Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Compiling Handlebars to C#

Bachelor Thesis of

Jakob Demler

At the Department of Computer Science
Chair for Computer Science II
Software Engineering

Reviewer:	Prof. Dr. Samuel Kounev
Advisor:	Dipl. Inf. Jürgen Walter

Duration: 11. October 2015 – 21. December 2015

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Würzburg, 18.12.2015

.....

(**Jakob Demler**)

Contents

1 Introduction

Before the Model-View-Controller (MVC) architectural pattern gained popularity in development of web-based applications, many problems existed within that domain: application logic embedded into HTML (i.e. PHP, JavaServerPages) "makes both the programming code and the HTML markup more difficult to understand, maintain, and debug." [21] MVC tried to solve this problem by decoupling business logic and user interface which results in applications that are "more loosely coupled, have higher cohesion, and minimize intra and inter-crosscutting" [21].

Nevertheless, even in modern MVC web-frameworks, like Ruby on Rails or ASP.NET MVC, there is still code mixed into HTML in form of view logic inside the view. As web applications grow and get more complex, their view logic often turns non-trivial, making the views more difficult to develop and maintain.

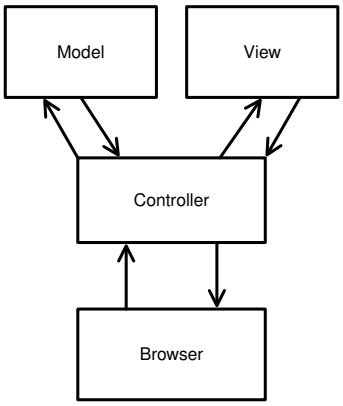


Figure 1: Schematic representation of the MVC-Architecture in the Web context: The browser sends a request which the controller processes by rendering the corresponding view with the data of the corresponding model

Following the rationale of the introduction of the MVC architectural pattern to web applications, yet another decoupling mechanism is expected to further improve their development speed and maintainability: The full seperation of the backend code and the HTML.

Using ViewModels which contain all the information needed to be displayed and passing them to logic-less Views will ensure this separation.
For these logic-less templates a suitable language is needed: "Establishing new languages is a powerful strategy for controlling complexity in engineering design; we can often enhance our ability to deal with a complex problem by adopting a new language that enables us to describe (and hence to think about) the problem in a different way, using primitives, means of combination, and means of abstraction that are particularly well suited to the problem at hand." [1]

One example for such logic-less templating language is Handlebars [9]. It offers sufficient functionality to write convenient templates, while preventing the inclusion of logic within the view. Alas, the Handlebars reference implementation is written in JavaScript and thus not trivially and without drawbacks usable for rendering templates on the server side.
As server-side rendering of templates has lots of advantages and other Handlebars implementations for C# fail to transfer the advantages of statically typed languages (e.g. performance and type safety) to

the domain of logic-less templates, this work will provide a tool that will not only allow the user to render Handlebars templates on the server-side, but also will compile them into native, typechecked and performant C# code.

The thesis is outlined as follows: Section 2 offers the reader an introduction to Handlebars and used libraries and tools. Section 3 sets goals and explains an approach reaching them. In section 4 design decisions are discussed: project architecture, Handlebars grammar and the compilation state as an essential concept. Implementation details are described in section 5, including AST-Classes, the code generation visitor and an example of a template and resulting generated code. Validation and performance comparison to alternatives are examined in section 6. Finally, section 7 summarizes and concludes results.

2 Foundations

In this section foundations are explained to follow subsequent sections easily. First, the wording in the title of this thesis is discussed followed by a brief introduction to Handlebars and Roslyn. Finally three parser generators are presented.

2.1 Wording

In many cases the term "compilation" is associated with the result of it being machine language or something close to it. Nevertheless "compilation" is defined as “read[ing] a program in one language - the *source* language - and translat[ing] it into an equivalent program in another language - the *target* language” [2] and is thus an umbrella term for all translation processes of computer languages. Terms like "Transpiler" or "Source-to-Source-Compiler" would be more precise, but they are rarely used. The usage of the term "compilation" can be boiled down to three main reasons:

1. The term is widely known and used
2. The process described in this thesis falls under its definition
3. It is associated with type safety and performance, which is one emphasis of this thesis

2.2 Handlebars

Handlebars is a logic-less templating language with a reference implementation in JavaScript. Due to its lack of expressive power and complexity Handlebars templates are easy and effective to read and write as they force the separation of the view-logic and the view: All the information needed to display an entity needs to be available inside the ViewModel which is then passed to the template.

2.2.1 Examples

These brief Handlebars examples are aiming for a basic understanding of the Handlebars language. Syntax and semantics will be discussed in detail in section 4.2 and section 5.3, respectively. See [9] for detailed introduction to Handlebars.

Handlebars	Output
Basic Output	
{{Member}} {{{Member}}} {{Parent.Child}} {{this}}	HTML-encoded contents of "Member" raw contents of "Member" HTML-encoded contents of member "Child" of member "Parent". HTML-encoded contents of the current context.
Comments	
{{!Comment}} {{!– {{Comment}} –}}	singleline comment multiline comment which allows Handlebars inside itself.
Blocks	
{{#if X}}{{/if}} {{#each List}}{{/each}} {{#with Parent}}{{/with}}	body of the block if X is truthy enumerates over "List" and renders the blocks' body setting the context to the current item checks "Parent" for truthiness and renders the body with the context to "Parent"
Miscellaneous	
{{> PartialName Parameter}} {{HelperName Parameter}}	contents of a partial. Its context is set to "Parameter" contents of a helper function to which "Parameter" is passed

2.2.2 Implementations

Besides the reference implementation in JavaScript Handlebars implementations exist for C#. Yet they do not compile the templates at design or compile-time but parse and interpret them at run-time and thus inherit problems dynamically typed languages have (i.e. type-errors at run-time, performance etc.). Examples include [10] which is a wrapper around the original JavaScript implementation, [11] and [13] rely on reflection to resolve templates at run-time.

2.3 Technical Foundations

This section focuses on two components that provide functionality needed for the compilation process: compiler libraries and parser generators.

2.3.1 Compiler library

Information about the typesystem, in which the Handlebars-template is rendered needs to be available, as parts of the compilation depend on it. This information can be provided by a library.

Roslyn [16] is a compiler as a library, offering APIs for syntactically and semantically analysing and generating C# and Visual Basic code. It was published under the open-source Apache License 2.0 in April 2014 and is since then being developed further by Microsoft with additional efforts from Roslyn's community.

2.3.2 Parsergenerators

Parsergenerators or compiler-compilers are tools which take as input a grammar and output a corresponding parser. There exist a vast number of parsergenerators. Distinctive features include grammar family, parsing algorithm and output languages. A short summary of considered parser generators follows. The selection process focused on the usage of parsing expression grammar (PEG) and integratability into C#.

Pegasus [14] is a parsergenerator that uses a PEG described by Brayan Ford in 2002 [6]. Its resulting parser has the feature to output user defined types. Its code is being actively maintained and is in a stable state.

IronMeta [12] is similar to Pegasus yet its syntax is more verbose and it lacks the feature to use user-defined types.

Katahdin [23] generates the parser at run-time and thus allows mutation of the PEG. It is build on the Mono implementation of the .NET framework.

3 Approach

This chapter sets goals for this work and describes possible approaches to problems occuring during the compilation process.

3.1 Goals

The context in which this work is being done sets several requirements: relatively large teams working on relatively large and long living projects, deployed as horizontally scaleable cloud applications. In order to be able to make rational decisions about architecture and implementation choices a set of goals is needed as guideline. The following goals will set the limits and directions of these decisions:

1. **Integration into existing technologies:** A tool which can not be used with the usual web development stack is useless.
2. **Consistency with the Reference Implementation:** To allow reuse of existing Handlebars Templates and also rendering on the client-side, the resulting code needs to be consistent with the reference implementation.
3. **Flexibility:** As the handlebars language is relatively new and constantly growing and changing, the resulting compiler has to be flexible and extensible.
4. **Robustness:** Users will produce malformed Handlebars-Templates. The compiler has to react in a suitable way and produce understandable error messages.
5. **Performance:** In times of cloud computing and horizontally scalable applications performance overhead for trivial tasks like rendering HTML is not acceptable. Resulting code must therefore be performing well in terms of CPU-cycles and memory usage.

3.2 The Compilation Process

According to [2] the compilation process can be divided into the following six distinct phases:

1. Lexical Analysis
2. Syntactical Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation

For step one and two a parser will be used which will be generated by a parser generator and a grammar. The result of this parser will be an abstract syntax tree which can be traversed in step three to ensure type safety and gather information about used types. As the aim for this work is not to produce machine code, there will be no need for an intermediate code to be generated. Thus step four, five and six will be handled in one step: the generation of an optimized C# AST which represents the related C# code.

3.2.1 Lexical and Syntactical Analysis

Neither lexing nor parsing are an essential step in the compilation process and to meet the requirement of flexibility the lexing and parsing code has to be interchangeable. This will be achieved by making the result independent of the parser used: the resulting AST will consist entirely of entities defined inside the project, not inside the parser.
As Pegasus natively supports the output of user defined types, it is chosen as a suitable parsergenerator.

3.2.2 Semantic Analysis

In order to be able to check the used types to meet the requirement of robustness and as the resulting C# of a Handlebars template relies on the types at hand, information about the type system containing the Handlebars template must be retrievable in this step. As C# does not offer compile-time reflection and run-time reflection would conflict with the requirement of performance, a non native tool is the only option to get information about used types at design time. Two tools fall into this category: the EnvDTE Interface [20] with its CodeModel implementation and the Microsoft Roslyn compiler platform. As the EnvDTE Interface does not support generics natively, has no good documentation and is no longer maintained, Microsoft Roslyn will be used to achieve design time introspection.

Furthermore, following the dependency inversion principle (DIP) [17], to support flexibility and a possible change in the future, an interface will be defined and implemented which will encapsulate Roslyn's complexity and will allow type-system queries on a more abstract level. This will not only ensure more readable and thus maintainable code but will also allow future tools for design time introspection to be integrated seamlessly.

