Chair of
Software Engineering

Towards a Web Framework for an Automated Eiffel Code Teaching Assistant

Bachelor's Thesis

Christian Vonrüti
ETH Zurich
cvonruet@student.ethz.ch
11-930-914

January 1, 2015 - July 1, 2015

Supervised by:
Marco Piccioni
Prof. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Abstract

This thesis explores how to implement a web-based interactive teaching tool for the Eiffel programming language. The foundation for an implementation has been laid, in particular with parsing, AST construction and a partial semantic analysis. This implementation should serve as a platform upon which to build a teaching tool that can autonomously explain a program to a student, and guide them through the execution by breaking down the steps.

Acknowledgments

I would like to thank my supervisor Marco Piccioni for his advice, ideas his patience and for allowing me to pursue this idea as a thesis.

Further, I would like to thank Emmanuel Stapf, the lead engineer for the *Eiffel* language for the insights he provided into the internals of *Eiffel*.

Special thanks go to Alexander Viand and Lukas Häfliger for proof reading and advice with the report.

Contents

Chapter 1

Introduction

1.1 Motivation

During my time as a teaching assistant at ETH for the course *Introduction to Programming* I was teaching exercise sessions for a beginner's group. The programming language of choice in the course was *Eiffel*, invented by the lecturer himself, Prof. Dr. Bertrand Meyer.

Most of my students had never programmed before. I saw them develop an understanding and oftentimes misunderstanding of Eiffel which seemed to be rooted in intuitions they formed about certain program constructs based on example programs. When tasked to write their own programs as part of exercise sessions or just weekly assignments, they would then, at least that is my interpretation, extrapolate on their intuitions.

This would lead them to invent constructs and syntax that *Eiffel*, much to their dismay, would then reject. If *Eiffel* did in fact accept their program, it would often do something completely different from what they expected.

Prof. Dr. Bertrand Meyer, who is teaching *Introduction to Programming*, opens up his course by saying "The good news: Your computer will do exactly what your program says" only to follow up shortly after with "The bad news: Your computer will do exactly what your program says". As already alluded to, my students became very aware that *what the program says* and *what they think it says* do not necessarily coincide.

All of this combined seemed to make the students regard *Eiffel* as a magical black box, rather than a mechanical one following very precise rules. I found this belief to be confirmed further when I noticed that students' understanding would improve, when explaining certain lines of code thoroughly and mechanically, by explicitly mentioning every step, and its substeps recursively. However, explaining in such detail is a quite tedious and time consuming process. It is not well suited as an approach to use for all program explanations during exercise sessions.

Of course, this kind of explanation lends itself to automation since the steps themselves are all mechanical in nature. It's not dependent on human interpretation. After all, a program needs to be able to deterministically process the program text in the end. An automated tool would also allow a student to inspect every construct in every program they came across in detail, with just as much depth and time as needed.

And thus the idea to create a tool that could be used to aid students in understanding Eiffel programs, and the precise rules that govern it, was born.

1.2 Initial Considerations & Goals

Together with my supervisor Marco Piccioni we decided on the primary requirements for the tool.

One of the first concerns was that of software installation. Experience had shown that installing the Eiffel software proved troublesome for many students. As such, it was desirable for the tool to require only minimal installation effort or, better yet, no installation at all. As a result, the web was chosen to serve as a platform upon which to build the application. Since web browsers are ubiquitous, this allows for the application to be used without any complications or custom software installations.

It was decided, mostly to keep the scope of the project in check, that the application would only need to support a very limited subset of the *Eiffel* language, such that more time could be spent on automated assistants for the students and visualisations. We also recognized, that we would need to provide good error messages.

In terms of functionality, the following features were planned, in order:

Parser to process a subset of the Eiffel language. Primarily, the omission of generics, multiple inheritance, agents and tuples.

Static analysis Provide feedback to the student without running the program, such as duplicate class names, type errors and unknown features.

Interpreter with stepping Being able to execute a program within the browser, step by step for simple debugging.

Basic type emulation of the most fundamental types, integers, booleans and strings.

Stack visualization to show how feature calls and argument passing work, in particular to help the understanding of recursion and local variables.

Identifier lookup visualization to explain to a student how *Eiffel* determines, what an identifier within the program refers to.

Object graph Display how different objects are related by storing references in their attributes and how the dot-operator allows “jumping” from object to object.

AST expression visualization especially to show the structure of nested binary expressions and their evaluation semantics.

Most of these ideas were drawn from either my supervisor Marco Piccioni's or my own personal experience while assisting students.

Other topics such as multiple inheritance were reserved for extra achievements if time allowed.

1.3 Achieved Work

We had set out with multiple ideas for an automated *Eiffel* teaching assistant and had come to the conclusion, that this is best done in a fully client-side web application due to ease of use and performance.

Our initial goal of creating an automated teaching assistant that only supported a small subset of the language was soon found to be unsuitable for a teaching tool.

- We believe, that experimentation lies at the foundation of human learning. A student experimenting with Eiffel could very quickly overstep the limitations of the initially targeted *Eiffel* subset. As such, a tool that does not allow for experimentation would have been a poor teaching tool.
- A distinguishing feature of *Eiffel* is, that object orientation is applied throughout the entire language. Unlike in other popular languages such as Java, even seemingly simple features such as integers are objects with methods (features) attached to them. Further, the standard library makes heavy use of multiple inheritance which has highly complex semantics for conflict resolution.
- Approaches have been considered how programs that make use of these features could be executed inside the teaching assistant without actually supporting these language features, e.g. through some special cases in the implementation, or other approximative measures.

  During the thesis it became apparent, that these shortcuts themselves would be quite complicated in themselves and easily be the source of subtle deviations in semantics from an actual *Eiffel* compiler that supports these language features natively. Additionally, all the special cases for different features from different classes would need to be implemented separately, which in itself would be a large effort. This time could be better spent in actually implementing the underlying language features, which would in turn allow the use of other aspects of *Eiffel* without any need for additional special cases.

We therefore believe, that only a fully featured *Eiffel* compiler could serve as the foundation of a useful teaching tool. The focus of the thesis has been shifted to accommodate for this new insight.

However, a fully featured *Eiffel* compiler is much beyond the scope of a Bachelor's Thesis, as the language has many advanced and complex mechanisms that are heavily intertwined. Therefore, the planned didactic tools have been were relegated for future work in favor of work on the compiler. A virtually feature complete parser has been implemented along with a basic semantic analyzer. Additonally, there is a sample web application that shows how these components are connected and can be integrated in a website.

1.4 Contribution

The primary insight of this thesis is, that an *Eiffel* teaching tool should be built upon a fully featured *Eiffel* compiler. Further, that a client-side web application is an ideal platform for such an endeavour. Approaches have been evaluated in how *Eiffel* could be brought into the browser, with the result that an *Eiffel* compiler should be custom built with web technologies. This allows the compiler's metadata to be tailored to didactic needs, as well as effortless direct interaction with the web application. Further, the first steps to implement such a web application have been made.

The vibrancy of the web development community and ecosystem provide excellent stepping stones for further work. Knowledge about the involved technologies are widespread and well documented, the underlying technologies easy to understand as well as easy to debug, allowing this project to be extended easily.

1.5 Outline

In chapter 2, gives an overview into the major decisions that have been made during the duration of the process. Chapter 3 showcases some of the encountered problems and how they were solved. The different parts of the implementation are explained in chapter 4. Chapter 5 concludes the thesis with experience gathered during this thesis and future work.

Chapter 2

Process

This section outlines the major decisions made throughout the project that shaped its direction, and determined the technologies upon which and with which the application has been built.

Client vs Server. The possible implementations for the planned step-by-step program execution can be divided into two categories: client-side execution and server-side execution. A server side approach offers tremendous benefits from the developer's perspective: there are existing *Eiffel* implementations that could be used to parse, validate, run and debug programs on the server. However, this would make the application dependent on a possibly hefty server infrastructure and would degrade the user experience noticeably. Every action when debugging and compiling could possibly, and most would, involve a request to the server, processing and waiting for the reply. As such, the user experience would most likely have felt very sluggish.

Ignoring the server infrastructure, sandboxing concerns, and communication between a debugger run server-side and the client, I ruled this choice out primarily from a usability perspective, especially considering that we wanted to provide more detailed error messages and better program explanations.

The approach I chose was to implement all functionality client side. This implies that parsing, static analysis, and interpretation/execution would all have to be done inside the browser, i.e. in *JavaScript*.

