



Assignment of master's thesis

Title:	WooWoo Language Server
Student:	Bc. Michal Janeček
Supervisor:	Ing. Tomáš Kalvoda, Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

The aim of this thesis is to implement a Language Server Protocol (LSP) for the WooWoo language, which is used at FIT CTU in Prague for creating study materials in several courses.

1. Familiarize yourself with the syntax of the WooWoo language and the document templates for creating study materials.
2. Familiarize yourself with the domain of language servers and LSP.
3. Design a WooWoo language server that provides autocompletion, go-to-definition, syntax error detection and other common features.
4. Implement and test the language server designed in the previous task.
5. Integrate the language server into a VSCode extension and ensure that it is easy to use and install.

Master's thesis

WOOWOO LANGUAGE SERVER

Bc. Michal Janeček

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Tomáš Kalvoda, Ph.D.
May 7, 2024

Czech Technical University in Prague
Faculty of Information Technology

© 2024 Bc. Michal Janeček. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Janeček Michal. *WooWoo Language Server*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

Acknowledgments	viii
Declaration	ix
Abstract	x
Introduction	1
1 The WooWoo Language	3
1.1 Introduction to WooWoo	3
1.1.1 The Need for WooWoo	3
1.1.2 Inspired by PreTeXt	4
1.1.3 Present and Future	4
1.2 Syntax and Semantics	5
1.2.1 Indentation and Whitespace	5
1.2.2 Document	5
1.2.3 Document Part	5
1.2.4 Include Statement	6
1.2.5 Meta Block	6
1.2.6 Wobject	6
1.2.7 Block	7
1.2.8 Text Block	7
1.2.9 Outer Environments	7
1.3 The Building Blocks of WooWoo	8
1.3.1 Dialect	9
1.3.2 Templates	12
1.3.3 Project	13
2 Language Server Protocol	15
2.1 The Need for LSP	15
2.2 LSP Foundations	16
2.2.1 JSON-RPC	17
2.2.2 LSP Specification	17
2.2.3 Document Synchronization	19
2.2.4 Workspace Synchronization	19
2.2.5 Language Features	19
3 Requirements	23
3.1 Functional Requirements	23
3.2 Non-functional Requirements	24

4	Design & Technologies	26
4.1	Overview of Existing Language Servers	26
4.1.1	TypeScript Language Server	27
4.1.2	Rust-Analyzer	27
4.1.3	Bash Language Server	27
4.2	High-Level Language Server Architecture	27
4.3	Parser	28
4.3.1	Existing WooWoo Parsers	28
4.3.2	Parser Generators	29
4.3.3	Parser Choice	30
4.4	Analysis Engine	30
4.4.1	Technology	31
4.4.2	Architecture Overview	32
4.4.3	WooWooAnalyzer	33
4.4.4	Parser	35
4.4.5	WooWooDocument	36
4.4.6	DialectManager	36
4.4.7	Feature Components	37
4.4.8	Enabling Python Integration	39
4.5	Communication Layer	40
4.5.1	Pygls	40
4.6	VSCoDe Extension Integration	42
5	Implementation	43
5.1	Parser Implementation	43
5.1.1	Defining Grammar Rules	43
5.1.2	External Scanner	46
5.1.3	Parser Integration	48
5.2	Analysis Engine Implementation	48
5.2.1	Build Configuration with CMake	49
5.2.2	WooWooAnalyzer	50
5.2.3	DialectManager	51
5.2.4	Reference Mapping in DialectedWooWooDocument	52
5.2.5	Feature Components	53
5.2.6	Python Bindings	55
5.3	Communication Layer Implementation	56
5.3.1	Utility Functions	56
5.3.2	WooWooLanguageServer Implementation	56
5.3.3	Feature Registration and Launching the Server	57
5.4	VSCoDe Extension Implementation	58
5.4.1	Extension Manifest	58
5.4.2	Starting the Language Server Process	59
5.5	Future Work	60
5.5.1	Tree-sitter Grammar	60
5.5.2	Working with Woofiles	60
5.5.3	Expanding Current Features	61
5.5.4	Adding New Features	61

6	Distribution	62
6.1	Distributing <code>wuff</code>	62
6.1.1	Cross-Platform Wheel Building	63
6.1.2	Supported Systems	64
6.2	Extension Distribution	65
6.2.1	VSCoDe Marketplace	65
6.2.2	Open VSX Registry	65
7	Testing	67
7.1	Grammar Tests	67
7.1.1	Test Corpus	67
7.1.2	Manual Grammar Validation with Existing Projects	68
7.2	User Testing	69
7.2.1	Methodology	69
7.2.2	Results	70
7.3	Performance Tests	70
7.3.1	Datasets	70
7.3.2	Methodology	70
7.3.3	Replication	71
7.4	Unit Testing <code>wuff</code>	72
7.4.1	Testing Framework	72
7.4.2	Structure	72
7.4.3	Use in Wheel-Building Pipeline	73
8	Conclusion	74
A	Installation Guide	75
B	Feature Demonstration	78
B.1	Diagnostics	78
B.2	Completion	80
B.3	Go To Definition	81
B.4	File Rename	82
B.5	Folding Range	83
B.6	Rename Symbol	84
B.7	Semantic Tokens	85
B.8	Hover	85
B.9	Find All References	85
	Contents of the Enclosed Medium	90

List of Figures

1.1	Illustration of syntactic variations in expressing the Pythagorean identity equation using PreTeXt, LaTeX, and Write Only Once, Write Only Once (WooWoo). . . .	4
1.2	Examples of Inner Environment Forms	8
1.3	Difference between classic and fragile outer environments	9
1.4	The structure of the WooWoo ecosystem, its dialects, and the specific templates designed for the FIT-Math dialect.	10
1.5	Output generated using the fit-pdf template.	12
1.6	Output generated using the fit-html template.	13
1.7	Output generated using the fit-knowledge template.	13
1.8	Output generated using the fit-words template.	14
1.9	Example folder structure of a WooWoo project	14
2.1	A comparison between the pre-Language Server Protocol (LSP) fragmented setup and the post-LSP unified approach, adapted from the conceptual illustration provided in [11].	16
2.2	Example of communication between a tool and a language server during a routine editing session [12].	17
2.3	Communication between LSP client and language server upon user renaming files.	20
2.4	Demonstration of the <i>Find All References</i> feature in Visual Studio Code (VSCoDe) for a WooWoo document part label.	21
2.5	Demonstration of the <i>Completion</i> feature: Suggestion and placeholder insertion .	22
4.1	Key Components of the WooWoo Language Server	28
4.2	Comparison of average workspace initialization times between Python and C++ implementations across projects containing sources for various subjects at Faculty of Information Technology, Czech Technical University in Prague (FIT CTU). For further description and sizes of the datasets, see Table 7.1.	32
4.3	Feature-wise comparison of average durations, highlighting the time required for <i>Hover</i> and <i>Completion</i> features in Python and C++ implementations. Analysis conducted on sources for the BI-MA1 course.	33
4.4	Comparison of average durations for processing <i>Semantic Tokens</i> in Python and C++. The analysis encompasses all projects referenced in Figure 4.2.	34
4.5	The core classes present in the wuff analysis engine.	35
4.6	Components of the Dialect Class	38
5.1	Directory tree of the wuff analysis engine	49
5.2	Obtaining the description of the <i>Question</i> environment defined by the FIT-Math dialect.	54
6.1	GitHub Actions pipeline jobs for cross-platform wheel generation and Python Package Index (PyPI) upload.	63
6.2	Summarization of the steps that cibuildwheel takes on each platform. [53] . . .	64
6.3	The WooWoo extension listed in the VSCoDe Marketplace, accessible from within the editor.	65

7.1	Structure outline of the corpus folder	69
7.2	Unit Tests Directory Structure	72
B.1	Demonstration of the <i>Diagnostics</i> feature, specifically focusing on error reporting based on the detection of Tree-sitter MISSING nodes.	78
B.2	Demonstration of the <i>Diagnostics</i> feature, specifically focusing on error reporting based on the detection of Tree-sitter ERROR nodes.	79
B.3	Demonstration of the <i>Completion</i> feature for inner environments capable of referencing, focusing on body completion.	80
B.4	Demonstration of the <i>Go to Definition</i> feature, illustrating how users are directed to labeled structures within the FIT-Math dialect.	81
B.5	Demonstration of the <i>Rename Files</i> feature, illustrating the process from initiation to review of proposed changes.	82
B.6	Demonstration of the <i>Folding Range</i> feature, which allows users to fold document parts, blocks, and wobjects.	83
B.7	Demonstration of the <i>Rename Symbol</i> feature, enabling project-wide refactoring of references.	84

List of Tables

4.1	DialectManager Class Public Interface	37
6.1	Supported Systems for the wuff Package	64
7.1	Testing Datasets Used for Performance Measurements	70

List of code listings

1.1	Start of a document part in WooWoo	6
1.2	Use of include statements in WooWoo	6
1.3	An example <i>Question</i> wobject from the FIT-Math dialect	6
1.4	Example of a <code>text_block</code> containing <code>math_environments</code>	7
1.5	Simplified excerpt illustrating the structure of the FIT-Math WooWoo dialect	11
2.1	Example of composing basic JavaScript Object Notation (JSON) structures in LSP, represented in TypeScript.	18
4.1	Example of <code>WooWooAnalyzer</code> use within C++.	34
4.2	Use of the <code>WooWooAnalyzer</code> within Python, equivalent to the sequence in Listing 4.1	40
4.3	Instantiating <code>LanguageServer</code> with a <i>TextDocumentCompletion</i> LSP feature. [43]	41
4.4	Example of decorated feature functions used for registration and delegating requests to the <code>WooWooLanguageServer</code>	41
5.1	The rule in <code>tree-sitter-woowoo</code> capturing wobject.	44

5.2	Setting right associativity in the block rule to resolve ambiguity.	45
5.3	Failed parser generation due to ambiguity in the grammar.	45
5.4	Structure of the <code>scanner.cc</code> file	48
5.5	Redirecting Requests to Feature Components	50
5.6	Finding projects folder by scanning for Woofiles.	51
5.7	Snippet from the <code>Dialect</code> class's deserialization method.	52
5.8	Query (S-expression) for finding key:value pairs within <code>meta_blocks</code>	53
5.9	Verifying the presence of a <code>meta_block</code> at feature execution position.	54
5.10	Project-wide search for references.	55
5.11	Defining Python bindings.	55
5.12	Converting Position from the <code>wuff</code> module to Position from <code>lsprotocol.types</code>	56
5.13	Typical method of <code>WooWooAnalyzer</code> , constructing parameters for <code>wuff</code> , and then returning the result in pygls-compatible type.	56
5.14	The structure of the <code>server.py</code> file. Instantiating <code>WooWooLanguageServer</code> , registering methods, and providing the <code>start</code> function for client to launch the server.	57
5.15	Defining the WooWoo Language and Grammar Contributions within the Extension Manifest.	58
5.16	Snippet from the <code>createServer</code> function showing how the server and <code>LanguageClient</code> are configured and instantiated.	59
7.1	An example of a Tree-Sitter WooWoo test case (<code>simple_nesting.txt</code>).	68
7.2	Testing <i>Completion</i> suggestions (snippet from <code>test_complete.py</code>)	72

I would like to express my deepest thanks to my supervisor, Ing. Tomáš Kalvoda, Ph.D., for the incredible opportunity to contribute to the WooWoo project. His dedication to the faculty is truly inspiring, and I am honored to have been involved in his meaningful work. His guidance and support throughout this thesis have been invaluable.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Section 2373(2) of Act No. 89/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the "Work"), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work and for any purpose (including use for profit). This authorisation is unlimited in time, territory and quantity.

In Prague on May 7, 2024

Abstract

This thesis enhances the authoring experience for the WooWoo language, extensively used at the Faculty of Information Technology at CTU to create mathematical study materials.

It offers a comprehensive exploration of WooWoo’s history, vision, syntax, and semantics, and introduces the concept of ‘WooWoo Dialects’—a crucial abstraction for developing language tools for WooWoo while preserving its domain-agnostic nature.

The core contribution of this work is the design and implementation of a Language Server Protocol (LSP) developed specifically for WooWoo, featuring a Tree-sitter grammar that underpins an efficient parser. This Language Server is equipped with several features, including syntax error detection, auto-completion, and go-to-definition.

Furthermore, the thesis details the integration of the Language Server into a Visual Studio Code extension and its deployment on popular marketplaces, ensuring easy accessibility and installation. As of the publication of this thesis, the developed tools are already being actively used by WooWoo authors.

Keywords WooWoo, Language Server Protocol, Language Server, Tree-sitter grammar, cross-language development, VSCode extension, developer tools

Abstrakt

Tato diplomová práce vylepšuje proces tvorby obsahu v jazyce WooWoo, který je hojně využíván na Fakultě informačních technologií ČVUT ke tvorbě matematických studijních materiálů.

Práce poskytuje ucelený přehled historie, vize, syntaxe a sémantiky jazyka WooWoo a zavádí koncept „WooWoo dialektů“—klíčovou abstrakci pro vývoj jazykových nástrojů pro WooWoo, při zachování jeho doménově agnostické povahy.

Hlavním přínosem této práce je návrh a implementace jazykového serveru pro WooWoo, který implementuje protokol Language Server Protocol (LSP) a zahrnuje Tree-sitter gramatiku, tvořící základ pro efektivní parser. Vytvořený jazykový server nabízí několik funkcí, včetně detekce syntaktických chyb, auto-doplnění a funkce přechodu k definici.

Práce dále podrobně popisuje integraci jazykového serveru do rozšíření pro Visual Studio Code a jeho distribuci na populárních platformách, což zajišťuje snadnou dostupnost a instalaci. V době zveřejnění této práce jsou již tyto nástroje aktivně využívány autory ve WooWoo.

Klíčová slova WooWoo, protokol jazykového serveru, language server protocol, jazykový server, Tree-sitter gramatika, vývoj napříč jazyky, rozšíření pro VSCode, vývojářské nástroje

Acronyms

ANTLR ANOther Tool for Language Recognition.

API Application Programming Interface.

AST Abstract Syntax Tree.

CI Continuous Integration.

CST Concrete Syntax Tree.

DSL Domain-Specific Language.

ERB Embedded Ruby.

FIT CTU Faculty of Information Technology, Czech Technical University in Prague.

IDE Integrated Development Environment.

JSON JavaScript Object Notation.

JSON-RPC JavaScript Object Notation - Remote Procedure Call.

LSP Language Server Protocol.

PDF Portable Document Format.

PyPI Python Package Index.

RLS Rust Language Server.

RPC Remote Procedure Call.

TCP Transmission Control Protocol.

URI Uniform Resource Identifier.

VSCoDe Visual Studio Code.

Wasm WebAssembly.

WooWoo Write Only Once, Write Only Once.

XML eXtensible Markup Language.

YAML YAML Ain't Markup Language.

Introduction

Write Only Once, Write Only Once (WooWoo) has become the preferred standard for producing math study materials at Faculty of Information Technology, Czech Technical University in Prague (FIT CTU). Created in 2017 by Ing. Tomáš Kalvoda, Ph.D., a deputy head at the Department of Applied Mathematics, WooWoo's core strength lies in its content-centric approach. Authors define the structure and meaning of their material once, and WooWoo's output generators produce consistent presentations across a wide variety of outputs.

WooWoo's positive impact is evident in the enthusiasm of students, who appreciate the availability of materials in formats ranging from classic PDFs to interactive HTML and Anki cards. This enthusiasm, reflected in post-semester surveys¹, has even inspired community contributions such as Bc. Matěj Frnka's Mind Map template [1], adding another output generator alongside the existing ones.

Necessity of Language Tooling for WooWoo

While WooWoo offers significant advantages for authoring study materials, its potential for wider adoption is hampered by the lack of robust language tooling. For a language community to thrive and gain acceptance beyond its initial creators, essential tools like syntax error detection, go-to definition, and code completion are crucial. Without these, new authors face a steep learning curve, and even experienced authors contend with ongoing inefficiencies.

Past efforts, such as Bc. David Straka's Wootom extension [2] for the since-discontinued Atom editor, provided valuable support. However, its editor-specific nature and the editor's discontinued status severely limit its utility, highlighting the need for a more flexible, long-term solution for WooWoo language tooling.

Introducing the Language Server Protocol

To address the need for WooWoo language tooling that surpasses the limitations of single-editor solutions, this thesis leverages the Language Server Protocol (LSP), which was introduced in 2016 by Microsoft, Red Hat, and the Eclipse Foundation's Che project. The LSP standardizes the way language-aware tools, like Integrated Development Environments (IDEs), communicate with language smartness providers (Language Servers), which are equipped with a deep understanding of specific languages.

This standardized communication protocol allows a single WooWoo Language Server to integrate seamlessly with a variety of editors. The development of a WooWoo Language Server using

¹As evidenced by frequent mentions in the CTU Surveys at <https://anketa.is.cvut.cz/html/anketa/results>

the LSP forms the central focus of this thesis, aiming to provide the basis for efficient, accessible and editor-agnostic language support.

Goals & Summary

The goal of this thesis is the design, implementation, and testing of a Language Server for WooWoo, with the ultimate goal of integrating it into a publicly available Visual Studio Code (VSCode) extension. This extension is intended to simplify the setup and usage process for new authors and to showcase the functionality of the developed language server.

Chapter 1 provides a comprehensive overview of WooWoo’s history, its current state, syntax, semantics, and the tools available. It also introduces the concept of “WooWoo Dialects,” an important abstraction designed to ensure that the tooling can be adapted to specific domains without being restricted to any single one in particular. Chapter 2 covers the necessary technical background of the LSP, setting the foundation for subsequent developments. Chapter 3 outlines the specific requirements for the language server.

Chapters 4 and 5 cover the entire development process, from the initial design and selection of suitable technologies to meet the performance and usability needs of the language server, to the implementation of the server itself. Chapter 5 also includes a section on future work, discussing potential enhancements and extensions.

Establishing distribution channels and ensuring that the language tooling is functional and accessible for all potential users are addressed in Chapters 6 and 7.

..... Chapter 1

The WooWoo Language

Write Only Once, Write Only Once (because why write it twice?)

1.1 Introduction to WooWoo

Current tools for authoring structured study materials often constrain authors to specific formats or embed presentation-focused instructions into the content itself. WooWoo is a markup language designed to overcome these limitations, focused on being content-centric while being extendable and not locked on to a specific domain.

1.1.1 The Need for WooWoo

WooWoo was created in 2017 by Tomáš Kalvoda, a deputy head at the Department of Applied Mathematics with extensive experience in teaching and authoring math textbooks at FIT CTU. Traditionally, departmental textbooks were written in LaTeX for generating Portable Document Format (PDF) versions.

Limitations of LaTeX

While LaTeX excels at typesetting and producing visually consistent PDFs, it presents challenges in addressing modern learning preferences. Students increasingly favor dynamic and interactive formats over static PDFs, desiring options like web-based textbooks or mind maps to personalize their learning experience.

Furthermore, LaTeX's focus on layout means the source documents contain styling instructions intertwined with the content. This limits its flexibility in generating diverse outputs and would introduce complexity when targeting multiple formats.

WooWoo Approach

While technically possible, generating fundamentally different outputs from a single LaTeX document is impractical. LaTeX's primary purpose is to facilitate high-quality typesetting for written materials. Accommodating diverse outputs would require extensive markers and format-specific instructions, leading to increased complexity and reduced maintainability of the source files.

To address these limitations, WooWoo offers a cleaner, content-oriented approach. WooWoo documents prioritize describing the structure and meaning of the content, not its visual presentation. This separation of concerns allows for easier generation of different outputs from the

same source. The same source code can be used to generate multiple views of the same material, without the need to clutter the source with instructions tailored to specific output generators.

1.1.2 Inspired by PreTeXt

WooWoo shares its vision with several projects aimed at simplifying the creation of educational materials. A prominent example is the PreTeXt project [3], previously known as MathBook XML. Embracing the “Write Once, Read Anywhere” philosophy, PreTeXt has significantly influenced WooWoo’s development. It allows authors to craft content in a singular source format, from which various outputs like interactive websites, static PDFs, and ePubs can be generated. This flexibility was a key inspiration during WooWoo’s initial development phase.

Despite its innovative approach, PreTeXt faces challenges, particularly in its user experience for authors. The necessity to write documents in eXtensible Markup Language (XML) [4] format adds a layer of complexity and can diminish the readability of extensive texts. In response, WooWoo introduces a streamlined syntax that capitalizes on whitespace and indentation to enhance document clarity and author-friendliness. This is especially advantageous for content dense with mathematical expressions. Unlike the cumbersome markers required in LaTeX or PreTeXt, WooWoo simplifies the process by employing indented blocks to denote equations and similar structured elements, thereby eliminating the need for explicit markers and making the authoring process more intuitive for math-intensive documentation, as can be seen on Figure 1.1.

```
1 The Pythagorean identity equation:
2 <me>
3   \sin^2(x) + \cos^2(x) = 1
4 </me>
```

(a) PreTeXt Representation [4]

```
1 The Pythagorean identity equation:
2 \begin{equation}
3   \sin^2(x) + \cos^2(x) = 1
4 \end{equation}
```

(b) LaTeX Representation

```
1 The Pythagorean identity equation:
2
3   \sin^2(x) + \cos^2(x) = 1
```

(c) WooWoo Representation

■ **Figure 1.1** Illustration of syntactic variations in expressing the Pythagorean identity equation using PreTeXt, LaTeX, and WooWoo.

1.1.3 Present and Future

WooWoo offers inherent flexibility, with dialects providing a powerful tool for tailoring its syntax and vocabulary to specific domains. Currently, the FIT-Math dialect is the foundation of the WooWoo ecosystem, enabling the creation of mathematical textbooks and supporting a variety of output formats (traditional textbooks, interactive websites, flashcards, etc.).

Tools supporting the WooWoo ecosystem are continually evolving, including new output generators (templates) and the Language Server developed as part of this thesis. Notably, the *WooWoo Catalogue* (currently in its alpha version) automatically analyzes projects in the FIT-Math dialect and creates a centralized index of entities (*Definitions, Lemmas, Exercises*, etc.). This enables authors to easily search across projects by label and assists students in finding definitions and concepts across mathematical subjects [5].

The future of WooWoo is promising. A core goal of this thesis is to provide tools that will make authoring in WooWoo easier. This has the potential to attract new contributors interested in creating custom dialects, expanding the WooWoo ecosystem and its applications beyond the current focus on mathematics.

1.2 Syntax and Semantics

This section provides an overview of WooWoo’s syntax and semantics, focusing on the practical use of its constructs rather than a full presentation of formal grammar rules. Examples and explanations will illustrate how these constructs are used to organize and define content. For a complete formal specification, refer to the WooWoo Tree-sitter grammar¹, which is part of this thesis. The grammar and this section is sourced from and paraphrases from the WooWoo Specs document [6] as well as numerous discussions with the core developer of WooWoo, Tomáš Kalvoda.

Importantly, apart from the `include` statement, WooWoo itself does not define keywords. These are determined by specific WooWoo dialects. The examples in this section use the currently popular FIT-Math dialect. This means that dialect-specific keywords are used for construct names, but the syntax remains the same, more on dialects in Section 1.3.1.

Each construct in this section also includes its **formal name** (corresponding to the grammar rules) for clarity and easy referencing throughout the thesis.

1.2.1 Indentation and Whitespace

Inspired by the readability of Python, WooWoo utilizes indentation and whitespace to structure and separate content within a document.

- **Indentation:** 2-space indentation is used to organize components into hierarchies, indicating nesting within other elements.
- **Empty lines:** Serve as content separators and have two primary uses:
 - **Single empty line:** Separates elements within a structure.
 - **Two or more empty lines:** Signal a transition to a new block or structural element.

1.2.2 Document

A WooWoo document (`source_file`) is a single file with a `.woo` extension. It consists of a series of `document_parts`, `include` statements and optional `comments`. Comments are marked by the `%` character at the beginning of a line and are ignored during processing.

1.2.3 Document Part

A document part (`document_part`) is a top-level construct for organizing content within a WooWoo document. Start of a document part is marked by a dot, followed by a document

¹Available at: <https://gitlab.fit.cvut.cz/woowoo/lsp/tree-sitter-woowoo>.

part type, which has to start with capital letter, followed by a title (`.Type Title`). On the next line, an indented `meta_block` may follow. The body contains a series of `blocks` and `wobjects` and is terminated by a start of another `document_part`.

```
1 .Section Natural, Integer, and Rational Numbers
2   label: sec-NIR
3
4 Body.
```

■ **Code listing 1.1** Start of a document part in WooWoo

1.2.4 Include Statement

The include statement (`include`) allows authors to organize their work across multiple files by including the content from one file into another. It comprises the `.include` keyword, followed by a relative or absolute path to the file whose content is to be included. A line with this statement must not contain any other elements.

A common application is to create a separate file for each chapter of a textbook and then include these files in a root document to compile the entire textbook.

```
1 .include bi-ma1-01-real-numbers.woo
2
3 .include bi-ma1-02-functions.woo
4
5 .include bi-ma1-03-sequences.woo
```

■ **Code listing 1.2** Use of include statements in WooWoo

1.2.5 Meta Block

A meta block (`meta_block`) provides additional information associated with a WooWoo construct. Formatted in YAML Ain't Markup Language (YAML), it is positioned beneath the construct and indented relative to it. The specific content and structure of a meta block are entirely determined by the active dialect.

An example of its application is shown in Listing 1.1, where a `meta_block` is utilized to assign a label to a `document_part`. In Listing 1.3, a `meta_block` is used to provide various metadata about the *Question* wobject.