3.2.3 Code Optimization and Generation

For the code generation, like in the semantic analysis, there are two basic options available: The CodeDOM library [19] and again the Microsoft Roslyn compiler platform. The decision is similar to step 3: as CodeDOM is not complete and not maintained anymore and will be superseded by Roslyn, Roslyn will be used to generate the C#-AST. If it would be feasible to abstract from the AST-structure and thus also make the Roslyn code generation interchangeable is not fully decided yet. Building a class structure that is able to represent C# code is an enormous task and probably beyond the limits of this work.

The optimization of the resulting C# code could be done by preallocating memory for the resulting string or parallel execution of partials. Yet it is not clear if these methods are feasible and yield significant performance improvements. They will be assessed during the course of this work. Generating a string that represents the C# code of the C#-AST is just a call to a function, as Roslyn already implements this functionality.

3.2.4 Working with ASTs

In many of these steps the AST has to be traversed and certain functions must be called depending on the type of element. Using polymorphic inheritance is certainly a simple, fast and intuitive solution to this problem, but it is also problematic, as it is not extensible, it introduces dependencies and it may be hard to read and maintain, as code with the same task is scattered around class implementations. Possible alternatives include the visitor pattern [18], pattern matching [25] and multi-methods [22]. For an extensive comparison see [24]. Both multi-methods and pattern matching can only be implemented

into C# with the use of its dynamic type deduction ability and miss thus out on some advantages: performance, type checking and completeness checking at compile time.

Because of that, an adapted version of the visitor pattern, the hierarchical visitor pattern [5] will be used for all tasks that include dependencies (e.g. semantic analysis and code generation). For tasks which do not include dependencies to other types or libraries polymorphic inheritance and single dispatch functions will be used.

3.3 Integration

Resulting code needs to be able to call partial templates and helper functions, thus Handlebars templates need to be aware of other templates and helper functions.

One way to achieve this is to implement a Handlebars specific ViewEngine and let the MVC framework handle the overhead of routing and finding the correct template. Nevertheless, that approach introduces a dependency to the ASP.NET MVC framework. Therefore and because resolving partial calls with the MVC framework introduces massive performance overhead in contrast to native function calls, calls to partials and helper functions will be resolved at design-time, during the semantic analysis. This approach will yield independent and performant code, which can be used in any context, not just for web development.

Also the compilation process has to be seamlessly integrable into existing workflows. Visual Studio offers the possibility to add custom tools to project files which then are able to generate corresponding files. This approach is already used by the RazorGenerator [15] project which precompiles RazorViews into C# code and proves to be robust terms of conflicts and source control tools. Custom Tools are invoked when there associated files change.

3.4 Ensuring Robustness and Consistency with Unit Tests

As the JavaScript reference implementation has a large set of unit tests [8] it is possible to ensure robust and consistent code by adopting these test cases. Also new test cases will be introduced in order to test introspection and type checking capabilities.
However it is not trivial to implement unit tests that are able to test generated code, as the result needs to be compiled or interpreted in order to run it. This problem will again be solved with Roslyn: it has the capabilities to dynamically compile C# code into an assembly, which can then be invoked.

4 Design

This section will discuss design choices and their alternatives which were made with regard to the goals in section refsec:goals. First the large-scale architecture of the project and the newly developed Handlebars grammar will be introduced. Then the concept of how state is being handled during the compilation process is explained.

4.1 Project Architecture

The HandlebarsCompiler itself consists of four distinct sub-projects:

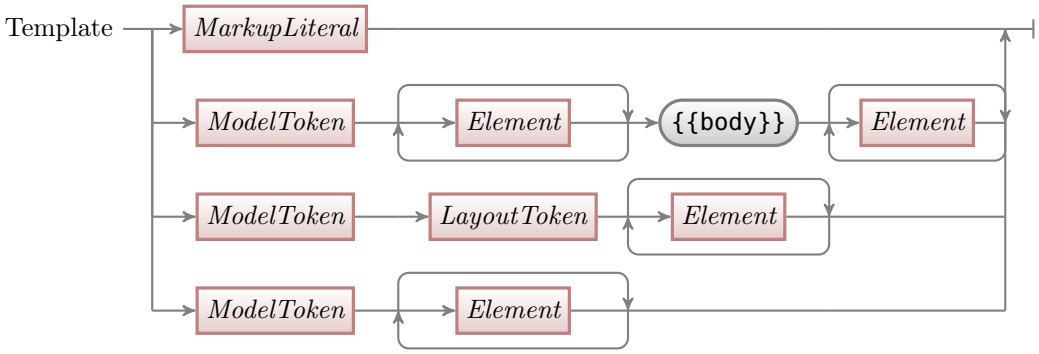
1. **Compiler:** the compiler itself as a class library. The compilation is done inside this sub-project and can be called using a simple interface.
2. **RuntimeUtils:** provides necessary functions used by the compiled templates at runtime. May be installed as a NuGet package. If it is not installed, the runtime functions are provided inside the compiled templates as static private methods.
3. **CustomTool:** enables integration into VisualStudio as a CustomTool. CustomTools are bound to single files and invoked when these files are saved. The custom tool then invokes the compiler.
4. **Commandline Interface:** this tool allows the HandlebarsCompiler to be used independently from VisualStudio and the CustomTool project. It expects the path to a solution file as input and then compiles all containing HandlebarsTemplates.
5. **CompilerTests:** contains the UnitTests for the Compiler project.

This division has the advantage, that more than one way of using the compiler can be easily implemented. Interfacing the compiler is possible through the CustomTool or the CommandLine, but more options would be possible. Also portability through several operating systems can be achieved that way. While the CustomTool is certainly bound to a Windows with a VisualStudio installation, the command line could be used on other operating systems as well.

4.2 Handlebars Grammar

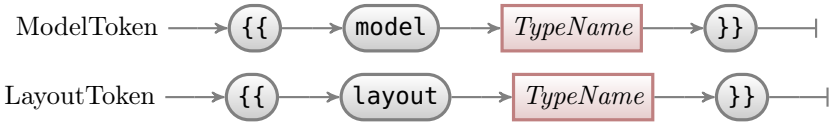
This section will describe the grammar which is passed to a parsergenerator. The rules described here are simplified and stripped from type information and syntax error handling rules; they are presented in form of syntax or railroad diagrams. As PEGs, unlike context free grammar (CFG)s, are not ambiguous one can read these railroad diagrams from top to bottom. A path will only be taken if non of the overlying paths match.
Non-terminals are displayed as red squares while terminals are displayed as blue squares with rounded corners. In some cases the grammar seems overly verbose; this is due to the fact that different paths may yield objects of different classes.

Template:
There are four basic options to a Handlebars-Template:

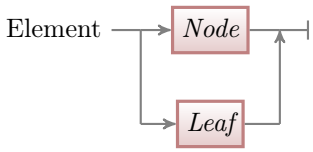


Except for the first these template-types share a common ModelToken, containing the namespace and classname of the ViewModel which will be passed to the template at run-time. The first option represents a StaticHandlebarsTemplate which does not contain Handlebars but only markup. The second option, a Handlebars-Layout, contains a special token ({{body}}) which signals where the content of the layout should be rendered. This token is surrounded by sequences of elements. The second and third option, a layouted Handlebar-Template and a Handlebars-Template, only differ in the LayoutToken, which determines the layout in which the template is to be rendered.

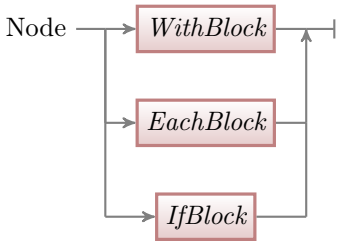
ModelToken and LayoutToken:
The ModelToken is, like the LayoutToken a special token which does not appear in the JavaScript implementation of Handlebars. They are used to specify information about types for the compiler. This information is captured by the TypeName rule which yields a TypeName that can be passed to Roslyn.



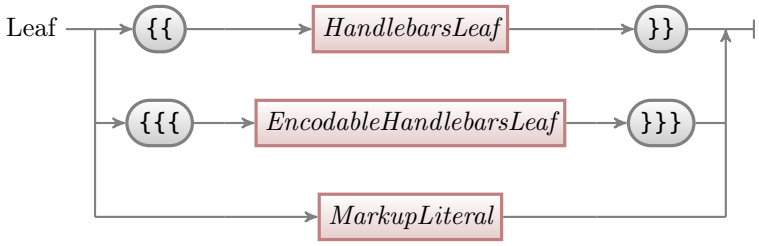
Element: Node or Leaf:
Elements are the most basic building block of Handlebars-Template as they represent every single element of the Handlebars-AST. Thus Elements can be either leafs or nodes in the AST:



Nodes are either WithBlocks, EachBlocks or IfBlocks:

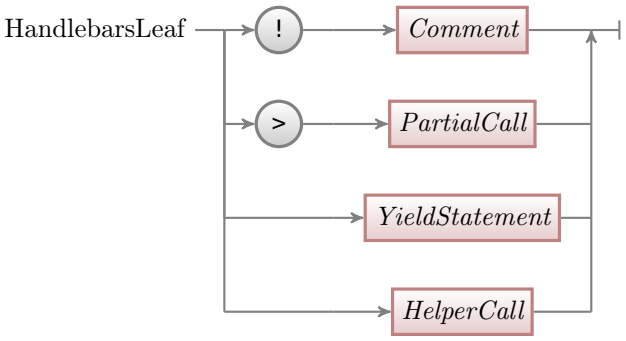


Leafs are either HandlebarsLeafs surrounded with double braces or an EncodableHandlebarsLeaf in triple braces, otherwise it is a MarkupLiteral:



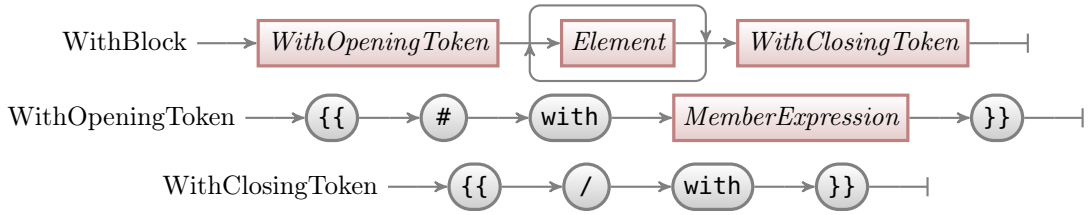
EncodableHandlebarsLeafs are either YieldStatements or HelperCalls.