Even with that choice, I was considering options that would have allowed me to avoid writing a parser and semantic analyzer myself, but rather leverage existing projects. One project, that I was aware of, which would have allowed to use the actual *Eiffel* compiler was *emscripten*.

emscripten[1] is a way to compile languages such as C or C++ into JavaScript. To be more precise, it takes *LLVM* bytecode and compiles that into *asm.js*[2], which is a subset of JavaScript, which can be highly optimized by browsers. Despite finding earlier thoughts on the idea of using emscripten to run *Eiffel*

[1] http://kripken.github.io/emscripten-site/
[2] http://asmjs.org/

programs in the browser[3], I decided that this approach contained too many unknowns and seemed to be generally infeasible: If I succeeded in getting programs to run through emscripten inside the browser, I would need to somehow be able to read debug output either directly from the runtime representation that emscripten generated, or, have *Eiffel*'s own debugger itself be run, again with the help of emscripten, inside the browser and run the *Eiffel* program through that, which would again incur the same problems already discussed in the server-side approach. Then of course there was the question as to how to translate *Eiffel* code, in which the *Eiffel* compiler code is written, to *LLVM* bytecode. In the previously mentioned discussion that I found about *Eiffel* and emscripten, there was also a link[4] to a source code repository containing what appeared to be a solution to that problem, scarcely documented however. Fortunately though, the *Eiffel* compiler can also output C code, which could then be run through clang, a C/C++ frontend for *LLVM* and could therefore be used in combination with emscripten to go from *Eiffel* source code to actual JavaScript code that would execute in the browser. However, this would only work for the compiler itself. The goal for the project, however, is for the student to provide his own *Eiffel* source code and run that code. This would imply, that the very same process would need to happen inside the student's browser. As such, clang and emscripten themselves would need to be run inside the browser. Luckily, this had already been done[5,6]. In the end, however, I decided against this approach because I was unsure whether I could get this process to run at all, considering that emscripten itself has certain limitations and all the unknowns on how to string all the components together if I could indeed get them all to run in the browser. Further, this approach could have cost a lot of time solely in experimentation without any clear results or means of assessing progress.

This left me with the approach that I took in this thesis: Implementing everything in JavaScript. A benefit of this approach is, that all the gathered metadata can be exposed directly to the application such that didactic components can access them. Also, more metadata can easily be added to the compiler if needed for additional didactic features. With the other considered approaches, the addition of more metadata, as well as all the current metadata, would also need to be exposed to the web application code. This would most likely have needed a lot more additional work with many changes to the original compiler.

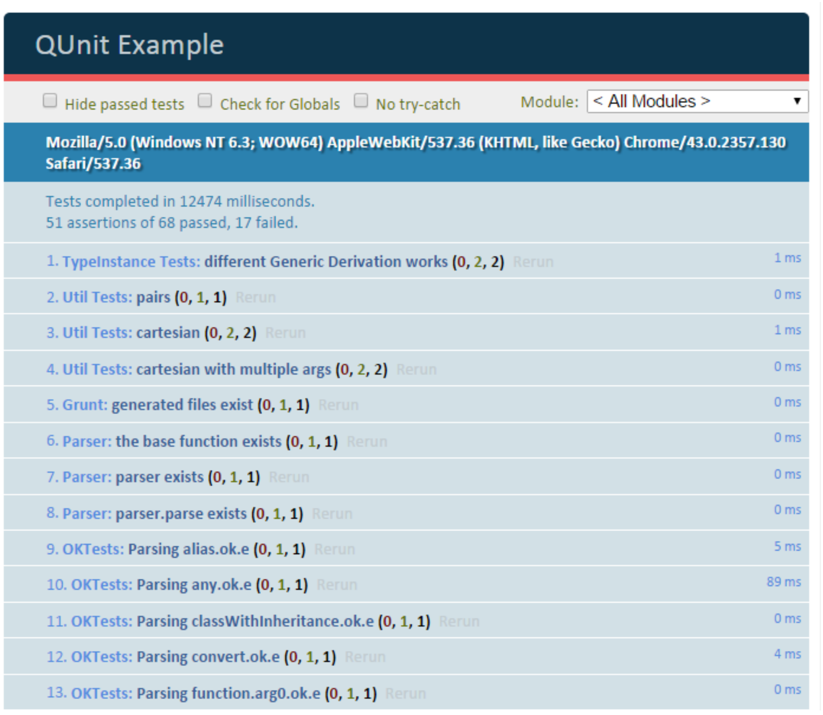
Parser. In the beginning, I immediately started working on the parser. All the other features would need to be built on top of it, except for the UI of course. I was already aware of *PEG.js*'s existence due to an article I had read several years ago, and chose the aforementioned library to serve as a parser generator. I had briefly reviewed alternatives such as Jison and JS/CC, but I chose *PEG.js* for it's more concise syntax and its very clear documentation and tight integration with JavaScript. This choice would later prove to be misguided, however.

Initially, I was making use of *PEG.js*'s demo page to get immediate feedback on the state of my grammar. The demo page features two editors side by side,

[3]https://room.eiffel.com/blog/colinadams/compiling_to_llvm
[4]https://svn.eiffel.com/viewvc/eiffelstudio/trunk/Src/framework/eiffel_llvm/
[5]http://kripken.github.io/clangor/demo.html
[6]http://kripken.github.io/llvm.js/demo.html

one for the grammar and another for the input to be parsed, along with one part of the page dedicated to the parser output. This allowed me to iterate very quickly and identify on the fly which parts of the grammar would be needed in order to cope with traditional programs that students in the course would see and make use of. This was done while still aiming to implement only a subset of *Eiffel*, which is why I developed the grammar mostly based on examples with very little reference to the *Eiffel* syntax given in its standard.

Testing. As the grammar grew more complex, I wanted to be able to test the grammar across many different small examples such that I could quickly evaluate what changes in the grammar would break which feature. Since the application was to be written in JavaScript, I opted for a JavaScript testing framework called *qunit*[7] which presents all the tests in the form of a webpage, as can be seen in Figure 2.1.



```
Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/43.0.2357.130
Safari/537.36

Tests completed in 12474 milliseconds.
51 assertions of 68 passed, 17 failed.

1. TypeInstance Tests: different Generic Derivation works (0, 2, 2) Rerun      1 ms
2. Util Tests: pairs (0, 1, 1) Rerun                                           0 ms
3. Util Tests: cartesian (0, 2, 2) Rerun                                       1 ms
4. Util Tests: cartesian with multiple args (0, 2, 2) Rerun                    0 ms
5. Grunt: generated files exist (0, 1, 1) Rerun                                0 ms
6. Parser: the base function exists (0, 1, 1) Rerun                            0 ms
7. Parser: parser exists (0, 1, 1) Rerun                                       0 ms
8. Parser: parser.parse exists (0, 1, 1) Rerun                                 0 ms
9. OKTests: Parsing alias.ok.e (0, 1, 1) Rerun                                 5 ms
10. OKTests: Parsing any.ok.e (0, 1, 1) Rerun                                 89 ms
11. OKTests: Parsing classWithInheritance.ok.e (0, 1, 1) Rerun                 0 ms
12. OKTests: Parsing convert.ok.e (0, 1, 1) Rerun                              4 ms
13. OKTests: Parsing function.arg0.ok.e (0, 1, 1) Rerun                        0 ms
```

Figure 2.1: Screenshot of qunit

This would also allow me to later add tests for my JavaScript application code and have all tests be in one place. Until that time, I had made use of the *PEG.js*'s demo page to compile the grammar into JavaScript source code. This change in testing also encouraged automating this process.

grunt & Node.js. This led me to investigate JavaScript build tools. The entire JavaScript ecosystem of tools is built around the *Node.js*[8] platform. *Node.js* is, among other things, a way to run JavaScript code outside of a

[7] https://qunitjs.com/
[8] https://nodejs.org/

browser, which allows building command line tools directly in JavaScript. One of these build tools that I found was *grunt*[9], which could be readily installed through *Node.js*'s package manager *npm*. *npm* can be instructed to keep track of all the installed packages for a certain project in a file called `package.json`. This enables the installation of all (*npm*-based) dependencies through a single command `npm install`, provided that *Node.js* is installed of course.

Relatively early on, I started to work in parallel on building an AST representation and the basics of the semantic analyzer. As the project kept evolving, supporting more and more *Eiffel* constructs, the analyzer code had to be adjusted repeatedly to reflect the new and changed constructs.

While I already had automated unit tests in place, the dynamically typed nature of JavaScript meant that I would only see the effects of a e.g. missing updated variable name when running the unit tests and only if one of the test actually exposed that error.