1.2.6 Wobject

A wobject (`wobject`) is an atomic environment, meaning it cannot contain other wobjects. Its declaration starts with a capitalized type name surrounded by a dot and colon (`.Type:`) and can optionally include a `meta_block`. The wobject's content is indented and consists of `blocks` separated by at least two consecutive empty lines. An example of `wobject` containing a single block is demonstrated in Listing 1.3.

```
1 .Question:
2   label: que:simple-addition
3   title: Simple addition
4   solution: 5
5
6   What is the sum of 2 and 3?
```

■ **Code listing 1.3** An example *Question* wobject from the FIT-Math dialect

1.2.7 Block

A block (`block`) serves as a container for organizing textual content and structural elements within a WooWoo document. It consists of a series of consistently indented `text_blocks` and `outer_environments`.

1.2.8 Text Block

A text block (`text_block`) is a series of consistently indented lines of text. A text line consists of plain text without special meaning and may include inline environments (`inner_environment`) and math environments (`math_environment`). Optionally, a `text_block` can be accompanied by a `meta_block`.

1.2.8.1 Inner Environments

Inner environments (`inner_environment`) are used to annotate segments of text by assigning each segment a specific type and, optionally, metadata. These environments can reference fields from a `meta_block` associated with the same `text_block` or any parent structure.

When searching for a field, the search begins in the innermost structure containing the inner environment, proceeding to parent structures in ascending order. The first matching field found is used, allowing fields to be overridden by redeclaring them in a descendant structure's `meta_block`.

There are three different forms of `inner_environment`, which are visually compared in Figure 1.2 and described below:

Verbose Form Text surrounded with quotation marks, followed by the type and optional metadata (providing a way to reference information in an accompanying `meta_block`).

Short Form Begins with a dot, followed by the type, a semicolon, and the environment body (which must contain no whitespace).

Shorthands WooWoo provides shorthanded versions using the `#` or `@` characters as placeholders for the *environment type*. The specific meaning of these shorthands is defined within each WooWoo dialect. For instance, in the FIT-Math dialect, `@` is used when referencing external web pages, instead of using the *reference inner_environment* in the verbose form.

1.2.8.2 Math Environment

Another tool for text annotation is the math environment (`math_environment`), indicated by the `$` symbol. Its content can span multiple lines. Importantly, the interpretation of this environment's content is entirely determined by the active WooWoo dialect. In the FIT-Math dialect, the environment is conveniently used for mathematical expressions and inline equations.

```
1 The set of real numbers along with the symbols  $+\infty$  and  $-\infty$ ,
2 i.e., the set  $\mathbb{R} \cup \{+\infty, -\infty\}$ , is called the extended
3 set of real numbers and it is denoted by the symbol  $\mathbb{eR}$ .
```

■ **Code listing 1.4** Example of a `text_block` containing `math_environments`

1.2.9 Outer Environments

Outer environments (`outer_environment`) are structural elements used for annotating larger portions of text within a WooWoo document. They begin with a dot, for classic environments, or an exclamation mark for fragile environments, followed by a lowercase *environment name* and

```
1 We will talk more about natural, integer and rational numbers
2 in Chapter .reference:sec-NIR.
```

(a) Text containing the **short** form of inner environment

```
1 "Pythagoras".reference.1 (570 -- 495 B.C.) was a Greek philosopher.
2   1:
3     link: https://en.wikipedia.org/wiki/Pythagoras
```

(b) **Verbose** form of inner environment, with additional information provided in a `meta_block`

```
1 "Pythagoras"@1 (570 -- 495 B.C.) was a Greek philosopher.
2   1:
3     link: https://en.wikipedia.org/wiki/Pythagoras
```

(c) **Shorthanded** form of an inner environment

■ **Figure 1.2** Examples of Inner Environment Forms

a semicolon (`.name:`). An outer environment may include a `meta_block`. The body of an outer environment is always indented and its structure depends on the environment type:

Classic The body of a classic outer environment consists of a `block`. Classic environments can be nested, allowing for complex hierarchical structures. (See Figure 1.3a for an example.)

Fragile The body of a fragile outer environment contains raw text. This text is not parsed as WooWoo content and is typically intended as input for other tools. The end is indicated by a decrease in indentation (dedent) or two or more empty lines. Fragile outer environments often include metadata to guide further processing (see Figure 1.3b).

Implicit Outer Environment

Outer environments can also be declared implicitly, meaning the environment name can be omitted. The specific *environment name* is determined by the active dialect. For example, the **FIT-Math** dialect designates the `equation` outer environment as implicit. This enhances the conciseness of math-focused documents and provides convenience for authors (see Listing 1.1c).

1.3 The Building Blocks of WooWoo

To leverage the options of the WooWoo ecosystem, it is important to understand the difference between *WooWoo syntax*, *WooWoo dialects*, and *dialect templates*. The relationships between these components are captured in Figure 1.4.

WooWoo syntax defines how the documents should be written. It is defined by a formal grammar and introduces language constructs like `wobjects` and `blocks`, described in Section 1.2.

WooWoo dialect can be thought of as a set of specific instances of constructs defined by the grammar. It is typically designed for a specific domain. For example, in a **Math** dialect, you might find predefined `wobjects` such as *Definition*, *Theorem*, and *Equation*, designed to make writing math-related content more straightforward.

```

1  .itemize:
2
3  .item:
4
5      We say that a formula  $F$  is "true under evaluation".notion  $v$ ,
6      if and only if  $v(F) = 1$ .
7
8  .item:
9
10     We say that a formula  $F$  is "false under evaluation".notion  $v$ ,
11     if and only if  $v(F) = 0$ .

```

(a) Example of use of nested classic outer environments.

```

1  !tikz:
2      filename: fig_diagram_relace_diag1p
3      options: 'd/.style={circle,draw}'
4
5      \draw(0,0)node[d](a){ $a$ } (1.5,0)node[d](b){ $b$ };
6      \draw (a) edge[->,thick,out=10,in=170] (b);
7      \draw (b) edge[->,thick,out=-170,in=-10] (a);

```

(b) Example of a tikz (FIT-Math) fragile environment with metadata intended for a template.

■ **Figure 1.3** Difference between classic and fragile outer environments

Template is an output generator operating with a specific *dialect*. An example could be the `fit-pdf` template, which generates PDF version of a textbook given a valid WooWoo project written in accordance with the `FIT-Math` dialect.

1.3.1 Dialect

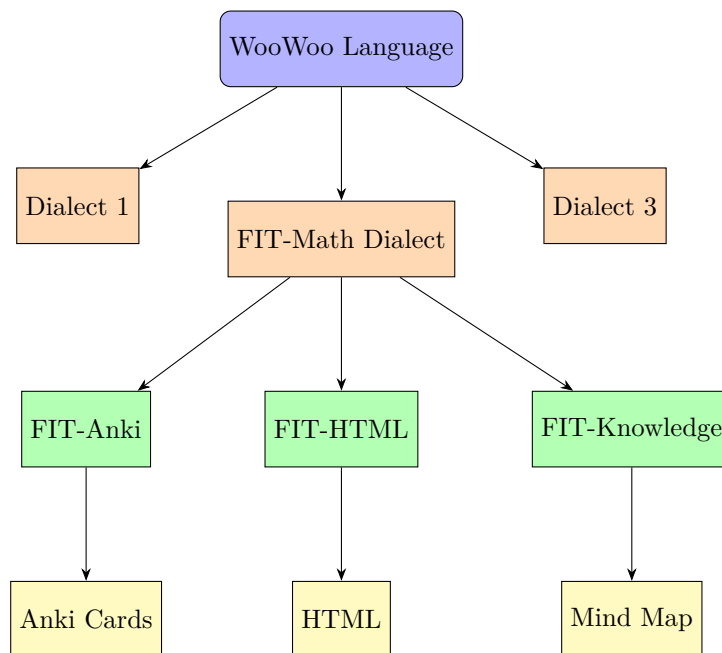
Apart from the `include` statement, the WooWoo language itself does not define any keywords. Instead, keywords are introduced through *dialects*, which extend the base WooWoo syntax. Authors are encouraged to write WooWoo documents following a specific dialect. While it is technically possible to create WooWoo documents without conforming to a dialect, this approach is impractical due to the absence of templates for output generation.

Each dialect must clearly define a list of instances available for use with WooWoo constructs. The dialect serves as a guideline for both document authors and template (output generators, further discussed in Section 1.3.2) creators. Authors refer to the dialect to understand the available options, while template creators ensure their templates can accommodate all elements defined within the dialect.

The term “dialect” in the context of WooWoo was first defined by this thesis, as previously, templates and dialects were not clearly separated and the term “template” was used to reference both, which lead to confusion.

1.3.1.1 Dialect Structure

A dialect can be represented by a file in YAML format. The formalization was created as part of this thesis. This format was chosen because it is easily readable by both computers and humans, and the format is already widely used within the WooWoo ecosystem. The following list provides a structural overview of a dialect, illustrating the types of elements it includes.



■ **Figure 1.4** The structure of the WooWoo ecosystem, its dialects, and the specific templates designed for the FIT-Math dialect.

name: A string representing the name of the dialect.

version_code: An integer denoting the version code of the dialect.

version_name: A string indicating the semantic versioning of the dialect. *Example:* "1.0.0".

description: A string providing a description of the dialect's purpose and usage.

implicit_outer_environment: Specifies the default outer environment for the dialect. *Example:* "equation".

document_parts: An array of objects defining the different parts of a document, such as chapters, sections, and subsections. Each object contains:

- **name:** The name of the document part.
- **description:** A brief description of the document part.
- **meta_block:** An object listing required and optional metadata fields that may be present in a document part's meta block.

wobjects: An array of objects representing different wobjects. Each object contains:

- **name:** The name of the wobject.
- **description:** A description of the wobject's purpose.
- **meta_block:** An object listing required and optional metadata fields that may be present in a wobject's meta block.

environments: An array of objects representing different environments. Each environment has the following properties:

- **name:** The environment name.

- **description:** What the environment is used for.
- **fragile:** A boolean flag indicating how the environment’s content should be processed.
- **meta_block:** An object listing required and optional metadata fields for the environment.
- **references:** (If applicable) Metadata fields to which the environment refers.

shorthands: Contains one object for each shorthand:

- **at:**
 - **description:** Explains what the shorthand does and how to use it.
 - **references:** (If applicable) Metadata fields to which the shorthand refers.
- **hash:** Same as the previous shorthand.

1.3.1.2 FIT-Math Dialect

The FIT-Math dialect, created by WooWoo’s core developer, Tomáš Kalvoda, is designed specifically for authoring mathematical documents. It is currently used for courses such as BI-DML, BI-LA1, BI-LA2, BI-MA1, and BI-MA2 at the FIT CTU, with plans to extend its application to additional courses. These courses collectively engage over a thousand students each semester.

The dialect’s formal specification is outlined in the `fit-math.yaml` file,² which was composed as part of this thesis after consulting with Tomáš Kalvoda, and various available sources [6, 7]. A simplified excerpt can be seen in Listing 1.5.

```

1  name: "FIT Math"
2  version_code: 1
3  version_name: "1.0.0"
4  description: |-
5    Dialect used in math courses on FIT CTU.
6  implicit_outer_environment: "equation"
7
8  document_parts:
9    - name: "Chapter"
10     description: "Top-level document part with the chapter's textual
11       content and list of sections."
12     ...
13  wobjects:
14    - name: "Definition"
15     description: "Mathematical environment containing a definition."
16     ...
17  environments:
18    - name: equation
19     description: "A simple math equation"
20     ...
21    - name: eqref
22     description: "Equation reference."
23     references:
24       - structure_type: outer_environment
25         structure_name: equation
26         meta_key: label
27    - name: cite
28     description: "Citation of a bibliography source."

```

■ **Code listing 1.5** Simplified excerpt illustrating the structure of the FIT-Math WooWoo dialect

²See the up-to-date version here: https://gitlab.fit.cvut.cz/woowoo/lsp/woowoo-language-server/-/blob/main/dialects/fit_math.yaml

1.3.2 Templates

Templates are the key to WooWoo’s versatility. In the context of WooWoo, a template serves as an output generator, processing a WooWoo project authored in a specific dialect. The primary requirement for a template is compatibility – it must be able to correctly interpret any valid document that conforms to the designated dialect. Beyond this, template authors have full freedom in terms of implementation and internal structure.

The term *template* is used because most of current WooWoo templates are using the Embedded Ruby (ERB)³ templating system to generate the output.

FIT-Math Templates

There are many templates available for the FIT-Math dialect. This allows authors to generate wide variety of outputs from just one source, leveraging the power of WooWoo.

What follows is a brief overview of currently used templates as of February 2024. It is worth noting that as per Tomáš Kalvoda, other templates are currently being developed, like `fit-slides` for generating lecture slides, but there is not a stable version yet.

fit-html Generates interactive websites with clickable elements and seamless reference navigation within the content.

fit-pdf Produces traditional, static PDFs in both desktop and mobile optimised versions.

fit-anki Creates Anki flashcards for memorization, extracting content from structures like *Definition*, *Lemma*, *Question* `wobjects`, and others.

fit-knowledge Generates an interactive website in mind-map format. It extracts key concepts (*Definitions*, *Theorems*, etc.) and their relationships from a WooWoo project, providing a visual representation to aid navigation and understanding. It was created by Matěj Frnka in 2022 [1].

fit-words Generates a single-page visual word cloud, highlighting the frequency of terms within the WooWoo project.

Refer to Figures 1.5, 1.6, 1.7, and 1.8 to view how different templates influence the output. Each figure illustrates the result of applying a unique template to the same source content, showcasing variations in presentation and style.

Definice 5.2 (Limita funkce / limit of a function): Mějme funkci $f: A \rightarrow \mathbb{R}$, hromadný bod $a \in \overline{\mathbb{R}}$ množiny A a bod $b \in \overline{\mathbb{R}}$. Funkce f má v bodě a limitu rovnou b , právě když pro každé okolí U_b bodu b existuje okolí U_a bodu a takové, že pokud $x \in U_a \cap A$ a $x \neq a$ pak $f(x) \in U_b$.

Formálně tento požadavek vyjadřuje formule

$$(\forall U_b) (\exists U_a) (\forall x \in (A \cap U_a) \setminus \{a\}) (f(x) \in U_b).$$

Tuto skutečnost symbolicky zapisujeme následovně

$$\lim_{x \rightarrow a} f(x) = b, \quad \text{případně} \quad \lim_a f = b.$$

■ **Figure 1.5** Output generated using the `fit-pdf` template.

³Learn more about ERB here: <https://docs.ruby-lang.org/en/2.3.0/ERB.html>

Definice 5.2 (Limita funkce / limit of a function) 📌

Mějme funkci $f : A \rightarrow \mathbb{R}$, hromadný bod $a \in \overline{\mathbb{R}}$ množiny A a bod $b \in \overline{\mathbb{R}}$. Funkce f má v bodě a limitu rovnou b , právě když pro každé okolí U_b bodu b existuje okolí U_a bodu a takové, že pokud $x \in U_a \cap A$ a $x \neq a$ pak $f(x) \in U_b$.

Formálně tento požadavek vyjadřuje formule

$$(\forall U_b) (\exists U_a) (\forall x \in (A \cap U_a) \setminus \{a\}) (f(x) \in U_b).$$

Tuto skutečnost symbolicky zapisujeme následovně

$$\lim_{x \rightarrow a} f(x) = b, \quad \text{případně} \quad \lim_a f = b.$$

■ **Figure 1.6** Output generated using the `fit-html` template.

■ **Figure 1.7** Output generated using the `fit-knowledge` template.

1.3.3 Project

A WooWoo project consists of source files (text files with a `.woo` extension) and a *Woofile*. Source files located in the same directory or its subdirectories as the *Woofile* are considered part of the project. This setup is important because files often refer to each other through *environments* and may `include` one another. Including files from outside the project directory is technically possible but is generally discouraged. An example of a WooWoo project's folder structure is shown in Figure 1.9.

Woofile

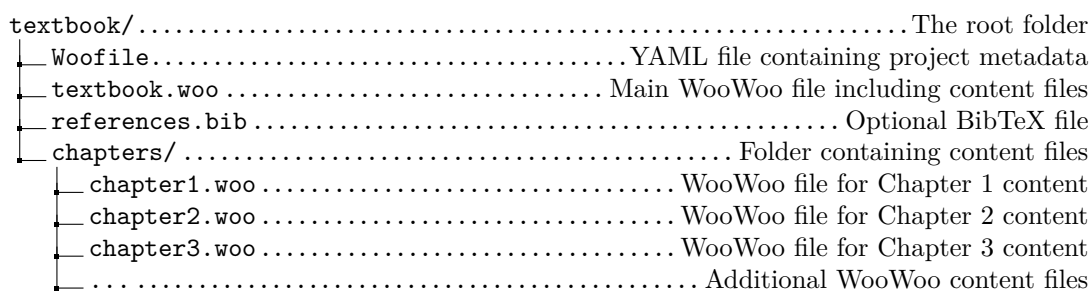
A *Woofile* is a YAML configuration file containing various data about the project, including:

- **root_file**: The main file of the project, usually including other files and serving as the entry point.
- **document**: Metadata about the project such as list of authors, title, subject code, semester, language, etc.
- **dialect specifics**: Specifies the dialect used in the project. As dialect formalization is a recent development (see Section 1.3.1.2), the *Woofile* currently defines the dialect inline. This might change in future versions.



■ **Figure 1.8** Output generated using the `fit-words` template.

- **bibtex (optional):** Specifies the location of the file that holds bibliographic references utilized in the project.



■ **Figure 1.9** Example folder structure of a WooWoo project

Language Server Protocol

“The Language Server protocol is used between a tool (the client) and a language smartness provider (the server) to integrate features like auto complete, go to definition, find all references and alike into the tool.” [8]

Having robust language support is crucial for a thriving language ecosystem. Historically, providing advanced language-specific tooling often required customized solutions for each individual tool [9]. People like to use all kinds of different editors and IDEs, and having each of them support a language required a significant development effort and maintenance. This creates an obstacle especially for small and starting projects like WooWoo.

To address these challenges, the LSP emerged as a solution. It defines a standardized communication mechanism between language-aware tools (editors, IDEs, etc.) and language servers. A language server possesses in-depth knowledge of a specific programming language or structured text format. By adhering to the LSP, a single language server can provide rich language support to a wide range of compatible tools.

This chapter introduces the LSP, exploring its underlying principles, architectural design, and the key features it defines.

2.1 The Need for LSP

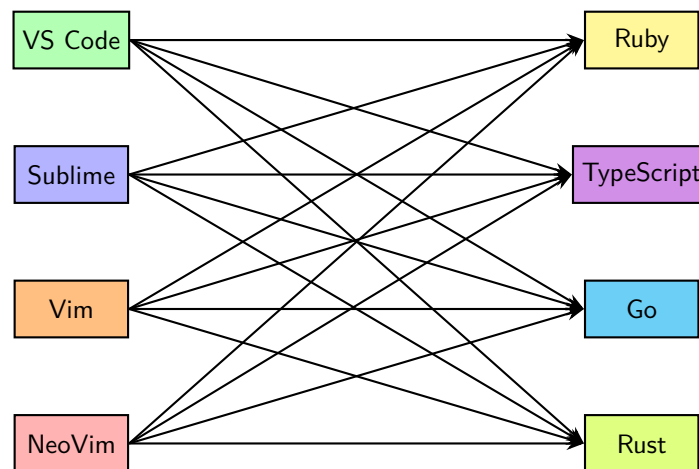
Traditional practice involved building highly customized, language-specific plugins for each editor to provide features like syntax highlighting, code completion, and navigation. Since there was no unified interface, supporting M languages in N editors led to $M \times N$ complexity. The LSP addresses this by introducing a standardized interface for both clients and servers, effectively reducing the complexity to $M + N$ [9, 10]. Figure 2.1 shows a comparison of the setup before and after the introduction of LSP [11].

“The idea behind the Language Server Protocol is to standardize the protocol for how tools and servers communicate, so a single Language Server can be re-used in multiple development tools, and tools can support languages with minimal effort” [12].

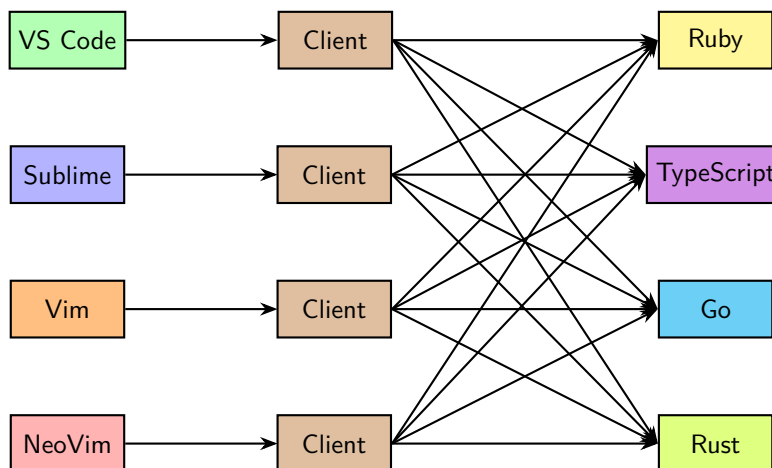
The LSP resulted from a collaboration between the Eclipse Che project, Microsoft, and Red Hat. Initially supported by Microsoft’s Visual Studio Code and the Eclipse Che IDE, it was released publicly in 2016 [13]. Rapid adoption by other tools and providers followed.

As of February 2024, nearly every major programming language has its own language server, a tool implementing the LSP. Many languages boast multiple alternative language server implementations, often written in different programming languages. LSP adoption is also widespread among IDEs, with most major IDEs providing support. The official Microsoft LSP website lists

over 30 different editors that implement the protocol [14].



(a) Before the introduction of LSP, each editor required a distinct plugin for every programming language.



(b) With the adoption of LSP, editors utilize a unified client layer that can interface with any language server.

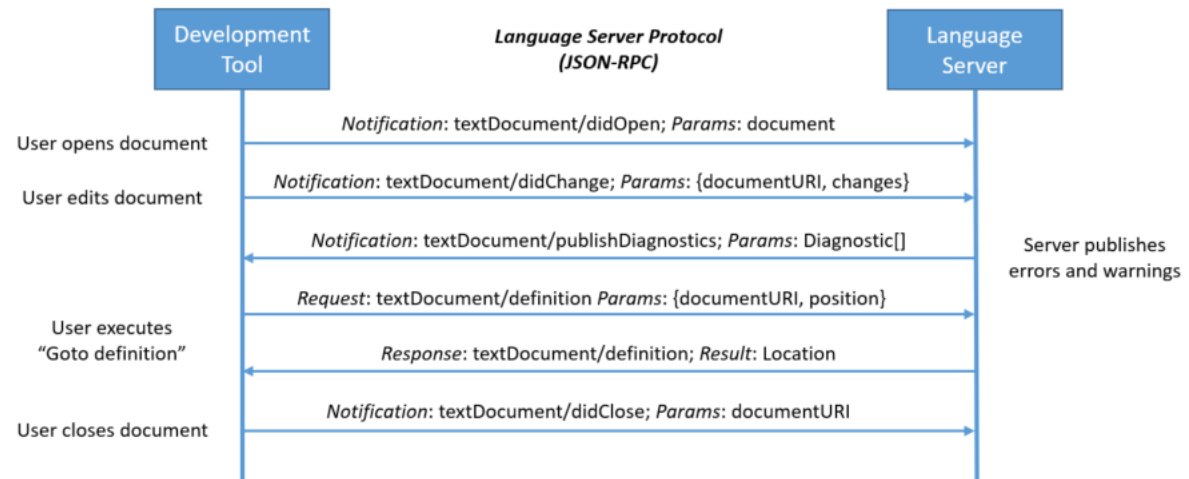
■ **Figure 2.1** A comparison between the pre-LSP fragmented setup and the post-LSP unified approach, adapted from the conceptual illustration provided in [11].

2.2 LSP Foundations

The LSP employs JavaScript Object Notation - Remote Procedure Call (JSON-RPC) as its underlying communication protocol. A language server that implements the LSP is typically initiated by the IDE or other development tool at the beginning of a coding or editing session as a separate local process. It is common for a user to work with multiple languages simultaneously, leading to the operation of several distinct language server processes. Although theoretically possible, running a language server remotely would be impractical due to the potential network latency. [12]

Communication is initiated with an *Initialize* request from the tool to the language server. Upon successfully processing this request, the language server should be prepared to handle subsequent requests for language support. The client notifies the server of every significant

user action, such as document changes, deletions, or action executions, requiring the server to continually update its internal state to provide accurate responses. An illustration of this communication process is depicted in Figure 2.2. [12]



■ **Figure 2.2** Example of communication between a tool and a language server during a routine editing session [12].

2.2.1 JSON-RPC

JSON-RPC is a simple, stateless protocol designed to facilitate Remote Procedure Calls (RPCs). It leverages JavaScript Object Notation (JSON) as the data format for communication, ensuring compatibility between different software components, even if they are implemented in different programming languages. [15]

RPC is a programming mechanism that allows a program to seamlessly execute a procedure (or subroutine) in a different address space. This could be on a remote machine or, as with the LSP, in a separate process on the same computer. RPC mechanisms abstract away the complexities of inter-process communication. [16]

The communication in JSON-RPC relies on the following structured message types [15]:

Request A JSON object sent from a client to a server to invoke a method. The Request object must contain the following fields: **method** (a *string* containing the name of the method to be invoked), **params** (a structured value holding the parameters for the method), and **id** (an identifier that matches the response with the request; it can be a *number*, *string*, or *null*).