HandlebarsLeafs can be either Comments, PartialCalls, YieldStatements or HelperCalls. Comments and PartialCalls have identifiers "!" and ">" to be distinguishable to YieldStatements.

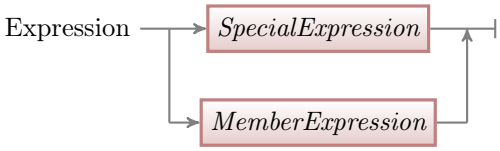


YieldStatements and HelperCalls with implied parameter are ambiguous however, as the Expression and TypeName rule can match the same values. Therefore the HelperCall-Rule does not include its implied parameter version. This ambiguity can only be resolved during the semantic analysis phase, with information about the containing type system.

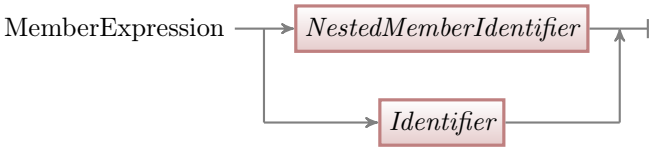
Representing all Block-Rules only the WithBlock-Rule will be described here in detail: Its body, a series of Elements, is enclosed by a WithOpeningToken and a WithClosing Token. While the WithOpening-Token holds information about the Member to which the context is set, the WithClosingToken just determines the end of the block and, thus carries no semantic value.



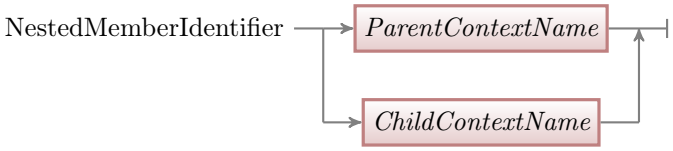
Expressions are the base of all parameters of HandlebarsTokens. They can either be a SpecialExpression which gets its value from the context it is used in, or a MemberExpression which represents a Member of the ViewModel.

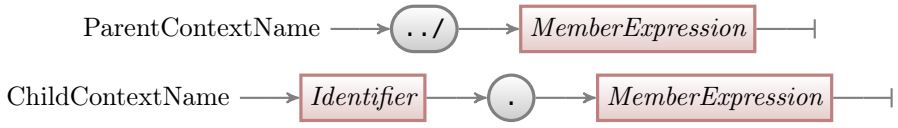


While SpecialExpressions are represented just by a keyword, MemberExpressions are more complex: they need to be able to represent a path through the context.



This path is either nested or not. If it is nested it is either a member of a child or a member of a parent. This pattern goes on recursively and is able to represent all possible paths, semantically correct or not, through the ViewModel.





4.3 CompilationState

During the traversion of the Handlebars-AST by a `CodeGenerationVisitor` (see section 5.5 for details) a lot of data accumulates and needs to be handled properly. This data is bundled and stored in a `CompilationState` object (for implementation details see section 5.6) which accompanies every compilation process. There are three basic types of data which need to be handled:

1. generated code
2. ViewModel context
3. Truthyness context

How these different types of data are stored and handled will now be discussed in detail.

4.3.1 Stacks of Stacks of Generated Code

The main result of the code generation process is code. To be more accurate, the `CodeGenerationVisitor` yields sequences of `StatementSyntax` objects which are part of the C#-AST provided by Roslyn.
Nevertheless, these sequences of `StatementSyntax` object need to follow a certain structure: code blocks are `StatementSyntax` objects which, in turn, themselves contain `StatementSyntax` objects. During traversal of the AST, the `CodeGenerationVisitor` does not, and should not, know about the block which will contain the generated statement for the current AST-Element.
However if a block is to be generated, the containing statements need to be included. So the information which `StatementSyntax` resides in what block needs to be available.
This problem has an elegant solution: if generated statements are stored in a stack of lists and every statement is appended to the list on the top of the stack the block operations become trivial: At the start of a new block, an empty list of statements is pushed to the stack; at its end the stack is popped and the resulting list of statements are now the block body. See section 5.5 for an example.

4.3.2 Keeping Track of the Context

In a Handlebars-Template the context, the current scope of the ViewModel, can change. It determines which members are directly accessible or only accessible through pathed member expressions. Initially the context is the ViewModel itself; #with- and #each-blocks can change the context. If, for example, the ViewModel has a member "A", {{#with A}} will change the context from "ViewModel" to "ViewModel.A". This is also possible over more than one level (i.e. {{#with A.B.C}} sets the context to "ViewModel.A.B.C").

Paths can contain one or more path-up-elements (e.g. "../"); their semantics is not one step back in the current context, but switch to the parent context (i.e. in the previous example "../" switches to the context "ViewModel" not "ViewModel.A.B").

Hence, a data-structure has to exist, that can easily store and manipulate the context and all its parents. Again, a stack is used to solve the problem; operations such as looking up the parent context or changing to a new context become trivial (e.g. push and pop). A #with-block, for example, pushes its context on the context-stack on entering and pops it on leaving; a path-up-element pops the context-stack and passes the result to the next element.

4.3.3 Optimize Truthiness Checks and Detect Unreachable Code

During the process of rendering a Handlebars-Template, objects need to be checked for truthyness. For example "{{#if A.B}}" will check "ViewModel" as well as "ViewModel.A" and "ViewModel.A.B" for truthyness. The naive implementation would cause redundant checks to happen inside #if- and #with-blocks. To be able to identify which checks are necessary and which are not, a data-structure is needed to keep track of that information. And again it is a stack which works similar to the context-stack described in section 4.3.2, with the difference that #if-blocks and #each-blocks perform different operations on it. The implications of such a truthyness-stack however, are not to be underestimated. Not only is it now possible to distinguish between necessary and unnecessary thruthyness checks but also unreachable code can be detected.

5 Implementation

This chapter covers implementation details of the compiler and the code generated for a handlebars template by it. First, an overview over the compilation process and the involved components is given, followed by an instruction on how to use this works compiler. To better understand the subsequent sections, we provide an example of the resulting code of the compilation process. Next, some components are discussed in more detail: the Handlebars-AST class hierarchy representing a handlebars template, the `IASTVisitor` interface and its primary implementation, the `CodeGenerationVisitor` traversing the Handlebars-AST and generating code and the `CompilationState` which holds data accumulating during the compilation process. Finally, two components that directly interact with Roslyn are introduced: the `SyntaxHelper` and the `RoslynIntrospector`.

5.1 Overview

Figure 2 shows the schematic representation of components central to this project. The compilation process itself starts with a handlbars template ("template.hbs" in figure 2) which is parsed by a Parser. This parser handles lexical as well as syntactical analysis and returns a fully initialized `HandlebarsTemplate` holding the Handlebars-AST which represents the template. Section 5.4 describes these classes in detail. The parser itself is generated by a parsergenerator, for this project Pegasus was chosen (see section 3.2.1, and a grammar which has already been discussed in section 4.2("grammar.peg" in figure 2).
The next step is the traversal of the create Handlebars-AST by the `CodeGenerationVisitor`. This step involves the semantic analysis which is happening in the `RoslynIntrospector` (see section 5.8 for details) as well as the code generation handled by the `SyntaxHelper` (see section 5.7 for details).
All accumulated data is stored in a `CompilationState` object (see section 4.3 for design and section 5.6 for implementation details) that also handles the final assembly of the resulting C# document("template.hbs.cs" in figure 2).

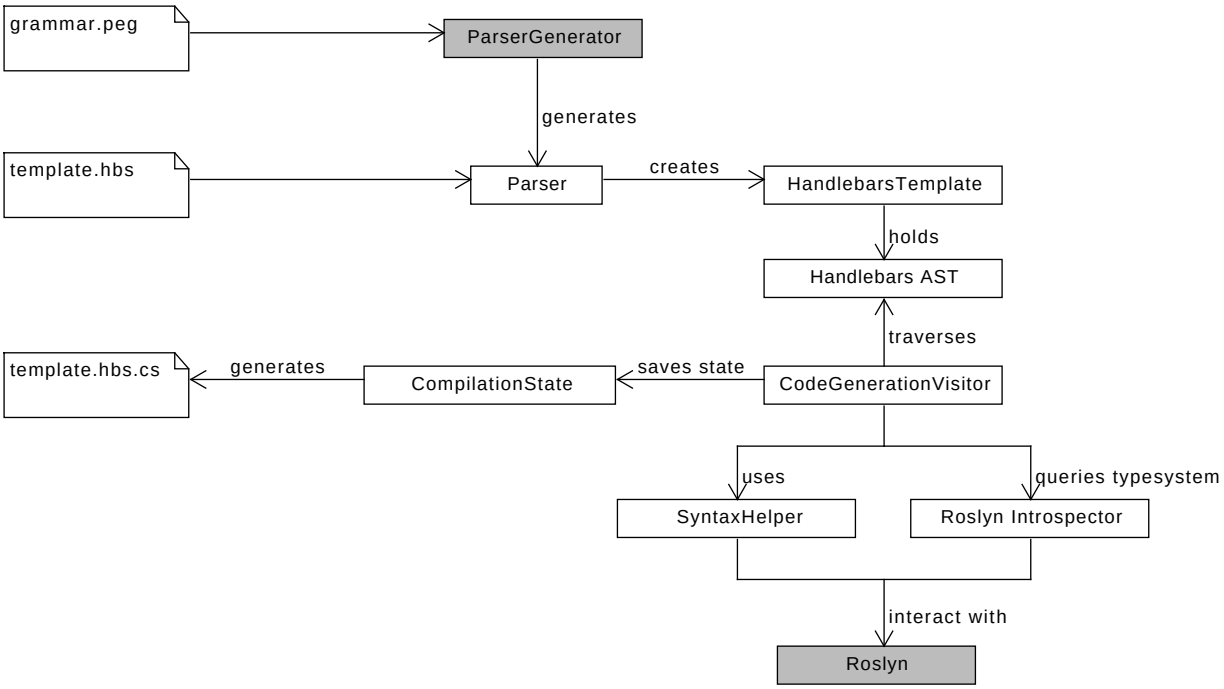


Figure 2: Schematic representation of this project's components and their interaction. Components with white background are part of this project's implementation.