TypeScript. To make development easier, I went looking for alternatives. I found a very extensive list[10] with many compile-to-JavaScript languages and spent some time reviewing several languages, focusing on those with static typing. Of particular interest were *Elm*[11], *Nim*[12] and *TypeScript*[13]. *Elm* due to its powerful debugging with "time-travel-like" capabilities which I hoped to be able to use for stepping through *Eiffel* programs[14], *Nim* for expressive syntax and powerful macro system and *TypeScript* for its IDE support and straight forward JavaScript interoperability, not mentioning the numerous new methods for abstractions that the languages support. In the end, I chose *TypeScript* for its interoperability, IDE support and familiarity. While the other two languages brought many more powerful language features than *TypeScript* did, they seemed too experimental. Adding to this, I had no experience building more than toy applications with *Nim*'s paradigm, and in *Nim*'s case, not all language features, its standard library in particular, could be ported to JavaScript.

This led me to rebuild the entire codebase, fortunately still relatively small at that time, in *TypeScript*. Even though *TypeScript* is a superset of JavaScript, I redesigned the entire codebase from scratch, to make use of *TypeScript*'s additional features as much as possible and to employ type annotations wherever applicable but not easily inferable by the *TypeScript* compiler.

Focus Shift. As my implementation progressed, I came to realize that only building the parser and semantic analyzer while not yet considering the implementation of the *Eiffel* interpreter/debugger would not work. I wanted to think ahead such as to avoid another rewrite. The more I explored the details of what I would need, the more I realized that further extensions such as multiple inheritance and generics would not be easily achievable unless out the entire concept was fleshed out beforehand. Further, I noticed that by only supporting a subset of *Eiffel*, I would need to simulate more parts of the *Eiffel* standard library. It would also mean, that my implementation would need to

[9] http://gruntjs.com/
[10] https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS
[11] http://nim-lang.org/
[12] http://elm-lang.org/
[13] http://www.typescriptlang.org/
[14] http://debug.elm-lang.org/

cheat and special-case with parts such as arrays, because the features depend on generics, and/or the internal implementation relied on multiple inheritance. Furthermore, the omission of special cases that required too much work would severely limit what the student could do. When experimenting, they could easily go beyond those limitations. This is exacerbated by the fact that *Eiffel* uses the full spectrum of very advanced language features for the seemingly most simple elements of the language, such as integers.

For these reasons, I decided, that I would instead try to implement as much of *Eiffel* as possible, including and planning with multiple inheritance and generics from that point onwards.

This decision meant, that completing the original goals had become infeasible. However, the advantages of having a consistent implementation without special-casing anything that students would come across, except for *Eiffel* internals, seemed to justify this shift. It meant that instead of writing code to simulate *Eiffel*, I could spend my time implementing *Eiffel* language features which would then allow running even more *Eiffel* code.

I started by letting my parser and analyzer run over `ANY`, the base class from which everything else inherits. The analyzer would then complain about all classes that `ANY` required. This kicked off an iterative process of adding classes and their dependencies recursively, while at the same time fixing bugs that were exposed through the new classes from the *Eiffel* standard library. This also marked the deviation from my hand-crafted grammar to implementing most of the new rules according to their specification in the *Eiffel* standard [2].

During development, I've made use of features which preclude older browsers. In particular, I've made extensive use of ECMAScript 2015's, the next version of JavaScript, `Map`[15] and `Set`[16]. However, all functionality that I've used from `Map` can be emulated with JavaScript itself with a so-called "*polyfill*" or less specifically a "*shim*"[17], albeit with much worse performance. By that time, I was already exclusively testing my application in Chrome. Even though the original decision was that it should run anywhere, I knew[18] that if there was a problem in a different browser, it could be resolved through a polyfill. Later, these features would be supported by the browsers natively. These new upcoming JavaScript features allowed me to be more productive.

I also wanted to automate more of my build process: recompilation of *TypeScript*, e.g. automatic browser refreshes when those changes were complete and better automated tests. I switched my build tool from *grunt*, a configuration-over-code base build tool, to *gulp*[19], which uses code-over-configuration, because it was much easier to customize and did not rely as heavily on plugins as grunt did.

Demo. Nearing the end of my thesis, my supervisor Marco Piccioni recom-

[15] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map`
[16] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set`
[17] `https://remysharp.com/2010/10/08/what-is-a-polyfill`
[18] `http://kangax.github.io/compat-table/es6/#Map`
[19] `http://gulpjs.com/`

mended I implement a small demo website where some features of what I had accomplished during this time could be seen and presented. Until this point, the output of my program was solely in the form of debugging information.

Due to my previous web development experience, I knew that even small website features can take up a substantial amount of time, which is why I went looking for solutions to assist me. One criteria that I had, was that whatever solution I chose had to make little assumptions about how the application logic was structured, since that part was already written. I was also aware, that the JavaScript ecosystem is highly in flux, new frameworks and best practices appearing weekly[20,21]. It's one of the reasons why the "*TodoMVC*" project[22] has been created. It shows the same, simple application built with different frameworks. After looking at several popular contenders such as *Angular*[23], *ember*, [24] and *backbone*[25], I was unsure as to how well my existing code could be integrated with these frameworks. Finally, I settled on *react.js*, it's promise to be only responsible for the view while making almost no assumptions about the rest of one's code sounded like what I was looking for. Additionally, it supports reusable and composable UI components while being extremely fast.

As a next step, I went over the list of community-built UI components[26], most of which were single components for very special purposes, instead of a comprehensive library with generic UI components that I was looking for. I settled on *material-ui*[27], a project implementing Google's *material design*[28] standard for *react.js*

These dependencies brought two new major dependencies with them: *browserify*[29] and *babel*[30], both of which I was able to integrate fairly easily into my new build toolchain.

The last choice regarding what technology to choose that I've had to make was between the two in-browser code editors: *CodeMirror*[31] and *Ace*[32]. Both editors are very popular. *Ace* is used in a cloud based IDE and *CodeMirror* powers the developer tools in both Chrome and Firefox. With *Ace*, I had done some earlier experiments in combination with *require.js* and it didn't seem to work well.

Lastly there was the question of easily importing code into the application. In addition to the traditional open file dialog, I've chosen to support the Chrome-exclusive drag & drop support for folders. Unfortunately, this API is not (yet) standardized. However, in other browsers multi-file drag & drop works as well, as does the open-file-dialog.

[20] http://www.allenpike.com/2015/javascript-framework-fatigue/
[21] https://blog.andyet.com/2014/08/13/opinionated-rundown-of-js-frameworks
[22] http://todomvc.com/
[23] https://angular.io/
[24] http://emberjs.com/
[25] http://backbonejs.org/
[26] https://github.com/facebook/react/wiki/Complementary-Tools#ui-components
[27] http://material-ui.com/
[28] http://www.google.ch/design/spec/material-design/introduction.html
[29] http://browserify.org/
[30] https://babeljs.io/
[31] https://codemirror.net/
[32] http://ace.c9.io/

I've also wanted to provide students with the ability, to easily import sample projects. For that, I envisioned a GitHub[33] integration, such that code could be read directly from repositories or Gists[34] on GitHub. However, due to GitHub restrictions[35] and having to run this application on a server with a custom backend[36], this idea was scrapped.

[33] https://github.com/
[34] https://gist.github.com/
[35] https://developer.github.com/v3/rate_limit/
[36] http://blog.vjeux.com/2012/javascript/github-oauth-login-browser-side.html

Chapter 3

Obstacles

This chapter gives an overview of some selected problems that were encountered during the thesis and how they were solved. The grammar examples in this section are all simplified from their actual implementation unless they are specifically referred to, as well as references to the official Eiffel grammar specifications as per the standard are simplified unless relevant to the subject at hand.

3.1 Parsing

PEG.js was found to have several limitations and simplifications, which result in some unnecessarily complex grammar rules and degrade performance significantly.

Backtracking. *PEG.js* only does a very limited amount of backtracking: Only the choice operator `/` backtracks if an alternative fails. Otherwise, the parser is greedy and as soon as a subrule has matched somehow, it will not be reevaluated for different alternatives if its calling rule should fail at a later point. In general, this means that the longest possible match always needs to come first.

For instance, consider Listing 3.3 on the next page. If features were parsed with a grammar such as the one in Listing 3.1, the parser would fail. The

Listing 3.1: *Pegjs*: Incorrect rule order

```
Features = Feature*

Feature
  = Attribute
  / Function
  / Procedure
```

Listing 3.2: *Pegjs*: Correct rule order

```
Features = Feature*

Feature
  = Function
  / Procedure
  / Attribute
```

Listing 3.3: *Eiffel*: Parsing example

```
1 class EXAMPLE
2
3   attr: TYPE
4
5   func: TYPE
6     do
7
8     end
9 end
```

`Attribute` rule would match the beginning of the line 3 in Listing 3.3. Then, another `Feature` would be expected as per the `Features` rule. However, no feature would match the remaining text, so the parse fails, because it won't backtrack to choose something different than `Attribute`. The corrected grammar can be seen in code Listing 3.2.

