Masaryk University
Faculty of Informatics

# Custom Roslyn Tool for Real-Time Static Code Analysis

Master's Thesis

**Zuzana Dankovčíková**

Brno, Spring 2017

Masaryk University
Faculty of Informatics

# Custom Roslyn Tool for Real-Time Static Code Analysis

Master's Thesis

**Zuzana Dankovčíková**

Brno, Spring 2017

*This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.*

# Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Zuzana Dankovčíková

**Advisor:** Bruno Rossi PhD

# Acknowledgement

TODO: This is the acknowledgement…

# Abstract

TODO: This is the abstract ...

# Keywords

roslyn, C#, compilers, code review, .NET compiler platform, Kentico, analyzer, code fix..., …

# Contents

# List of Tables

# List of Figures

# 1 Introduction

[1-2 pages]

TODO...

Ideas:

What is code quality, why is it important, tool that support it.. compilers, diversion ... aaaand here comes Roslyn which provides compiler as a platform.

In the .NET world, the compiler used to be a black box that given the file paths to the source text, produced an executable. In order to do that, compiler has to collect large amount of information about the code it is processing. This knowledge, however, was unavailable to anyone but the compiler itself and it was immediately forgotten once the translated output was produced [**roslyn-overview-github**].

Why is this an issue when for decades this black-boxes served us well? Programmers are increasingly becoming reliant upon the powerful integrated development environments (IDEs). Features like IntelliSense, intelligent rename, refactoring or "Find all references" are key to developers' productivity; and even more so in an enterprise-size systems.

This gave a rise to number of tools that analyze the code for common issues and are able to suggest a refactoring. The problem is that that such tool needs to parse the code first in order to be able to understand and analyze it. As a result companies need to invest fair amount of resources to duplicate the logic that the .NET compiler already possesses. Not only is it possible that the compiler and the tool may disagree on some specific piece of code, but with every new version of C# the tool needs to be updated to handle new language features[**dot-net-development-using-the-compiler-api**].

With roslyn.. etc. etc. .. API for analysis.. use in companies for custom analyzers... etc. etc.... https://github.com/dotnet/roslyn/wiki/Roslyn Overview – motivation Make sure to stress out that ".NET Compiler Platform" and "Roslyn" names will be used interchangably as it is in Roslyn Succinctly on page 11.

# 2 Compilers

This chapter will provide a high level overview of compiler internals that is important for the following two chapters on static code analysis and .NET Compiler Platform, both of which build upon compiler's fundamentals.

As per [**dragon-book**], compiler is a program that can read a program in a *source* language and translate it into a semantically equivalent program in a *target* language while reporting any errors detected in the translation process. The compiler may sometimes rely on other programs. For example, *preprocessor* is responsible for collecting the source code to be fed to compiler by expanding shorthands (macros) into source language statements.

The compilation process can be divided into two parts: *analysis* and *synthesis*.

The purpose of the analysis part is to break up the source program into chunks and build up a grammatical structure that it corresponds to, based on the source language grammar. This structure is subsequently transformed into an intermediate representation of the source program. Along the way, compiler collects information about the program and stores it to a data structure called *symbol table*. If any errors in syntax or semantics are encountered, analysis part shall inform programmer about the problem. Otherwise, both intermediate representation and symbol table are passed to the synthesis part which is responsible for constructing the target program from them.

The two main steps of compilation process internally consist of different phases as shown in Figure 2.1. Each phase transforms one representation of source language into another, and passes it to the following phase, while working with the symbol table during the process. In synthesis phase, an optional machine-independent optimizations can take place and are done on the top of intermediate representation. After target-machine code is generated, additional machine-dependent code optimizations are performed.

For the purpose of this thesis, only analysis part is relevant and following section will elaborate on its respective phases.