5.2 Usage

This project is shipped as a VisualStudio extension and can be installed by simply running the "HandlebarsCompiler.vsix" file found on the releases tab of its Github repository [7].
In order to be able to use this work's compiler, one first needs a ViewModel and a handlebars-template. Assuming following simple ViewModel:

```
namespace ViewModels
{
  public class PersonModel
  {
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
    public List<string> EMailAddresses { get; set; }
  }
}
```

An appropriate handlebars-template that displays the information of a person is contained in a .hbs file and might look like this:

```
{{model ViewModels.PersonModel}}
<h1>Hello {{FirstName}} {{LastName}}</h1>
<p>You are {{Age}} years old and you have
{{#if EMailAddresses}}
  following email addresses:
  <ul>
    {{#each EMailAddresses}}
      <li>{{this}}</li>
    {{/each}}
  </ul>
{{else}}
  no email addresses.
{{/if}}</p>
```

To invoke the compilation process the property "CustomTool" in VisualStudio of the file containing the handlebars template has to be set to "HandlebarsCompiler". The compilation will then be triggered every time the file is saved and will produce a .hbs.cs file next to the .hbs file containing the template. Now, the template can be invoked by calling `TemplateName.Render(person)` which will return the rendered HTML as string.

5.2.1 Using Partials

Partials can be used in the same way as in handlebars.js: `{{> PartialName ParameterName}}` or with `this` as implied parameter: `{{> PartialName}}`. Assume the ViewModel from the above example and following handlebars template residing in "EMailList.hbs":

```
{{model System.Collections.Generic.List<System.String>}}
{{#if this}}
  following email addresses:
  <ul>
    {{#each this}}
      <li>{{this}}</li>
    {{/each}}
  </ul>
{{else}}
  no email addresses.
{{/if}}
```

The template used in the section above can now be simplified:

```
{{model ViewModels.PersonModel}}
<h1>Hello {{FirstName}} {{LastName}}</h1>
<p>You are {{Age}} years old and you have
{{> EMailList EMailAddresses}}</p>
```

As template names are not necessarily unique these calls can be ambiguous. Therefore parts of the namespace of the template can be specified. Assume the "EMailList.hbs" template resides in the namespace "Project.Views.Partials". Any of the following calls would call the correct partial, assuming there is no other "Partials" namespace:

- `{{> Partials.EMailList}}`
- `{{> Views.Partials.EMailList}}`
- `{{> Project.Views.Partials.EMailList}}`

5.2.2 Using Layouts

The concept of layouts is unique to this implementation of handlebars. There is no equivalent feature implemented in handlebars.js yet. It allows the embedding of one template into another. This is especially helpful for elements that must be rendered for many or all templates, for example the HTML head. In order to use layouts one must first declare a handlebars-layout: As described in section 4.2 a handlebars-template differs from a handlebars-layout only in a `{{body}}` token. An exemplary HandlebarsLayout,residing in "MainLayout.hbs" could look like this:

```
{{model ViewModels.IPageModel}}
<!DOCTYPE html>
  <head>
    <title>{{Title}}</title>
    {{#if Keywords}}<meta name="keywords" content="{{Keywords}}">{{/if}}
  </head>
  <body>
    {{body}}
  </body>
</html>
```

The contents of an embedded template would be rendered inside the HTML-body-tags. Assume following ViewModel:

```
namespace ViewModels
{
  public class TitlePageModel : IPageModel
  {
    public string Title { get; set; }
    public string Keywords { get; set; }
    public string Headline { get; set; }
    public string Content { get; set; }
  }
}
```

And handlebars-template:

```
{{model ViewModels.TitlePageModel}}
{{layout MainLayout}}
<h1>{{Headline}}</h1>
<p>{{Content}}</p>
```

This template is equivalent to

```
{{model ViewModels.TitlePageModel}}
<!DOCTYPE html>
  <head>
    <title>{{Title}}</title>
    {{#if Keywords}}<meta name="keywords" content="{{Keywords}}">{{/if}}
  </head>
  <body>
    <h1>{{Headline}}</h1>
    <p>{{Content}}</p>
  </body>
</html>
```

This feature proves to be useful, especially as with server-side rendering, opposed to client-side rendering, full HTML-pages have to be rendered.

5.2.3 Using Helpers

Helpers provide a way to call C# (or JavaScript using handlebars.js) code from handlebars templates. Though this contradicts the concept of logic-less templates and should used with caution, it can be very convenient for some cases.

A helper has to be accessible from the template, static and return a string, other from that there are no further restrictions. For the compiler to recognize the function as a helper, it has to be attributed with a `CompiledHandlebarsHelperMethod`-attribute. Assume following helper:

```
[CompiledHandlebarsHelperMethod]
pulic static string FullName(PersonModel model)
{
  return string.Concat(model.FirstName, " ", model.LastName);
}
```

The "FullName" helper can now be used inside a template like this:

```
{{model ViewModels.PersonModel}}
<h1>Hello {{FullName this}}</h1>
<p>You are {{Age}} years old and you have
{{#if EMailAddresses}}
  following email addresses:
  <ul>
    {{#each EMailAddresses}}
      <li>{{this}}</li>
    {{/each}}
  </ul>
{{else}}
  no email addresses.
{{/if}}</p>
```

Helpers with multiple parameters are possible:

```
[CompiledHandlebarsHelperMethod]
```

```
pulic static string FullName(string firstName, string lastName)
{
  return string.Concat(firstName, " ", lastName);
}
```

5.3 GeneratedCode

Even though the generated code differs in detail for each different kind of `HandlebarsTemplate`, they all share the same structure: A static class containing a static method named "Render" which returns a string.

```
[CompiledHandlebarsTemplate]
public static class BasicTest
{
  public static string Render(/*...*/)
  {
    /*...*/
  }
}
```

The "Render" method always starts with the declaration of a local instance of a `StringBuilder` and ends with a return statement that returns the value of the "ToString" method of this `StringBuilder`:

```
public static string Render(/*...*/)
{
  var sb = new StringBuilder();
  /*...*/
  return sb.ToString();
}
```

Between these two statements the actual rendering happens, which is now explained by example.

Handlebars	Resulting Statement (assuming context is "viewModel")
HTML	`sb.Append("<p>HTML</p>")`
{{Member}}	`sb.Append(WebUtility.HtmlEncode(viewModel.Member))`
{{{Member}}}	`sb.Append(viewModel.Member)`
{{Parent.Child}}	`sb.Append(WebUtility.HtmlEncode(viewModel.Child.Parent))`
{{this}}	`sb.Append(WebUtility.HtmlEncode(viewModel))`
{{> Partial Param}}	`sb.Append(Partial.Render(viewModel.Param))`
{{{HelperName Parameter}}}	`sb.Append(HelperNamespace.HelperName(viewModel.Parameter))`

Blocks:
Following examples are more complex and show the code generated for blocks. Whitespace is ignored for simplicity.

Handlebars	Resulting C# code

{{#if Member}}
  {{Member}}
{{else}}
  No Member
{{/if}}
	if(IsTruthy(viewModel) && IsTruthy(viewModel.Member))
	{
	  sb.Append(WebUtility.HtmlEncode(viewModel.Member));
	} else
	{
	  sb.Append("No Member");
	}

{{#each List}}
  {{this}}
{{/each}}
	if(IsTruthy(viewModel) && IsTruthy(viewModel.List))
	{
	  foreach(var item0 in viewModel.List)
	  {
	  sb.Append(WebUtility.HtmlEncode(item0));
	  }
	}

{{#with Parent}}
  {{Child}}
  {{this}}
  {{../Member}}
{{/with}}
	if(IsTruthy(viewModel) && IsTruthy(viewModel.Parent))
	{
	  sb.Append(WebUtility.HtmlEncode(viewModel.Parent.Child));
	  sb.Append(WebUtility.HtmlEncode(viewModel.Parent));
	  sb.Append(WebUtility.HtmlEncode(viewModel.Member));
	}

5.4 Holding the Information - Handlebars Abstract Syntax Tree

At first the class hierarchy of the Handlebars-AST is discussed. Starting with the `HandlebarsTemplate`, a central class which holds every AST, and then following with an overview over the class hierarchy and a set of examples from it.

5.4.1 HandlebarsTemplate

`HandlebarsTemplates` are classes that store and manage Handlebars-AST. They come in four flavors: A `StaticHandlebarsTemplate` contains no Handlebars and is just a raw string. `HandlebarsTemplates` are compatible to handlebars.js while `LayoutedHandlebarsTemplates` and `HandlebarsLayouts` are constrained to this work and allow a `HandlebarsTemplate` to be embeded into another.
Every `HandlebarsTemplate` contains information about its name, namespace, AST and the errors, which occurred during the parsing process. Furthermore, every non-static `HandlebarsTemplate` holds information about its model.
The `LayoutedHandlebarsTemplate` additionally contains information about the layout it should be rendered in and the `HandlebarsLayout` contains the position at which its child template is rendered.
Every `HandlebarsTemplate` has also an accept method; the entry point for a visitor.

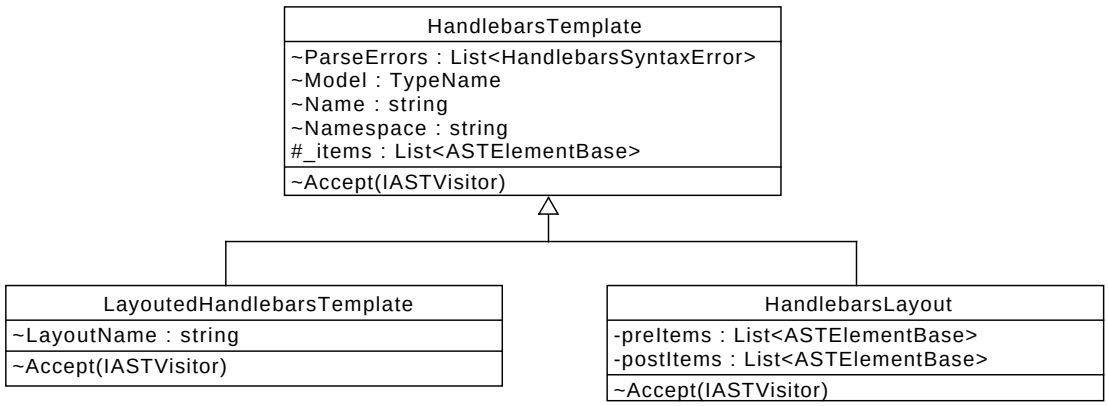


Figure 3: UML-Diagramm of `HandlebarsTemplate`, `LayoutedHandlebarsTemplate` and `HandlebarsLayout`

5.4.2 Abstract Syntax Tree Class Hierarchy

The parsing process of a handlebars-template returns a Handlebars-AST as an object structure. These objects are instances of classes which are shown in the class hierarchy diagram in figure 4.