Whitespace. *PEG.js* also does not do a separate lexing pass. As a result, whitespace has to be manually specified everywhere. Coupled with the greediness of the parser and lack of backtracking, one has to be very careful not to match too much whitespace at some point, when one expects whitespace before a certain location. Otherwise, the parser will obviously fail, because there isn't any more whitespace available to be matched. As a result, I've been tried to consistently allow and/or demand for whitespace *before* other rules, rather than a mix of the two. After certain elements, however, one must make certain that there actually follows whitespace, but this can be solved using a lookahead that does not actually consume any characters.

Since whitespace must be manually specified throughout the entire grammar, I've specified two short convenience rules `w` and `W` that stand for optional and obligatory whitespace, respectively.

At a later time I've noticed, however, that my rule for obligatory whitespace should not be necessary very often: The greedyness of the algorithm actually works in one's favor in some instances. For example, the grammar always requires whitespace to occur between two words. However, when matching identifiers, they are matched greedily, so it's not possible that there isn't any whitespace between two words. This would impact the grammar only very little though.

Listing 3.4: *Eiffel*: Parsing example

```
 1 class EXAMPLE
 2 
 3   func: TYPE
 4     obsolete
 5       "[Tag] message"
 6     do
 7 
 8     end
 9 note
10   copyright: "[
11     copyright
12   ]"
13 end
```

Verbatim strings. Eiffel supports a special kind of string that can span multiple lines with special delimiters[1]. For example `"alpha[content]alpha"` where `content` can span over multiple lines and `alpha` is a sequence of letters that appears once at the beginning and again and the end.

PEG.js Does not support backreferences which would allow parsing this easily. Instead, custom JavaScript code is inserted after certain rules, which constructs and stores a lookahead to match the corresponding ending of the string `]alpha"`. After every following character, there is a negative lookahead performed for that string. Since that rule only fails if the lookahead fails, i.e. the previously stored lookahead is actually present, everything until the next quotation mark is then matched.

Another thing of note is, that the specification does not seem to explicitly force verbatim strings to stretch over multiple lines, for the verbatim string opener to be only preceded by whitespace until the next linebreak, or for the closer to be preceded by only whitespace until the previous line break. The EiffelStudio compiler enforces all of these rules. Omitting them, as the specification would seem to suggest, results in at least one of the standard library classes not parsing. My grammar would detect a verbatim string opener on line 5 of Listing 3.4. This ending would be found on line 12, which would obviously invalidate the class.

Free operators. This is another case where the specification[2] seems to differ from what *EiffelStudio*'s compiler does, or the specification has been worded poorly. One free operator that does not seem to be covered is the interval operator `|..|`.

Every character in the operator must be an "*operator symbol*", as per the specification. To qualify as an *operator symbol*, it must satisfy any of the three properties given in **8.32.20**[3]. However, it "appears in" a special symbol "`.`", it goes against "[...]is neither a dot `.` nor[...]" and clearly is none of the listed

[1]See the Eiffel Standard, ECMA-367, **8.29.10** Manifest strings
[2]See the Eiffel Standard, ECMA-367, **8.32.21** Free operator
[3]See the Eiffel Standard, ECMA-367, **8.32.20** Operator symbol

Listing 3.5: *Pegjs*: recognizes simple arithmetic expressions

```
additive
  = multiplicative "+" additive
  / multiplicative

multiplicative
  = primary "*" multiplicative
  / primary

primary
  = integer
  / "(" additive ")"

integer "integer"
  = digits:[0-9]+
```

Listing 3.6: *Pegjs*: Excerpt of implementation of precedence

```
ImpliesExpr
  = OrExpr ("implies" OrExpr)*

OrExpr
  = AndExpr ("or" AndExpr)*

AndExpr
  = CompExpr("and" CompExpr)*
```

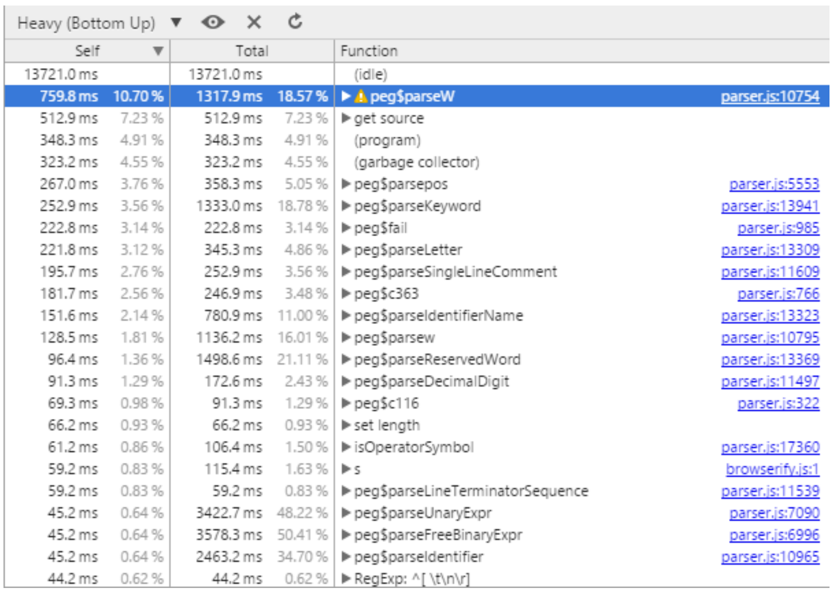
symbols in the last property.

It follows, that `|..|` would be an invalid operator, and yet still had to be supported. I've changed the grammar to allow for this operator to be parsed, but it's unknown whether this new grammar now allows operators, that the *EiffelStudio* compiler does not.

Operator Precedence. *PEG.js* does not offer any assistance for defining operator precedence or associativity as other parser generators might. The first implementation was modelled after the grammar demo for *PEG.js* as can be seen in Listing 3.5. This specifies precedence in a very straight forward manner as well as right-associativity for the operators. However, for left associativity, this the operands in the rules cannot be merely switched: This would allow `additive` to invoke itself an arbitrary amount of times without consuming any input. This attempt made the browser very unresponsive until it would forcefully stop script execution. I also tried manually reversing the associativity but that also resulted in performance issues. Instead, I settled on an approach, where a rule never calls itself recursively as can be seen in Listing 3.6. The hierarchical structure is constructed manually afterwards.

Error messages. Unfortunately, the goal to provide good parser error messages could not be achieved overall. In some places, the syntax was relaxed to allow for theoretically invalid ASTs where the intent of the programmer can still be certainly inferred, but this entails writing an analysis for each such case. *PEG.js* provides relatively good error messages for a parser generator, in that it gives a lot of output of it would have been expecting in order to continue, but these messages are completely unhelpful to students, my supervisor Marco Piccioni and I found.

Performance. The biggest remaining issue is the amount of time it takes to parse the roughly 100 classes of the included standard library. The primary reason for this seems to be the lack of a lexer-pass in the parser. For every rule alternative, whitespace (`parseW`) and all identifiers have to be reparsed over and over again, as well as verified that identifiers are not keywords. Naturally, these rules tend to dominate the performance. Also, the position function (`parsepos`) is called over and over again, constructing new objects all the time. There are two graphs, Figure 3.1 is ordered by how much time is spent in a function itself in total, and Figure 3.2 on the next page is ordered by how much time is spent in which function in total, including all subroutine calls.