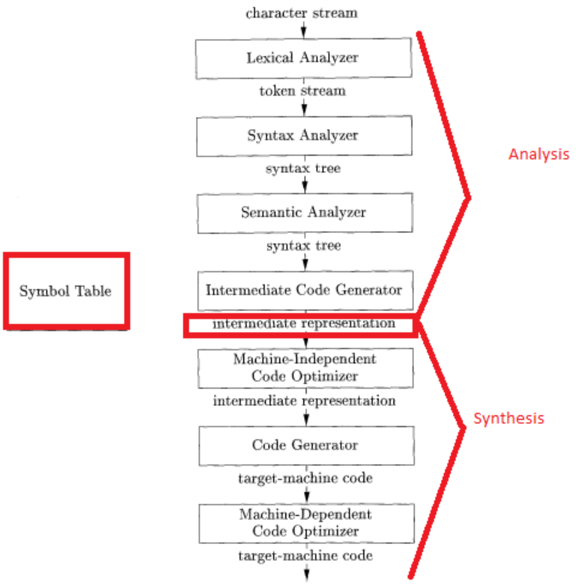


Figure 2.1: TODO: Phases of the compiler [**dragon-book**]

## 2.1 Lexical analysis

The compilation process starts with *lexical analysis* or *scanning*. The scanner transforms stream of characters of the source program, as written by the programmer, into the series of meaningful sequences called *lexemes*. Most programming languages allow for an arbitrary number of white spaces to be present in the source text to aid readability. However, white spaces, similarly as comments, are unimportant for the target code generation itself, and thus lexical analyser is responsible for discarding them completely.

In order to be able to correctly recognize the lexeme, lexical analyzer may need to read ahead. For example, in C–like languages if the scanner sees < character, it cannot decide whether it is a lexeme for *"less then"* operator or it is s part of *"less then or equal to"* lexeme. In order to do that, it needs to read ahead and see if the following character is = or not. Reading ahead is usually implemented with an input buffer which the lexical analyzer can read from and push back to. The use of buffer also boosts the performance as fetching block of characters is more efficient as fetching one at a time [**dragon-book**].

The lexical analyzer typically uses regular expressions to identify the lexemes and for each lexeme, it outputs a *token* (or *token object*) of the form

$$\langle \textit{token-name}, \textit{attribute-name} \rangle \tag{2.1}$$

For an input sequence

$$total = 42 + base * interest \tag{2.2}$$

the scanner output could be

$$\langle id, 0\rangle \langle = \rangle \langle num, 1\rangle \langle + \rangle \langle id, 2\rangle \langle * \rangle \langle id, 3\rangle \tag{2.3}$$

Lexemes can be divided into logical groups such as identifiers, relational operators, arithmetical operators, constants or keywords as seen in the example above. Scanner often uses regular expressions to identify tokens.

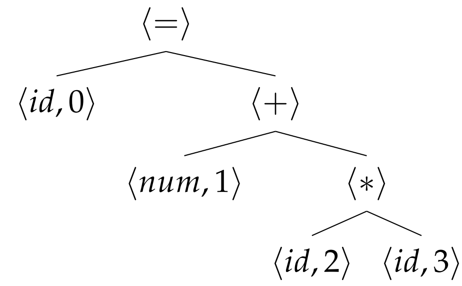
Each identifier (`id`) has an attribute which points to the entry of the symbol table, where information about identifier name, type or position in the source text is stored. Similar holds for constants like "42" in the example. In the (2.2) example, the assignment and addition symbols do not have attributes but different representation can be used, such as $\langle \textit{bin-op}, 2\rangle$. In this case, `bin-op` would denote it is a binary operator and number two would be a pointer to symbol table with all symbols for binary operations with the second index suggesting that it represents addition.

## 2.2 Syntax Analysis

The stream of token objects along with partially populated symbol table is an input for the subsequent compiler phase – *syntax analysis* or *parsing*. The parser has to verify that the sequence of token names can by produced by the grammar for the source language and for a well-formed program, it shall output a *syntax tree* or often referred to as an abstract syntax tree (AST)[1].

[1] The AST is an intermediate representation of source program in which each interior node represents an operation (programming construct) with the children of the node representing the arguments of that operation. As opposed to *parse syntax tree*, in which interior nodes are nonterminals of the grammar, ASTs are more lightweight and the might omit some nodes which exist purely as a result of grammar's production rules.

The resulting AST for the token stream generated in (2.2) is depicted in figure XY and it shows how multiplication precedence rule was applied on the expression.



The syntax analyzer uses a context free grammar (CFG) to from the syntax tree. The CFG is defined by a 4-tuple consisting of:

**Terminals** – token names (first component of the token) as obtained from previous compilation step.

**Nonterminals** – syntactic variables that help to impose the hierarchical structure of the language and represent set of stings.

