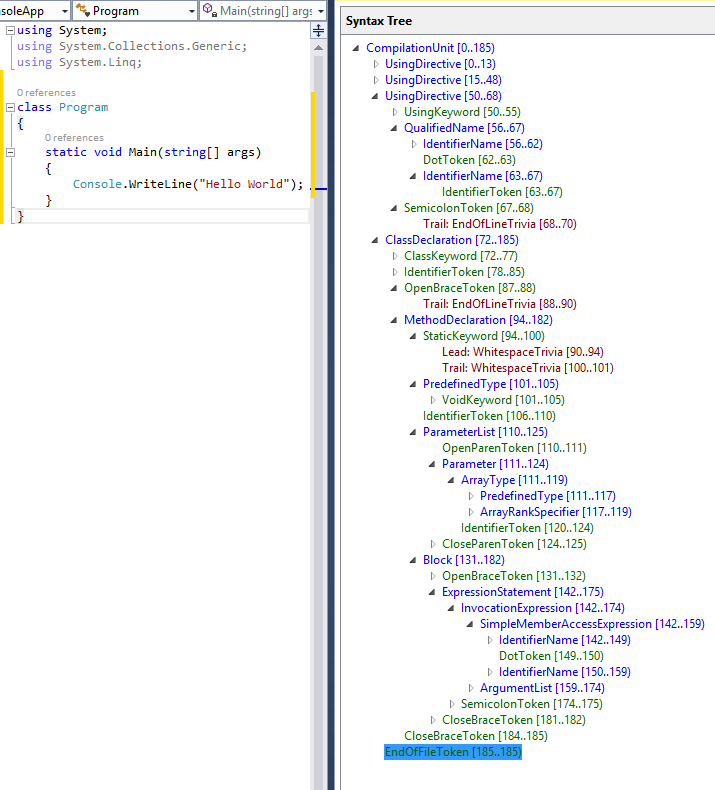
The **Syntax API** exposes the syntax trees the compilers use to understand Visual Basic and C# programs. They are produced by the same parser that runs when a project is built or a developer hits F5. The syntax trees have full-fidelity with the language; every bit of information in a code file is represented 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) it tree view would look like this:

SyntaxNode: Blue  
SyntaxToken: Green  
SyntaxTrivia: Red



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.

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

## 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.

            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();

# Syntax Transformation

This walkthrough builds on concepts and techniques explored in the **Getting Started: Syntax Analysis** and **Getting Started: Semantic Analysis** walkthroughs. If you haven’t already, it’s strongly advised that you complete those walkthroughs before beginning this one.

In this walkthrough, you’ll explore techniques for creating and transforming syntax trees. In combination with the techniques you learned in previous **Getting Started** walkthroughs, you will create your first command-line refactoring!

## Immutability and the .NET Compiler Platform

A fundamental tenet of the .NET Compiler Platform is immutability. Because immutable data structures cannot be changed after they are created, they can be safely shared and analyzed by multiple consumers simultaneously without the dangers of one tool affecting another in unpredictable ways. No locks or other concurrency measures needed. This applies to syntax trees, compilations, symbols, semantic models, and every other data structure you’ll encounter. Instead of modification, new objects are created based on specified differences to the old ones. You’ll apply this concept to syntax trees to create tree transformations!

This example uses the **SyntaxFactory** class methods to construct a **NameSyntax** representing the **System.Collections.Generic** namespace.

**NameSyntax** is the base class for four types of names that appear in C#:

* **IdentifierNameSyntax** which represents simple single identifier names like **System** and **Microsoft**
* **GenericNameSyntax** which represents a generic type or method name such as **List<int>**
* **QualifiedNameSyntax** which represents a qualified name of the form <left-name>.<right-identifier-or-generic-name> such as **System.IO**
* **AliasQualifiedNameSyntax** which represents a name using an assembly extern alias such a **LibraryV2::Foo**

## Modifying Nodes with With\* and ReplaceNode Methods

Because the syntax trees are immutable, the **Syntax API** provides no direct mechanism for modifying an existing syntax tree after construction. However, the **Syntax API** does provide methods for producing new trees based on specified changes to existing ones. Each concrete class that derives from **SyntaxNode** defines **With\*** methods which you can use to specify changes to its child properties. Additionally, the **ReplaceNode** extension method can be used to replace a descendent node in a subtree. Without this method updating a node would also require manually updating its parent to point to the newly created child and repeating this process up the entire tree – a process known as *re-spining* the tree.