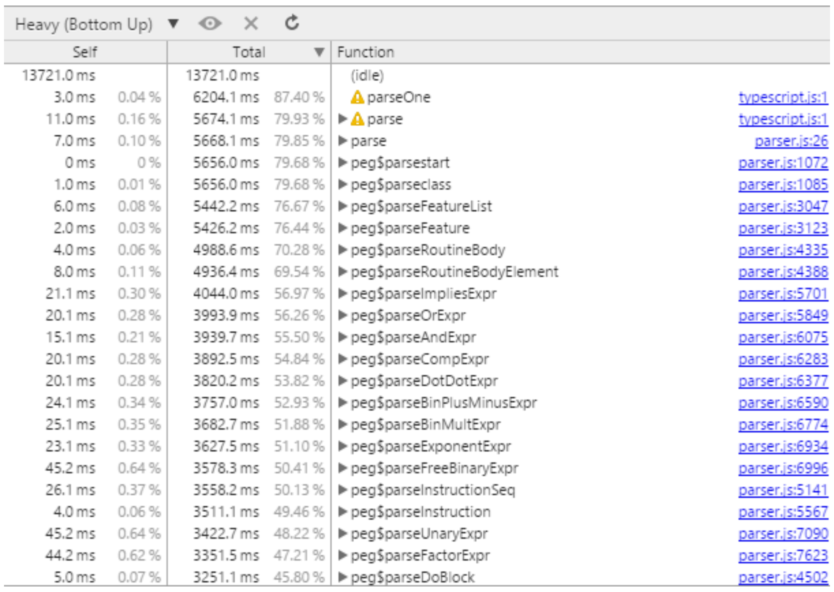
```
Heavy (Bottom Up) ▼
         Self ▼     |        Total       | Function
 13721.0 ms         |  13721.0 ms        |   (idle)
   759.8 ms 10.70 % |   1317.9 ms 18.57 % | ▶ ⚠ peg$parseW                         parser.js:10754
   512.9 ms  7.23 % |    512.9 ms  7.23 % | ▶ get source
   348.3 ms  4.91 % |    348.3 ms  4.91 % |   (program)
   323.2 ms  4.55 % |    323.2 ms  4.55 % |   (garbage collector)
   267.0 ms  3.76 % |    358.3 ms  5.05 % | ▶ peg$parsepos                        parser.js:5553
   252.9 ms  3.56 % |   1333.0 ms 18.78 % | ▶ peg$parseKeyword                    parser.js:13941
   222.8 ms  3.14 % |    222.8 ms  3.14 % | ▶ peg$fail                            parser.js:985
   221.8 ms  3.12 % |    345.3 ms  4.86 % | ▶ peg$parseLetter                     parser.js:13309
   195.7 ms  2.76 % |    252.9 ms  3.56 % | ▶ peg$parseSingleLineComment          parser.js:11609
   181.7 ms  2.56 % |    246.9 ms  3.48 % | ▶ peg$c363                            parser.js:766
   151.6 ms  2.14 % |    780.9 ms 11.00 % | ▶ peg$parseIdentifierName             parser.js:13323
   128.5 ms  1.81 % |   1136.2 ms 16.01 % | ▶ peg$parsew                          parser.js:10795
    96.4 ms  1.36 % |   1498.6 ms 21.11 % | ▶ peg$parseReservedWord               parser.js:13369
    91.3 ms  1.29 % |    172.6 ms  2.43 % | ▶ peg$parseDecimalDigit               parser.js:11497
    69.3 ms  0.98 % |     91.3 ms  1.29 % | ▶ peg$c116                            parser.js:322
    66.2 ms  0.93 % |     66.2 ms  0.93 % | ▶ set length
    61.2 ms  0.86 % |    106.4 ms  1.50 % | ▶ isOperatorSymbol                    parser.js:17360
    59.2 ms  0.83 % |    115.4 ms  1.63 % | ▶ s                                   browserify.js:1
    59.2 ms  0.83 % |     59.2 ms  0.83 % | ▶ peg$parseLineTerminatorSequence     parser.js:11539
    45.2 ms  0.64 % |   3422.7 ms 48.22 % | ▶ peg$parseUnaryExpr                  parser.js:7090
    45.2 ms  0.64 % |   3578.3 ms 50.41 % | ▶ peg$parseFreeBinaryExpr             parser.js:6996
    45.2 ms  0.64 % |   2463.2 ms 34.70 % | ▶ peg$parseIdentifier                 parser.js:10965
    44.2 ms  0.62 % |     44.2 ms  0.62 % | ▶ RegExp: ^[ \t\n\r]
```

Figure 3.1: Profiling parsing and analysis: Self time

3.2 Analysis

Incremental parsing & analysis. Initial tests for the parser showed it to run extremely fast. Hence, the analyzer and AST were initially designed to be re-initialized with each run. Only when the grammar and tested input files grew, performance bogged down. Since parsing a part of the included standard library amounted to over a five second runtime on an Intel i7-4770k, it was no longer feasible to re-run everything for every change a student makes.

To tackle this, the standard library is only parsed and analyzed once during



Heavy (Bottom Up)

Self		Total		Function	
13721.0 ms		13721.0 ms		(idle)	
3.0 ms	0.04 %	6204.1 ms	87.40 %	⚠ parseOne	typescript.js:1
11.0 ms	0.16 %	5674.1 ms	79.93 %	▶ ⚠ parse	typescript.js:1
7.0 ms	0.10 %	5668.1 ms	79.85 %	▶ parse	parser.js:26
0 ms	0 %	5656.0 ms	79.68 %	▶ peg$parsestart	parser.js:1072
1.0 ms	0.01 %	5656.0 ms	79.68 %	▶ peg$parseclass	parser.js:1085
6.0 ms	0.08 %	5442.2 ms	76.67 %	▶ peg$parseFeatureList	parser.js:3047
2.0 ms	0.03 %	5426.2 ms	76.44 %	▶ peg$parseFeature	parser.js:3123
4.0 ms	0.06 %	4988.6 ms	70.28 %	▶ peg$parseRoutineBody	parser.js:4335
8.0 ms	0.11 %	4936.4 ms	69.54 %	▶ peg$parseRoutineBodyElement	parser.js:4388
21.1 ms	0.30 %	4044.0 ms	56.97 %	▶ peg$parseImpliesExpr	parser.js:5701
20.1 ms	0.28 %	3993.9 ms	56.26 %	▶ peg$parseOrExpr	parser.js:5849
15.1 ms	0.21 %	3939.7 ms	55.50 %	▶ peg$parseAndExpr	parser.js:6075
20.1 ms	0.28 %	3892.5 ms	54.84 %	▶ peg$parseCompExpr	parser.js:6283
20.1 ms	0.28 %	3820.2 ms	53.82 %	▶ peg$parseDotDotExpr	parser.js:6377
24.1 ms	0.34 %	3757.0 ms	52.93 %	▶ peg$parseBinPlusMinusExpr	parser.js:6590
25.1 ms	0.35 %	3682.7 ms	51.88 %	▶ peg$parseBinMultExpr	parser.js:6774
23.1 ms	0.33 %	3627.5 ms	51.10 %	▶ peg$parseExponentExpr	parser.js:6934
45.2 ms	0.64 %	3578.3 ms	50.41 %	▶ peg$parseFreeBinaryExpr	parser.js:6996
26.1 ms	0.37 %	3558.2 ms	50.13 %	▶ peg$parseInstructionSeq	parser.js:5141
4.0 ms	0.06 %	3511.1 ms	49.46 %	▶ peg$parseInstruction	parser.js:5567
45.2 ms	0.64 %	3422.7 ms	48.22 %	▶ peg$parseUnaryExpr	parser.js:7090
44.2 ms	0.62 %	3351.5 ms	47.21 %	▶ peg$parseFactorExpr	parser.js:7623
5.0 ms	0.07 %	3251.1 ms	45.80 %	▶ peg$parseDoBlock	parser.js:4502

Figure 3.2: Profiling parsing and analysis: Total time

an initial loading screen. When the student's code is analyzed, the analyzer only accesses other classes through a single entry point: `AnalysisContext.classWithName()`. Thanks to this, this function could easily be adjusted to delegate the request to a different instance of `AnalysisContext`, that was used to analyze the standard library.

Inheritance Cycles. This problem was solved through a very straight forward depth-first-search approach. If a class is encountered that has already been visited, a cycle has been detected. The application is able to distinguish between cycles and classes that inherit from a cycle, but do not participate in it. Currently, such classes are ignored in further analysis steps.

Multiple Inheritance. I have tried implementing this as the standard describes[4]. However, the first major problem came up when reading the standard. Then standard seemed to be very imprecise about a particular issue with repeatedly inherited features in point 1 of the previously referenced section **8.16.12** of the standard. After reasoning through several different possible interpretations of several of the referenced rules therein, I contacted my supervisor Marco Piccioni for clarifications. He redirected my questions to Emmanuel Stapf, the lead engineer on the Eiffel compiler.

He told us, that the standard was merely the "goal", and that their Eiffel compiler did not adhere to the standard in that case. With that new knowledge, I continued. Further, I discovered the an algorithm could not follow the steps as outlined in **8.16.12**, because some of the information that it required were not available before the inherited features were determined.

In particular, the standard talks about merging different features, based on

[4]See the Eiffel Standard, ECMA-367, **8.16.12** Inherited Features

the signatures that they have. However, the signatures of features making use of anchored types are not known until after this step is complete.

Instead, features are combined by name, and a history of which features have been merged to what name is kept, such that afterwards the merges can be verified when the all type information is available.

For every parent, the following steps are taken:

- The features are renamed.
- The adaptions are indexed in a best-faith approach. For example, if an adaption refers to a feature by its old name, and there is no other feature that has been renamed to that name, the adaption is accepted but the user is warned about it. Further, adaptions can appear in any order (the standard defines a different order than what the official Eiffel compiler requires).
- Every feature is inserted into a data structure given by the class `FeaturePretenders`. It collects all features competing for the same name.