**Start symbol** – reword – a special nonterminal which set of strings represents the language generated by the grammar.

**Productions** – rules that specify how nonterminals can be rewritten to sequences of zero or more terminal and nonterminal symbols.

An example of a production denoting the construction of a `while-cycle` would be

$$\mathit{stmt} \to \textbf{while (}\ \mathit{expr}\ \textbf{)}\ \textbf{\{}\ \mathit{stmt}\ \textbf{\}}, \tag{2.4}$$

where nonterminals *stmt* and *expr* stand for a statement and expression respectively (defined by other productions). Symbols in bold represent terminals of the grammar – open and close parenthesis and curly braces, plus `while` keyword.

### 2.2.1 Error Handling

There are several types of errors that can be encountered during the compilation process. *Lexical errors* such as misspelling the identifier name, *syntactic errors* like missing semicolon, *semantic error* for example wrong number of function arguments or *logical errors* that do not really prevent the program from compiling but can indicate possible mistakes (for instance using the assignment operator = instead of the comparison operator == in condition of an if-statement).

It's parser responsibility to report the presence of potential syntactic error, recover from the error in order to continue with syntactic analysis and be able to detect any subsequent errors. There are two main strategies for the error recovery:

Panic-Mode Recovery

In this method, after parser encounters an error, it searches for a *synchronizing token* (usually delimiters such as semicolon or }) and until found, all the symbols are thrown away one by one. Even though panic-mode recovery often discards significant amount of input while searching for the synchronization token, it is guaranteed not to end up in an infinite loop.

Phrase-Level Recovery

Another approach the parser can take to recover from an erroneous input is trying to perform a local correction. This can be done by replacing the prefix of the following input by some tokens that would enable syntactic analyzer to continue parsing. A prime example of phrase-level recovery is inserting a missing semicolon or replacing coma with a semicolon. Even though this technique is very powerful, as it can cope with all possible problems in the input, it might lead to infinite loops (e.g. always inserting symbols ahead of current symbol).

## 2.3 Semantic Analysis

## 2.4 Intermediate Code Generation

## 2.5 Symbol Table Management

11+85

# 3 Code Quality and Static Code Analysis

Software quality is [**software-engineering-practicioners-approach**]: *"An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it."* This definition can be viewed from two perspectives:

- **user (customer) perspective** – *a useful product*
- **developer perspective** – *an effective software process*

Software quality is represented by internal and external software characteristics [**code-complete**].

External quality characteristics are ones that the user of the software product is primarily concerned with. These are for example correctness, usability, efficiency, reliability, integrity, adaptability, accuracy and robustness.

On the other hand, internal characteristics such as maintainability, flexibility, portability, reusability, readability and testability; are only important for programmers and have no visible customer value.

However, these attributes influence the external ones. For example, if software is not readable internally, it is hard to find and fix bugs which directly affect users perception of software's reliability and correctness. For software company, high internal quality means less maintenance effort, faster time-to-market, fewer bug and thus reduced customer support. It enables engineers to focus on developing new features rather then dealing with unmaintainable code base.

While external quality, or conformance to customer requirements, is mostly -?- checked -?- by functional testing, there is more to software quality. Next sections will take a look at who overall quality of code can be raised... OMG this is such a wierd paragraph...

## 3.1 ...

## 3.2 Code review process

- Code review process

## 3.3 Static Code Analysis

- compilers and static code analysis (some data from the dragon book)

## 3.4 Static Code Analysis Tools available on .NET platform

[2-3 pages?] - Resharper,
- FxCop
- StyleCop
- new DotNetAnalyzers available thanks to Roslyn

# 4 .NET Compiler Platform

In the .NET world, the compiler used to be a black box that given the file paths to the source text, produced an executable. This perception was changed in 2015 when Microsoft introduced the .NET Compiler Platform (commonly referred to as "Project Roslyn").

Not only have been compilers for both Visual Basic and C# rewritten into the entirely managed code, they also expose the internals of the compiler pipeline via a public .NET API [1]. This makes them a platform (also known as *compiler-as-as-service*) with rich code analysis APIs that can be leveraged by developers to perform analysis, code generation or dynamic compilation in their own programs [**roslyn-succinctly**]. Those can be then easily integrated into the Visual Studio all without the hard work of duplicating compilers' parsing logic.