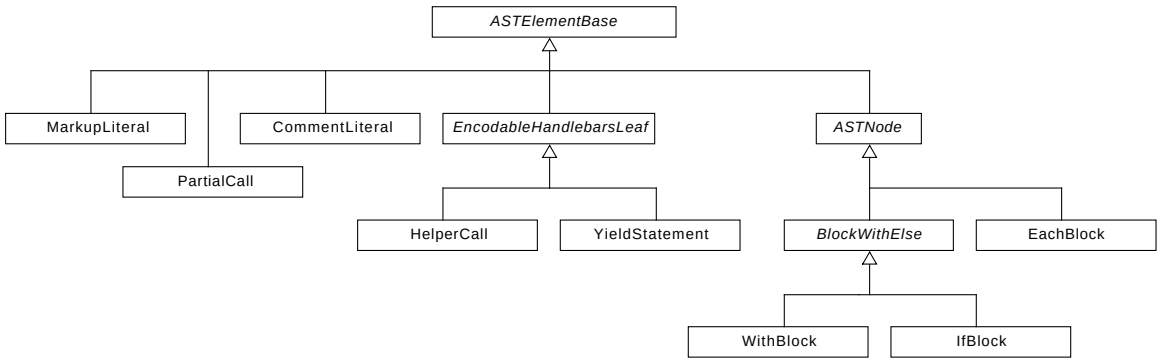


Figure 4: Diagram showing the class hierarchy of the Handlebars-AST

5.4.3 ASTElementBase

`ASTElementBase` is the base of every element in the Handlebars-AST. It contains information about the original position of the token in the Handlebars template, which is needed for error handling. Otherwise it defines two abstract methods that every AST-Element has to implement:

- Accept(IASTVisitor visitor): the Accept-Method as described in the visitor pattern [5]

- HasExpressionOnLoopLevel<T>(): is supposed to provide information about the type of expression used in a Handlebars-Token. This is needed to determine if `SpecialExpressions` are used inside loops.

```
internal abstract class ASTElementBase
{
  internal readonly int Line;
  internal readonly int Column;

  internal ASTElementBase(int line, int column)
  /*...*/

  internal abstract void Accept(IASTVisitor visitor);
  internal abstract bool HasExpressionOnLoopLevel<T>();
}
```

5.4.4 Nodes

Handlebars-templates can contain three kinds of nodes, as shown in section 4.2, with only their children as common information:

- `WithNode`
- `IfNode`
- `EachNode`

Their base class is therefore simple:

```
internal abstract class ASTNode : ASTElementBase
{
  protected readonly IEnumerable<ASTElementBase> _children;

  internal ASTNode(IEnumerable<ASTElementBase> children, int line, int column) :
      base(line, column)
  {
    _children = children;
  }
}
```

The `EachBlock` class will serve as concrete example for an `ASTNode`:

```
internal class EachBlock : ASTNode
{
  internal enum ForLoopFlags { None = 0, First = 1, Last = 2, Index = 4 }
  internal readonly MemberExpression Member;

  internal ForLoopFlags Flags { get {/*...*/} }
```

```
  internal EachBlock(MemberExpression member, IList<ASTElementBase> children, int
      line, int column) : base(children, line, column)
  /*...*/

  internal override void Accept(IASTVisitor visitor)
  /*...*/

  internal override bool HasExpressionOnLoopLevel<T>()
  /*...*/
}
```

Additional to the child elements, which are handled by the base class, the `EachBlock` has a `MemberExpression` and `ForLoopFlags`. The `MemberExpression` determines the member which is enumerated while the `ForLoopFlags` store information about `SpecialExpressions` (i.e. @index) which are used inside the loop.

5.4.5 Leafs

There are five kinds of AST-Leafs, elements of the AST that do not have children: `MarkupLiterals` represent the HTML that wraps around Handlebars-tokens in a template. The only information a `MarkupLiteral` object has is its position and its contents. `Comments` carry no semantic value, yet the contents of a `Comment`-object are inserted into the generated code as comment in order to simplify the debugging of a template. `PartialCalls` contain information about the partial templates name and namespace and the expression that will be passed to the partial at run-time. `YieldExpressions` facilitate the output of values to the render result. The expression it stores is evaluated and appended to the resulting string at run-time. `HelperCalls` point to functions inside the surrounding compilation and thus allow the user to invoke C# code from Handlebars-Templates.

5.4.6 Expressions

`Expressions` are classes that hold information about types and members, which come in different variants. Their abstract base class offers a method to evaluate an `Expression` using a `CompilationState` object:

```
internal abstract class Expression
{
  internal abstract bool TryEvaluate(CompilationState state, out Context context);
}
```

This methods out parameter context is a `Context`-object that represents the semantic value of the `Expression`. If the `Expression` could be evaluated this context has a symbol containing information about the evaluated type and a path how the expression can be called inside the rendered code. If the evaluation fails, errors are raised to inform the user about a type-error.

Covering all cases that involve members of the ViewModel, is the `MemberExpression` which itself holds an `IdentifierElement` to store information and handling the actual evaluation. Also, it has,

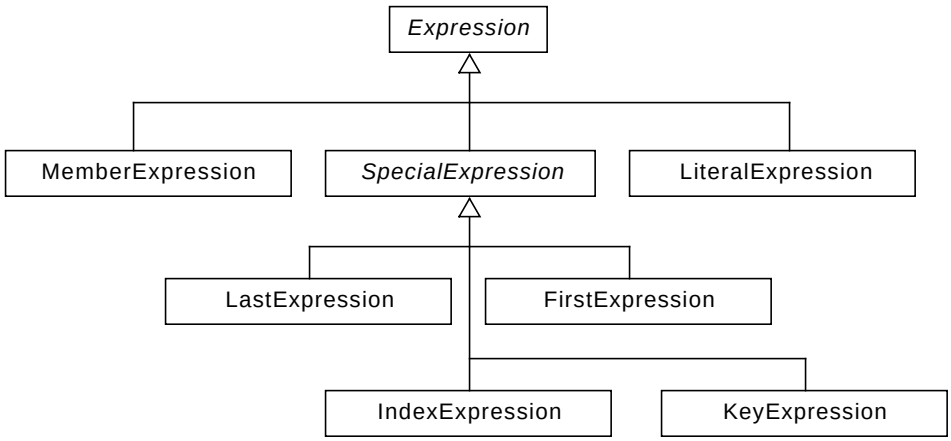


Figure 5: Overview of the Class Hierarchy of Expressions

additional to the `TryEvaluate` method an `EvaluateLoop` method that returns the `Context` of a loop over its expression:

```
internal class MemberExpression : Expression
{
  public readonly IdentifierElement Path;
  internal MemberExpression(IdentifierElement path)/*...*/

  internal override bool TryEvaluate(CompilationState state, out Context
      context)/*...*/

  internal Context EvaluateLoop(CompilationState state)/*...*/
}
```

For every special expression (e.g. `{{@first}}, {{@last}}, {{@index}}, {{@key}}`) exists a separate class. In their `TryEvaluate` methods they create a context by getting created symbols from a `RoslynIntrospector`-object.
The last variant is the `LiteralExpression` which holds information about a string-literal passed to a helper.

5.5 IASTVisitor and CodeGenerationVisitor

This section describes the `IASTVisitor` interface and its, currently only, implementation the `CodeGenerationVisitor`.

5.5.1 IASTVisitor

The `IASTVisitor` interface is derived from the hierarchical visitor described in [5]. It allows its implementations to traverse the Handlebars-AST. This traversion does not only include extra methods for entering and leaving a node, as defined in the hierarchical visitor pattern, but also methods for visiting Handlebars-token which are not represented by objects in the Handlebars-AST (e.g. else, body).

```
internal interface IASTVisitor
{
  void VisitEnter(EachBlock astNode);
  void VisitLeave(EachBlock astNode);

  void VisitEnter(WithBlock astNode);
  void VisitLeave(WithBlock astNode);
  void VisitElse(WithBlock astNode);

  void VisitEnter(IfBlock astNode);
  void VisitLeave(IfBlock astNode);
  void VisitElse(IfBlock astNode);

  void Visit(YieldStatement astLeaf);
  void Visit(MarkupLiteral astLeaf);
  void Visit(CommentLiteral astLeaf);
  void Visit(PartialCall astLeaf);
  void Visit(HelperCall astLeaf);

  void VisitEnter(HandlebarsTemplate template);
  void VisitLeave(HandlebarsTemplate template);

  void VisitEnter(LayoutedHandlebarsTemplate layoutedTemplate);
  void VisitLeave(LayoutedHandlebarsTemplate layoutedTemplate);

  void VisitRenderBody(HandlebarsLayout layout);
  void VisitLeave(HandlebarsLayout layout);

  void VisitLeave(StaticHandlebarsTemplate staticTemplate);
}
```

5.5.2 CodeGenerationVisitor

Implementing the `IASTVisitor` interface, the `CodeGenerationVisitor` is responsible for triggering the semantic analysis and generating the resulting code. Every `CodeGenerationVisitor` object has its own `CompilationState` object, which is created in its constructor. The Visit-Methods for `HandlebarsLeafs` follow a certain procedure:

1. Set the cursor of the `CompilationState` to the visited AST-Element
2. Trigger the semantic analysis and retrieve its result

3. If the semantic analysis succeeds: Append the appropriate code

As an example the Visit-Method for a `YieldStatement` will serve. This method handles unencoded as well as encoded `YieldStatements`:

```
1  public void Visit(YieldStatement astLeaf)
2  {
3    state.SetCursor(astLeaf);
4    Context yieldContext;
5    if (astLeaf.Expr.TryEvaluate(state, out yieldContext))
6    {
7      state.PushStatement(SyntaxHelper.AppendMember(
8        yieldContext.FullPath,
9        yieldContext.Symbol.IsString(),
10       astLeaf.Type == TokenType.Encoded)
11     );
12   } else
13   {
14     //Unknown Member could also be a HelperCall with implied "this" as Parameter
15     if (astLeaf.Expr is MemberExpression)
16       astLeaf.TransformToHelperCall().Accept(this);
17   }
18 }
```

In line 3 the curser of the `CompilationState` object is set to the visited leaf. In line 5 the semantic analysis is triggered. Based upon its success, either the resulting code is pushed to the result stack (line 7 to 11) or the `YieldStatement` is transformed into an `HelperCall` object and the Visitor is forwarded to it (line 15 and 16).