This data structure consists of four lists: "*effective*", "*deferred*", "*redefined*" and "*selected*". Every feature inside the data structure is in exactly one of the first three lists. The decision in which list the feature is inserted is made in the following order, the first condition that matches takes precedence. It is in `redefined`, if the feature has been marked for redefinition. If the feature is marked for undefinition, it goes into the `deferred` list, also, if the feature was already deferred in the indicated parent class. All other features go into `effective`, as they were inherited as effective from their parents.

Similarly, features are also inserted into the `selected` list, if they were marked for selection.

In the next step, all `FeaturePretenders` datastructures are processed. Now, a single feature is chosen from all the pretenders to serve as the inherited feature. In general, there can either only be one feature in `effective` or one or more entries in the `redefined` list, otherwise, all features must be in `deferred` and/or `redefined`. Also, if there are features in `redefined`, the current class must have a redeclaration for a feature of said name, i.e. in a feature list, otherwise it is an error.

Multiple features in `effective` is obviously a conflict. If there is a feature in `effective` and there is a redeclaration, those two features are in conflict. If there are only features in `deferred` and `redefined`, possible conflicts can only be determined once all type information is available, which is only after this stage.

If none of the above cases apply and there is exactly one effective feature, this will be determined the inherited feature, similarly, if there is no effective feature and there is a redeclaration, that feature will be the inherited feature.

Generics. Generics could have been implemented in a very simple manner: By directly replacing every feature with a new copy where all generic parameters have been substituted for their substitutions. However, this did not sound very appealing from a performance point of view and also, it would break the connection from the actual source code to what *Eiffel* "*saw*". In particular, for explanations for students it would be important to have all the substitution information available wherever they were required.

This led to a different approach. Originally, I implemented a `ClassSymbol` class that was to represent one Eiffel class. It would have information about all its features, it parents and so forth. However, with generics in mind, something that had originally been considered an "if-time-permits-feature", the concept of every variable and feature simply referring to `ClassSymbol`s was no longer possible unless for every generic derivation, there was a different ClassSymbol, all with its own features and other members. Instead, I opted to create a `TypeInstance` class, which, as its base type, would always refer to one `ClassSymbol`, and also have a list of parameters. These parameters would be of type `TypeInstance` recursively. A generic type parameter was modelled by having a `TypeInstance`, whose base type was a `ClassSymbol` that represented a generic parameter, with an empty parameter list.

However, this approach would led to different problems once inheritance was introduced, because not enough information was being stored. This led to the final implementation that is now used. An interface `ActualType` was created which unified `TypeInstance` with the new class `FormalGenericParameter`. `ClassSymbol` would no longer be used for generic parameters. This led to many changes throughout the source code. The type system could now distinguish between the two, an instance of refactoring where TypeScript was particularly useful. Further, every `TypeInstance` now carries an instance of the `Substitution` class. It contains a mapping between `FormalGenericParameter`s and `ActualType`s. This way, all replacements are recorded in one datastructure and an easy lookup is possible for explanation purposes.

This `Substitution` datastructure has also been added to `FeatureSymbol`, which represents a feature. Since types are only resolved after inheritance, for every inherited feature it must know how to resolve generic type parameters therein appearing type usages. Since a class is not allowed to inherit from different generic derivations[5], the union of all substitutions performed across the different parents would not yield any conflicting substitutions. However, when explaining to a student how the type of one particular feature was derived, all the other substitutions would not be necessary, since only the relevant information would be present.

Initially, it was planned that a `TypeInstance` would receive its own generic instantiations of certain features, i.e. there would have been copies or another class `FeatureInstance`. However, they themselves again would have needed to have `TypeInstance`s to refer to their features (or rather, `ActualType`s, with the new implementation). This, however, without some sort of global registry for `TypeInstance`s with a much more complicated initialization to satisfy all mutual relationships, would have created an infinite object graph. Instead, it was decided that all feature accesses are implemented lazily through functions, and new `TypeInstance`s are created on the fly.

[5]See the Eiffel Standard, ECMA-367, **8.6.13** Parent rule(5)

Anchored Types. This is a powerful Eiffel feature that breaks modularity. It allows for an entity's type to be defined in terms of the type of another. Let's consider to features `depending: like defining` and `defining: ANY`. Suppose the class assigns something to the depending attribute. At a later time, a subclass is introduced, which redefines `defining` into `defining: STRING` since covariant redefinitions are allowed as per the specification[6]. If the parent class did not assign a `STRING`, after all, it could assign anything since everything inherits from `ANY`, this will lead to a compile error once the new class is introduced. Eiffel will recompile the feature containing the anchored type and re-typecheck the feature.

This shows, that anchored types break modularity. It follows, that the types of a feature making use of anchored types must be determined for every class that inherits this feature. Further, the feature that the anchored type refers to might have been renamed, and `select` will need to be respected.

The implementation currently only works for features, that do not depend on a renamed feature or one that has been affected through the use of `select`.

The algorithm constructs a dependency graph between all the features return types including their pretenders, their local variables and parameters.

Then, Tarjan's algorithm[1] is applied to that graph. Tarjan's algorithm determines the strongly connected components (SCCs) of a graph. The SCCs form a disjoint set of all nodes, where all nodes in one particular SCC can be reached from any other node in that same SCC. In other words, all the nodes inside one SCC form a cycle. An SCC consisting of one component has no cycles, a node cannot have an edge from itself to itself, this would immediately create a cycle in the dependency graph and can already be detected upon the construction of said graph.

The resulting list of SCCs will contain exactly as many SCCs as there are nodes if and only if there are no cycles in the dependency graph, and thus no cycles in the anchored types. By virtue of how the algorithm operates, these SCCs now form a topological sort on all the nodes, which directly correlates to the order in which the types of all entities must be initialized.

Further, attention must be paid that generic parameters are resolved with the respect to the generic type variables from a feature's declaring class.

3.3 GUI

CodeMirror & react. As I introduced the tabbed interface to support editing multiple files, something curious would happen: The editor would reset into its initial state. I discovered that it didn't just reset, it would completely reinitialize itself. When using CodeMirror, you put a simple `<textarea>` element onto your page, then instruct CodeMirror to initialize itself in its place.

I was sure that this was a *react.js*-specific problem, because react was re-rendering the component it's in. From react's perspective, it can only see the

[6]See the Eiffel Standard, ECMA-367, **8.10.26** Redeclaration rule (2)

this `<textarea>` element that is then replaced by CodeMirror. When react then re-creates the component, it will reinstate the `<textarea>`, which will cause CodeMirror to re-initialize.

This is a problem because you would lose edit history and therefore the ability to undo changes, cursor position, and selections.

To solve this, I overrode the `shouldComponentUpdate` function[7] to always return `false`, such that *react.js* keeps what is already in the DOM.

material-ui. This UI component library is still under heavy development. It lacked several customization abilities that I needed such as custom colors on some elements and bugs were found. Some of these enhancements[8] and fixes[9] have already been contributed back to the project and have been accepted.

Syntax highlighting. CodeMirror already had the ability to highlight Eiffel code, thanks to fellow ETH student Yassin Hassan's earlier contribution to CodeMirror. Activating the highlighting should have been as simple as including the additional JavaScript containing the relevant code. However, since my build process uses "*browserify*", the syntax highlighting file was not able to register itself with the CodeMirror data structures, it just failed silently. The solution was to include the JavaScript file through a `require()` call which instructs browserify to pull in the therein mentioned module.

AST visualization. For this feature, I needed to be able to map a cursor position to nodes in the AST. I added short rule `pos` to the parser that I would use twice for every rule the constructs an `AST` node: Once to mark the start of the characters the node represents, and one to mark the end. Initially, this rule returned an object consisting of three fields: `offset`, `line` and `column`, all information which is directly exposed by *PEG.js* through eponymous functions. However, this greatly impacted the parse times. Not having a lexer-pass meant, that these positions were recomputed over and over again for every position and for every wrong branch that was taken. I settled to using only the `offset` function. Luckily, CodeMirror could provide the cursor formation as an offset as well. Upon a successful parse, which is triggered a few milliseconds after a user has made some changes, the AST is transformed into a segment tree utilizing the s-tree[10] library.

When the cursor is moved, this segment tree is queried for all intervals wherein the cursor is contained, thereby finding the surrounding AST nodes. The hierarchy is then established by sorting by start position and length. Currently, it's possible that they do not form a strict hierarchy, if the cursor happens to fall exactly onto the boundary of two adjacent AST nodes.