This chapter will take a look at how the Roslyn API layers are structured, how the original source code is represented by the compiler and how developers can build tools upon the compiler's API. Note that although Roslyn provides equivalent APIs for both VisualBasic and C#, this thesis will only focus on the latter since only that one is relevant for the practical part of the thesis.

## 4.1 The Compiler Pipeline

Roslyn compilers expose an API layer that mirrors the traditional compiler pipeline (see 4.1). However, instead of a single process of generating the target program, each compilation step is treated as a separate component [**roslyn-overview**]:

- **Parse phase** consists of *lexical analysis* (*scanner*) and *syntactic analysis* (*parser*). First, the lexical analyzer processes the stream of characters from the source program and groups them into meaningful sequences called *lexemes*. Those are subsequently processed by the *syntax analyzer* that creates a tree-like structure of tokens based on the language's grammar [**dragon-book**].

[1] Application Programming Interface

- **Symbols and metadata phase** where named symbols are generated based on the declarations from the source and imported metadata.

- **Bind phase** in which the identifiers from the source code are matched to their respective symbols.

- **Emit phase** where all the gathered information is used to emmit an assembly.

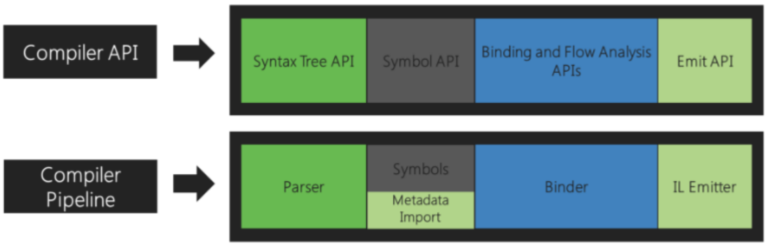


Figure 4.1: Compiler pipeline [**roslyn-overview**]

In each phase, the .NET Compiler Platform creates an object model containing gathered information and exposes it through the API in form of .NET objects. These object are also used internally by Visual Studio [2] to support basic IDE functionality. For instance **syntax tree**, that is the result of the parse phase, is used to support formatting and colorizing the code in the editor. The result of the second phase – **hierarchical symbol table**, is the basis for *Object browser* and *Navigate to* functionality. Binding phase is represented as an **object model that exposes the result of the semantic analysis** and is utilized in *Find all references* or *Go to definition*. Finally, the Emit phase produces the Intermediate Language (IL) byte codes and is also used for *Edit and Continue* feature [**roslyn-overview**].

[2] The new generation of Visual Stuio leveraging from the Roslyn compiler are called vNext and first one was VS 2015.

## 4.2 The .NET Compiler Platform's Architecture

The Roslyn's architecture consists of two main layers - Compiler and Workspaces APIs, and one secondary layer - Features API, as seen on Figure 4.2.

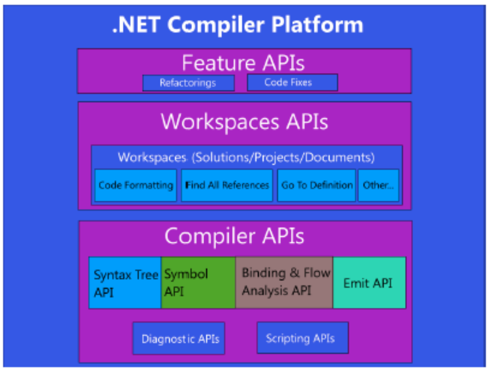


Figure 4.2: .NET Compiler Platform Architecture [**roslyn-succincly**]

One of the key concepts of .NET Compiler Platform is immutability. The compiler exposes hundreds of types that represent all information about source code from `Project` and `Document` to `SyntaxTrees` with almost all of those types being immutable. This means, that once created, the object cannot change. In order to alter it in any way, new instance must be created, either manually, or from an existing instance by applying one of many `With()` methods that the API provides.

The immutability enables the compiler to perform parallel work without need to create duplicate objects or apply any locks on them. This concept is useful for the command line compiler but it is considered extremely important for IDEs where it enables for one document to be handled by multiple analyzers in parallel.