Response A JSON object sent from a server to a client in reply to a Request. The Response object includes **result** (the data returned by the invoked method) in case of successful execution, or **error** (an object describing the error) if the method failed or couldn't be executed. Every Response must also contain the **id** field, matching it to the original Request.

Notification A special kind of Request sent from a client to a server or vice versa, intended to inform about events. Notifications are similar to Requests but do not have the **id** field, as they do not require a Response.

2.2.2 LSP Specification

The LSP Specification, maintained by Microsoft, defines the set of JSON-RPC messages used in the protocol. It outlines the types of requests a client can make, the responses a server should

provide, and the notifications used. As of March 2024, the latest version is 3.17, and it is the version used in this thesis. This section and its subsections, unless otherwise noted, are sourced from the specification [8].

The specification is designed to be flexible, with numerous configuration options for both the client and server. Capabilities are exchanged during the initialization phase. Neither clients nor servers are required to support all defined features, as some might not even be applicable to a given language. Most language server projects list their implemented features in the project description.

The following subsections provide an overview of the core message categories and structures defined by the specification.

2.2.2.1 Basic JSON Structures

The LSP Specification defines various basic JSON structures that are reused as building blocks within different requests, responses, and notifications. An example (see Listing 2.1) is the `Location` object, which represents a text range within a resource, such as a specific word in a text file. The specification uses TypeScript interfaces to strictly describe the structures.

```
1 export type uinteger = number;
2 type DocumentUri = string;
3 interface Position {
4     line: uinteger;
5     character: uinteger;
6 }
7 interface Range {
8     start: Position;
9     end: Position;
10 }
11 interface Location {
12     uri: DocumentUri;
13     range: Range;
14 }
```

■ **Code listing 2.1** Example of composing basic JSON structures in LSP, represented in TypeScript.

2.2.2.2 Lifecycle Messages

The lifecycle of a language server, from start to termination, is managed by a client. For example, in VSCode, opening a file with a specific extension might trigger server startup, while a user command could initiate a shutdown. In addition, the LSP supports dynamic feature registration and configuration changes, such as setting log tracing levels.

The initialization process begins with the client sending an *Initialize* request (providing its capabilities and workspace information), and the server responds with an `InitializeResult` object, indicating its own capabilities. This exchange allows clients and servers to negotiate the features they will use during the session. Finally, the client sends an *Initialized* notification to signal the completion of the handshake.

Following initialization, the client and server interact throughout the session, exchanging messages related to language features and the workspace state. The session concludes when the client sends a *Shutdown* request to initiate termination. After processing the *Shutdown* request, the client then sends an *Exit* notification to instruct the server to fully exit.

2.2.3 Document Synchronization

The LSP recognizes two document types: *text* and *notebook*. Each has a specific set of synchronization messages. While *notebook* documents offer advantages for interactive computing with programming languages, WooWoo, as a markup language, does not support this format. Therefore, this thesis focuses exclusively on *text* documents.

Document synchronization relies on notifications sent to the server to track user actions on a specific document. Some actions have paired “will” and “did” notifications to signal their status.

Examples of key document synchronization messages include (see the LSP Specification¹ for a complete list):

textDocument/didOpen Indicates a newly opened text document. Upon sending, the document’s content is managed by the client, and the server should not fetch it using the Uniform Resource Identifier (URI). Subsequent synchronization relies solely on notifications.

textDocument/didChange Signals changes made to a text document. The server updates its state based on the included change information (e.g., edits, deletions, additions).

textDocument/willSave and textDocument/didSave Sent before/after a document save. This allows the server to perform actions like applying formatting and performing linting.

2.2.4 Workspace Synchronization

In the LSP, a workspace provides the project-level context within which documents exist. It often corresponds to a folder opened in an editor, but the LSP also supports multiple root folders per workspace. Workspace synchronization employs notifications to maintain consistency between the client and server regarding the workspace’s structure, contents, and configuration. This includes informing about user actions like creating, deleting, or renaming files and folders.

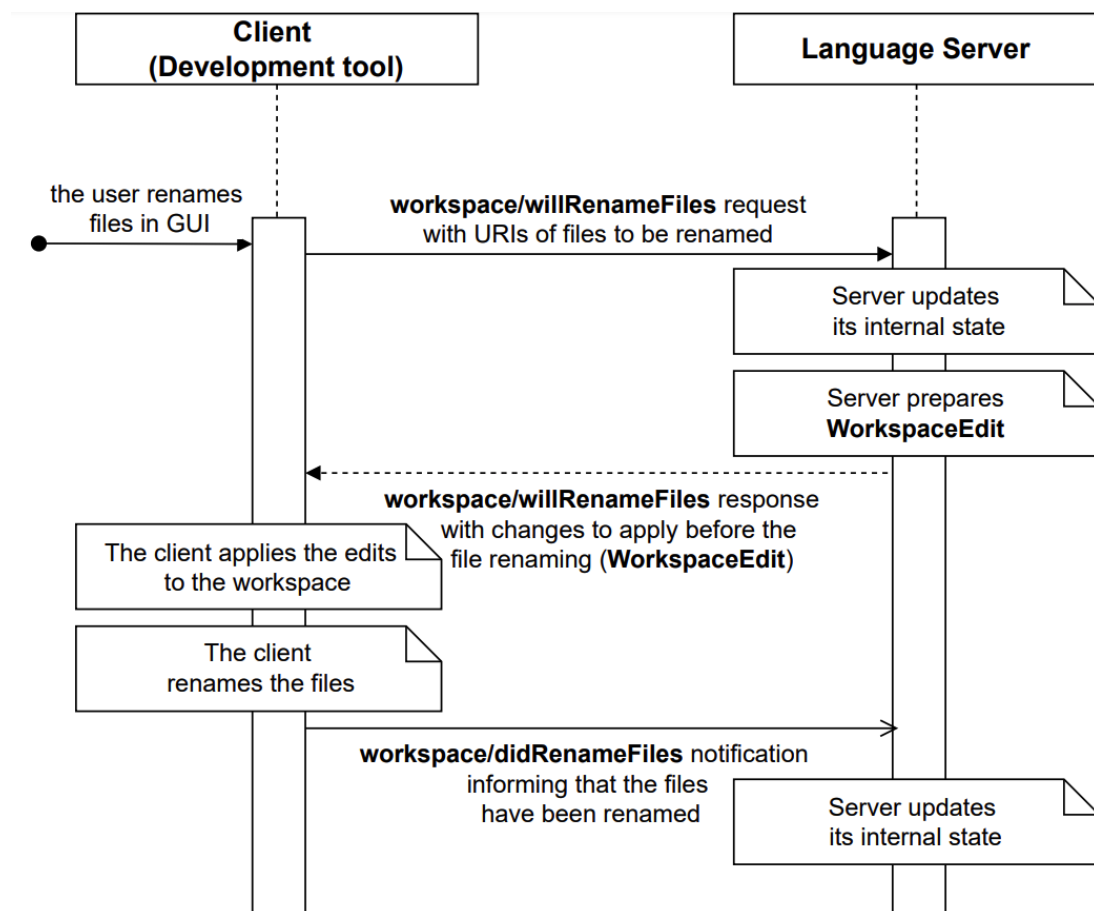
Similarly to document synchronization, most actions use paired “will” and “did” messages. For example, when a user renames one or more files, the following sequence occurs (as illustrated in Figure 2.3):

1. The client sends a **workspace/willRenameFiles** request to the server, providing old and new URIs of files to be renamed.
2. The server can apply changes to the workspace before the files are renamed (for example, refactoring references to the renamed files from other documents). It responds with a **WorkspaceEdit** detailing these changes.
3. The client applies the edits from the **WorkspaceEdit** included in the response and then proceeds with the file renaming.
4. The client sends a **workspace/didRenameFiles** notification to the server to inform it that the files have been renamed. The server can now update its internal state to reflect the changes.

2.2.5 Language Features

Language features form the core of the protocol, as they provide the actual smartness and language tools, which are the motivation for the LSP in the first place. As opposed to lifecycle management and workspace document synchronization, all language features are optional and every implementor simply chooses what they want to support. Currently, there are more than

¹Available at: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_synchronization



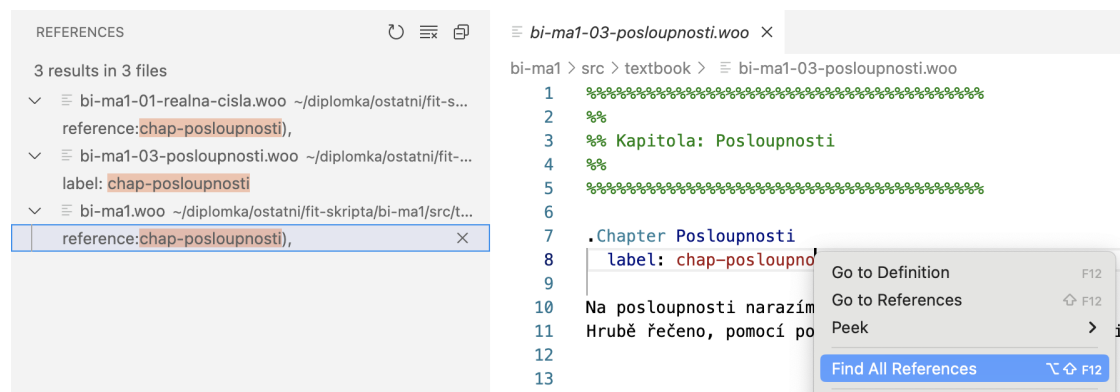
■ **Figure 2.3** Communication between LSP client and language server upon user renaming files.

30 features with new being added with newer versions. Which features are being used at runtime is negotiated during initialization, as described in Section 2.2.2.2. Language features can be divided into two types, described in following subsections.

2.2.5.1 Code Comprehension Features

Code comprehension features focus on understanding existing code, with the goal of helping the user navigate and understand the relationships within the codebase. A typical example is the *Find All References* feature introduced in the first version of the protocol. It is a widely supported functionality among language servers, as evidenced by listings on langserver.org [9]. Within the context of WooWoo, this feature is commonly used to locate all references of a specific label, as illustrated in Figure 2.4.

Closely related to *Find All References* is the *Go to Definition/Declaration* feature. This functionality enables users to navigate directly to the declaration of a symbol from its reference. Additionally, the *Hover* feature enhances code understanding by providing detailed information about symbols under the cursor when the user hovers over them. This is achieved by the client sending a request containing the text document URI and the cursor position. In WooWoo documents, the *Hover* feature displays dialect-specific descriptions of structure types.



■ **Figure 2.4** Demonstration of the *Find All References* feature in VSCode for a WooWoo document part label.

2.2.5.2 Coding Features

On the other hand, coding features are mostly focused on assisting the user at the moment of typing and on making it easier to refactor and make changes to the document. Key features in this category include *Completion*, *Diagnostics*, and *Rename Symbol*.

Completion can be triggered by events including typing a specific character or manual invocation. Upon triggering, the user is displayed a list of possible auto-completions offered by the server based on context-sensitive analysis. This helps the users to write faster and see which symbols are valid at the given position. The text can be inserted in either simple `PlainText` or `Snippet` mode, which allows for complex completions containing placeholder sections. For example, see Figure 2.5, which shows a code completion in `Snippet` mode for the `.include` keyword and a placeholder for a file to be included.

Diagnostics are managed by the `textDocument/publishDiagnostics` notification sent from the server to the client. It is the server's responsibility to re-compute the diagnostics when needed. *Diagnostics* provide a list of errors, warnings, information, and hints, along with the position to which they apply. This offers the user valuable insight about the code, like warning them about an incorrect state. Note that since version 3.17, a functionality allowing the client to request specific diagnostics for a specific document at any point in time was added.

The *Rename Symbol* feature enables consistent refactoring across a project. When executed, the server finds all references to the selected symbol (such as a variable, function, or in the context of WooWoo a meta-block field value) and allows the user to change its name. This maintains code integrity throughout the document, saving time and preventing errors that manual renaming might introduce.

Figure 2.5(a) shows a code editor with the text `.inc` on the first line. A blue completion menu is open, showing the suggestion `.include`. Below the menu, the code is: `%%
%% Chapter
%%
.Chapter Přehled použitého značení
label: chap-notation`. The cursor is positioned at the end of the first line.

(a) Initial code completion suggestion for `.include` keyword.

Figure 2.5(b) shows the same code editor as in (a), but now the text is `.include bi-ma1-01-realna-cisla.woo`. A completion menu is open, listing several file names: `bi-ma1-01-realna-cisla.woo`, `bi-ma1-02-funkce.woo`, `bi-ma1-08-aplikace.woo`, `bi-ma1-05-spojitosť.woo`, `bi-ma1-appendix.woo`, `bi-ma1-07-prubeh.woo`, and `bi-ma1.woo`. The first item is highlighted. Below the menu, the code is: `%%
%% Chapter
%%
.Chapter Přehled použitého značení
label: chap-notation`. The cursor is positioned at the end of the first line.

(b) Placeholder for filename after accepting `.include` suggestion.

■ **Figure 2.5** Demonstration of the *Completion* feature: Suggestion and placeholder insertion

..... Chapter 3

Requirements

This chapter outlines the functional and non-functional requirements for the WooWoo Language Server, serving as the technical foundation for its design and implementation. These requirements were developed through discussions with Tomáš Kalvoda, the creator of WooWoo, focusing on core authoring needs. They also incorporate insights from authoritative sources on language servers (citations provided in relevant sections).

3.1 Functional Requirements

“Functional requirements are product features or functions that developers must implement to enable users to accomplish their tasks [17].” They must be specific and measurable [18]. In the case of a language server, which doesn’t have its own user interface, functional requirements focus primarily on the expected core features of the server.

Dialect

F1.1 Dialect Independence: The server shall not be restricted to a single WooWoo dialect.

F1.2 Custom Dialect Selection: The user shall be able to provide their own WooWoo dialect which the server will use.

Project Management

F2.1 Project Awareness: The server shall recognize the concept of a WooWoo project and adjust language features accordingly. This includes adapting *Completion*, *Go to Definition*, *Find All References*, and other relevant features to be sensitive to the project’s scope.

F2.2 Multiple Project Support: The server shall support workspaces containing multiple WooWoo projects.

F2.3 Standalone File Handling: The server shall provide applicable language features for WooWoo files that are not associated with a specific project.

Workspace Lifecycle

All the functionalities in this category must be aligned with the LSP.

F3.1 File Creation: The server shall recognize newly created WooWoo files, integrating them into its analysis and providing appropriate language features.

F3.2 File Deletion: The server shall remove analysis data associated with deleted WooWoo files.

F3.3 File Renaming: The server shall update its internal references and analysis when WooWoo files are renamed.

F3.4 File Changes: The server shall re-analyze modified WooWoo files to ensure language features remain accurate.

Language Features

All the functionalities in this category must be aligned with the LSP.

F4.1 Code Completion:

- The server shall offer suggestions for file paths when completing the `include` statement.
- The server shall suggest references within applicable inner environments.

F4.2 Find All References: The server shall be able to locate all references to a given referencable symbol (meta-block field) within a project.

F4.3 Go to Definition/Declaration: The server shall be able to navigate directly to the declaration of a symbol. This includes references within inner environments and meta-blocks, as well as jumping to an included file via the `include` statement.

F4.4 Rename Symbol: The server shall be able to rename referencable symbols, with the server updating all references consistently.

F4.5 Syntax Error Detection: The server shall identify and report syntax errors in WooWoo files, providing the user with a best-effort indication of the error location.

F4.6 Hover Information: The server shall provide descriptions or documentation for WooWoo structure types. Description content shall be obtained from the active dialect.

3.2 Non-functional Requirements

“Nonfunctional requirements detail the system’s operational qualities, focusing on *how* it performs rather than *what* it does. These requirements are pivotal in ensuring the system’s efficiency, reliability, and overall usability, thereby significantly influencing the user experience.” [17]

NF1 Performance: The server must exhibit minimal latency when responding to LSP requests in standard operating conditions, ensuring a fluid user experience. For example, code completion suggestions should be presented promptly to avoid interrupting the user’s workflow.

NF2 Robustness in Invalid States: Most of the time, the code in the editor is incomplete and syntactically incorrect [10]. Despite this, users expect the server to provide relevant language features. The server shall handle these situations gracefully.

NF3 Platform Compatibility: The server shall be easily installable on Linux, Windows, and macOS operating systems and support both x86 and ARM architectures.

NF4 Integration and Interoperability: The server shall adhere to the LSP, enabling integration with various LSP-compatible editors.

NF5 Extensibility: The design and architecture of the server should facilitate easy extension and integration of new features and improvements without requiring significant changes to the existing code.

NF6 Documentation: Adequate documentation, comprising README files and in-line code comments, must be provided. This documentation should explain the server's installation process, basic operation, and provide a synopsis of its primary features, assisting users and future developers in navigating and utilizing the server effectively.

Design & Technologies

The LSP imposes constraints on a language server’s Application Programming Interface (API) to ensure compatibility across different editors and tools. Beyond that, developers have broad flexibility in design and implementation choices. This includes selecting the programming language, determining the internal architecture, and choosing how the server analyzes code. These design decisions, guided by the non-functional requirements established in the previous chapter, directly influence how effectively the server can process files and provide language-specific features.

In practice, file analysis begins with an error-tolerant parser. The VSCode Language Server Guide [10] emphasizes the importance of this component, as code within an editor often exists in incomplete or partially incorrect states during development. The parser must be able to continue analysis despite errors, making its best effort to parse the remainder of the file as accurately as possible to create a Concrete Syntax Tree (CST). A CST (often called a parse tree), as opposed to Abstract Syntax Tree (AST), preserves the exact syntax of the original code, including formatting, whitespace, and the precise location of every token in the document. This detailed representation is important for applications which manipulate the code itself and not just extracting its abstract meaning [19]. The quality of the language features depends directly on the accuracy of the CST.

Responsiveness is another key requirement for a smooth user experience. A language server should feel like a natural extension of the code-editing process, offering guidance and suggestions in real-time as the code evolves. Delays, even short ones, can disrupt the user’s thought process, forcing them to break focus and wait for the tool. This hinders efficient coding and can even lead to disengagement. If the tool feels consistently sluggish, the user might be less inclined to utilize its features to their full potential.

Finally, cross-platform compatibility should be a key design consideration for language servers. The goal is to support all common development environments where the target language is relevant. For WooWoo, this means supporting popular desktop operating systems and architectures. Additionally, the language server should be easy to install and configure, even for users with limited technical expertise.

4.1 Overview of Existing Language Servers

Existing language servers demonstrate diverse approaches to common design challenges. Some leverage existing parsers for code analysis, often building upon infrastructure surrounding their compilers and interpreters. Where suitable parsers are unavailable, language servers may either integrate modified parsers or utilize ones specifically designed for efficient, error-tolerant parsing within the server context. This section provides a brief overview of several language server

architectures, highlighting these variations.

4.1.1 TypeScript Language Server

The TypeScript Language Server utilizes existing components to parse the code. It wraps a component called `tsserver` [20]. “The TypeScript standalone server (aka `tsserver`) is a node executable that encapsulates the TypeScript compiler and language services, and exposes them through a JSON protocol. `tsserver` is well suited for editors and IDE support [21].” This approach reduces the size of the language server’s codebase and allows it to focus primarily on the LSP interface and adding additional features to existing infrastructure. The TypeScript Language Server (including `tsserver`) is written in TypeScript.

4.1.2 Rust-Analyzer

The `rust-analyzer`, an LSP implementation for the Rust programming language, began development in 2017 and eventually replaced the Rust Language Server (RLS), an earlier implementation of LSP for Rust. The original RLS was built upon infrastructure of `rustc`, Rust’s official compiler. However, its architecture limited its ability to provide the low-latency, high-quality responses required for interactive environment [22]. The underlying parser was not optimized for code editor use and lacked the built-in error resilience needed in this context.

The `rust-analyzer` employs a custom-built, incremental, and error-resilient parser, specifically designed for Rust [23], enhancing its effectiveness over more general solutions, albeit at the cost of substantial development efforts.

The language server itself is written in Rust. Precompiled binaries for Windows, Linux, and macOS can be found on their releases page¹.

4.1.3 Bash Language Server

The Bash Language Server adopts a mixed approach. It leverages existing tools like ShellCheck² for linting and Explainshell³ for static analysis. Additionally, it also employs a parser generated from the `tree-sitter-bash`⁴ grammar using the Tree-sitter parser generator [24]. Tree-sitter is an incremental, error-tolerant parser generator specifically designed for language tooling within code editors [25]. Not having to maintain a custom-made parser combined with the modular approach simplifies development and improves maintainability.

The Bash Language Server is written in TypeScript and utilizes the WebAssembly (Wasm) version of the `tree-sitter-bash` parser for cross-platform compatibility.

4.2 High-Level Language Server Architecture

A modular design approach was adopted for the WooWoo Language Server architecture. This decision aligns with best practices observed in existing language servers and leverages the benefits of maintainability, extensibility (NF5), and the potential for component reuse. The following three main components are central to a language server:

Parser Responsible for efficiently parsing WooWoo source files and producing CSTs.

¹Releases available at: <https://github.com/rust-lang/rust-analyzer/releases>

²For more on ShellCheck, visit: <https://www.shellcheck.net/>

³For more on Explainshell, visit: <https://explainshell.com/>

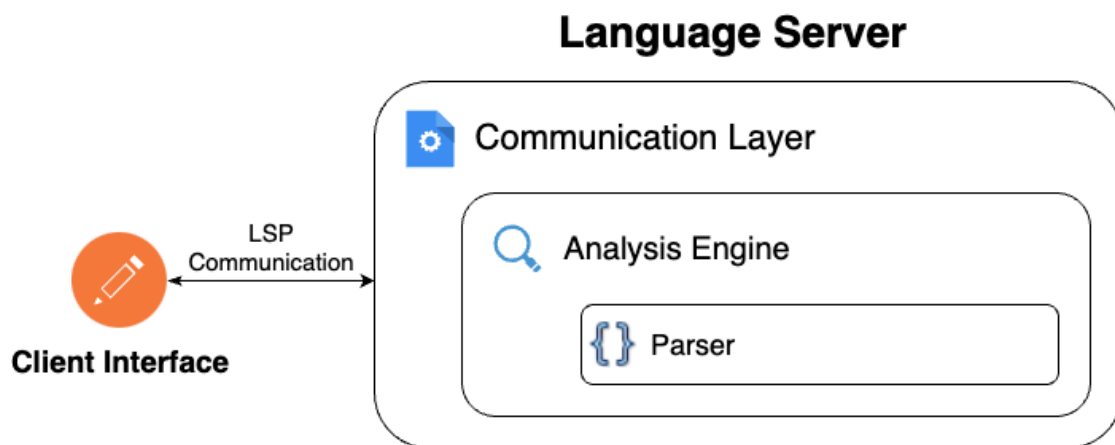
⁴Available at: <https://github.com/tree-sitter/tree-sitter-bash>

Analysis Engine Leverages the CSTs generated by the parser for in-depth language analysis. This component encapsulates comprehensive knowledge of WooWoo’s semantics and exposes a custom API for client interactions.

Communication Layer Implements the LSP interface, translating client requests (e.g., from VSCode) into Analysis Engine API calls and formatting responses according to the LSP.

This architecture (see Figure 4.1) offers significant advantages:

- **Component Reusability:** The parser and analysis engine are not tightly coupled to the LSP implementation. They could be used in other contexts, such as powering a standalone WooWoo linter or serving as components within different LSP implementations.
- **Flexibility:** The modular design facilitates adaptation to future requirements. Should a different parser technology or LSP framework become necessary, the impact on the overall system would be minimized compared to a tightly integrated architecture.
- **Maintainability:** Clear separation of concerns enhances maintainability, making debugging, updating, and extending individual components easier.



■ **Figure 4.1** Key Components of the WooWoo Language Server

4.3 Parser

Parser choice is a foundational decision in language server design, as it is the essential building block, forming the basis for all other language features. As outlined in the requirements, the parser must be fast (NF1), error-resilient (NF2), and platform-independent (NF3). This section describes the available options, presents the final design choice, and explains the rationale behind it.

4.3.1 Existing WooWoo Parsers

As described in Section 4.1, language servers often leverage existing parsers or components from other infrastructure, such as compilers. While WooWoo has two existing parsers, neither is suitable for this use case. The reasons are outlined in the following sections.

4.3.1.1 The WooWoo Library Parser

The WooWoo library serves as the initial stage in a pipeline for generating outputs, processing WooWoo projects before handing them off to output generators (templates) [26]. However, it is not optimized for speed. Built with `parslet`, a Ruby-based parser engine, it can take seconds to parse source files for a course like BI-MA1 at FIT CTU. This aligns with the official `parslet` GitHub page: “Parslet is slow because of the way it is constructed internally . . . often resulting in execution times in the second range instead of the subsecond range” [27]. Additionally, making this parser error-resilient would be challenging. Therefore, while useful for referencing grammar rules, this parser is not suitable for a language server.

4.3.1.2 The Wootom Extension Parser

Wootom is a package (extension) for the now-discontinued Atom text editor. Created by David Straka as part of his 2021 bachelor’s thesis, it pioneered language support for WooWoo within a text editor [2, 28].

Written in TypeScript, the Wootom extension utilizes a custom basic recursive descent parser. These parsers can, in the worst-case scenario, have exponential complexity relative to the length of the input [29]. Additionally, the implementation lacks support for incremental parsing and error recovery. It also exhibits inconsistencies with the WooWoo specification, preventing it from recognizing certain syntax errors. These factors, combined with its use of a high-level language (TypeScript) which limits its performance, make it unsuitable for a use in a language server.