For `HandlebarsNodes`, additionally following tasks have to be executed:

- Push and pop a new block to the result stack
- Push and pop the appropriate context to the context stack
- Push and pop the appropriate context to the truthyness stack

Serving as an example are the three Visit-Methods for a `WithNode`: `VisitEnter`, `VisitLeave` and `VisitElse`:

```
1  public void VisitEnter(WithBlock astNode)
2  {
3    state.SetCursor(astNode);
4    state.PushNewBlock();
5    Context context;
6    if (astNode.Expr.TryEvaluate(state, out context))
7    {
8      state.PromiseTruthyCheck(context);
9      state.ContextStack.Push(context);
```

```
10    }
11  }
```

Already known from above example are the call to `SetCursor` and `TryEvaluate` in line 3 and 7. The result-stack is pushed to in line 4 while the context and truthyness stack are pushed to in line 8 and 9.

Exactly the opposite is done in the `VisitLeave`-method:

```
1  public void VisitLeave(WithBlock astNode)
2  {
3    state.ContextStack.Pop();
4    var latestBlock = state.PopBlock();
5    if (astNode.HasElseBlock)
6      state.DoTruthyCheck(state.PopBlock(), latestBlock, IfType.If);
7    else
8      state.DoTruthyCheck(latestBlock, ifType: IfType.If);
9  }
```

In line 3 and 4 the context and result stacks are popped. Depending on the existence of an `ElseBlock`, `DoTruthyCheck` is called with either one or two blocks of `SyntaxStatements`; the first for the if-part, the second for the else part. Inside the `DoTruthyCheck`-method the truthyness stack is popped and the actual statement for performing the truthyness check is constructed. See section 5.6 for details.

The `VisitElse`-method now is obvious:

```
1  public void VisitElse(WithBlock astNode)
2  {
3    state.ContextStack.Pop();
4    var truthyContext = state.TruthyStack.Pop();
5    truthyContext.Truthy = !truthyContext.Truthy;
6    state.TruthyStack.Push(truthyContext);
7    state.PushNewBlock();
8  }
```

A new block is pushed (the else block) in line 7, the context stack is popped in line 3 and current truthyness of the current context is reversed in line 4 to 6.

5.6 CompilationState

The concepts behind the `CompilationState` have already been explained in section 4.3. It keeps track of lots of data accumulated during the compilation process. This section discusses the implementation details of this central class essential for understanding the compilation process as a whole.

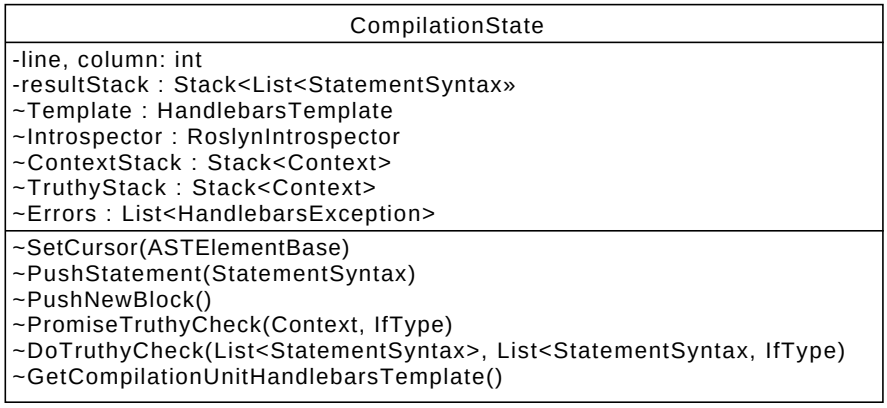


Figure 6: Class-Diagramm of the `CompilationState` class with a selection of members and methods that support understanding of its purpose and use

While some methods of the `CompilationState` are trivial (e.g. `SetCursor`, `PushStatement`, `PushNewBlock` and `PromiseTruthyCheck`) and merely serve as abstraction layers for conveniently interacting with members of the `CompilationState`, others contain complexity and logic and are thus relevant for understanding. These methods are now discussed briefly.

`PromiseTruthyCheck` and `DoTruthyCheck` work as a pair, while `PromiseTruthyCheck` pushes a context to the truthyness-stack, `DoTruthyCheck` pops this context and constructs an If-statement for it. This If-statement contains as few checks as possible. Imagine a context with a path of "ViewModel.Member" lying on the truthyness stack. Naively, the correct If-statement would have check both "ViewModel" and "Member" for truthyness, but as information about truthyness is stored, DoTruthyCheck checks if "ViewModel" has already be checked for thruthyness and omits the check in that case. This case occurs for example in following handlebars-template:

```
{{model System.String}}
{{#if this}}
  {{#if Length}}
    The Input has a Length > 0
  {{/if}}
{{/if}}
```

Another central method that resides inside the `CompilationState` is `GetCompilationUnitHandlebarsTemplate`. This method embeds the generated `SyntaxStatements` into an C#-AST that represents a complete and correct C#-document: it creates the declarations for containing namespace, class and method.

5.7 SyntaxHelper and Roslyn's C#-Abstract-Syntax-Tree

As discussed in section 3.2.3 Roslyn will be used to generate C# code. Specifically a C#-AST is created from Roslyn's AST-classes. These AST classes follow the concept of immutability: a change to an object does not change the object, but yields a new copy of the old object with changes applied. With the static `SyntaxFactory` class Roslyn offers the ability to create these objects. Nevertheless, the overhead to create them turns out to be inconvenient and is thus outsourced in another, project-specific, static class: the `SyntaxHelper`. Both `CodeGenerationVisitor` and `CompilationState` can now call the methods inside the `SyntaxHelper`, only needing to pass a minimum of parameters.
As an example the method that generates the class declaration is shown. It only takes two inputs: the class name and its attribute. "SyntacFactory" is aliased to "SF" for convenience in this example and in the whole `SyntaxHelper` class.

```
1  internal static ClassDeclarationSyntax CompiledHandlebarsClassDeclaration(string
       templateName, string attribute)
2  {
3    return
4      SF.ClassDeclaration(
5        new SyntaxList<AttributeListSyntax>().Add(
6        SF.AttributeList(new SeparatedSyntaxList<AttributeSyntax>().Add(
7          SF.Attribute(SF.ParseName(attribute)))
8        )),
9        SF.TokenList(
10         SF.Token(SyntaxKind.PublicKeyword),
11         SF.Token(SyntaxKind.StaticKeyword)),
12       SF.Identifier(templateName),
13       default(TypeParameterListSyntax),
14       default(BaseListSyntax),
15       default(SyntaxList<TypeParameterConstraintClauseSyntax>),
16       default(SyntaxList<MemberDeclarationSyntax>)
17     );
18 }
```

The attribute of the class is set in line five to eight, it is declared as public and static in line ten and eleven. Finally, in line twelve the name of the class is set.

5.8 RoslynIntrospector

Retrieving information about the typesystem that surrounds the Handlebars-template to be compiled, the `RoslynIntrospector` is created in the constructor of a `CompilationState`. This information can be extracted from Compilation-objects provided by Roslyn. As the creation of these Compilation-objects is, especially for large projects, time consuming and they might not change between two invocations of the compiler, they are stored as static variables. Yet it is important to keep these objects synchronized as the type system can change significantly between two invocations, resulting in different results. As Roslyn offers the functionality to check for changes between two solutions, the check if the loaded solution is up-to-date is trivial. Mainly the `RoslynIntrospector` encapsulates complexity for resolving partial template and helper-calls and the lookup for members of the ViewModel. It can also generate

symbols for variables that only appear at run-time. This is especially helpful for SpecialExpressions, as they can then be treated the same as every other Expression. One example method that is important for finding partial templates and layouts is `FindClassesWithNameAndAttribute`. It will serve as example and shows how powerful Roslyn is.

```
1  private INamedTypeSymbol FindClassesWithNameAndAttribute(string name, string
       attribute)
2  {
3    foreach (var comp in projectCompilations.Values)
4    {
5      INamedTypeSymbol template =
6          comp.GetSymbolsWithName(x => x.Equals(name))
7              .OfType<INamedTypeSymbol>()
8              .FirstOrDefault(x => x.GetAttributes()
9                                .Any(y => y.AttributeClass.Name.Equals(attribute)));
10     if (template != null)
11       return template;
12   }
13   return null;
14 }
```

The foreach-loop walks through all Compilation-objects that are referenced in the project which contains the handlebars-template that is compiled. The first symbol that is found that has the correct name (`GetSymbolsWithName`), have the correct type (`OfType`) and have the attribute provided as parameter is returned. The simplicity of the statement in line 6 to 9 astonishes and shows that working with type-systems can be as simple as working with any other data-structure.

6 Evaluation

This chapter will evaluate the results of this project by validating compatibility and functionality of the compiler with UnitTests in section 6.2 and comparing the performance against the reference implementation in section 6.3.

6.1 Test Infrastructure

The machine we used to run the following tests and benchmarks is a dedicated machine. It has two Intel Xeon X5650 processors with a total of 12 cores clocking at up to 3.0Ghz. The processors L1 data and instruction cache have each 32Kb per core and 256Kb L2 and 12288Kb L3 cache. It has 24GB of ECC-RAM in 6 modules clocked to 1333 MHz and runs on Windows 8.1 Enterprise Edition. For the benchmark Node.js was used in version v4.2.2, Handlebars.js in version v4.0.5 and Benchmark.js version v1.0.0.

6.2 Validation

To be able to validate compatibility to the reference implementation and functionality UnitTests where implemented using the MSTest-Framework. How to implement UnitTests for code generators and the results of these tests are topic of this section.