### 4.2.1 The Compiler APIs

As discussed in the previous section, the Compiler APIs offer an object model representing the results of syntactic and semantic analysis produced by the respective phases of the compiler pipeline. Moreover, it also includes an immutable snapshot of a single compiler invocation, along with assembly references, compiler options, and source files.

This layer is agnostic of any Visual Studio components and as such can be used in stand-alone applications as well. There are two separate, though very similar, APIs for Visual Basic and C#, each providing functionality tailored for specific language nuances.

Diagnostic APIs

Apart from parsing code and producing an assembly, the compiler is also capable of raising diagnostics, covering everything from syntax to semantics, and report them as errors, warnings or information messages [**roslyn-succinctly**]. This is achieved through the compilers' Diagnostics APIs that allow developers to effectively plug-in to compiler pipeline, analyze the source code using the exposed object models, and surface custom diagnostics along with those defined by the compiler itself. These APIs are integrated to both MSBuild [3] and Visual Studio (2015 and newer). providing seamless developer experience. The practical part of this thesis relies on Diagnostic APIs to provide custom diagnostics and the details will be discussed in Chapter 5.

Scripting APIs

As a part of the compiler layer, Microsoft team has introduced new Scripting APIs that can be used for executing code snippets. These APIs were not shipped with .NET Compiler Platform 1.0 and are part of v2.0.0 RC3[4].

### 4.2.2 Workspaces APIs

Workspace represents a collection of solutions, projects, and documents. It provides a single object model containing information about the projects in a solution and their respective documents; exposes all configuration options, assembly and inter-project dependencies, and provides access to syntax trees and semantic models. It is a start-

[3] The Microsoft Build Engine https://github.com/Microsoft/msbuild

[4] Release candidate 3, as per https://github.com/dotnet/roslyn/wiki/Scripting-API-Samples [26-02-2017].

ing point for performing code analysis and refactorings over entire solutions.

Although it is possible to use the `Workspace` outside of any host environment, the most common use case is an IDE providing an instance of `Workspace` that corresponds to the open solution. Since the instances of `Solution` are immutable, the host environment must react to every event (such as user key stroke) with an update of the `CurrentSolution` property of the `Workspace`.

### 4.2.3 Feature APIs

This layer relies on both compiler and workspaces layers and is designed to provide API for offering code fixes and refactorings. This layer was also used while working on the practical part of this thesis.

## 4.3 Syntax Tree

As mentioned in the previous sections, the product of the syntactic analysis is a syntax tree. It enables developers to work with the code in a managed way instead of working against plain text. Syntax trees are used for both analysis and refactorings, where the new code is generated either manually or as a modified version of the existing tree. While being immutable, syntax trees are thread-safe and analysis can be done in parallel.

It is important to point out, that in a same way the compiler constructs a syntax tree from the source text, it is also possible to round-trip back to the text representation. Thus, the source information is always preserved in full fidelity. This means that every piece of information from source must be stored somewhere within the tree, including comments, whitespaces or end-of-line characters.

Figure 4.3 shows a syntax tree of an invocation expression as obtained from Syntax Visualizer[5] extension available in Visual Studio. This tool is useful for understanding how Roslyn represents particular language constructs and is widely utilized whenever one needs to analyze the code. Following sections explain what are the main building blocks of such syntax tree and will refer to this figure:

[5] https://roslyn.codeplex.com/wikipage?title=Syntax%20Visualizer

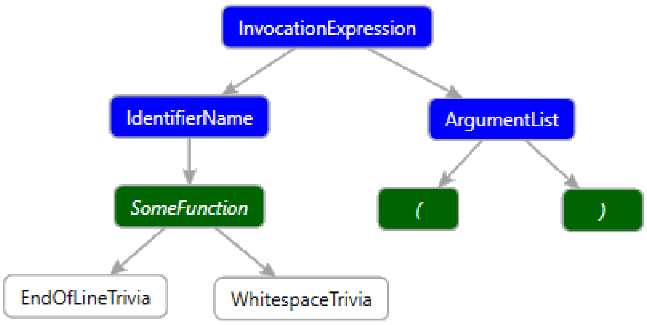


Figure 4.3: Syntax tree of an invocation expression

### Syntax Nodes

Syntax nodes (blue color) are non-terminal nodes of a syntax tree, meaning they always have at least one other node or token as a child. Nodes represent syntactic constructs of a language such as statements, clauses or declarations. Each type of node is represented by a single class deriving from `SyntaxNode`. Apart from common properties `Parent`, `ChildNodes` and utility methods like `DescendantNodes`, `DescendantTokens`, or `DescendantTrivia`, each subclass exposes specific methods and properties. As shown in Figure 4.3, `InvocationExpression` has two properties, `IdentifierName` and `ArgumentList` both of which are `SyntaxNodes` themselves.

### Syntax Tokens

As opposed to nodes, syntax token (green color) represent terminals of the language grammar, such as keywords, punctuation, literals and identifiers. For the sake of efficiency, `SyntaxToken` is implemented as a value type (C# structure) and there is only one for all kinds of tokens. To be able to tell them apart, tokens have `Kind` property. For example, `SomeFunction` is of kind `IdentifierName`, whereas "(" character is `OpenParenToken`.

### Syntax Trivia

In order to enable refactoring features, syntax trees must also store information about whitespaces, comments and preprocessor directives that are insignificant for compilation process itself. This information is represented by another value type – `SyntaxTrivia` (white color). Trivia are not really parts of the tree itself, rather they are properties of tokens accessible by their `LeadingTrivia` and `TrailingTrivia` collections.

## 4.4 Semantics of the Program

Even though syntax trees are enough to describe proper form of the program (compliance to the language grammar), they cannot enforce all language rules, for example, type checking. In order to tell whether a method is called with the right number of arguments, or operator is applied to operands or right type, it's inevitable to introduce semantics.

As described in [**dragon-book**], one of the core responsibilities of a compiler is to collect information about all elements and their properties from the source program. These are attributes such as identifier name, type, allocated storage, scope or for method names the number and types of arguments and their return values.

All this data is being incrementally collected when parsing the source code (analysis phase) and is stored in *symbol tables*. These are later used in synthesis where intermediate language representation is produced.

### Symbols

In .NET Compiler Platform, a single entry of a symbol table is represented by a class deriving from `ISymbol`. The symbol represents every distinct element (namespace, type, field, property, event, method or parameter) either declared in the source code or imported as metadata from a referenced assembly. Each specific symbol has its own methods and properties often directly referring to other symbols. For example `IMethodSymbol` has a `ReturnType` property specifying what is the type symbol the method returns.

### Compilation

An important immutable type, that represents everything needed to compile a C# (or Visual Basic) program is a `Compilation`. It contains all source files, compiler options and assembly references. Compilation provides convenient ways to access any discovered symbol. For instance, it is possible to access the entire hierarchical symbol table rooted by global namespace or look up type symbols by their common metadata names.

### Semantic Model

When analyzing a single source file of a compilation, all its semantic information is available through a *semantic model*. The `SmeanticModel` object can answer many questions such as:

- What symbol is declared at the specific location in the source?
- What is the result type of an expression?
- What symbols are visible from this location?
- What diagnostics are reported in the document?

This makes semantic model very useful when performing static code analysis concerned with more than just syntax.

## 4.5 Analysers and Code Refactorings

[3 pages?]

# 5 Implementation of Custom Analyzers

## 5.1 CMS Internal Guidelines

[3-4 pages] What is Kentico CSM?
Current situation & motivation
- how code reviews are done at kentico
- tools that aid code reviews (??)
- original BugHunter
- use of FxCop and ReSharper
[5-7 pages] How was the tool implemented,
Project structure,
What it contains
Concerns about Performance

# 6 Measuring and Optimizing the Performance

[up to 7 pages?]

Why tool needs to be super-fast (refer to chapter 4 where this should have been said)

Talk about /ReportAnalyzer switch of MSBuild process (csc.exe)

How the performance of the slowest analyzers (SystemIO, BaseChecks) was improved

Talk about how analyzers deployment influenced the build time

Questionares sent to development team, feedback from senior developers

# 7 Conclusion

[1-2 pages]

# A Questionnaires

TODO...

# B Deployment and Versioning

[2 pages]