4.3.2 Parser Generators

Parser generators automate much of the work involved in creating a parser. They work by taking a formal definition of a language’s grammar as input and automatically generating source code capable of parsing text according to that grammar. This generated parser analyzes the input and builds a structured representation called a parse tree. The parse tree organizes the components of the input text in a way that reflects the grammar’s rules. [30]

Parser generators make parser creation and maintenance less complex. They benefit from ongoing community contributions, leading to optimizations, resources, and extensive support. Each parser generator offers its own strengths and trade-offs. Some of the most widely known choices are described in the subsections below.

4.3.2.1 ANTLR

ANOther Tool for Language Recognition (ANTLR) is a mature, powerful and widely-used parser generator that helps developers build interpreters and compilers. It uses a specialized Domain-Specific Language (DSL) for defining grammars. This, combined with its advanced LL(*) parsing algorithm, enables ANTLR to handle a broad range of languages, even those with complex, context-sensitive elements. Unlike simpler LL parsers, ANTLR can look ahead an arbitrary number of tokens, giving it greater flexibility. [31]

The ability to generate parsers in multiple programming languages (like Java, C#, and Python) makes ANTLR adaptable to various projects. It has extensive documentation and resources available, including The Definitive ANTLR 4 Reference, a 300 page book dedicated to it. Furthermore, ANTLR’s distribution under the BSD license makes it accessible for both research and commercial use.

4.3.2.2 Bison

Bison is a classic parser generator with a long history (originally released in 1985) as part of the GNU software suite. It takes a context-free grammar definition as input and generates a

deterministic LR or generalized LR (GLR) parser [32]. Bison’s established presence means it is often used in projects where compatibility with legacy tools or YACC (Yet Another Compiler Compiler) is important [32]. It is released under the GNU General Public License (GPLv3 or later).

4.3.2.3 Tree-sitter

Tree-sitter is a relatively recent parser generator, originally released by GitHub in 2018 for integration into the Atom text editor [33]. Specifically designed for use within text editors, it describes itself as “An incremental parsing system for programming tools” [25]. Tree-sitter is open-source and released under the MIT License.

While generated parsers are written in C, bindings are available for common programming languages, enabling integration into a wide variety of applications [34]. The parsers produce CSTs that retain token positions within the original document [19].

Tree-sitter employs a GLR parsing algorithm, providing the flexibility to parse any programming language. This algorithm’s non-backtracking nature contributes to its speed. Additionally, Tree-sitter supports incremental parsing, efficiently reusing portions of the syntax tree after code edits. [35]

It also features a novel error-recovery technique, allowing the parser to produce useful results even when the code is incomplete or incorrect [19].

Despite its relatively recent initial release, Tree-sitter has quickly gained popularity. As of March 2024, its official website lists 138 grammars covering various programming languages. Its found widespread use in projects, including prominently semantic-aware highlighting, folding, and indentation-assistance within the Neovim community [36]. GitHub.com also employs Tree-sitter for code highlighting of several languages [37].

4.3.3 Parser Choice

As previously discussed, reusing an existing WooWoo parser wasn’t viable.

Developing a new parser from scratch would demand extensive effort and time, beyond the scope of this thesis, to achieve the desired error tolerance and performance for an effective language server.

Therefore, a parser generator was the optimal choice, with Tree-sitter being the preferred option. Tree-sitter was designed explicitly for use within programming tools. This focus and its capabilities, particularly in error resilience and incremental parsing, align well with the needs of creating a responsive language server. Tree-sitter’s design also simplifies the handling of context-sensitive elements, like indentation level.

Furthermore, Tree-sitter’s growing adoption demonstrates its effectiveness and suitability for this project. Notably, GitHub’s use of Tree-sitter for code highlighting, along with their detailed rationale [38], underscores its strengths. The project’s momentum and active development community provide further assurance of continued support.

Another advantage in choosing Tree-sitter is that the grammar itself becomes a contribution to the WooWoo ecosystem, enabling its use in tools like Neovim and others.

4.4 Analysis Engine

The analysis engine (analyzer), referred to as `wuff`⁵, is a core component responsible for parsing source code (using the Tree-sitter generated parser) and analyzing the resulting CSTs. This analysis enables language features such as *Completion*, *Go to Definition*, and *Diagnostics*, ensuring a fluid and robust user experience (NF1, NF2).

⁵Inspired by the playful sound of the name `ruff` (<https://github.com/astral-sh/ruff>)

The first key design decision for this component involved selecting the underlying implementation technology. This choice was crucial for meeting performance requirements (NF1) and ensuring platform compatibility (NF3).

The internal architecture needed to be designed to satisfy all functional requirements while ensuring modularity, flexibility, and performance.

4.4.1 Technology

The performance requirements (NF1) and the need for platform compatibility (NF3) were key factors in selecting the programming language and technology for implementing the analysis engine. While Tree-sitter ensures efficient parsing, the analyzer’s responsibilities extend beyond parsing. For example, it must accurately map and re-index references across the entire project to support features like *Find All References*, even in the context of code changes.

To identify the most suitable technology, two implementations were explored and evaluated through performance tests and design considerations.

4.4.1.1 Python Implementation

Python is featured on the official VSCode site as a popular choice for implementing the language server communication layer [39], using the `pygls` library (a generic language server written in Python). Implementing the analysis engine in Python, alongside its use in the communication layer, would simplify development due to seamless integration between components.

The `pygls` official repository lists many concrete implementations [40]. Some less prominent projects, like the Helios language server, use Python for both the LSP layer and analysis. Conversely, projects like `ruff-lsp`, a LSP implementation for `ruff` (a Python linter known for its exceptional speed), delegate all analysis work to an external module written in Rust [41]. Another notable example is YLS, Avast’s LSP implementation for the YARA language. It employs a hybrid approach, with some analysis performed by an external module written in C++ and other parts handled in Python [42].

Given this information, it was worth exploring a Python implementation for the analyzer. However, although sufficient in most situations, it sometimes exhibited noticeable lag.

4.4.1.2 Switching to Compiled Language

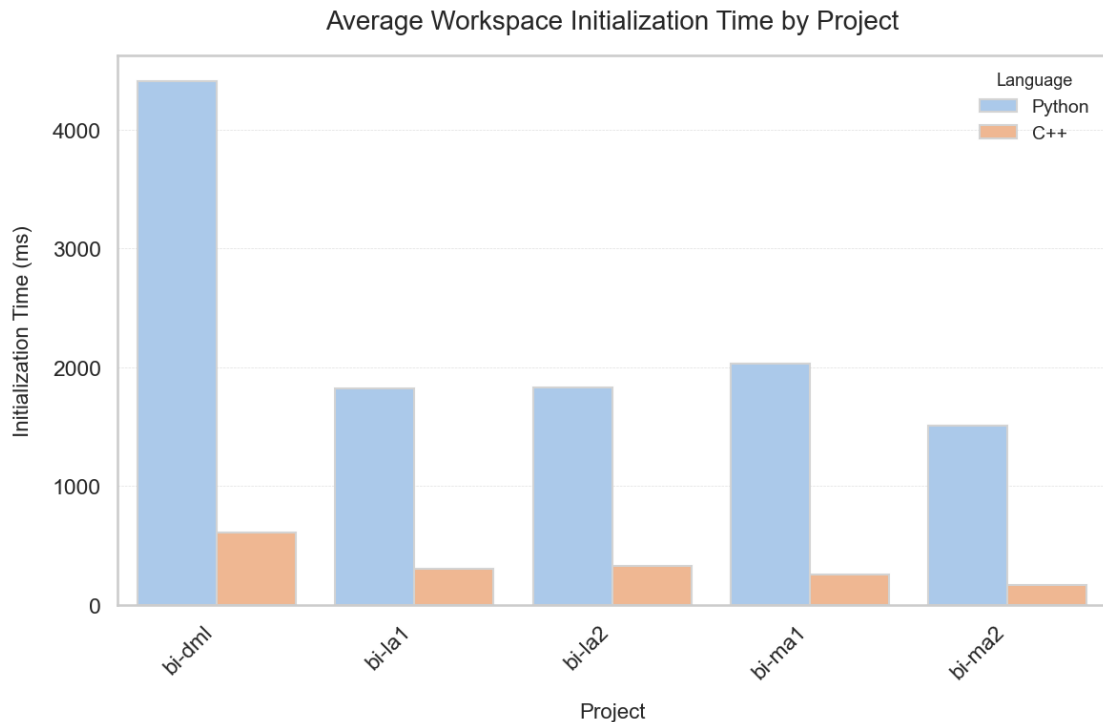
Given the noticeable lag of the Python-based analyzer, especially on less powerful machines, it was deemed sub-optimal in terms of responsiveness. A second version of the analyzer was developed in C++ to address these concerns. Performance tests were conducted to compare initialization time and the speed of core features. These tests were performed on practical samples, primarily existing projects used for generating math textbooks at FIT CTU.

Importantly, the tests were run on an Apple MacBook Pro with an M2 Pro chip and 32 GB RAM, providing a realistic benchmark on the higher end of consumer notebooks. The test scripts, testing datasets, results, and detailed version information related to the performance evaluation are described in more detail in Chapter 7 and included in the attachment accompanying the thesis.

The C++ implementation demonstrated significant performance advantages. Workspace initialization (the time it takes for the analyzer to become responsive) was almost 10 times faster (see Figure 4.2).

Features like *Hover* and *Completion*⁶ were roughly 5 times faster in C++ (see Figure 4.3). While absolute speeds for *Hover* were in the submillisecond range for both implementations, the improvement in *Completion* times is particularly important for user experience. In the Python

⁶ *Completion* testing involved repeated changes to force re-indexing, providing a realistic assessment of performance under editing conditions.



■ **Figure 4.2** Comparison of average workspace initialization times between Python and C++ implementations across projects containing sources for various subjects at FIT CTU. For further description and sizes of the datasets, see Table 7.1.

version, *Completion* often took hundreds of milliseconds, leading to noticeable delays. The C++ implementation significantly reduces this delay.

Lastly, the *Semantic Tokens* feature saw a two-order-of-magnitude improvement in the C++ implementation (see Figure 4.4). This eliminates the delayed semantic highlighting observed in the Python version.

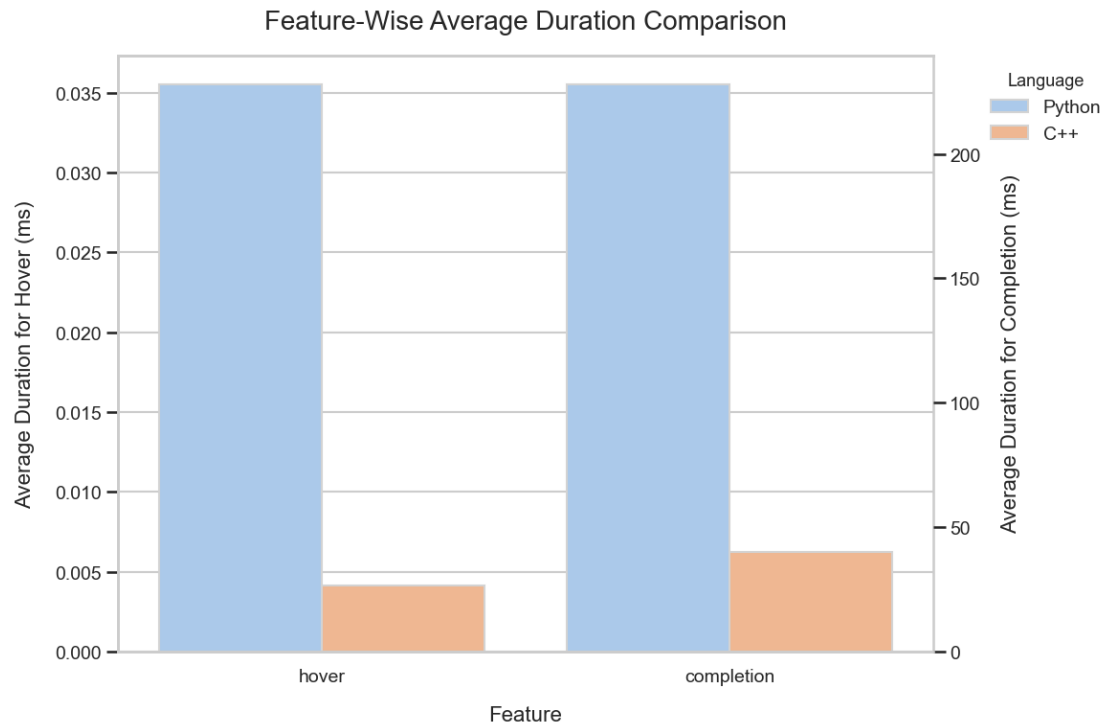
Based on these results, the Python version was discontinued and C++ was adopted for the analyzer, ensuring a smooth and responsive user experience for authors in WooWoo. Platform compatibility is ensured through the distribution of precompiled wheels, discussed further in Chapter 6.

4.4.2 Architecture Overview

The analysis engine’s architecture centers on three key principles: dialect-independence, centralized API including workspace management, and modular feature implementation. These principles are driven by the requirements outlined in Chapter 3.

Firstly, language tooling for WooWoo is highly dependent on the currently used dialect. For the purpose of dialect independence (F1.1) and custom dialect selection (F1.2), a `DialectManager` component is introduced to handle all dialect-related work. The analyzer will get information regarding a dialect only through this component, using its dialect-independent API.

Secondly, as the analyzer must manage the state and respond to lifecycle events (F3.1 - F3.4), there will be a single stateful object exposing the API for this, for easy use by clients. This component, `WooWooAnalyzer`, will be the entrypoint to the application and manage the workspace state, including storing references to structures representing the individual WooWoo documents.



■ **Figure 4.3** Feature-wise comparison of average durations, highlighting the time required for *Hover* and *Completion* features in Python and C++ implementations. Analysis conducted on sources for the BI-MA1 course.

Finally, the language features (F4.1 - F4.6) will be realized by independent specialized components (called *Feature Components*), to which the analyzer will redirect all the work. This ensures modularity and easy extensibility (NF5), as the components can be replaced and new ones easily added.

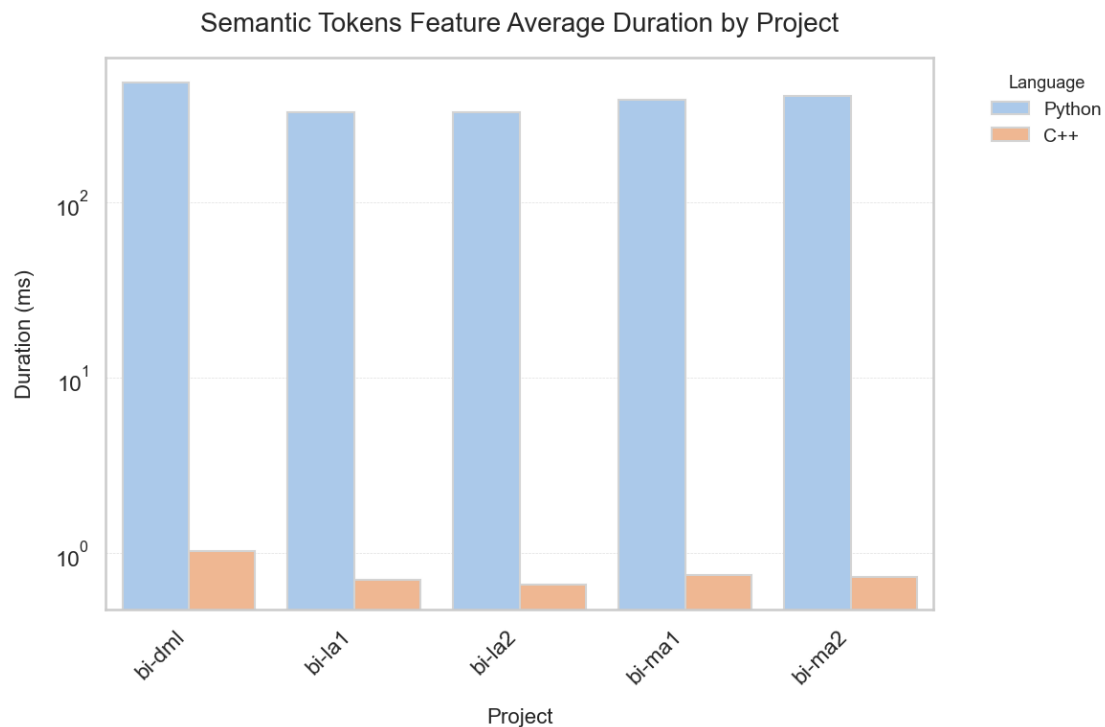
Figure 4.5 illustrates the structure and relationships between core components in `wuff`. For clarity, some details have been omitted. This section describes these components, outlining their responsibilities and how they interact.

4.4.3 WooWooAnalyzer

The `WooWooAnalyzer` is the heart of the analysis engine, serving as the primary interface for client interactions. While it directly manages workspace-related tasks, it delegates dialect handling to the `DialectManager` and uses feature components to provide language-specific functionality. This centralized design simplifies client interaction while ensuring a modular and flexible architecture.

The `WooWooAnalyzer` is responsible for the following:

1. **Dialect component:** Maintains a reference to a `DialectManager`, enabling client-driven dialect switching via the public `setDialect` function. Provides the `DialectManager` to other components and documents as needed.
2. **Project detection:** Scans the workspace for WooWoo projects upon initialization. Automatically associates `.woo` files with their respective projects and groups unassigned files into an artificial project for management.



■ **Figure 4.4** Comparison of average durations for processing *Semantic Tokens* in Python and C++. The analysis encompasses all projects referenced in Figure 4.2.

3. **Document management:** Tracks open documents, determines reparsing logic, handles project switching, and provides a document-related API for client interactions (such as responding to edits).
4. **Feature components:** Instantiates and manages feature components (Completer, Navigator, Highlighter, etc.). Acts as the exclusive gateway for client interaction with these components, exposing their combined API.

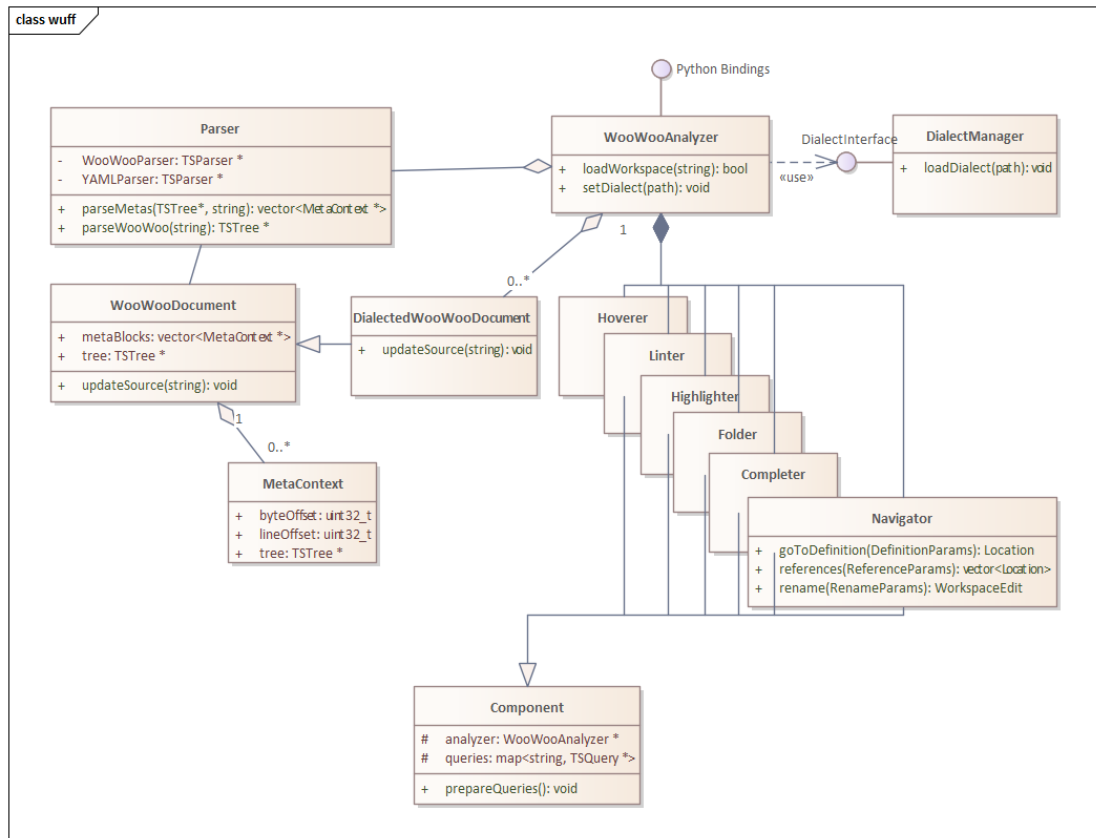
The following example demonstrates these concepts in practice, showing how to set a dialect, load a workspace, and utilize language features. See Listing 4.1:

```

1 // Create an instance of WooWooAnalyzer
2 auto *analyzer = new WooWooAnalyzer();
3 // Set a custom dialect
4 analyzer->setDialect("dialects/fit_math.yaml");
5 // Load the workspace
6 analyzer->loadWorkspace("file:///bi-ma1/textbook");
7 // Use language features
8 Location definitionLocation = analyzer->goToDefinition(
9     DefinitionParams(
10         TextDocumentIdentifier("file:///bi-ma1-real-numbers.woo"),
11         Position{50, 150});

```

■ **Code listing 4.1** Example of WooWooAnalyzer use within C++.



■ **Figure 4.5** The core classes present in the `wuff` analysis engine.

4.4.4 Parser

Component responsible for all parsing-related work. It will use the Tree-sitter parser generated from the WooWoo Tree-sitter grammar (`tree-sitter-woowo`). Parsing will produce CSTs, realized by the `TSTree` structure from Tree-sitter.

Importantly, WooWoo documents are essentially multi-language, as they can contain an arbitrary number of YAML sub-documents (`meta_blocks`). The `tree-sitter-woowo` parser will not parse the `meta_blocks`; instead, they will be recognized as single `meta_block` nodes. The Parser component will use `tree-sitter-yaml`, a publicly available Tree-sitter parser for YAML⁷, released under the MIT License, to parse these `meta_blocks`. Consistency motivated the choice of a Tree-sitter parser for YAML.

Therefore, parsing a single WooWoo source file should result in one WooWoo CST and multiple YAML CSTs, one for each `meta_block`. Tree-sitter provides the following objects to represent these structures:

TSTree Represents the CST of an entire source code file. It consists of `TSNode` instances. [34]

TSNode A single node within the `TSTree`. It tracks its start and end position in the source code, as well as its hierarchical relationships to other nodes (parent, siblings, and children). [34]

⁷Available at: <https://github.com/ikatyang/tree-sitter-yaml/>

4.4.5 WooWooDocument

The `WooWooDocument` class represents an up-to-date state of a single document within the workspace. It contains the document's source code, its CST representation, meta blocks (as CSTs), its file path, and other relevant metadata.

The core function of this class is to maintain synchronization with source updates. When the analyzer receives a notification of changes to the document, it will provide the new source or relevant changes to the `WooWooDocument`, allowing it to update itself accordingly.

The analyzer manages instances of this class, grouping them by project. These instances serve as the primary input for analysis by feature components, enabling the provision of language features.

The meta blocks in the document will be stored in a vector and represented by the `MetaContext` class.

MetaContext This class encapsulates a single `meta_block` (YAML sub-document) within a WooWoo source file (represented by the `WooWooDocument`). It contains the `meta_block`'s `TSTree` (CST representation), its position within the parent document, and the type/name of the node it belongs to. This metadata is essential for targeted searches within `meta_blocks`, such as retrieving only ones belonging to a specific structure (e.g., `document_part`) with a given name (e.g., *Chapter*).

DialectedWooWooDocument

To optimize performance, it's essential for documents to be processed with dialect-specific knowledge. This includes tracking references and analyzing `meta_blocks` for *referencables* (nodes, that can be referenced). This information is frequently requested by individual feature components. On-demand calculations would slow down responsiveness.

Therefore, the `DialectedWooWooDocument` subclass extends the base `WooWooDocument`. This specialized class adds dialect awareness, tracking references within the document, and proactively re-evaluating them as changes occur.

4.4.6 DialectManager

Class responsible for loading and processing a dialect definition file (YAML file describing syntax and semantics of a specific dialect, as described in Section 1.3.1).

It will utilize the `yaml-cpp` library, released under the MIT License, to extract the dialect information into structures representing the different node kinds, like `Environment` or `Wobject`.

The `DialectManager` stores all information about the active (loaded) dialect, but the dialect is not accessible from outside classes. Instead, other components can obtain information only via the interface described in Table 4.1.

4.4.6.1 The Concept of Reference

The concept of `Reference` is important for understanding the `DialectManager`'s design. A `Reference` defines what a WooWoo structure can reference, based on structure types and `meta_block` content. It consists of three attributes:

1. **metaKey:** The key within a `meta_block` that identifies the target of the reference (e.g., *label*).
2. **structureType:** The type of WooWoo structure being referenced (e.g., `outer_environment`).
3. **structureName:** The name of the WooWoo structure being referenced (e.g., *equation*).

Importance for Feature Components

Feature components, such as the **Completer**, rely on **References** to offer contextually relevant suggestions. Structures can define multiple **References**. If *structureType* and *structureName* are omitted, the **Reference** applies to any structure containing the specified *metaKey* in its *meta_block*.

Example (FIT-Math)

In the FIT-Math dialect, an *eqref* environment is used to reference mathematical equations. Therefore, it includes a **Reference** targeting an *equation outer_environment*, within which a *meta_block* contains the key *label*. The value associated with the *label* key represents the specific target of the reference.