Another issue that became apparent, was that of whitespace processing during parsing and how the start and end positions were set. It seemed to make sense, to never include any additional whitespace within the start and end boundaries. This, however, did not seem give satisfiable explanations. In

[7] https://facebook.github.io/react/docs/component-specs.html#updating-shouldcomponentupdate
[8] https://github.com/callemall/material-ui/pull/953
[9] https://github.com/callemall/material-ui/pull/965
[10] https://github.com/rogah/s-tree

Listing 3.7: *Eiffel*: Parsing example

```
 1
 2  class
 3    APPLICATION
 4
 5  inherit
 6    ARGUMENTS
 7
 8  create
 9    make
10
11  feature {NONE} -- Initialization
```

Listing 3.7 on lines 7 and 10, it would only report the AST node pertaining to the entire class. When explaining this code however, I would explain line 7 as belonging to the part, where parents for the class are specified, however, and line 10 belongs to the area, where valid creation procedures are specified. This meant, that additional whitespace had to be included before the ending position marker in some select places.

However, I could not just insert my rule for optional whitespace, because my grammar sometimes required whitespace to precede certain rules. I created a new whitespace rule `S`, which would match all remaining whitespace including comments except for the final whitespace character.

GitHub import. One feature I had been working on for the GUI had to be abandoned, unfortunately. The idea was to provide the ability to easily import sample projects from GitHub and such that students could share their own through it or through GitHub's Gists. However, I had to discover that the public API which I had been using only allowed for 60 requests per hour[11] which enforced based on the IP. Since every file had to be fetched in one request along with several metadata calls per import, this was infeasible. Authorized access entailed running my own server, or have the students sign up with GitHub, manually perform steps on GitHub to generate an access token for their account, and then to input that token into my application. Instead, students can just download a ZIP from GitHub containing the source code. With a small addition to the application, it would then also be able to read ZIP files directly. Due to security restrictions, JavaScript is unable to download the aforementioned ZIP file directly from GitHub. If browsers were to allow such requests, I could have circumvented GitHub's API limit.

[11] https://developer.github.com/v3/#rate-limiting

Chapter 4

Deliverables

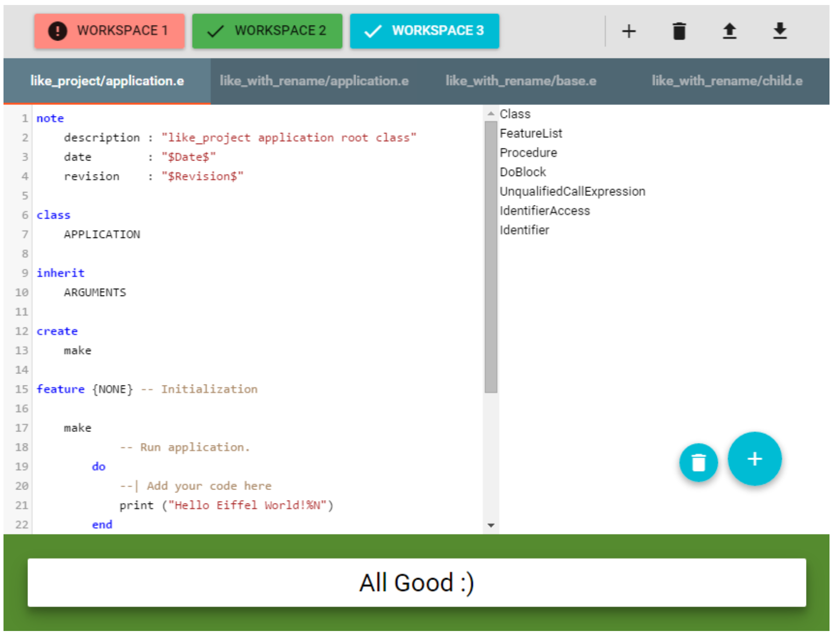
This chapter describes what has been achieved and gives an overview of the delivered project's structure.

The project, as already mentioned in the introduction section 1.3 on page 7, the focus of the project has changed during the thesis. However, this is largely due to a shift in focus of the project. Instead, the parser is virtually feature complete: It can parse `ANY` and all its direct and indirect dependencies and map that into an AST data structure. Thus, this particular goal has been largely exceeded. The newly added goal of multiple inheritance and generics have tremendously increased the effort for the semantic analyzer. Most of its pieces are there, such as to serve as a solid foundation for future improvements. The primary obstacle as of now, are unclear semantics of the official *Eiffel* compiler in some cases with multiple inheritance, particularly in relationship with `select`.

The analyzer is capable of recognizing cyclic inheritance, it performs feature merging for multiple inheritance complete with a history as to where each feature came from and what features have been merged into one. It supports generics with a mapping that keeps track of all applied substitutions. Anchored types are resolved including a cycle detection, except for those who depend on a feature that was renamed during inheritance. This ties into the aforementioned point of some *Eiffel* semantics yet remaining unclear.

Work on the interpreter with stepping has also begun. It is based on nested graphs, where every graph outlines the steps that need to be taken to execute a particular AST node. Every node can consist of a subgraph, which in turn consists of the steps to execute nested AST nodes. The graph also doubles as an explanation for the instructions, by providing labelled edges and the possibility of dummy nodes that only hold explanatory data. This also ensures, that the actual execution also matches the documentation provided to the student. Further, this will allow for easy stepping, simply by keeping track of a stack of parent nodes.

Further, a webpage demonstrating some of the implemented feature has been implemented as can be seen in Figure 4.1 on the following page. Other visualizations there were initially targeted have been relegated to future work as described in section 1.3 on page 7.



```
note
    description : "like_project application root class"
    date        : "$Date$"
    revision    : "$Revision$"

class
    APPLICATION

inherit
    ARGUMENTS

create
    make

feature {NONE} -- Initialization

    make
            -- Run application.
        do
            --| Add your code here
            print ("Hello Eiffel World!%N")
        end
```

```
Class
FeatureList
Procedure
DoBlock
UnqualifiedCallExpression
IdentifierAccess
Identifier
```

All Good :)

Figure 4.1: Screenshot of application

4.1 Code

The application is divided into three primary parts and technologies.

PEG.js is used for generating the parser.

TypeScript All the application code: AST data structures, semantic analysis and business logic for the web application is implemented in *TypeScript*.

React is used to construct the UI and partition it into separate components.

Here follows an overview of the folder structure, along with the most important files.

```
/
├── package.json ............................... contains dependencies
├── bower.json ................................. contains dependencies
├── gulpfile.json ............................. controls build process
├── gulp .................................. tasks for the build process
├── .editorconfig .................... controls editor's indentation etc
├── src ....................................... the application code
│   ├── css ..................................................... SCSS files
│   ├── eiffel ............................... contains standard library
│   ├── grammar
│   │   └── eiffel.pegjs ............................................ grammar
│   ├── react ........................................ react components
│   ├── ts ........................................ TypeScript source code
│   └── www ...................................................... HTML code
└── test ................................... qunit based test code
```

The file `semantics.ts` in the folder `src/ts` contains the entry point for the analysis: The `analyze` function. From there, it proceeds through a multitude of steps, that perform the implemented analysis.

The entry point for the GUI of the demo application is in the file `app.jsx` inside the folder `src/react`. It initializes the business logic for the web application, which is encapsulated by the `Model` class inside `src/ts/app.ts`. `app.jsx`'s code is automatically executed by its inclusion inside `index.html` in the `src/www` folder. It is included last (`browserify.js`) such that all other scripts have been loaded by the time it executes.

Care has been taken, such that the code is as self documenting as possible.

4.2 Dependencies

This section gives a short overview of the most important dependencies upon which the project or its build process is based.

Node.js A JavaScript engine for server-side applications. Needed to execute tools written in JavaScript.

PEG.js is the parser generation that is used for this project. It uses the PEG syntax format intermixed with JavaScript.

TypeScript is a superset of JavaScript. The fundamental addition is that of type annotations, which allows for better static analysis and refactoring support in IDEs.

react.js is a UI library for easily rendering reusable components. It brings its own custom JavaScript extension called JSX, in order to write *react.js* code as if it were HTML.

CodeMirror is a code editor for the web.

material-ui is a collection of UI components implemented with *react.js*

gulp is a JavaScript build too, used to automate all build steps in the project.

browserify brings the upcoming JavaScript module system to the web by statically analyzing the imports. Compatible with Node's module system which means most libraries that were written for node can now be used in the web effortlessly.

babelify A *browserify* plugin which transforms next-generation JavaScript code into today's JavaScript, including JSX-to-JavaScript translation.