6.2.1 Testing Generated Code

As the result of the compilation process is just a string, containing the generated code, it is not trivial to test that code. Roslyn offers the capability to compile code at run-time into an assembly. These assemblies can then be reflected upon and thus it is possible to invoke methods inside this generated assembly.
This process if far from performant; triggering the compilation process can take up to several hundred milliseconds. To allow acceptable test-run times, all handlebars-templates for one TestClass are compiled into one assembly and then saved as static variable for that TestClass.
Handlebars-Templates are registered for compilation using attributes that are evaluated in the static constructor of each TestClass using reflection. Also helper methods exist that allow the convenient test of a templates result with given input. An actual UnitTest-case can thus be easily and effectively implemented:

```
[TestMethod()]
[RegisterHandlebarsTemplate("BasicTest", "{{Name}}", _marsModel)]
public void BasicTest()
{
  ShouldRender("BasicTest", MarsModelFactory.CreateFullMarsModel(), "Mars");
}
```

6.2.2 Handlebars.js UnitTests

As already described in section 3.4, some of the UnitTests of the reference implementation were copied into this work to ensure compatibility and robustness. It is not possible to transfer all UnitTests, as the JavaScript implementation has niches and specialties that do not make sense or have no equivalent in C#. The UnitTests that were transferred though show that the results of both implementations are the same at least for these tests.

BasicPartials	✓
Comments	✓
CompileWithUndefinedContext	✓
CompilingWithBasicContext	✓
CompilingWithStringContext	✓
Each	✓
EachDataPassedToHelpers	✓
EachObjectWithFirst	✓
EachObjectWithIndex	✓
EachObjectWithLast	✓
EachWithFirst	✓
EachWithIndex	✓
EachWithLast	✓
EachWithNestedFirst	✓
EachWithNestedIndex	✓
EachWithNestedLast	✓
Escaping	✓
If	✓
MostBasic	✓
PartialInAPartial	✓
PartialPrecedingASelector	✓
PartialsWithContext	✓
PartialsWithSlashAndPointPaths	✓
PartialsWithSlashPaths	✓
PartialWithStringContext	✓
With	✓
WithWithElse	✓
Zeros	✓

6.2.3 Project-Specific UnitTests

As UnitTests from the handlebars.js project cannot cover features and technical details specific to this project, some more UnitTests were written in order to ensure robustness also for these parts of the software. This includes for example error handling at design-time (e.g. type errors, syntax errors) and resolving of partial templates or helper calls.

BaseAsParameterTest	✓
BasicHelperTest	✓
BasicLayoutTest	✓
BasicPartialTest	✓
BasicTest	✓
CaretIsElseTest	✓
CommentTest	✓
ContextErrorTest	✓
EachOverObjectTest	✓
EachTest	✓
EmptyListsAreFalsyTest	✓
EmptyTemplateTest	✓
FirstTest	✓
HtmlEncodeTest	✓
IfElseTest	✓
IfTest	✓
ImplicitThisParameterTest	✓
ImpliedThisParameterTest	✓
IndexTest	✓
InterfaceAsParameterTest	✓
IterateOverNullTest	✓
LastTest	✓
LayoutsWithInterfaceModelTest	✓
LiteralParameterTest	✓
MalformedMemberExpressionTest	✓
MalformedModelTest	✓
MalformedPartialCallTest	✓
MalformedWithBlockTest	✓
MultipleParametersHelperTest	✓
NamingConflictsInTruthynessQueriesTest	✓
NestedEachLoopsTest	✓
NestedIfTest	✓
NullParameterPartialTest	✓
PathTest	✓
RedundantTruthynessCheckTest	✓
RootTest	✓
SelfReferencingPartialTest	✓
StaticHandlebarsTemplateTest	✓
ThisTest	✓
UnencodedResultTest	✓
UnexpectedCharacterTest	✓
UnknownMemberTest	✓
UnknownViewModelTest	✓

UnknownPartialTest	✓
UnknownSpecialExpressionTest	✓
UnlessElseTest	✓
UnlessTest	✓
UnreachableCodeTest	✓
WhitespaceControlTest	✓
WithTest	✓

6.3 Performance Benchmark

The handlebars.js project uses microbenchmarks to compare against other JavaScript rendering libraries which are run on every continuous integration build. To compare the performance of this work agains the reference implementation in JavaScript, these microbenchmarks were copied to C# where possible. The handlebars.js version was benchmarked using Node.js (v.4.2.2) with the Benchmark.js library [3] (v.1.0.0), as this benchmark library is also used by the handlbars.js project. Benchmark.js measures the time it takes to execute a function and repeats this experiment until it has statistically significant results. For these measurement the sample size was around 100 samples.
The C# version was benchmarked using the Stopwatch-Class and measuring the time it takes to execute the Render method of a compiled template 100,000 times, averaged through 100 samples. During these benchmarks the CPU scaled to 3.0 GHz and the standard deviation of all used measurement series was below 1%. The results of this benchmark show the performance gain this project aimed for:

Microbenchmark	handlebars.js	CompiledHandlebars	speedup	two sample t-test p-value
arrayeach	391 ops/ms	3039 ops/ms	7.77	1.928×10^{-98}
complex	120 ops/ms	1180 ops/ms	9.83	1.615×10^{-97}
data	295 ops/ms	1333 ops/ms	4.51	5.532×10^{-98}
depth1	228 ops/ms	3693 ops/ms	16.20	3.885×10^{-99}
depth2	63 ops/ms	1515 ops/ms	24.04	2.231×10^{-100}
partial-recursion	125 ops/ms	1895 ops/ms	15.16	3.866×10^{-99}
partial	211 ops/ms	905 ops/ms	4.29	1.759×10^{-96}
paths	2060 ops/ms	4646 ops/ms	2.25	1.959×10^{-90}
string	5563 ops/ms	13964 ops/ms	2.51	7.549×10^{-93}
variables	1991 ops/ms	4027 ops/ms	2.02	2.669×10^{-88}

Table 3: Table showing the results of each microbenchmark for handlebars.js and CompiledHandlebars in operations in milliseconds. Additionaly, the p-value of a two sample t-test with a 99% confidence level is provided.

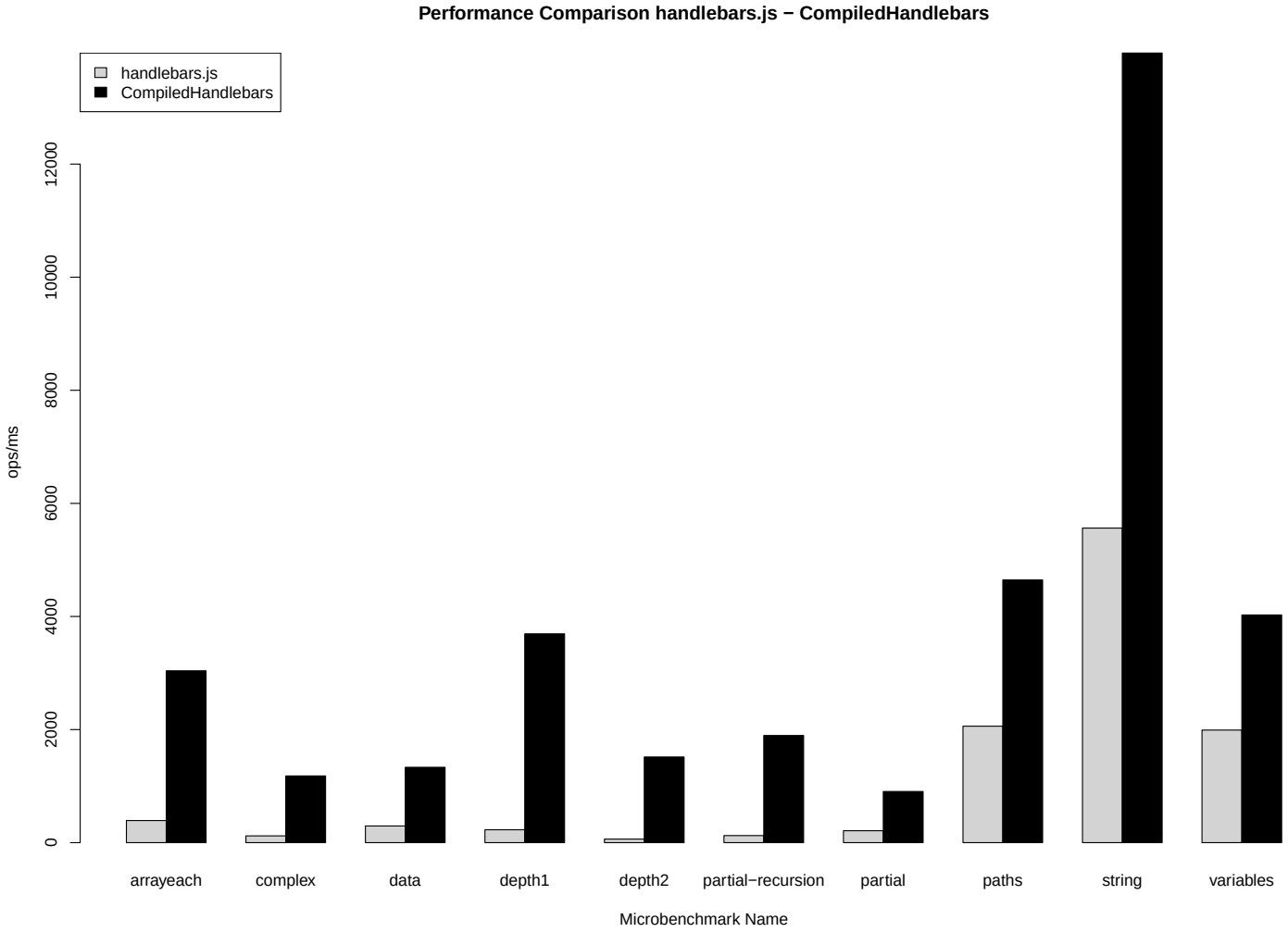


Figure 7: Graph showing the throughput in operations per millisecond for each microbenchmark