■ **Table 4.1** DialectManager Class Public Interface

Function Name	Parameters	Return Type	Description
loadDialect	string <i>dialectFilePath</i>	void	Loads and processes a dialect from the specified file path into the manager. Replaces the current dialect if already present.
getDescription	string <i>type</i> , string <i>name</i>	string	Retrieves a description for a given type and name combination from the current dialect (e.g., <i>document_part</i> , <i>Chapter</i>).
getReferencing- TypeNames	None	vector of string	Returns a list of names that are capable of referencing (e.g., <i>reference</i> or <i>eqref</i>).
getPossible- ReferencesBy- TypeName	string <i>name</i>	vector of Reference	Returns possible references for a given type name. For example, <i>eqref</i> in the FIT-Math dialect can only reference <i>equation</i> outer environments.

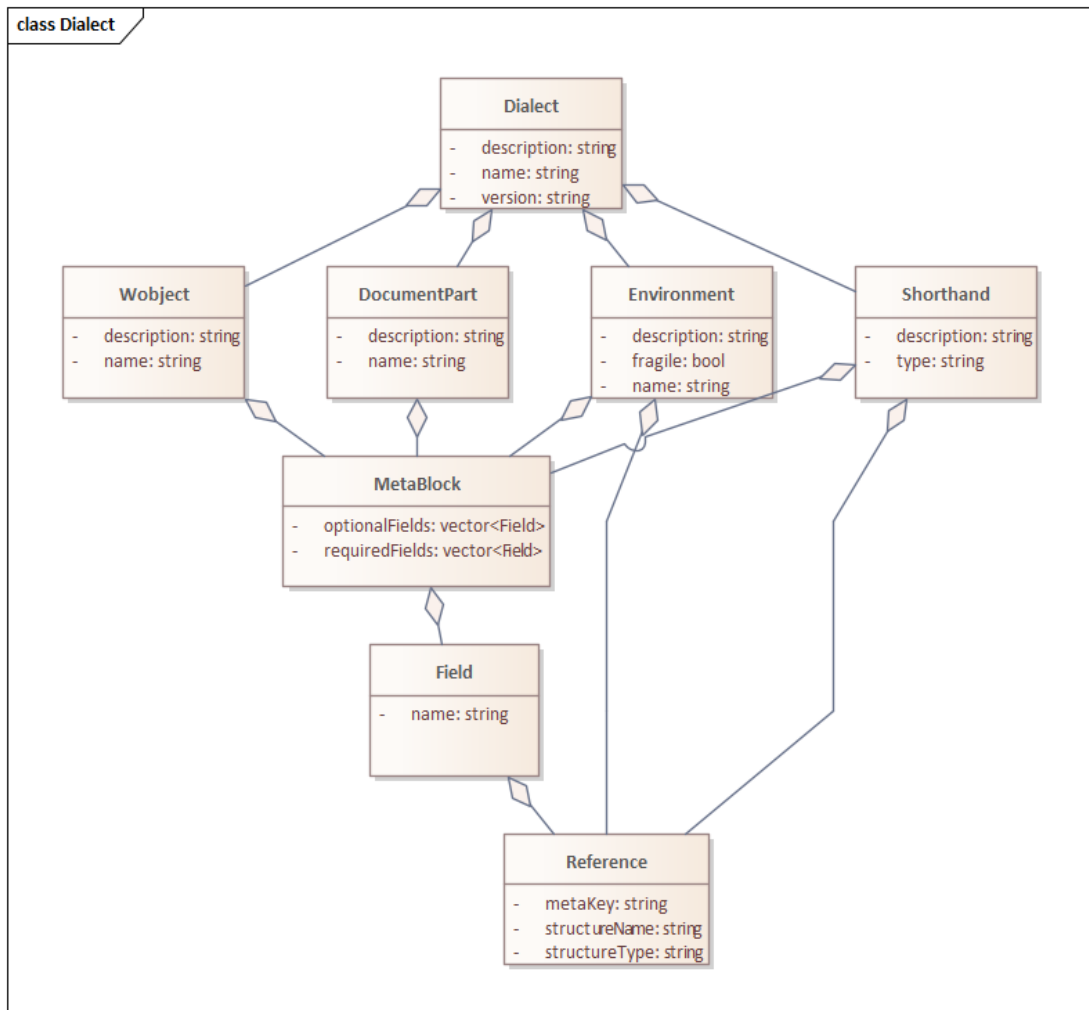
4.4.6.2 Dialect

The **Dialect** class serves as the primary data structure representing a complete dialect within the analysis engine. The **DialectManager** maintains a single active instance of this class at a time, stored in its *activeDialect* property. The **Dialect** class comprises various subclasses and directly mirrors the structure of the YAML dialect-definition file. Figure 4.6 illustrates the hierarchical composition of a **Dialect** object.

4.4.7 Feature Components

Feature components are responsible for delivering the language features. Each component specializes in a specific area of language tooling, and they are instantiated by the analyzer. The analyzer coordinates feature component actions based on requests from the client.

Leveraging the Tree-sitter Query API is crucial to the mechanism of every component.



■ **Figure 4.6** Components of the `Dialect` Class

TSQuery A core Tree-sitter object used for analyzing the structure of `TSTrees`. Queries are compiled from strings of *S-expressions*, and they remain immutable after compilation. The expression consists of one or more patterns that match a certain set of nodes in a syntax tree. To execute a `TSQuery`, a `TSQueryCursor` object is used. Execution yields a list of matches that can be easily iterated over. [25, 34]

All feature components share a common design pattern. Upon initialization, they pre-compile `TSQueries` relevant to their specific functionality. These queries are stored for repeated use, since their compilation is an expensive operation. Feature components also maintain a reference to the analyzer for workspace-wide access. A typical language feature involves executing a query on a document's CST, often restricting itself to a specific range or position, analyzing the results, and returning the outcome.

Common logic, such as query creation and lifecycle management, will be abstracted into a parent class `Component`, which all feature components will inherit from.

Input parameters and result values will align with the LSP, potentially omitting optional fields not needed for analyzing WooWoo documents. The full specification of LSP-related structures used by the feature components can be found in the `LSPTypes.h` file. For precise interface

definitions of each component, refer to their corresponding header files. The list of components and their responsibilities can be found below:

Completer Provides auto-completion functionality, including completion of `include` statements and environment references.

- **complete:** Suggests completions valid at the given file and position.

Hoverer Handles hover functionality. Works with `DialectManager` to obtain descriptions of hovered elements.

- **hover:** Returns a hover message for the specified file and position.

Navigator Class responsible for mapping, navigating, and manipulating references.

- **goToDefinition:** Finds and returns the location of a symbol's definition. If executed on an `include` statement, navigates to the top of the included file.
- **references:** Finds and returns all locations of symbols referencing a `meta_block` field (definition) at a given position.
- **rename:** Renames all references to a definition (including the definition itself) at a given position.
- **refactorDocumentReferences:** Refactors document file references to new names, given a list of old and new names. Applicable on `include` statements.

Highlighter Provides server-side code highlighting. Clients can configure token types and modifiers using the `Highlighters` API. This ensures compatibility with wide range of editor environments. Highlighted tokens are specified using Tree-sitter queries.

- **semanticTokens:** Generates a list of semantic tokens for a given document identifier, respecting client-specified token types and modifiers.

Folder Returns folding ranges within a given document. Foldable WooWoo structures include: `block`, `document_part`, and `wobject`.

- **foldingRanges:** Assembles and returns folding ranges for a given document.

Lint Class responsible for reporting syntax errors to the user. It leverages the Tree-sitter capabilities for error detection, including identifying positions of the syntax errors and missing nodes.

- **diagnose:** Provides a list of error diagnostics for a given document.

4.4.8 Enabling Python Integration

While the core analysis engine is implemented in C++ for optimal performance, providing Python bindings is essential to ensure seamless integration with the communication layer of the WooWoo Language Server, which is described in the following section.

Additionally, it increases the engine's usability for other potential projects that might prefer to interact with it using Python. Pybind11⁸ was chosen to create these bindings due to its proven reliability in creating Python bindings for C++ code.

⁸For more on pybind11, visit: <https://github.com/pybind/pybind11>

Module API

The entire public API of the `WooWooAnalyzer` class will be exposed for use within Python. This includes access to the class's public methods and their parameters, making them callable and usable within Python code. See Listing 4.2 for an example.

```

1  from wuff import WooWooAnalyzer
2  from wuff import DefinitionParams
3  from wuff import TextDocumentIdentifier
4  from wuff import Position
5
6  analyzer = WooWooAnalyzer()
7  analyzer.set_dialect("dialects/fit_math.yaml")
8  analyzer.load_workspace("file:///bi-ma1/textbook")
9  definition_location = analyzer.go_to_definition(
10     DefinitionParams(
11         TextDocumentIdentifier("file:///bi-ma1-01-real-numbers.woo"),
12         Position(50, 150)
13     ))

```

■ **Code listing 4.2** Use of the `WooWooAnalyzer` within Python, equivalent to the sequence in Listing 4.1

4.5 Communication Layer

The communication layer acts as a bridge between the LSP client and the analysis engine. It handles communication with the LSP client using the JSON-RPC protocol, translating and delegating requests and notifications to the analyzer. Responses from the analyzer are then formatted according to the LSP Specification and relayed back to the client.

Developing a custom communication layer from scratch would be a significant and unnecessary effort. Generic language server implementations exist for various languages, providing the communication logic and LSP-related objects.

4.5.1 Pygls

“Pygls (pronounced like “pie glass”) is a pythonic generic implementation of the Language Server Protocol for use as a foundation for writing your own Language Servers in just a few lines of code.” [40]

Pygls was chosen for this project due to its proven reliability, extensive resources, active community, and commitment to LSP compliance (NF4). It offers a lot of resources, including official guides on the VSCode website [39]. This aligns well with the goals of this thesis, as one of them is to integrate the WooWoo Language Server into a VSCode extension. Importantly, Pygls remains flexible and is not focused on VSCode, making it a good choice for supporting other editors as well.

As performance bottlenecks are primarily addressed by the analysis engine, Python's ease of installation and platform compatibility (NF3) make it a suitable choice for the communication layer.

4.5.1.1 LanguageServer Class

The pygls `LanguageServer` class provides the base functionality for language servers, including startup modes, logging, and feature registration. It also offers default implementations for lifecycle-related LSP features (*Initialize*, *Shutdown*, *Exit*), which can be overridden if necessary [43].

Feature registration is facilitated using the `LanguageServer.feature` decorator. Pygls utilizes the `lsp.protocol.types` module to import standard LSP method names and types. To ensure consistency, the implementation will employ these predefined pygls elements (see Listing 4.3 for an example) [43].

```

1 from pygls.server import LanguageServer
2
3 server = LanguageServer('example-server', 'v0.1')
4
5 @server.feature(TEXT_DOCUMENT_COMPLETION, CompletionOptions(
6     trigger_characters=[',']))
7 def completions(params: CompletionParams):
8     """Returns completion items."""
9     return CompletionList(
10         is_incomplete=False,
11         items=[
12             CompletionItem(label='Item1'),
13             CompletionItem(label='Item2'),
14             CompletionItem(label='Item3'),
15         ]
16     )

```

■ **Code listing 4.3** Instantiating `LanguageServer` with a `TextDocumentCompletion` LSP feature. [43]

4.5.1.2 WooWooLanguageServer

The `WooWooLanguageServer` class will extend the pygls `LanguageServer` class. It acts as an intermediary, wrapping the analysis engine (`wuff`) and managing the flow of requests and responses between the client and the engine.

Importantly, the analysis engine will not be called directly from the decorated feature registration functions. Instead, these functions primarily register features and forward calls to specific methods within the `WooWooLanguageServer`, where interactions with the analysis engine will occur (as illustrated in Listing 4.4).

```

1 @SERVER.feature(
2     TEXT_DOCUMENT_COMPLETION, CompletionOptions(trigger_characters=
3         trigger_characters)
4 )
5 def completions(ls: WooWooLanguageServer, params: CompletionParams) ->
6     CompletionList:
7
8     return ls.completion(params)

```

■ **Code listing 4.4** Example of decorated feature functions used for registration and delegating requests to the `WooWooLanguageServer`.

During server initialization, the `WooWooLanguageServer` will receive the path to the dialect-definition file. This configuration will remain in effect for the server's lifetime, requiring a restart to change dialects.

4.5.1.3 Running the Server

There are three methods to start and establish a connection with a pygls server [43]:

1. **TCP:** Primarily used for debugging and development. This method allows the server to run over a Transmission Control Protocol (TCP) connection, is easy to set up, and enables remote access if needed.

- 2. WEBSOCKET:** Similar to TCP, but specifically used to expose the language server to browser-based editors.
- 3. STDIO:** The preferred mode for production. It is used when the server is started by the client as a child process.

The `WooWooLanguageServer` will utilize STDIO mode. This choice aligns with its intended use, where it is initiated as a child process by a VSCode extension or a similar client application.

4.6 VSCode Extension Integration

The WooWoo VSCode extension integrates the WooWoo Language Server to provide language features directly within the VSCode editor. This design approach simplifies user setup by automatically managing the language server, thereby facilitating seamless distribution and usage.

To accelerate development, Microsoft's Python extension template was chosen as a foundation. This decision benefits from established language server integration patterns and access to Python tooling within VSCode [39].

The extension is written in TypeScript and utilizes the VSCode `LanguageClient` API to interact with the LSP. It depends on the Python extension by Microsoft, which is automatically installed if not already present. Notably, Python environment management is left to the user, offering flexibility in setup.

Client-side Syntax Highlighting

The LSP does not directly define support for syntax highlighting; this task is typically handled by the client. For example, VSCode utilizes TextMate grammars to parse and assign colors to tokens [44].

However, the LSP offers the *Semantic Tokens* feature, which is used to enhance traditional syntax highlighting. In contrast to TextMate grammars, which are constrained by the regex-based parsing mechanism and single-file scope, the *Semantic Tokens* can leverage the project-wide awareness and sophisticated parsing capabilities of semantic token providers (often language servers). This allows for capturing finer-grained details and nuances within the code. Semantic highlighting, powered by *Semantic Tokens*, is applied on top of the existing syntax highlighting provided by tools on the client side. [45, 46]

While it would be technically possible to do all highlighting on the server-side, it is not ideal, as language servers or other semantic token providers can take some time to load and cause the highlighting to have delays. This option was also explored in the WooWoo Language Server, but was deemed unsuitable due to the delays at launch and even during editing.

For optimal user experience, the WooWoo Language Server will adopt a hybrid highlighting approach. It will provide semantic highlighting limited to highly contextual parts of the language, which would be hard or impossible to capture using TextMate grammar. The rest of the highlighting will be done client-side using a TextMate grammar.

Implementation

This chapter outlines the implementation of the WooWoo Language Server, starting with the development of a Tree-sitter grammar for WooWoo. It then discusses the key components of the analysis engine, including vital build files, and details the development of the communication layer through pygls, which integrates the analysis engine.

Next, the chapter focuses on the integration of the language server into a VSCode extension, including the setup of the extension manifest and local grammar configurations.

Finally, the chapter concludes with a discussion of potential future improvements to various parts of the project.

5.1 Parser Implementation

The parser implementation resides in its own repository¹, named `tree-sitter-woowoo` to align with existing Tree-sitter grammar naming conventions.

Developing the Tree-sitter grammar involved two main components [47]:

- 1. Writing Grammar Rules:** Tree-sitter grammar rules are defined within the `grammar.js` file using JavaScript syntax. JavaScript offers familiarity and convenience for expressing regular expressions. “Of course during parsing, Tree-sitter does not actually use JavaScript’s regex engine to evaluate these regexes; it generates its own regex-matching logic as part of each parser.” [47]
- 2. Implementing a Scanner:** To recognize structures which are impossible or inconvenient to describe with regular expressions, as well as for storing state (e.g., indentation levels), it’s necessary to implement an external scanner. This part is optional, but needed for an indent-based language like WooWoo.

A C parser (`parser.c`) is generated from the grammar rules defined in `grammar.js`. If an external scanner is included, the parser relies on it to recognize specific tokens (these are configured within the grammar). The external scanner is also crucial for accurate error recovery. Poor scanner implementation can severely compromise the parser’s functionality.

5.1.1 Defining Grammar Rules

Each rule within the language is encapsulated within a distinct JavaScript function, and rules can reference each other using the `$` object. Terminal symbols are defined using JavaScript strings

¹Available at: <https://gitlab.fit.cvut.cz/woowoo/lsp/tree-sitter-woowoo>

and regular expressions.

Tree-sitter facilitates rule composition through functions that allow the combination of other rules into sequences (where rules must match consecutively), alternatives (where any one of the rules must match), repetitions (either 0 or more times, or 1 or more times), and optional matches.

A challenge in defining Tree-sitter grammar, given Tree-sitter’s close adherence to the LR(1) [47] parsing algorithm, was resolving conflicts between rules. Tree-sitter provides mechanisms to resolve these LR(1) conflicts by setting precedences or defining left or right associativity for rules. In the implementation of `tree-sitter-woowoo`, numerical precedences were unnecessary. Such precedences are primarily useful in resolving operator precedence conflicts in programming languages, a construct not present in WooWoo, which is a markup language.

5.1.1.1 Grammar Structure

The `source_file` rule acts as the starting point or root symbol of the grammar. It is composed of include statements, document parts, and empty lines. Each document part contains an optional meta block, followed by a sequence of blocks and wobjects, each defined by their respective rules. The structure and order of these elements reflect the syntax of the WooWoo language, as detailed in Section 1.2.

The rule structure is exemplified by the `wobject` rule presented in Listing 5.1. The rule begins with a “.”, followed by `wobject_type`², and a “:”.

It may, optionally, include a `meta_block`, followed by an indented `wobject_body`, which contains one or more blocks, each separated by multiple empty lines. To accurately represent indentation, `_ex_indent` and `_ex_dedent` tokens were introduced, signifying increases and decreases in indentation levels, respectively. This method draws inspiration from other indentation-sensitive languages, as evidenced by the `tree-sitter-python` grammar³.

It’s also important to mention the use of the `_ex` prefix in the rules, which denotes *external* tokens—those recognized by the scanner. This naming convention was specifically developed for `tree-sitter-woowoo` to facilitate the identification of scanner-processed tokens.

```

1  wobject: $ => seq(
2      '.',
3      $.wobject_type,
4      ':',
5      choice($_newline_char,
6          $.meta_block),
7      $_ex_indent,
8      $_wobject_body,
9      $_ex_dedent
10 ),
11
12  wobject_type: $ => seq(
13      $_uppercase_letter, // helper rule
14      repeat($_letter) // helper rule
15  ),
16
17  _wobject_body: $ => seq(
18      $.block,
19      repeat(
20          seq(

```

²`wobject_type` is constructed from helper rules that remain hidden in the parse tree to assist in defining elements like letters, including any Unicode letter to accommodate Czech characters. Rules starting with an underscore are not represented as separate nodes in the parse tree; instead, their characters are integrated into the parent node [47].

³See the `tree-sitter-python` grammar at <https://github.com/tree-sitter/tree-sitter-python/blob/master/grammar.js>.

```

21         $_ex_multi_empty_line ,
22         $.block
23     )
24 )
25 ),

```

■ **Code listing 5.1** The rule in `tree-sitter-woowoo` capturing `wobject`.

5.1.1.2 Setting Precedences

An LR(1) conflict arises when two rules overlap in a manner that creates ambiguity with one token of lookahead [47]. Tree-sitter reports such conflicts during parser generation and provides explanations of the ambiguous interpretations. Some LR(1) conflicts in `tree-sitter-woowoo` could be resolved by specifying left or right associativity. See Listing 5.2 for an example of how associativity is applied in the grammar.

```

1  block: $ => prec.right(seq( // specifying right associativity
2      choice($_outer_environment, $.text_block),
3      repeat(choice(
4          seq($_ex_empty_line,
5              choice($_explicit_outer_environment,
6                  $.text_block)
7          ),
8          $.implicit_outer_environment
9      )
10 )
11 ))),

```

■ **Code listing 5.2** Setting right associativity in the `block` rule to resolve ambiguity.

In the example, without the specified right associativity, a conflict would arise due to two possible interpretations for the same symbol sequence. Right associativity dictates that Tree-sitter prefers the longer rule, which means it will continue with the `block` when encountering a single empty line separator (`_ex_empty_line`). Without this associativity, specifically, Tree-sitter cannot determine whether to continue the `repeat` function in the `block` rule upon encountering an empty line—a situation that arises from the way the `tree-sitter-woowoo` grammar is constructed. See Listing 5.3 for potential interpretations and solutions for the `block` rule, as suggested by Tree-sitter, when associativity is not specified.

```

1  Unresolved conflict for symbol sequence:
2
3  ',.' document_part_type ',.' document_part_title _ex_empty_line
4  text_block + _ex_empty_line ...
5
6  Possible interpretations:
7
8  1: ',.' document_part_type ',.' document_part_title _ex_empty_line
9  (block text_block + block_repeat1)
10 2: ',.' document_part_type ',.' document_part_title _ex_empty_line
11 (block text_block) + _ex_empty_line ...
12
13 Possible resolutions:
14
15 1: Specify a left or right associativity in 'block'
16 2: Add a conflict for these rules: 'block'

```

■ **Code listing 5.3** Failed parser generation due to ambiguity in the grammar.

5.1.1.3 Allowing Conflicts

Some LR(1) conflicts are intended to exist in the grammar. These can be declared, and when they occur at runtime, Tree-sitter will use the GLR algorithm to explore all of the possible interpretations [47].

The conflicting rules are declared in the `conflicts` array of arrays, where each array specifies a set of rules involved in an LR(1) conflict. In `tree-sitter-woowo`, there are three conflict sets, each containing one rule.

The first two conflicts involve the `document_part` and `document_part.body` rules. Here, the ambiguity occurs in certain cases when a structure separator is encountered (`_ex_multi_empty_line`), making it unclear whether it signifies the start of a new document part or just separates elements within the existing document part.

The last conflict is within the body of an inner environment (`verbose_inner_environment_body` rule). With only one token of lookahead, it is ambiguous whether a quotation mark (") in the body of a verbose inner environment marks the end of the environment or the start of another, nested verbose inner environment.

5.1.1.4 External Symbols

The grammar file contains an `externals` array, which lists all the tokens recognized by the external scanner. “If a token in the `externals` array is valid at a given position in the parse, the external scanner will be called first before anything else is done. This means the external scanner functions as a powerful override of Tree-sitter’s lexing behavior, and can be used to solve problems that can’t be cracked with ordinary lexical, parse, or dynamic precedence. [47]”

Within the `tree-sitter-woowo` grammar, external symbols are used for tracking the indentation level using special `_ex_indent` and `_ex_dedent` tokens, emitted whenever an increase or decrease in indentation is detected. In addition to that, it handles the emission of newlines and empty lines, as these tokens have to be managed carefully within the context.

Additionally, the external scanner is used to disambiguate text-related tokens, particularly the dot character. This character could represent the start of a short inner environment, signify a start of meta-information of a verbose inner environment, or hold no special meaning. The scanner leverages lookahead to make the correct distinction.

5.1.2 External Scanner

Tree-sitter’s external scanner mechanism enables custom, language-specific parsing logic for tokens whose structure is complex or inefficient to define using regular expressions alone. Implementing a scanner involves defining five C functions, with names derived from the language’s name. In the `tree-sitter-woowo` implementation, a custom `Scanner` struct encapsulates the parsing logic and state management, with these core functions delegating work to it. Tree-sitter’s parser calls these functions as needed:

`tree_sitter_woowo_external_scanner_create` Called only once, used to instantiate and return a pointer to the `Scanner` object, which maintains parsing state. For stateless languages, this function can return `NULL`.

`tree_sitter_woowo_external_scanner_destroy` Simply frees any memory used by the scanner. It receives the pointer returned by the `create` function.

`tree_sitter_woowo_external_scanner_serialize` Receives a pointer to the scanner and a byte buffer. Stores the scanner’s entire current state within the buffer, allowing Tree-sitter to restore this state later.

tree_sitter_woowoo_external_scanner_deserialize Used to deserialize the scanner's state. It's called with a pointer to the `Scanner` object and the byte array obtained from the `serialize` function.

tree_sitter_woowoo_external_scanner_scan The function responsible for recognizing external tokens, as detailed in the following section.

The scanner implementation is located in `scanner.cc` and is written in C++.

Important Note: In February 2024, Tree-sitter deprecated C++ for scanner development, favoring C⁴. While the `tree-sitter-woowoo` implementation was completed prior to this announcement, migration to C should be straightforward if needed for future versions.

5.1.2.1 Scan Method Overview

The `scan` method within the `Scanner` struct is the core function responsible for recognizing external tokens. It has two important parameters, briefly described below, with the full specifications available in the Tree-sitter documentation [25, 47],

TSLexer * lexer This struct contains methods for accessing the current parsing state. It allows lookahead by one character, advancement to the next character, and marking the end of the current token. It cannot go back; every advanced character is considered part of the current token unless `mark_end` was called beforehand.

const bool * valid_symbols This array contains information about which tokens (as defined in the `externals` array) are valid at the current position. It also includes a marker token indicating whether the parser is in *error recovery* mode.

The `scan` method returns either `FALSE` (no token found) or `TRUE` (token found). If `TRUE`, the token type is indicated to the lexer by setting its `result_symbol` property. The function can only return one token at a time. Its implementation is divided into segments with comments for guidance, and it branches based on the lexer's position, valid symbols, and the `Scanner` state (including the current indentation level and unprocessed tokens).

Unprocessed tokens refer to a situation where the previous scan detected an indentation level change of more than one level. Due to the scanner returning only one token at a time, unprocessed tokens are emitted immediately upon the next call to the `scan` method.