    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;  
using Microsoft.CodeAnalysis.CSharp.SyntaxFactory;  
   
namespace ConstructionCS  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            NameSyntax name = IdentifierName("System");  
            name = QualifiedName(name, IdentifierName("Collections"));  
            name = QualifiedName(name, IdentifierName("Generic"));  
   
            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 oldUsing = root.Usings[1];  
            var newUsing = oldUsing.WithName(name);  
   
            root = root.ReplaceNode(oldUsing, newUsing);  
        }  
    }  
}

## Transforming Trees using SyntaxRewriters

The **With\*** and **ReplaceNode** methods provide convenient means to transform individual branches of a syntax tree. However, often it may be necessary to perform multiple transformations on a syntax tree in concert. The **SyntaxRewriter** class is a subclass of **SyntaxVisitor** which can be used to apply a transformation to a specific type of **SyntaxNode.** It is also possibleto apply a set of transformations to multiple types of **SyntaxNode** wherever they appear in a syntax tree. The following example demonstrates this in a naïve implementation of a command-line refactoring which removes explicit types in local variable declarations anywhere where type inference could be used. This example makes use of techniques discussed in this walkthrough as well as the **Getting Started: Syntactic Analysis** and **Getting Started: Semantic Analysis** walkthroughs.

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;  
using Microsoft.CodeAnalysis.CSharp.SyntaxFactory;  
   
namespace TransformationCS  
{  
    public class TypeInferenceRewriter : CSharpSyntaxRewriter  
    {  
        private readonly SemanticModel SemanticModel;  
   
        public TypeInferenceRewriter(SemanticModel semanticModel)  
        {  
            this.SemanticModel = semanticModel;  
        }  
   
        public override SyntaxNode VisitLocalDeclarationStatement(  
 LocalDeclarationStatementSyntax node)  
        {  
            if (node.Declaration.Variables.Count > 1)   
            {  
                return node;  
            }  
            if (node.Declaration.Variables[0].Initializer == null)  
            {  
                return node;  
            }  
   
            VariableDeclaratorSyntax declarator = node.Declaration.Variables.First();  
            TypeSyntax variableTypeName = node.Declaration.Type;  
              
            ITypeSymbol variableType =   
 (ITypeSymbol)SemanticModel.GetSymbolInfo(variableTypeName)  
 .Symbol;  
              
            TypeInfo initializerInfo =   
 SemanticModel.GetTypeInfo(declarator  
 .Initializer  
 .Value);  
              
            if (variableType == initializerInfo.Type)  
            {  
                TypeSyntax varTypeName =   
 SyntaxFactory.IdentifierName("var")  
                                     .WithLeadingTrivia(  
 variableTypeName.GetLeadingTrivia())  
                                     .WithTrailingTrivia(  
 variableTypeName.GetTrailingTrivia());  
   
                return node.ReplaceNode(variableTypeName, varTypeName);  
            }  
            else  
            {  
                return node;  
            }  
        }  
    }  
}

using System;  
using System.Collections.Generic;  
using System.IO;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
using Microsoft.CodeAnalysis;  
using Microsoft.CodeAnalysis.CSharp;  
  
namespace TransformationCS  
{  
 internal class Program  
 {  
 private static void Main()  
 {  
 Compilation test = CreateTestCompilation();  
  
 foreach (SyntaxTree sourceTree in test.SyntaxTrees)  
 {  
 SemanticModel model = test.GetSemanticModel(sourceTree);  
  
 TypeInferenceRewriter rewriter = new TypeInferenceRewriter(model);  
  
 SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());  
  
 if (newSource != sourceTree.GetRoot())  
 {  
 File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());  
 }  
 }  
 }  
  
 private static Compilation CreateTestCompilation()  
 {  
 SyntaxTree programTree =  
 CSharpSyntaxTree.ParseFile(@"..\..\Program.cs");  
 SyntaxTree rewriterTree =  
 CSharpSyntaxTree.ParseFile(@"..\..\TypeInferenceRewriter.cs");  
  
 SyntaxTree[] sourceTrees = { programTree, rewriterTree };  
  
 MetadataReference mscorlib =  
 new MetadataFileReference(typeof(object).Assembly.Location);  
 MetadataReference codeAnalysis =  
 new MetadataFileReference(typeof(SyntaxTree).Assembly.Location);  
 MetadataReference csharpCodeAnalysis =  
 new MetadataFileReference(typeof(CSharpSyntaxTree).Assembly.Location);  
  
 MetadataReference[] references = { mscorlib, codeAnalysis, csharpCodeAnalysis };  
  
 return CSharpCompilation.Create("TransformationCS",  
 sourceTrees,  
 references,  
 new CSharpCompilationOptions(  
 OutputKind.ConsoleApplication));  
 }  
 }  
}