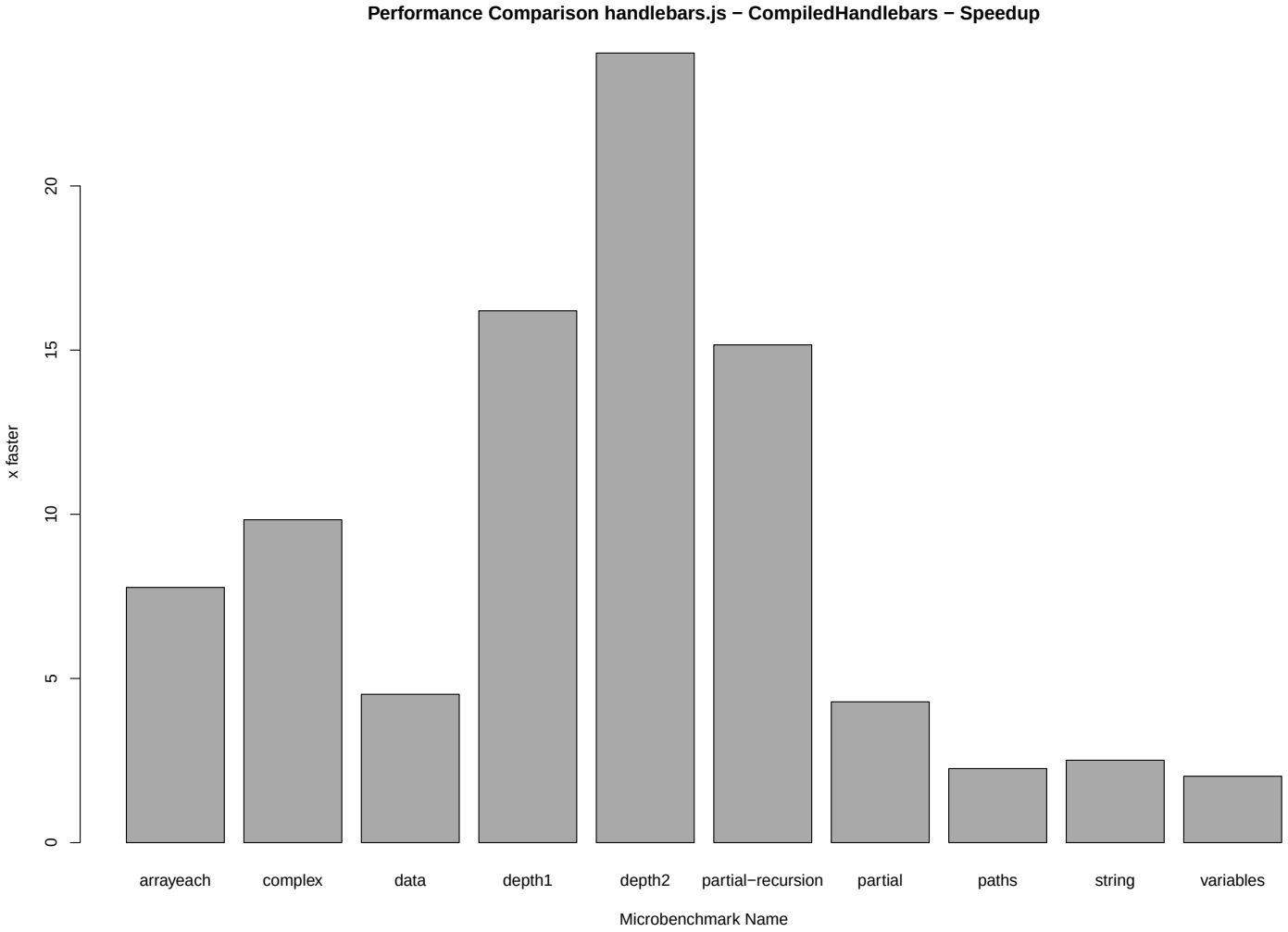


Figure 8: Graph showing the speedup from handlebars.js vs CompiledHandlebars for each microbenchmark

The results show a significant performance increase for compiling handlebars to native C# code compared to the JavaScript based reference implementation of Handlebars. For faster microbenchmarks, when the render times for the handlebars.js implementation are several hundret nanoseconds, the speedup factor is more around two. Nevertheless, for handlebars templates that perform more complex and time consuming tasks and take handlebars.js to render more than one μs , the C# alternative is at least four and up to 24 times faster, as the potential for improvement grows.
This performance gain is possible because some of the necessary computation is done at compile-time: both the C# and the JavaScript implementation have to inspect and analyze the passed ViewModel, but, as C# is statically typed and the type of the ViewModel is known before run-time, this analysis and inspection does not entail any performance costs for run-time if it is done at compile or design-time. Additionally, as C# code is generated at design-time, this generated code also leverages the optimizations produced by the C# compiler.

7 Conclusion

In this thesis, we created a compiler from Handlebars to C# code. Compared to the existing, JavaScript based, reference implementation of Handlebars, our approach enables fast and type-safe logicless views. We consider this a major improvement for web development, as it allows further decoupling of HTML and code compared to a traditional MVC-architecture.

This work discussed possible approaches to the compilation process and design decisions including used tools or project architecture. Also implementation details of the resulting compiler were described. Finally, we present an experimental validation of our implementation and additionally show an evaluation of its performance.

Compared to the Handlebars reference implementation, our software provides two major advantages: firstly, it improves the user experience by reporting type errors at compile- and not at run-time. Secondly, the benchmark experiments showed an eminent performance increase. For fast microbenchmarks with render times in handlebars.js of several hundred nanoseconds, the speedup was around two. For slower and more complex handlebars-templates however, the speedup was between two and 24. This can be attributed to two basic advantages of generating code at compile- or design-time against interpreting a handlebars template at run-time:

1. Calculations on the typesystem can be done at compile-time and have zero cost at run-time
2. Compiler optimizations can be leveraged on generated code

These advantages are not solely true for the particular problem of rendering handlebars-templates, but rather apply to a whole set of other problems, like serialization, logging, factories and many more. For a more thorough analysis of the capabilities and use-cases of compile-time metaprogramming see the papers of the C++ committee reflection subgroup, especially [4].
Additionally, tools that allow interaction with type-systems, like Roslyn, become more sophisticated and more easy to use. Some languages even provide static reflection capabilities. Furthermore, integration of tools that generate code into an existing workflow shows to be feasible. Therefore, we come to the conclusion that the compile-time generation of code should grow in importance as its advantages are most significant.

Concerning future work, we recommend three possible directions: speedup, usability and further comparisons. Despite good performance results, we expect potential for further optimizations, as the current implementation was mainly feature driven. Estimating resulting string length, unrolling for-loops and rendering partial templates in parallel are just some ideas that could be implemented. Besides performance, usability and integration into existing frameworks and tools are important features. Implementing a ViewEngine for the ASP.NET MVC framework or an OWIN-middlewear, much like they exist for RazorViews, would further enhance these traits. Finally, to compare performance against other server-side rendering technologies a benchmark could be designed and run against RazorViews, for example.

Acronyms

MVC Model-View-Controller

PEG parsing expression grammar

AST abstract syntax tree

DIP dependency inversion principle

CFG context free grammar

References

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science and information processing. Addison-Wesley, 2006.

[3] BenchmarkJS. benchmark.js website. `http://benchmarkjs.com/`, 2015. [Online; accessed 7-December-2015].

[4] Matuš Chochlik. A case for strong static reflection. *Programming Language C++, SG7, Reflection*, April 2015.

[5] Cunningham & Cunningham Inc. Hierarchical visitor pattern. `http://c2.com/cgi/wiki?HierarchicalVisitorPattern`, 2011. [Online; accessed 27-September-2015].

[6] Bryan Ford. Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking. Master's thesis, Massachusetts Institute of Technology, 2002.

[7] Github. Compiledhandlebars for c# repository. `https://github.com/Noxum/CompiledHandlebars`, 2015. [Online; accessed 10-December-2015].

[8] Github. Handlebars homepage. `http://handlebarsjs.com/`, 2015. [Online; accessed 14-September-2015].

[9] Github. Handlebars specification on github. `https://github.com/wycats/handlebars.js/tree/master/spec`, 2015. [Online; accessed 9-September-2015].

[10] Github. Handlebars.net. `https://github.com/james-andrewsmith/handlebars-net`, 2015. [Online; accessed 22-September-2015].

[11] Github. Handlebars.net. `https://github.com/rexm/Handlebars.Net`, 2015. [Online; accessed 22-September-2015].

[12] Github. Ironmeta on github. `https://github.com/kulibali/ironmeta`, 2015. [Online; accessed 7-September-2015].

[13] Github. Mustache#. `https://github.com/jehugaleahsa/mustache-sharp`, 2015. [Online; accessed 22-September-2015].

[14] Github. Pegasus on github. `https://github.com/otac0n/Pegasus`, 2015. [Online; accessed 7-September-2015].

[15] Github. Razorgenerator. `https://github.com/RazorGenerator/RazorGenerator`, 2015. [Online; accessed 22-September-2015].

[16] Github. Roslyn github repository. `https://github.com/dotnet/roslyn`, 2015. [Online; accessed 25-November-2015].

[17] Robert Martin. The dependency inversion principle. *C++ Report*, May 1996.

[18] Robert Martin. *Agile principles, patterns, and practices in C*. Prentice Hall, Upper Saddle River, NJ, 2007.

[19] MSDN. Codedom documentation. `https://msdn.microsoft.com/en-us/library/system.codedom%28v=vs.110%29.aspx`, 2015. [Online; accessed 14-September-2015].

[20] MSDN. Envdte documentation. `https://msdn.microsoft.com/en-us/library/envdte.aspx`, 2015. [Online; accessed 14-September-2015].

[21] Joline Morrison Nick Heidke and Mike Morrison. Assessing the effectiveness of the model view controller architecture for creating web applications. 2009.

[22] Peter Pirkelbauer, Yuriy Solodkyy, and Bjarne Stroustrup. Open multi-methods for c++. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, pages 123–134, New York, NY, USA, 2007. ACM.

[23] Christopher Graham Seaton. *A Programming Language Where the Syntax and Semantics Are Mutable at Runtime*. PhD thesis, University of Bristol, 2007.

[24] Yuriy Solodkyy. Accept no visitors. `https://github.com/CppCon/CppCon2014/tree/master/Presentations/Accept%20No%20Visitors`, 2014. CppCon 2014 [Online; accessed 15-September-2015].

[25] Bjarne Stroustrup Yuriy Solodkyy, Gabriel Dos Reis. Open pattern matching for c++. In *12th international Conference on Generative programming: Concepts & Experiences*.