5.1.2.2 Scanner Structure and State Management

The WooWoo external scanner maintains parsing state primarily by tracking indentation levels. The mechanism of serializing the state, along with the file's overall structure, is illustrated in Listing 5.4.

The five functions mandated by Tree-sitter are located within the `extern "C"` block. Functionality is delegated to the custom `Scanner` class, which handles state tracking and serialization. The remaining functions discussed in Section 5.1.2 are handled in similar manner.

Because WooWoo does not allow for variable indentation length (unlike Python for example), there is no need to store the indentation in a stack. It is sufficient to store just one numerical value representing the current indentation level. Spaces needed for one indentation level can be easily configured in the `Scanner` by setting the `INDENT_LEN` variable. However, if indentation of different lengths within the same source file was allowed, it would require significant changes to the scanner's detection of `_ex_indent` and `_ex_dedent` tokens.

⁴C++ deprecation commit: <https://github.com/tree-sitter/tree-sitter/commit/74812ced1b0bec57f010bb240f35742fdcf1d20a>

```

1 // Custom scanner to store state and handle the scanning process
2 struct Scanner {
3     int16_t indent_level;
4     int16_t unprocessed_indentation;
5
6     unsigned serialize(char *buffer) {
7         size_t i = 0;
8         buffer[i++] = indent_level;
9         buffer[i++] = unprocessed_indentation;
10        return i;
11    }
12
13    bool scan(TSLexer * lexer, const bool * valid_symbols){ ... }
14
15    /* Other attributes and methods (omitted) ... */
16 }
17
18
19 extern "C" {
20 // Functions called by the parser generated by Tree-sitter
21 unsigned tree_sitter_woowoo_external_scanner_serialize(void *payload,
22     char *state) {
23     Scanner *scanner = static_cast<Scanner *>(payload);
24     return scanner->serialize(state);
25 }
26 /* Functions for create, deserialize, destroy and scan (omitted) ... */
27
28 }

```

■ Code listing 5.4 Structure of the scanner.cc file

5.1.3 Parser Integration

Before transitioning to C++, the Tree-sitter parser was used directly from Python. To avoid the need for a C++ compiler on the client to compile the parser, a cross-compilation pipeline using the dockcross⁵ cross-compiling toolchain Docker images was established on GitLab. This pipeline compiled the parser for the most common systems, and the resulting binaries were utilized by the Tree-sitter Python bindings. Although this pipeline is no longer in use, its configuration can still be found in the repository's past commits⁶.

Switching to using C++ for the analysis engine simplified the entire process. The parser, written in C, along with the scanner in C++, are now easily integrated directly into the project's build process.

5.2 Analysis Engine Implementation

The analysis engine (referred to as wuff) implementation, resides within a dedicated Git repository⁷. This repository contains both the C++ core of the analysis engine and the Python packaging infrastructure needed for seamless integration into the WooWoo Language Server.

⁵More about dockcross: <https://github.com/dockcross/dockcross>

⁶See <https://gitlab.fit.cvut.cz/woowoo/lsp/woowoo-language-server/-/blob/ac92738232faeb825d7f95496a54729d7b62d6d8/.gitlab-ci.yml>

⁷Available at: <https://gitlab.fit.cvut.cz/woowoo/lsp/wuff>

The `src/` directory contains the C++ codebase, with a clear separation between source (`.cpp`) and header (`.h`) files. `CMakeLists.txt` manages the build configurations. Python integration is achieved using `pybind11`, with bindings definition files located in the same directory. To promote modularity, the codebase is organized into subdirectories (e.g., `components/`, `dialect/`, `document/`, `parser/`), each representing a specific domain of the analysis engine’s functionality. See Table 5.1 for a complete directory structure.

Python packaging files reside alongside the C++ code in the repository’s root directory. This simplifies packaging the analysis engine as a Python module. While the `src/` directory could be separated into its own project, the current setup prioritizes ease of use within this thesis, as the analysis engine is exclusively accessed through the Python package.

This section explores the implementation of each component and outline the configuration of the Python packaging.

wuff/	The root folder of the analysis engine
├─ src/	Source files for the C++ codebase
│ └─ components/	Feature components of the analysis engine
│ └─ dialect/	<code>DialectManager</code> and <code>Dialect</code> nodes
│ └─ document/	Manages WooWoo documents
│ └─ lsp/	LSP structure definitions
│ └─ parser/	Parser component implementation
│ └─ tree-sitter/	Tree-sitter generated parsers
│ └─ utils/	Utility functions and classes
│ └─ Bindings.cpp	<code>pybind11</code> bindings for Python integration
│ └─ CMakeLists.txt	CMake build file
│ └─ WooWooAnalyzer.cpp	Implementation of the <code>WooWooAnalyzer</code>
│ └─ WooWooAnalyzer.h	Header file for the <code>WooWooAnalyzer</code>
├─ tests/	Test cases for the analysis engine
├─ MANIFEST.in	Manifest for Python package data
├─ pyproject.toml	Python project settings and dependencies
├─ README.md	Package description and instructions
└─ setup.py	Python setup script for package distribution

■ **Figure 5.1** Directory tree of the `wuff` analysis engine

5.2.1 Build Configuration with CMake

This project employs CMake (minimum version 3.12) to manage its build process and integrate external dependencies. The CMake configuration (`CMakeLists.txt`) defines builds for two targets: the primary `wuff` module and `WooWooTest`. The latter is designed for easy C++ debugging and excludes Python bindings. This section focuses specifically on the `wuff` module build.

5.2.1.1 Dependency Management

The project leverages CMake’s `FetchContent` module to manage dependencies effectively. The module automates downloading dependencies from various sources (Git repositories, URLs, etc.) and integrates them into the build process. If a fetched dependency has its own `CMakeLists.txt`, it can be easily included using `add_subdirectory()`. Otherwise, source files might need to be manually added to the build. [48]

Below is a list of dependencies fetched for `wuff`, along with brief descriptions of their integration:

tree_sitter Contains the Tree-sitter structures and API. As **tree_sitter** is not CMake-based, the **FetchContent** module fetches the designated tag from its Git repository, and its source files are then manually included in the **wuff** build system.

yaml-cpp A C++ YAML parser/emitter. Used for reading dialect-definition files and transforming them into C++ classes. It is integrated as a subdirectory after fetching.

pybind11 The library chosen to expose the C++ types and functionality in Python. Similarly to **yaml-cpp**, it contains its own **CMakeLists**, and it is simply added as subdirectory.

5.2.2 WooWooAnalyzer

Much of the **WooWooAnalyzer**'s implementation centers around filesystem interactions. This includes tasks like scanning for projects and loading the initial contents of **.woo** files during workspace initialization. Additionally, the **WooWooAnalyzer** manages loaded documents, handling requests for deletion or renaming. Deletion is implemented by removing the document from its container and calling its destructor. Renaming involves updating the document's ID (its path) within the associated maps, which are described in more detail later in this section.

Beyond these filesystem-related responsibilities, the **WooWooAnalyzer** primarily serves as a coordinator for feature components. It exposes a public interface and instantiates these components during its own construction. Client requests are then redirected to the appropriate feature component, as illustrated in Listing 5.5. Similarly, requests to **setDialect** are delegated to the **DialectManager**.

```

1 Location WooWooAnalyzer::goToDefinition(const DefinitionParams &params)
2     {
3     return navigator->goToDefinition(params);
4     }
5 std::vector<CompletionItem> WooWooAnalyzer::complete(const
6     CompletionParams &params) {
7     return completer->complete(params);
8     }

```

■ **Code listing 5.5** Redirecting Requests to Feature Components

5.2.2.1 Handling Paths and URIs

Ensuring consistent handling of file paths and URIs across different operating systems presented a significant development challenge. For example, on Windows, **file:///example.woo** and **file:///Example.woo** refer to the same file, while on Linux, they represent distinct files. Additionally, on macOS, the behavior depends on the specific file system configuration. Importantly, while the LSP primarily uses URIs for file identification [8], the analyzer component also requires actual paths to initially load file content and handle navigating **include** statements effectively.

A suite of utility functions was developed (residing within the **utils** namespace in **utils.h** and **utils.cpp**) to ensure consistent handling of filenames. These functions handle platform-specific conversions between URIs and file system paths. For example, the **uriToPathString** function converts a URI to a platform-appropriate file path, accounting for percent decoding, Windows drive letters, and leading slashes. Similarly, the **pathToUri** function generates valid file URIs from file system paths.

C++ platform macros (like **_APPLE_** or **_WIN32**) are used for platform-specific code execution. The implementation currently treats all macOS file systems as case-insensitive. This is an area for potential future refinement.

5.2.2.2 Loading a Workspace

The `WooWooAnalyzer` employs a structured process to load a workspace, ensuring the discovery and processing of all `.woo` files. This process is initiated by the `loadWorkspace` method, which accepts a `workspaceUri` pointing to the workspace's root directory (typically the user-opened folder in an IDE).

First, the analyzer identifies `WooWoo` project folders within the workspace. A project folder is defined by the presence of a `Woofile`. The `findProjectFolders` function (Listing 5.6) recursively scans the workspace for `Woofile` files, adding their parent directories to a list of project folders.

```

1  std::vector<fs::path> WooWooAnalyzer::findProjectFolders(const fs::path
    &rootPath) {
2      std::vector<fs::path> projectFolders;
3      for (const auto &entry: fs::recursive_directory_iterator(rootPath))
4          {
5          if (entry.is_regular_file() && entry.path().filename() == "
            Woofile") {
6              projectFolders.push_back(entry.path().parent_path());
7          }
8      }
9      return projectFolders;
}

```

■ **Code listing 5.6** Finding projects folder by scanning for Woofiles.

Next, the analyzer iterates through each project folder's contents. All `.woo` files are loaded using the `loadDocument` function, which creates and stores a `DialectedWooWooDocument` instance for each file. These documents are associated with their respective projects for project context aware analysis.

Finally, the analyzer gathers any `.woo` files within the workspace that are not part of a recognized project. These files are grouped together, providing a means to process them even if they lack explicit project association. An alternative approach would be to create a separate artificial project for each unassigned file. However, for the time being, the assumption that unassigned files may potentially be related provides a more flexible context for analysis.

Upon completion, the workspace is represented internally by two key maps: the `projects` map and the `docToProject` map. The `projects` map organizes `DialectedWooWooDocument` instances by their associated projects. The `docToProject` map facilitates quick lookups of a document's project based on its file path. Together, these maps enable efficient retrieval of any `DialectedWooWooDocument` within the workspace.

5.2.3 DialectManager

The primary implementation goal for the `DialectManager` was to deserialize the dialect-definition file into well-organized data structures and ensure an interface optimized for efficient querying by feature components. Table 4.1 outlines the provided interface.

The `Dialect` class and its components (like `DocumentPart` and `Wobject`, detailed in Figure 4.6) reside in separate source and header files. Each component encapsulates its attributes and a `deserialize` function for processing the YAML data.

5.2.3.1 Loading the Dialect

To specify a dialect, a path to its definition file (a YAML file) is provided. This path can be passed either through the `DialectManager`'s constructor or by invoking the `loadDialect` method with the path as an argument.

The loading operation starts with the file being read and parsed via the `LoadFile` function from `yaml-cpp`. This parsing results in an instance of the `Node` object (also provided by `yaml-cpp`). The entire dialect definition is then deserialized into a `Dialect` class instance. The `DialectManager` maintains a single instance of this `Dialect` class (stored in the `activeDialect` attribute), which has a `deserialize` method that accepts a `Node`. The `Dialect` class initiates the deserialization process by invoking the `deserialize` method on its associated components. These components, in turn, recursively invoke `deserialize` on their own nested components. See Listing 5.7 for a snippet from the `Dialect` class's deserialization method.

```

1 void Dialect::deserialize(const YAML::Node& node) {
2     // Check for required nodes (omitted)
3
4     // Deserialize attributes
5     name = node["name"].as<std::string>();
6     version_code = node["version_code"].as<std::string>();
7     // Other Dialect attributes (omitted)...
8
9     // Deserialize DocumentParts
10    if (node["document_parts"]) {
11        for (const auto& dpNode : node["document_parts"]) {
12            auto dp = std::make_shared<DocumentPart>();
13            dp->deserialize(dpNode);
14            document_parts.push_back(std::move(dp));
15        }
16    }
17    // Deserialize Wobjects, Environments and Shorthands (omitted)...
18 }

```

■ **Code listing 5.7** Snippet from the `Dialect` class's deserialization method.

5.2.3.2 Preprocessing for Efficiency

The `DialectManager` preprocesses the loaded dialect-definition file to optimize the performance of future function calls. It iterates over all valid parameter values and pre-calculates the results, storing them in hashmaps for instant retrieval. This approach avoids the need for repetitive, time-consuming searches.

For example, the `referencesByTypeName` hashmap is an important optimization. For each structure capable of referencing another structure, it assembles a vector of its possible `References` (value) by its `typeName` (key). This allows for instant lookup of what a given type (like the `reference` environment in `FIT-Math`) can reference. Without this map, the `DialectManager` would need to scan the entire dialect definition on each query, which would be particularly inefficient for components like the `Completer` that frequently perform reference lookups.

5.2.4 Reference Mapping in `DialectedWooWooDocument`

The `DialectedWooWooDocument` class builds upon the `WooWooDocument` class, which handles parsing using the `Parser` component and CST storage. To enable dialect-aware reference handling, it integrates with the `DialectManager` to obtain reference information and performs pre-processing. This pre-processing step allows language features to efficiently query the document for all referencable and referencing fields. The primary addition in this class is the use of `Tree-sitter` queries to index the document.

Firstly, the class iterates over each `meta_block` and uses the `fieldQuery` (see Listing 5.8) to obtain all keys within them. If a key within a `meta_block` matches a reference definition within the dialect (as detailed in Section 4.4.6.1), its position is stored in the `referencableNodes`

hashmap. For example, if a dialect defines an environment that can reference any structure containing a `meta_block` with a `label` key, all encountered `label` positions and their associated values are stored. This preprocessing significantly speeds up subsequent definition searches, enabling features such as retrieving all referencable fields by name (e.g. `label`) and their positions within a document.

```

1 (block_mapping_pair
2   key: (flow_node
3     [
4       (double_quote_scalar)
5       (single_quote_scalar)
6       (plain_scalar)
7     ] @key
8   )
9   value: (flow_node) @value
10 )

```

■ **Code listing 5.8** Query (S-expression) for finding key:value pairs within `meta_blocks`.

In addition to storing referencable fields, the indexing process locates all referencing fields. This involves querying for structures capable of containing references (meta block fields, short inner environments, and shorthands) and matching them against names defined within the dialect. For example, when used with the `FIT-Math` dialect, only the positions and values of `reference` and `eqref` environments are stored, as these are the only environments capable of referencing other document elements.

5.2.5 Feature Components

Feature components share a common implementation pattern. Each component defines a set of Tree-sitter queries tailored to the specific language features it provides. For example, the `Hoverer` component includes queries for identifying hoverable nodes, while the `Navigator` component includes queries for locating `include` statements and other definition points. These queries are precompiled during component instantiation using `ts_query_new` and being reused throughout the component lifecycle for efficiency.

Language feature methods construct tree cursors on-demand using these queries. Cursors are executed against the document CST, yielding nodes matching the query. To ensure the most relevant results, most queries are limited to a specific range, typically surrounding the position where the user triggered the feature. This range restriction is achieved by calling `ts_query_cursor_set_point_range` on the cursor before execution via `ts_query_cursor_exec`.

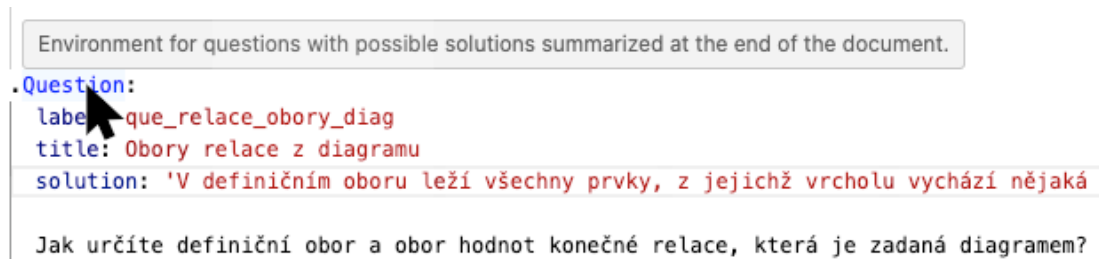
Following cursor execution, the returned matches are analyzed based on the specific requirements of the feature. For instance, the `semanticTokens` feature of the `Highlighter` component focuses on the position and type of matched nodes. Meanwhile, the `hover` feature also retrieves the node's text, such as the name of an outer environment. This text is necessary for a query to the `DialectManager` to obtain a description of the structure, identified by its type and name. It is ultimately displayed to the user (as shown in Figure 5.2).

Figures illustrating all other implemented features not shown in the main text are presented in Appendix B.

Example Feature Component Implementation

The `references` method in the `Navigator` demonstrates the typical workflow of feature components. This feature's purpose is to find all references to a given structure across the entire project.

Upon a user executing the *Find All References* feature at a given position in the editor, the following sequence occurs:



■ **Figure 5.2** Obtaining the description of the *Question* environment defined by the FIT-Math dialect.

1. **Meta_block Presence Check:** The feature must be executed on a `meta_block`, as that is where the key being referenced is defined (e.g., `label: section1`). This is checked by running a simple query (see Listing 5.9) limited to the executed position, verifying the presence of a `meta_block` node.
2. **Meta_block Field Extraction:** If a `meta_block` exists at the position, an additional query is performed to identify the exact key-value pair within the `meta_block` where the feature was triggered. Note that this is a YAML query, distinct from the WooWoo query in the previous step. The query is captured in Listing 5.8.
3. **Searching Project Documents:** Once the key-value pair is obtained, the feature iterates over all documents within the same project as the one where the feature was executed. Using the hashmaps within `DialectedWooWooDocument` objects (as described in Section 5.2.4), it queries each document for references matching the given key and its associated value. This process is demonstrated in Listing 5.10.

```
1 // Line and character are given as arguments.
2 TSPoint start_point = {line, character};
3 // end_point is next to start_point to specify the exact position, not a
  range.
4 TSPoint end_point = {line, character + 1};
5 // Limiting the search to the position where the action was executed.
6 ts_query_cursor_set_point_range(cursor, start_point, end_point);
7 // queries[findReferencesQuery] contains the "(meta_block) @type" string
  .
8 ts_query_cursor_exec(cursor, queries[findReferencesQuery],
  ts_tree_root_node(document->tree));
9
10 TSQueryMatch match;
11 std::string nodeType;
12 if (ts_query_cursor_next_match(cursor, &match)) {
13     if (match.capture_count > 0) {
14         TSNode node = match.captures[0].node;
15         nodeType = ts_node_type(node);
16
17         if (nodeType == "meta_block") {
18             // The user executed "Find All References" on a meta-block.
19             return findMetaBlockReferences(params);
20         }
21     }
22 }
```

■ **Code listing 5.9** Verifying the presence of a `meta_block` at feature execution position.

```

1 void
2 Navigator::searchProjectForReferences(std::vector<Location> &locations,
3   WooWooDocument *doc, const Reference &reference,
4   const std::string &referenceValue)
5   {
6   for (auto projectDocument: analyzer->getDocumentsFromTheSameProject(
7     doc)) {
8     for (auto refLocation: projectDocument->
9       findLocationsOfReferences(reference, referenceValue)) {
10      projectDocument->utfMappings->utf8ToUtf16(refLocation);
11      locations.emplace_back(refLocation);
12    }
13  }
14 }

```

■ **Code listing 5.10** Project-wide search for references.

5.2.6 Python Bindings

To implement Python bindings, the `PYBIND11_MODULE` macro was utilized to create a `wuff` module. Within this module, various classes were defined, with the primary focus being the `WooWooAnalyzer`. This class exposes a constructor, enabling instantiation from Python, and its entire public API. This API encompasses methods for setting up the workspace and accessing the language features.

In addition to the `WooWooAnalyzer`, bindings were also implemented for all parameter types necessary to invoke these language features. Listing 5.11 showcases the creation of the module, the exposure of language features, and an example of the `Position` class binding. This class is extensively utilized across the project to represent the position of text in documents.

```

1 PYBIND11_MODULE(wuff, m) {
2   // The main Analyzer class.
3   py::class_<WooWooAnalyzer>(m, "WooWooAnalyzer")
4     .def(py::init<>())
5     .def("set_dialect", &WooWooAnalyzer::setDialect)
6     .def("load_workspace", &WooWooAnalyzer::loadWorkspace)
7     .def("hover", &WooWooAnalyzer::hover)
8     // Other methods (omitted) ...
9
10  // Parameter classes
11  py::class_<Position>(m, "Position")
12    .def(py::init<uint32_t, uint32_t>())
13    .def_readwrite("line", &Position::line)
14    .def_readwrite("character", &Position::character);
15  // Other classes (omitted) ...
16 }

```

■ **Code listing 5.11** Defining Python bindings.

5.3 Communication Layer Implementation

The communication layer implementation resides in its own repository⁸. It builds mainly on the `pygls` and `wuff` Python modules. As most of the actual work is done by these packages, the implementation is quite simple and short.

It can be divided into three parts:

1. **Utility/conversions functions:** Convert data between LSP-specified types and the internal representations used by `wuff`.
2. **WooWooLanguageServer class:** The core logic where LSP types are adapted and analysis tasks are delegated to `wuff`.
3. **Feature registration:** Defines and registers the LSP features supported by the server. The entry point for LSP requests.

5.3.1 Utility Functions

For seamless conversion between parameter types imported from `lsprotocol.types` (used by `pygls`) and those expected by `wuff`, various conversion functions were implemented (located in `convertors.py`). These functions handle bidirectional conversion. To avoid class name conflicts, `wuff` structures are imported with a `Wuff` prefix (see Listing 5.12 for an example).

```

1 from wuff import Position as WuffPosition
2 from lsprotocol.types import Position
3
4 def wuff_position_to_ls(wuff_position: WuffPosition) -> Position:
5     return Position(line=wuff_position.line,
6                    character=wuff_position.character)

```

■ **Code listing 5.12** Converting `Position` from the `wuff` module to `Position` from `lsprotocol.types`.

Additionally, a platform-independent utility function for converting URIs to file paths is provided in `utils.py`. The `constants.py` file contains initial parameters for semantic token configuration and a preconfigured `FileOperationRegistrationOptions` instance. This instance, reused across multiple features, defines the file types the language server is interested in. Currently, the filter is set to accept all files, with the filtering logic delegated to the `wuff` module.

5.3.2 WooWooLanguageServer Implementation

The `WooWooLanguageServer`, inheriting from `pygls` provided `LanguageServer`, is implemented in the `woowoo_language_server.py` file. In its `__init__` method, it instantiates the `WooWooAnalyzer` (provided by the `wuff` module) and configures it (e.g. sets tokens and modifiers for semantic highlighting).

The core logic of the `WooWooLanguageServer` focuses on mediating between LSP types and those used by the `wuff` library. It contains dedicated methods for each registered LSP feature. These methods use conversion functions to transform data between LSP types and the types expected and returned by the `wuff` analyzer. See Listing 5.13 for an example.

```

1     def go_to_definition(self, params: DefinitionParams) -> Optional[
2         Location]:
3         wuff_params = WuffDefinitionParams(
4             WuffTextDocumentIdentifier(params.text_document.uri),

```

⁸Available at: <https://gitlab.fit.cvut.cz/woowoo/lsp/woowoo-language-server>

```

4         WuffPosition(params.position.line, params.position.character
5         ),
6         return wuff_location_to_ls(self.analyzer.go_to_definition(
7         wuff_params))
8     }

```

■ **Code listing 5.13** Typical method of `WooWooAnalyzer`, constructing parameters for `wuff`, and then returning the result in `pygls`-compatible type.

Setting a Dialect

As established in Section 4.5.1.2, the dialect must be set during server initialization. This is achieved through the `initializationOptions` field provided within the `InitializeParams` of the *Initialize* request in the LSP. This field allows the client to send any user-provided initialization options to the server during startup, as documented in the LSP Specification [8].

The server specifically expects a path to the dialect file in the `dialectFilePath` key within `initializationOptions`. If the user (client) does not provide a path to a dialect, the server falls back to `FIT-Math`, as currently, it is the most logical option due to its adoption. The `FIT-Math` dialect is included with the server.

The selected dialect file path is used to configure the `WooWooAnalyzer` before a `loadWorkspace` request is issued. Both the dialect setting and the `loadWorkspace` action are triggered in response to the client's *Initialize* request.

5.3.3 Feature Registration and Launching the Server

The `server.py` file provides the entry point for clients to interact with the `WooWoo Language Server`. It contains an instance of the `WooWooLanguageServer` class and defines the `start` function, used to launch the server in `STDIO` mode (as discussed in Section 4.5.1.3).

Importantly, `server.py` is where language features are registered to the instance. Listing 5.14 demonstrates this structure. Adding a new feature is streamlined:

1. **Feature Registration (`server.py`):** Decorate a function using `pygls`' `@SERVER.feature` decorator.
2. **Mediator (`WooWooLanguageServer`):** Create a method to handle the feature request, using converters to bridge LSP and `wuff` types.
3. **Analysis (`wuff`):** Implement the analysis logic within the `WooWooAnalyzer`.

```

1 from woowoo_language_server import WooWooLanguageServer
2
3 SERVER = WooWooLanguageServer("WooWoo Language Server", "v1.0")
4
5 # registering INITIALIZE feature
6 @SERVER.feature(INITIALIZE)
7 def initiliaze(ls: WooWooLanguageServer, params: InitializeParams) ->
8     None:
9     logger.debug("[INITIALIZE]")
10
11     ls.initialize(params)
12
13 # other features registration (omitted) ...
14 # function to be used by client to start the server

```

```
15 def start() -> None:
16     SERVER.start_io()
```

■ **Code listing 5.14** The structure of the `server.py` file. Instantiating `WooWooLanguageServer`, registering methods, and providing the `start` function for client to launch the server.