Browsersync starts a webserver with which the website is served during development, and is able to instruct the browser to reload when the source files change.

font icons The "ok" and "error" icons are custom fonts and reside in the `assets` directory. There is a demo page indicating the CSS class names that can be used to display them.[1]

4.3 Build step

The build step produces 4 primary JavaScript files.

```
/dist
 ├─ builtin.js ................................. the standard library
 ├─ browserify.js ....................... react code and dependencies
 ├─ parser.js ................................ controls build process
 └─ typescript.js
```

Gulp is used by specifying a set of tasks. Every task can name certain other tasks as its dependencies. When running a task, Gulp will ensure that all their dependencies are performed first.

`builtin.js` contains the standard library as an array of JavaScript strings, it is generated by the task `builtin`. This is generated from the files in the `eiffel directory`. `browserify.js` contains all the user interface code along with many dependencies. It is generated by the task `browserify`. `typescript.js` is generated by `typescript`.

Using dependencies, there are tasks that will clean the `/dist` folder from generated files. Also, there is a master task `default` that is executed when `gulp` is run without arguments. It automatically starts two webservers, one for tests and the other for the demo application, and makes the browser refresh the respective pages when files that they depend on change.

There are other build steps to place other less important dependencies in the correct location.

[1]https://icomoon.io/app/#/select

4.4 Debugging

Even though the build process combines the different source files into huge blobs of concatenated JavaScript, debugging is still fairly straight forward. Due to a feature called ”source maps”, the browser can display the original file while it’s actually executing the processed file. This means that I can actually debug using the *TypeScript* sourcecode in Chrome, even though it doesn’t understand it. Not all features such as breakpoints or hovering over variables to see their values work as well, though.

4.5 Webpage

The created demo page[2] as seen in Figure 4.1 on page 28 showcases some of the features that have been implemented during this thesis. In particular, a code editor has been integrated with the developed parser. The application has only been tested in Chrome, as it is the only browser that currently supports all required features.

Workspaces. Since parsing the standard library takes several seconds (see section 3.2 on page 20), workspaces have been added, such that a student can work on multiple projects without the browser having to re-parse the standard library for every instance of the application. Workspaces are isolated from each other: Errors in a workspace do not affect the other workspaces, nor do they share any open files.

The different workspaces can be accessed over a list of dynamically generated buttons at the top of the page. The active workspace is indicated by a blue button, buttons to inactive workspaces are styled differently depending on whether the corresponding workspace currently has any errors, as can be seen in Figure 4.2.

New workspaces can be created by clicking the plus icon and deleted by clicking the bin icon respectively, as seen in Figure 4.3.

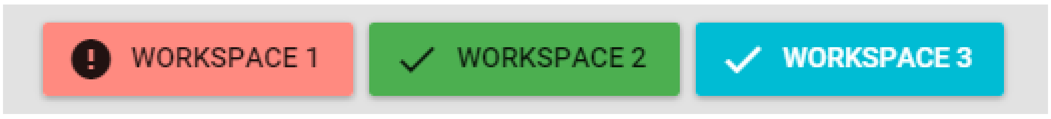


Figure 4.2: Workspace selector buttons with status

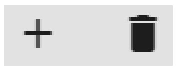
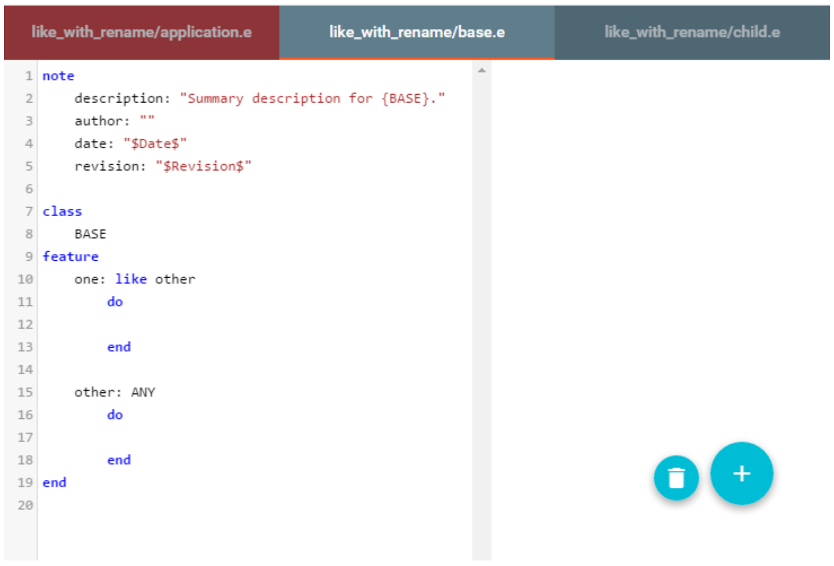


Figure 4.3: Buttons to create and delete workspaces

Multiple Files. Multiple files within one workspace are shown in a tabbed interface as can be seen in Figure 4.4 on the following page. Files that have

[2] http://eiffelweb.github.io/demo/

errors have it's corresponding tab marked with a red color, which can also be seen in the aforementioned picture. Further, there are "*new file*" and "*remove file*" button in blue situated in the lower right corner.



```
note
    description: "Summary description for {BASE}."
    author: ""
    date: "$Date$"
    revision: "$Revision$"

class
    BASE
feature
    one: like other
        do

        end

    other: ANY
        do

        end
end
```

Figure 4.4: Tabbed interface with error indicator and new & delete file buttons

Import and Export. Code can be imported by dragging files and folders[3] over the application. These files will be added to the currently active workspace. The original file name will be displayed in a file's corresponding tab. This is seen in Figure 4.4, where one folder containing three *Eiffel* files (extension `*.e`) has been imported. It is also possible to import files by opening a traditional "*Open File*"-dialog through the import button, displayed in Figure 4.5.

Also pictured is a button to export all files inside the currently active workspace. A zip file will be generated in memory and downloaded automatically. This is especially useful if several changes have been made or the student is working with newly created files directly inside the application.

Figure 4.5: Left: Import files, Right: Export files

Parsing. A file is automatically parsed after being imported or created. Further, the file is parsed several hundred milliseconds after the last keypress has been made inside the editor. The delay[4] has been chosen such that a moderately fast typist will not trigger the parser between subsequent keypresses. Triggering the parser after every keypress, albeit the parser is very fast for small files, would have degraded the experience for fast typists. This way, code can be

[3]Importing folders is a Chrome-exclusive feature
[4]`app.ts/EiffelFile.updateCode`

typed fluently and parser output is given immediately after the student stops typing.

Successful and unsuccessful parses trigger different events, to which other parts in the application can subscribe. One such subscriber sets a file's filename dynamically to the name of the parsed class, unless the file has been imported, in which case the file always reflects the original filename.

The parser integrated in the application also supports multipe classes in one file, a deliberate deviation from *EiffelStudio*. This is a convenience for working with multiple small classes.

The parser error message is displayed below the editor.

AST visualization. The AST hierarchy of the AST node that corresponds to the current cursor position is displayed to the right of the editor, as pictured in Figure 4.1 on page 28. This feature is disabled while the file cannot be parsed.

Chapter 5

Conclusions

5.1 Conclusions

The project as it stands can serve as a platform upon which to build a fully-featured web environment for an automated Eiffel teaching assistant. The basic structure has been laid out and implemented, along with a small demo page showcasing a small part of what is going on under the hood.

Working with these bleeding-edge technologies has been highly interesting, although somewhat exhausting at times. There were the occasional odd effects that would disappear after a few reloads, or still unfinished documentations. *TypeScript* in general was a huge boon despite its compiler sometimes indicating errors when there shouldn't be any, or not giving error messages in some situations but it does for the same situation in a different place. Thanks to Chrome's debugging abilities, though, such bugs could mostly be quenched quite quickly.

Also, this project has shown me yet again, that making predictions as to how long a feature will take to implement is extremely difficult, and it's very easy to underestimate the time it will take. Not necessarily because one is stuck, mostly because more cases one did not consider before will reveal themselves, which end up in consuming a large amount of time.

5.2 Future Work

There are many areas that can be expanded on and improved. First and foremost, the analyzer is to be completed. Of course, implementing the entire spectrum of the Eiffel language was a huge undertaking to begin with and there's still a lot missing. For example: correctly resolving selects, precursors and conversion semantics. The grammar for inline agents is still missing. Tuple support might not be straight forward forward because they take an arbitrary amount of type arguments and much more. The UI could be improved to expose more details, better error messages and better usability. The API used by the Traffic library to draw to the screen could be intercepted and translated into `<canvas>`

calls to make Traffic run in the browser.

Bibliography

[1] Tarjan, R. E. *Depth-first search and linear graph algorithms.* SIAM Journal on Computing 1 (2): 146–160, 1972.

[2] ECMA-367. *Eiffel: Analysis, Design and Programming Language.* 2nd Edition, 2006.