5.4 VSCode Extension Implementation

As outlined in Section 4.6, the extension is built upon Microsoft’s Python extension template. This choice simplifies development, as essential logic for managing Python tools is provided by the template. The template included marked sections indicating where customization was required. This involved configuring the extension metadata, specifying dependencies, and setting up the development environment. The following subsections detail the most significant implementations and additions made on top of this foundation.

5.4.1 Extension Manifest

“Every Visual Studio Code extension needs a manifest file `package.json` at the root of the extension directory structure [49].” This file defines various metadata and configurations for the extension. Notably, the dependencies are specified here. In this case, the dependencies were left unchanged from the extension template, as the primary dependency is the Python extension, already included.

Importantly, the manifest file contains a *contributes* section, where an extension declares how it extends VSCode’s features. For the WooWoo VSCode extension, there are several contributions (referred to as **Contribution Points**), described in the following sections.

5.4.1.1 Language Contribution

The *languages* contribution point is used to introduce a new language to VSCode (or enrich the knowledge about an existing one). This extension defines a new language with the `id` of `woowoo`, named WooWoo, and associates it with files having the `.woo` extension (as illustrated in Listing 5.15).

Additionally, it allows for the configuration of basic declarative features for the defined language. For this purpose, a `language-configuration.json` file was created. This file contains generally applicable settings such as `surroundingPairs`. This feature enables automatic enclosure of selected text with commonly used brackets and quotation marks, further enhancing the user’s editing experience.

```
1  "contributes": {
2      "languages": [
3          {
4              "id": "woowoo",
5              "extensions": [
6                  ".woo"
7              ],
8              "configuration": "./language-configuration.json",
9              "name": "WooWoo",
10             "aliases": [
11                 "WooWoo"
12             ]}],
13     "grammars": [
14         {
15             "language": "woowoo",
```

```

16         "scopeName": "text.woo",
17         "path": "./syntaxes/woo.json"
18     }],
19     ...

```

■ **Code listing 5.15** Defining the WooWoo Language and Grammar Contributions within the Extension Manifest.

5.4.1.2 Grammars Contribution

As discussed in Section 4.6, most of the code highlighting is done on the client-side, not by the language server. This is configured by providing a TextMate grammar for the language, using the *grammars* contribution point (see Listing 5.15).

For this reason, a TextMate grammar for WooWoo had to be implemented. The grammar included in this thesis was inspired by an existing TextMate grammar for WooWoo, created by David Straka in 2021 [2].

The resulting grammar (implemented in `syntaxes/woo.json`) accurately captures most of WooWoo's syntax, but there is room for improvement. For example, it recognizes all quotes (") as the same token, even if used within fragile environments where highlighting of quotes might not be expected. Additionally, it does not attempt to recognize meta blocks, as their interpretation is left entirely up to the language server.

5.4.1.3 Configuration Contribution

This contribution point defines keys that are exposed to the user, allowing customization through User Settings or Workspace settings.

Along with settings predefined in the Microsoft Python extension template (notifications settings, custom Python interpreter), the WooWoo VSCode extension defines a `dialect.file.path` configuration setting. Using this configuration setting, the user is expected to provide a path to a dialect definition file, which will be used by the server, as discussed in Section 5.3.2.

5.4.2 Starting the Language Server Process

The entire WooWoo Language Server repository is included as a submodule within the extension's root directory, specifically in the `bundled/tool/woowoo_pygl` subdirectory. This location is a standard convention for housing extension-specific tools.

The `bundled/tool/` directory also contains the `lsp_server.py` file. This file serves as the entry point for the language server and is executed directly by the Python interpreter. Its primary function is to import the core language server implementation and launch it using the `start` function (as described in Section 5.3.3).

The `createServer` function within `server.ts` is responsible for orchestrating the launch of the language server process from the `lsp_server.py` file. Additionally, it creates and configures an instance of VSCode's `LanguageClient`, establishing the communication channel between the editor and the newly launched language server process. See Listing 5.16 for the core configuration within the `createServer` function.

```

1  async function createServer(
2      settings: ISettings,
3      serverId: string,
4      serverName: string,
5      outputChannel: LogOutputChannel,
6      initializationOptions: IInitOptions,
7  ): Promise<LanguageClient> {
8

```



```

9      // Interpreter and arguments setup (omitted) ...
10
11     const serverOptions: ServerOptions = {
12         command, // Python interpreter
13         args, // python file to execute (lsp_server.py)
14         options: { cwd, env: newEnv },
15     };
16
17     // Options to control the language client
18     const clientOptions: LanguageClientOptions = {
19         // Register the server for woowoo documents
20         documentSelector: [{ scheme: 'file', language: 'woowoo' }],
21         outputChannel: outputChannel,
22         traceOutputChannel: outputChannel,
23         revealOutputChannelOn: RevealOutputChannelOn.Never,
24         initializationOptions, // options containing the dialectFilePath
25     };
26
27     return new LanguageClient(serverId, serverName, serverOptions,
        clientOptions);

```

■ **Code listing 5.16** Snippet from the `createServer` function showing how the server and `LanguageClient` are configured and instantiated.

5.5 Future Work

The primary goal of delivering a functional, accessible language server for WooWoo has been achieved. Since language servers are complex systems and the LSP keeps evolving, there’s always room to improve and add new features to the WooWoo tools created in this thesis.

5.5.1 Tree-sitter Grammar

“Writing a grammar requires creativity. There are an infinite number of CFGs (context-free grammars) that can be used to describe any given language. [47]”

The current grammar serves its purpose well, yet there remains potential for enhancement. This is particularly true for the scanner, where subtle optimizations could streamline its detection mechanisms and enhance the behavior during error recovery.

Another area for subtle refinement is the handling of comments within sequences of empty lines. When a comment is inserted between empty lines that are meant to be recognized as a multi-empty line token, it is currently included within this token.

Lastly, migrating the scanner to C represents a future improvement that should be straightforward. This change primarily involves converting the scanner’s member functions into standalone functions that accept the scanner as a parameter.

5.5.2 Working with Woofiles

Currently, Woofiles are used only to determine project folder locations, but their contents could be leveraged for additional insights. For example, Woofiles may declare the path to a bibliography file containing sources frequently referenced from certain WooWoo environments. Parsing Woofiles to extract this information could enable the language server to provide *Completion* and *Go to Definition* features for these sources.

5.5.3 Expanding Current Features

The implementation of some LSP features could be extended. Some promising enhancements include:

Autocompletion from Bibliography Suggest autocompletions from bibliography files. This enhancement would also necessitate extending the dialect definition format to allow specifying which environments can reference the bibliography.

Autocompletion of Structure Names Enable autocompletion for names of structures defined in the dialect file.

Highlighting of Special Environments Implement highlighting for the contents of math environments and fragile outer environments. This feature presents a challenge as the content of these structures varies across different dialects. Introducing components like `MathHighlighter` or `TikzHighlighter` to the analysis engine could be explored, specifying on a dialect-level which highlighter should be used for different environment contents.

Improved Error Reporting Enhance error reporting capabilities. Current limitations imposed by Tree-sitter could potentially be overcome in its future versions, leading to more precise and helpful error messages.

Diagnosis Extend the diagnostics to include linting features to check meta-blocks for required fields, and ensure that all structure names conform to the used dialect.

Folding Ranges Extend the folding range implementation to support `comment` and potentially `imports` kinds, in addition to the currently used `region` kind.


5.5.4 Adding New Features

The LSP encompasses many capabilities not yet utilized in the WooWoo Language Server. For example, the following features could significantly enhance its functionality:

Auto-formatting Adding a feature that automatically formats files in accordance with the latest WooWoo practices could provide a significant boost, particularly for new authors.

Code Lens This feature could enhance the document editing interface by displaying reference counts directly above document elements, for example, providing a quick overview of their usage without user interaction. *Code Lens* could also display other useful metadata, such as the last editing date for document parts.

Document Linking This would enable link detection within documents to facilitate navigation to external resources. This feature could be particularly useful for WooWoo, where external resources are heavily referenced. Implementing this feature might require extending the dialect definition.



Chapter 6

Distribution

An effective distribution strategy is crucial for language server adoption. While a language server itself is a significant step forward, it's often not enough on its own. Even though most modern editors support the LSP, configuring a specific language server within them may not be straightforward for all users, especially those unfamiliar with the LSP or nontechnical users in general.

The LSP does not specify how it should be integrated into a tool (IDE). “Some tools integrate language servers generically by having an extension that can start and talk to any kind of language server. Others, like VS Code, create a custom extension per language server... [50]”. From a user's perspective, these different integration techniques directly impact the setup process. This can range from simply searching for an extension within the editor's marketplace and clicking install, to manually downloading files and writing configuration settings.

This thesis focuses on VSCode, which, as discussed earlier, requires a custom-made extension (with design and implementation described in Sections 4.6, 5.4). While this approach necessitates more development effort, it greatly simplifies the installation process for end-users.

This chapter describes the details of how the WooWoo VSCode extension and its dependencies are distributed.

6.1 Distributing `wuff`

As detailed in Section 4.4, `wuff` is the core analytical component of the WooWoo Language Server, written in C++ for optimal performance. The choice of C++ over Python introduces the tradeoff of a more complex build process to ensure compatibility across different operating systems and architectures.

Requiring users to compile the package themselves would create a significant barrier, especially for those without a C++ compiler. Expecting users to install a C++ compiler is equally impractical, particularly for nontechnical individuals. Even if comprehensive guidance were provided, the compilation process itself would demand considerable effort from the user, as it can be quite complex and time-consuming, especially on platforms like Windows.

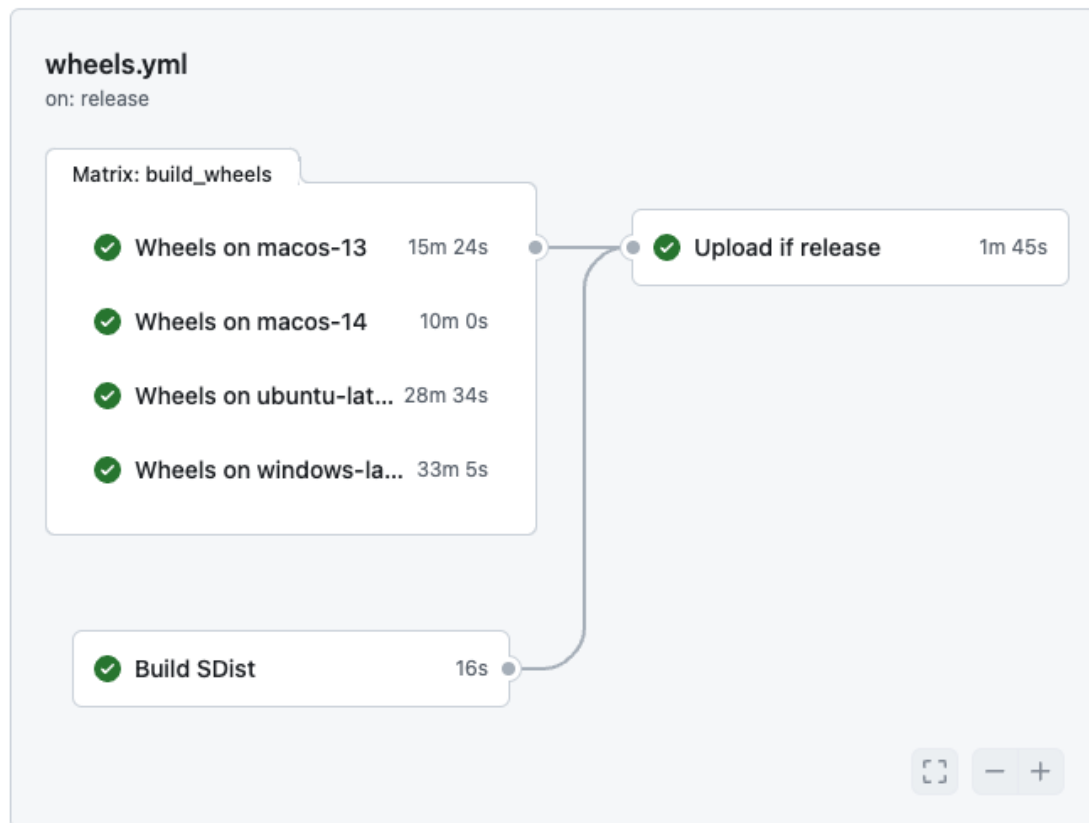
Pre-compiling the package is the most practical solution. However, shipping numerous binaries directly with the extension isn't feasible. The preferred approach is to upload pre-compiled wheels to the Python Package Index (PyPI). PyPI is the standard platform for distributing Python software, offering advantages like centralized discovery and ease of installation (using tools like `pip`). Python wheels are the established binary distribution format [51]. Their dominance is evident, with 359 out of the 360 most downloaded PyPI packages being distributed as wheels (as of March 2024) [52].

6.1.1 Cross-Platform Wheel Building

To build wheels for specific target platforms, direct access to those platforms is required. Continuous Integration (CI) services, like GitHub Actions or GitLab CI, are ideal for this purpose, as they provide the necessary platforms. Tools like `cibuildwheel`¹ automate the creation of platform-specific build environments and the generation of corresponding wheels. “Python wheels are great. Building them across Mac, Linux, Windows, on multiple versions of Python, is not [53].”

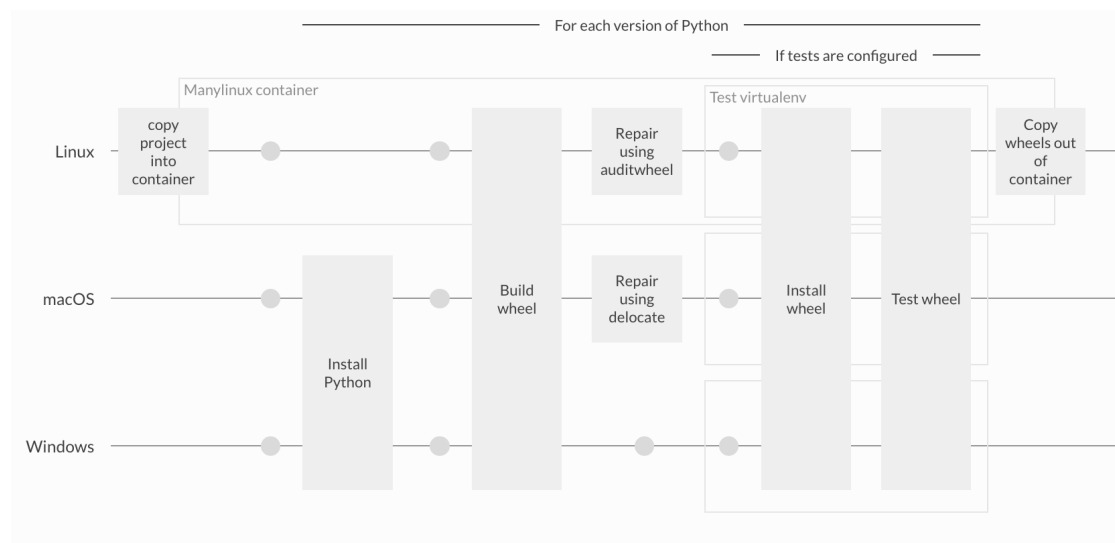
The `cibuildwheel` tool is compatible with various CI services but integrates particularly well with GitHub Actions, which offers broader platform support. The availability of free, standard GitHub-hosted runners for public repositories further strengthens the case for using GitHub Actions.

For these reasons, a pipeline has been established using GitHub Actions, leveraging templates from the official `cibuildwheel` repository. This pipeline configures a GitHub job for each target platform, executing the `cibuildwheel` command to build the wheels. For new releases, a final job aggregates the built wheels and uploads them to PyPI. See Figure 6.1 and Figure 6.2 for a visual representation of the pipeline and the build process. Note that to support both Intel and Apple Silicon chips on macOS, both `macos-13` and `macos-14` runners must be used, respectively.



■ **Figure 6.1** GitHub Actions pipeline jobs for cross-platform wheel generation and PyPI upload.

¹For more information about `cibuildwheel`, visit: <https://github.com/pypa/cibuildwheel>



■ **Figure 6.2** Summarization of the steps that `cibuildwheel` takes on each platform. [53]

6.1.2 Supported Systems

The `wuff` wheels are provided for a broad spectrum of common Python versions and implementations (CPython and PyPy), operating systems, and architectures, facilitated by the use of `cibuildwheel`. To ensure consistency and compatibility with `pygls`—utilized within the communication layer—a minimum of Python 3.8 is required by `wuff`.

For a comprehensive list of supported systems, refer to the `wuff` project page on PyPI² or consult Table 6.1. If a pre-built wheel is not available for a specific system, `pip` will attempt to build the package locally from the source distribution, also available on PyPI.

Note that in the context of wheel building, the tags `manylinux` and `musllinux` indicate compatibility with a wide range of Linux distributions based on the `glibc` and `musl libc` libraries, respectively.

■ **Table 6.1** Supported Systems for the `wuff` Package

Python Version	Platform Arch	Windows		manylinux/musllinux		macOS	
		x86-64	x86	x86-64	i686	ARM64	x86-64
PyPy 3.10		✓	✗	✓/✗	✓/✗	✓	✓
PyPy 3.9		✓	✗	✓/✗	✓/✗	✓	✓
PyPy 3.8		✓	✗	✓/✗	✓/✗	✓	✓
CPython 3.12		✓	✓	✓/✓	✓/✓	✓	✓
CPython 3.11		✓	✓	✓/✓	✓/✓	✓	✓
CPython 3.10		✓	✓	✓/✓	✓/✓	✓	✓
CPython 3.9		✓	✓	✓/✓	✓/✓	✓	✓
CPython 3.8		✓	✓	✓/✓	✓/✓	✓	✓

²`wuff` on PyPI: <https://pypi.org/project/wuff/#files>

6.2 Extension Distribution

To maximize the reach, adoption, and convenience for users of a VSCode extension, choosing the right distribution strategy is essential. There are two main options for distributing VSCode extensions [54]:

Direct Distribution Involves packaging the extension into the installable VSIX format and sharing it directly with users. VSCode provides the option to install an extension from a local VSIX file. However, this method places the onus of discovery and updates on the user.

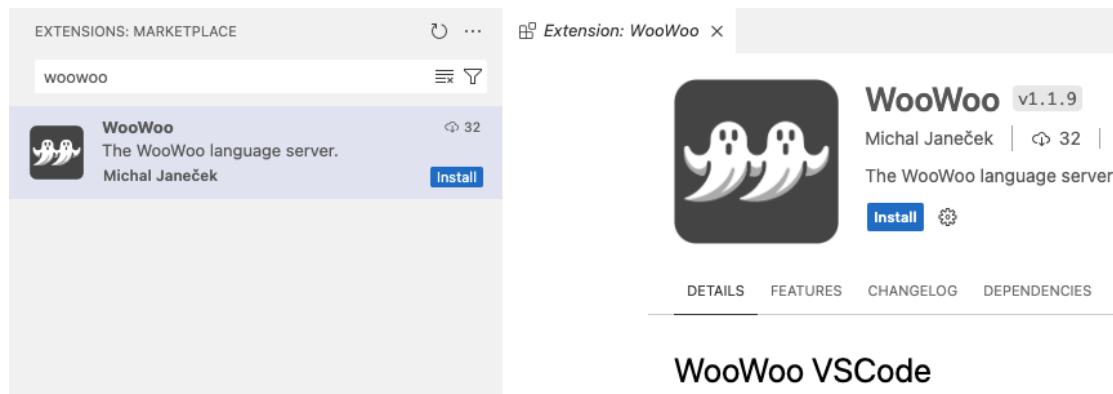
Marketplace Distribution Entails uploading the extension to a marketplace that is directly integrated within VSCode. This method significantly enhances the discoverability of the extension, as users can easily search for and install extensions relevant to their needs directly within the editor. It also streamlines the installation process, often requiring just a few clicks, and handles automatic updates to ensure users always have the latest version.

6.2.1 VSCode Marketplace

The VSCode Marketplace is a centralized platform integrated directly into the VSCode editor, offering a vast collection of over 55,000 extensions (as of March 2024). VSCode’s dominance as the preferred IDE (approximately 73% usage according to the 2023 Stack Overflow Annual Survey [55]) highlights the marketplace’s potential for broad distribution of language tools.

Publishing to the VSCode Marketplace requires creating an organization account in Azure DevOps, generating a Personal Access Token, and utilizing the `vsce publish` command for packaging and publishing [54]. Microsoft provides detailed instructions within their “Publishing Extensions” guide³.

The WooWoo Language Server extension is available on the marketplace⁴, meaning it can be easily installed directly within VSCode by searching for “WooWoo” in the *Extensions* tab (See Figure 6.3).



■ **Figure 6.3** The WooWoo extension listed in the VSCode Marketplace, accessible from within the editor.

6.2.2 Open VSX Registry

“The Open VSX Registry is a vendor-neutral, open source alternative to the Microsoft Visual Studio Marketplace for VS Code extensions. It’s built on the Eclipse Open VSX project, and

³Available at: <https://code.visualstudio.com/api/working-with-extensions/publishing-extension>

⁴<https://marketplace.visualstudio.com/items?itemName=michal-janecek.woowoo-vscode>

delivers on the industry’s need for a more flexible and open approach to VS Code extensions and marketplace technologies [56].”

This marketplace becomes particularly relevant due to the Microsoft VSCode Marketplace’s terms of use, which restrict its use to Microsoft-branded releases. However, there are other distributions of VSCode, such as the open-source Code - OSS. While Code - OSS serves as the foundation for Microsoft’s VSCode, it can be packaged and distributed by various entities, such as Arch Linux, and might have configurations that enable Open VSX by default. [57]

Publishing an extension on the Open VSX Registry requires creating an Eclipse account linked to a GitHub profile and signing the Publisher Agreement. An access token is then generated within the Open VSX profile. The `ovsx` CLI tool is used to create a namespace corresponding to the extension’s publisher field. Finally, the extension, packaged as a VSIX file, is uploaded to the namespace using the `ovsx publish` command. (Note: the `ovsx` tool internally leverages `vsce` for packaging). [58]

To ensure accessibility for all users, including those who prefer non-Microsoft VSCode distributions—which are popular among current WooWoo authors—the extension is also made available⁵ on the Open VSX marketplace.

⁵<https://open-vsx.org/extension/michal-janecek/woowoo-vscode>

Chapter 7

Testing

“Testing shows the presence, not the absence of bugs.” - Edsger W. Dijkstra

Testing is essential for ensuring the quality and reliability of software. This is particularly true for language servers, where errors in grammar parsing or performance lags can severely hinder the user’s editing experience. Throughout this chapter, diverse testing methodologies employed in development will be explored. Due to the project’s scale, testing primarily targets core components.

The chapter starts with grammar tests, the foundation of the language server’s accuracy. Then, user testing and feedback are examined, along with performance tests (including replication instructions). The chapter concludes with automated unit tests for the analysis engine.

7.1 Grammar Tests

Tree-sitter offers a convenient way to verify the generated parser’s correctness through the `tree-sitter test` command. This command executes the parser on the defined test cases and compares the parser’s output against the expected results. This simplifies development by enabling repeated testing, ensuring that changes to the grammar don’t introduce unintended errors or break existing functionality.

The official Tree-sitter manual states: “These tests are important. They serve as the parser’s API documentation, and they can be run every time you change the grammar to verify that everything still parses correctly. The recommendation is to be comprehensive in adding tests. If it’s a visible node, add it to a test file in your `test/corpus` directory. It’s typically a good idea to test all of the permutations of each language construct. This increases test coverage, but doubly acquaints readers with a way to examine expected outputs and understand the ‘edges’ of a language.” [47]

7.1.1 Test Corpus

To define grammar tests, each test case must reside in its own specially formatted text file within the `/corpus/` directory. Each file should adhere to the following structure (see Listing 7.1 for an example):

Name The name of the test case, displayed when running the tests.

Input Source Code The input to the parser.

Expected Output Syntax Tree Expected output, represented as an S-expression¹. Shows only *named* nodes, similar to the `tree-sitter parse` command output, which additionally shows the positions of nodes. Positions are not included in the test.

```

1 =====
2 Outer environment - nested
3 =====
4 .Chapter Test
5   label: outer-env-nested
6
7 .itemize:
8
9   .item:
10
11     Item 1 content.
12 ---
13 (source_file
14   (document_part
15     (document_part_type )
16     (document_part_title
17       (text ))
18     (meta_block )
19     (document_part_body
20       (block
21         (classic_outer_environment
22           (outer_environment_type )
23           (block
24             (classic_outer_environment
25               (outer_environment_type )
26               (block
27                 (text_block
28                   (text )))))))))))

```

■ **Code listing 7.1** An example of a Tree-Sitter WooWoo test case (`simple_nesting.txt`).

The WooWoo Language Corpus

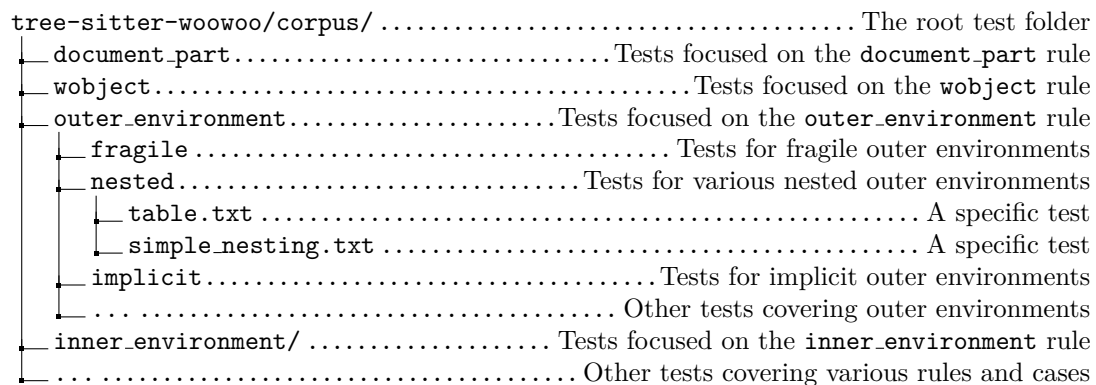
A comprehensive, structured corpus of over 60 tests was compiled for the WooWoo language, providing thorough coverage of its grammatical rules and edge cases. This corpus spans from simple test cases for individual rules to large, complex cases that cover edge cases of the language, including comments, varying newline styles, empty lines with spaces, and more.

These tests not only serve to continuously verify the parser's correctness during development but also act as a valuable resource for those learning WooWoo and its intricacies. For a visual overview of the corpus structure, see Figure 7.1.

7.1.2 Manual Grammar Validation with Existing Projects

During the development of the grammar, it was continuously verified by parsing existing WooWoo documents and checking for errors. This included every known existing WooWoo document as of January 2024, including sources for subjects at FIT CTU with codes BI-DML, BI-LA1, BI-LA2, BI-MA1, BI-MA2, BI-PKM, and the WooWoo `sandbox`.

¹More on S-expressions: <https://en.wikipedia.org/wiki/S-expression>



■ **Figure 7.1** Structure outline of the corpus folder

This manual testing shaped the final version of the grammar. The process was iterative, leading to many discussions with the creator of WooWoo syntax, Tomáš Kalvoda, about the language’s precise rules. These discussions played a key role in refining the grammar’s accuracy.

Many errors were encountered during development. Some were related to special or undocumented, but valid, WooWoo syntax, such as math/inner environments spanning multiple lines, empty lines containing whitespace, and the formatting of fragile environment bodies.

Additionally, several instances of invalid syntax were found in existing WooWoo documents. These had gone undetected by previous, less strict WooWoo parsers. These errors included incorrectly used empty line separators, empty outer environments, unclosed math environments, and text blocks in illegal positions. Their discovery led not only to grammar improvements but also corrections within the existing WooWoo documents themselves.

7.2 User Testing

Real-world testing with users is essential for validating the effectiveness and usability of language server features. This section outlines the user testing methodology employed during the development of the WooWoo VSCode extension. The extension is essentially a front-end to the language server, therefore the feedback gathered from using the extension could be directly applied to the language server.

7.2.1 Methodology

A user-focused, iterative approach was adopted for testing. The extension was made available early to a select group of initial users, primarily within the FIT CTU community who are actively involved in authoring WooWoo textbooks.

This initial user group included the creator of WooWoo, Tomáš Kalvoda, who provided valuable insights from his own use and feedback gathered from colleagues. The feedback, collected primarily through direct discussions, led to the identification of issues, which were then addressed in subsequent updates of the extension.

Additionally, a GitLab issue tracker supplemented the feedback process, allowing users to document bug reports and other observations. This iterative cycle of feedback, updates, and further testing ensured the continuous refinement of the WooWoo Language Server and the VSCode extension. The extension was made available from its early versions and was improved through this process over a months-long period.

7.2.2 Results

The user testing process led to significant improvements in the following areas:

1. **Code Highlighting:** Refinement of certain code highlighting rules to ensure accurate, visually helpful, and consistent code coloring.
2. **Performance Optimization:** Addressing performance bottlenecks that caused noticeable lags.
3. **Localization:** Improvements in the handling of Czech diacritics in language structures.
4. **Dialect Accuracy:** Resolving dialect-specific issues, such as inaccurate descriptions of elements.

7.3 Performance Tests

Performance tests were conducted on a selected subset of representative analysis engine features to guide the choice of underlying technology. Results of these tests, visually presented in the Design Chapter (Section 4.4.1.2), aided in making the design decision. This section outlines the datasets used, engine versions tested, the methodology employed, and the steps for replicating the experiments.

7.3.1 Datasets

The performance tests utilized nearly all existing WooWoo source codebases (see Table 7.1). Importantly, these datasets are currently only accessible to students and teachers at FIT CTU.

■ **Table 7.1** Testing Datasets Used for Performance Measurements

Code	Description	File Count	Total Lines	Commit Hash
bi-dml	BI-DML Course textbook ¹	16	35795	25b40d1
bi-la1	BI-LA1 Course textbook ²	9	16202	6f0b206
bi-la2	BI-LA2 Course textbook ³	10	14661	c1c237d
bi-ma1	BI-MA1 Course textbook ⁴	10	15693	c33df5b
bi-ma2	BI-MA2 Course textbook ⁵	7	12749	a2229c0

7.3.2 Methodology

This analysis utilized four testing scripts (located in the `/test/performance` directory) to evaluate key aspects of the analysis engine's performance. Importantly, for the purpose of these tests, one dataset was treated as a single workspace. Below is an overview of each script and its purpose:

initialization.py Measures the time required to execute `load_workspace` on each specified dataset (repeated ten times). This provides insight into how quickly the analyzer becomes operational and responsive.

¹<https://gitlab.fit.cvut.cz/BI-DML/bi-dml>

²<https://gitlab.fit.cvut.cz/BI-LA1/bi-la1>

³<https://gitlab.fit.cvut.cz/BI-LA2/bi-la2>

⁴<https://gitlab.fit.cvut.cz/BI-MA1/bi-ma1>

⁵<https://gitlab.fit.cvut.cz/BI-MA2/bi-ma2>

semantic_tokens.py This test provides a benchmark for a computationally intensive feature. For each dataset:

1. The workspace is loaded.
2. The `semantic_tokens` feature is executed on every file within the workspace.
3. The execution time for a single `semantic_tokens` operation is measured and repeated ten times for each file.

hover.py This test assesses the responsiveness of the `hover` feature under various conditions. It provides insights into the efficiency of a simple feature, as the hover information is primarily derived from the dialect file, without needing much computation resources. The following steps outline the test procedure:

1. A single, representative WooWoo file was selected.
2. Multiple predefined locations within the file were chosen to represent diverse code structures and positions (start, middle, end).
3. The `hover` feature is executed at each predefined location.
4. The execution time for each individual `hover` operation is measured and recorded.
5. The process is repeated ten times to ensure reliable results.

completion.py Similar in principle to the hover test, this script executes the `completion` feature at predefined locations representing a variety of autocompletion scenarios. Additionally, it simulates file changes to trigger cache invalidation, as this context-dependent, workspace-wide feature relies on up-to-date workspace state. This test aims to assess the performance of a workspace-state dependent feature as it adapts to code changes.

7.3.3 Replication

To replicate the performance tests, one should follow these steps:

1. Clone the `woowoo-language-server`² repository.
2. Check out one of the following commits:
 - **Python Version:** `f3939ee5` (last commit before migration to `wuff`)
 - **C++ Version:** `39d3a9c` (tested with `wuff==1.1.3`)
3. Place the datasets in the `/test/data/` directory in the root of the repository.
4. Ensure the datasets match the commits specified in Table 7.1.
5. Configure the script:
 - a. Locate the `tested_version` variable and set its value to either “python” or “C++” based on the version you’re testing.
 - b. Ensure that the `dialect_path` (for C++) and `template_path` (for Python) variables point to the correct YAML file within your local copy of the repository.
6. Run the scripts.
7. Results will be saved within the `/test/results` folder.
8. (Optional) Use the `/results/plots.ipynb` notebook to visualize the collected data.

²Available on FIT CTU’s GitLab: <https://gitlab.fit.cvut.cz/woowoo/lsp/woowoo-language-server>

7.4 Unit Testing wuff

“A *unit* test is an automated piece of code that invokes the unit of work being tested, and then checks some assumptions about a single end result of that unit. A unit test is almost always written using a unit testing framework. It can be written easily and runs quickly. It’s trustworthy, readable, and maintainable. It’s consistent in its results as long as production code hasn’t changed. [59]”

Unit testing aligns well with modular development practices, encouraging the creation of well-defined, independent components that are easier to test and reason about.

As described in Section 4.4, the analysis engine (**wuff**) is a core component within the WooWoo Language Server, responsible for the majority of language analysis tasks. Therefore, it was a primary focus of unit testing efforts.

7.4.1 Testing Framework

The Pytest framework³ was chosen for writing tests, as the unit tests target the **wuff** Python package. This approach effectively serves as a combined test of the C++ analysis engine’s core functionality and the integrity of the Python bindings. Pytest’s flexibility, ease of use and a lot of resources make it a well-suited choice for this project.

7.4.2 Structure

The unit tests are organized to match the components responsible for specific language features within the analysis engine (see Figure 7.2). Since each component requires a **WooWooAnalyzer** instance to operate within a specific workspace and dialect, a common configuration is established using Pytest *fixtures* for streamlined testing. A session-scoped fixture initializes the **WooWooAnalyzer**, ensuring a consistent testing environment for all subsequent tests.

```

tests/ ..... The root folder
├─ conftest.py ..... Utilities and pytest fixtures
├─ files ..... Folder with mock workspace for testing
├─ basic ..... Folder with basic tests
├─ components ..... Folder with tests organized by component
│   └─ completer ..... Folder containing test files targeting the Completer component
│       └─ navigator ..... Tests of the Navigator component
│           └─ test_definition.py ..... Tests for the Go to Definition feature
│               └─ test_references.py ..... Tests for the Find All References feature
└─ Other components ..... Other tests organized by component

```

■ **Figure 7.2** Unit Tests Directory Structure

Listing 7.2 demonstrates a core component test. It uses the analyzer fixture to get a configured **WooWooAnalyzer** instance, calls its `complete` method (utilizing the **Completer** component), and asserts that the expected labels were suggested.

```

1 def test_complete_reference(analyzer, file1_uri):
2     cp = create_completion_params(file1_uri, 4, 41)
3     results = analyzer.complete(cp) # call to the WooWooAnalyzer
4
5     labels = ["ct1", "ct2", "ct3"]
6     for label in labels:

```

³More on pytest: <https://docs.pytest.org/en/8.0.x/>

```
7     assert any(result.label == label for result in results), f"Label  
        {label} not found in results"
```

■ **Code listing 7.2** Testing *Completion* suggestions (snippet from `test_complete.py`)

7.4.3 Use in Wheel-Building Pipeline

A key advantage of targeting the Python module is seamless integration into the wheel build pipeline (described in Section 6.1.1). This allows automated testing after a wheel is built for a specific platform, safeguarding against platform-specific issues. If any tests fail, the build process halts, preventing the distribution of faulty wheels. See Figure 6.2 for a visualization of test integration within the process.

This approach acts as a safety net for code correctness and build integrity. During development, it frequently exposed issues ranging from programming errors to platform-specific build configuration problems or missing dependencies.

Conclusion

WooWoo stands as a powerful markup language with significant potential. However, prior to this thesis, its authoring tooling was lacking.

This work began with a comprehensive view of WooWoo, encompassing its history, syntax, semantics, and its role in creating the math study materials favored by students at FIT CTU. Building upon this foundation, this thesis also contributed by introducing the concept of “WooWoo Dialects”, a key abstraction for developing language tools for WooWoo while preserving its domain-agnostic nature.

The main focus of this thesis was to build a language server for WooWoo. Progressing from using Tree-sitter to generate a powerful WooWoo parser, which serves as the heart of the language server, to building an analysis engine for WooWoo workspaces—written in C++ for performance reasons—which enables the language features. Finally, this engine was incorporated into a Python-based language server, which integrates seamlessly with VSCode and has been adopted by WooWoo authors.

Furthermore, this thesis addressed distribution aspects, ensuring the analysis engine’s availability on PyPI and making the extension available on popular extension marketplaces used by various VSCode builds.

The project’s codebase is managed across four Git repositories, hosted internally on FIT CTU’s GitLab¹. The development process encountered many challenges and dead ends but ultimately yielded a tool that not only serves WooWoo authors but also provides a vast infrastructure to build upon, with many reusable components.

In summary, the contributions of this thesis provide lasting benefits to the WooWoo community, offering both immediate enhancements to the WooWoo authoring experience and resources to build upon for future innovations.

¹Located in the `woowoo/lsp` group: <https://gitlab.fit.cvut.cz/woowoo/lsp>

Appendix A

Installation Guide

This appendix details the installation procedures for the WooWoo VSCode extension, including alternative approaches primarily intended for developers and advanced users with specific technical requirements. For a more streamlined guide, refer to the extension’s description on the marketplace¹.

Step 1: System Compatibility

Ensure that your system meets the following requirements:

- Python 3.8 or higher.
- A C++ compiler for compiling the **wuff** package, if necessary. This is only required on systems where a pre-compiled **wuff** wheel is not available. Check Table 6.1 for compatible systems.

Step 2: Install wuff

Install the **wuff** package into the Python interpreter that the extension will use to launch the language server. This could be a Python virtual environment, the system-wide Python installation, or any other Python interpreter on your system. Use the following command for installation:

```
pip install wuff
```

wuff is available on PyPI². PyPI also provides a source distribution, which will be compiled using a C++ compiler on your system if a pre-compiled wheel is not available.

Developer Alternative: Compile From Source Locally

Alternatively, you can clone the **wuff** repository and install it locally:

```
git clone https://gitlab.fit.cvut.cz/woowoo/lsp/wuff.git
cd wuff
pip install .
```

¹<https://marketplace.visualstudio.com/items?itemName=michal-janecek.woowoo-vscode>

²<https://pypi.org/project/wuff/#files>

Step 3: Install the Extension

This step describes how to obtain and install the WooWoo extension in VSCode, using one of three methods based on your preference and setup needs.

Option 1: Install Directly via Marketplace (Recommended)

Install the WooWoo extension by opening the **Extensions** tab in VSCode. Type “WooWoo” into the search field to locate the extension. Once found, click **Install** to complete the installation process.

Option 2: Download the Extension from a Website

Alternatively, you can download the extension file (VSIX) from Open VSX or the Microsoft VSCode Marketplace. Once downloaded, install the extension manually using the **Install from VSIX** option in VSCode.

This method is particularly useful if you are using a custom build of VSCode that does not integrate with the standard marketplaces.

Option 3: Package the Extension Yourself

For those involved in development and testing, another option is to package the extension yourself. Begin by installing the `vsce` tool, which depends on Node.js. Then, clone the extension repository and use `vsce` to package it:

```
npm install -g @vscode/vsce
git clone https://gitlab.fit.cvut.cz/woowoo/lsp/vscode-woowoo-lsp
cd vscode-woowoo-lsp
vsce package
```

This process will generate a VSIX file in the extension folder, which you can then install manually in VSCode.

Step 4: Select Interpreter

After installing the extension, ensure the correct Python interpreter is selected by using the Command Menu.

Press **Ctrl + Shift + P** (Windows/Linux) or **Cmd + Shift + P** (Mac) and search for the **Python: Select Interpreter** command. Then, choose the Python interpreter where `wuff` was installed in Step 2.

If the language server isn't working, you may need to run the **WooWoo VSCode: Restart Server** command from the command menu to restart the language server.

Step 5: Configure Dialect (optional)

Language support in WooWoo is highly dependent on the specific dialect used. To ensure the extension operates correctly, you must specify your dialect unless you are using the default **FIT-Math** dialect.

- 1. Access Settings:** Open VSCode settings with **Ctrl + ,** (Windows/Linux) or **Cmd + ,** (Mac).
- 2. Search for WooWoo:** Type “WooWoo” in the settings search bar.

3. **Set Dialect File Path:** Locate the “WooWoo-vscode: Dialect_file_path” setting and input the absolute path to your dialect-definition file.
4. **Restart VSCode:** Restart VSCode to apply the changes.

The dialect file path you set is passed to the language server by the VSCode language server client upon startup.

..... Appendix B

Feature Demonstration

This appendix showcases various features of the WooWoo Language Server. It includes screenshots taken from the Visual Studio Code editor with the WooWoo extension installed, demonstrating these features in action.

B.1 Diagnostics

```
.Section Test
label: test
Text text .reference:
Text text
```

Syntax error: MISSING short_inner_environment_body woowoo-language-SERVER
View Problem (⌘F8) No quick fixes available

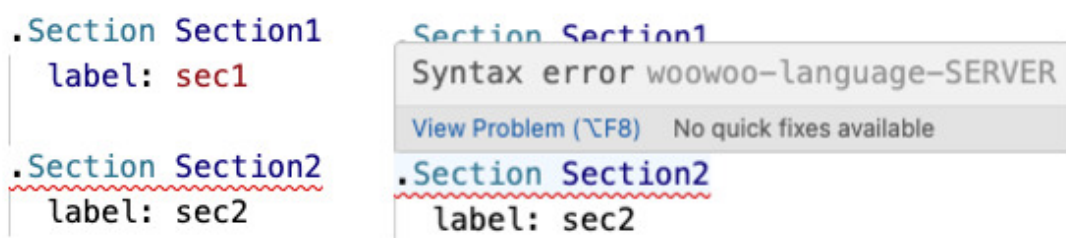
(a) Example of a short inner environment missing a body.

```
.Section Test
label: test
Text text $1 +
```

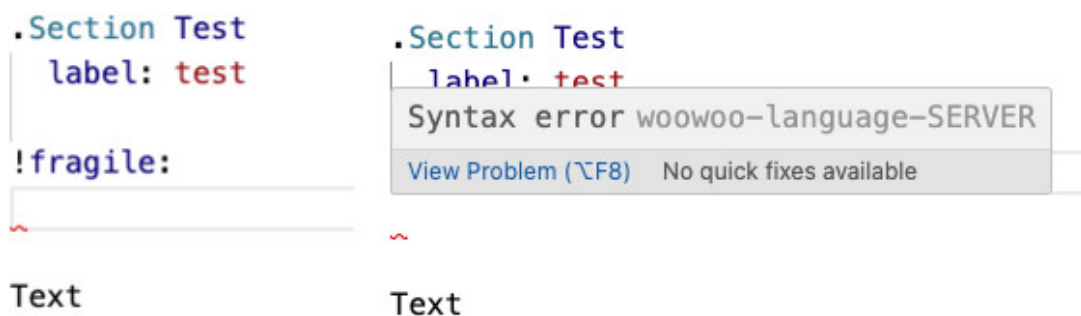
Syntax error: MISSING \$ woowoo-language-SERVER
View Problem (⌘F8) No quick fixes available

(b) Example of an unclosed math environment.

■ **Figure B.1** Demonstration of the *Diagnostics* feature, specifically focusing on error reporting based on the detection of Tree-sitter MISSING nodes.



(a) Syntax error caused by incorrect separation of document parts, which require a minimum of two empty lines between them. The error is identified based on the position of the `ERROR` node within the CST.



(b) Syntax error resulting from an invalid fragile outer environment that has an empty body, which is prohibited.

■ **Figure B.2** Demonstration of the *Diagnostics* feature, specifically focusing on error reporting based on the detection of Tree-sitter `ERROR` nodes.

B.2 Completion

Figure B.3(a) shows a code editor window with a LaTeX document. The cursor is at the end of the command `.reference:` inside a `ma` environment. The editor displays several lines of LaTeX code: `.row:`, `$U_a(\veps)$, resp. $U_{\va}(\veps)$`, and `(\veps)-okolí bodu a, resp. $\va \in \mathbb{R}^n$.`. A completion menu is open, showing a list of suggestions: `chap-notation`, `chap_extremy`, `chap_fce_vice_promenne`, `chap_kvadraticke_formy`, and `chap_linearni_rekurentni_rovnice`. The first suggestion, `chap-notation`, is highlighted in blue.

(a) Initial suggestions for completing the body of an inner environment that can reference another structure.

Figure B.3(b) shows the same code editor window as in (a), but now the user has typed `.reference: def`. The completion menu is filtered to show only suggestions that start with `def`: `def_charakter`, `def_definitnosti`, `def_deleni`, `def_derivace`, `def_derivace_ve_smeru`, and `def_dolni_horni_integral`. The first suggestion, `def_charakter`, is highlighted in blue.

(b) Filtered suggestions displayed after the user begins typing. Client-side filtering is performed by VSCode.

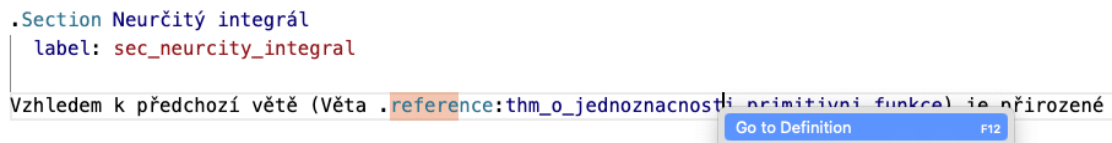
Figure B.3 Demonstration of the *Completion* feature for inner environments capable of referencing, focusing on body completion.

For a demonstration of the *Completion* feature applied to the `include` statement, see Figure 2.5 presented in Chapter 2.

B.3 Go To Definition

```
.Section Neurčitý integrál
  label: sec_neurcity_integral
```

Vzhledem k předchozí větě (Věta `.reference:thm_o_jednoznacnosti_primitivni_funkce`) je nřirozené



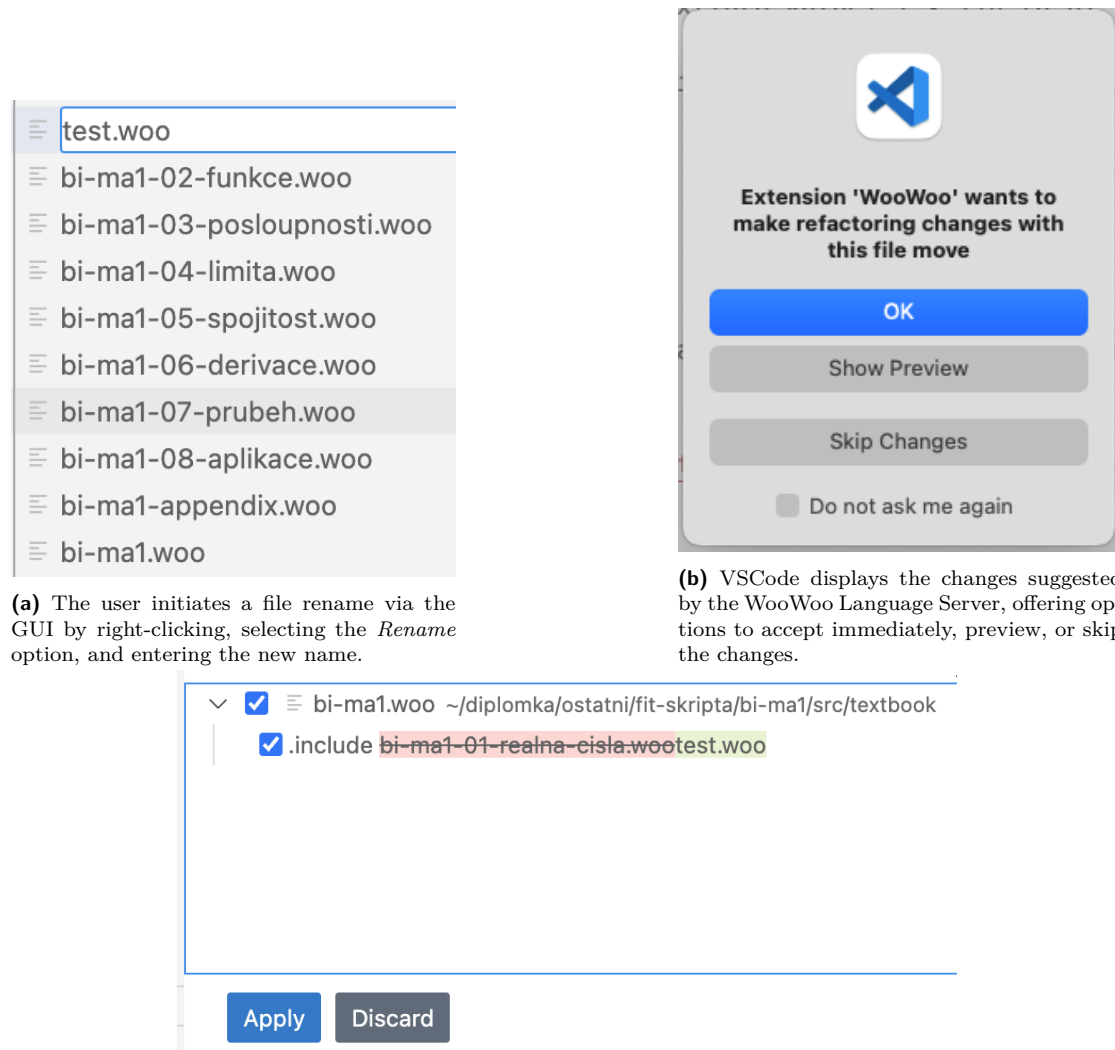
(a) User executes the *Go to Definition* option on a *reference* inner environment.

```
.Theorem:
  title: 0 jednoznačnosti primitivní funkce
  index: 'věta!0 jednoznačnosti primitivní funkce'
  label: thm_o_jednoznacnosti_primitivni_funkce
```

(b) After execution, the user is redirected to the corresponding structure. Behavior is dialect-dependent.

■ **Figure B.4** Demonstration of the *Go to Definition* feature, illustrating how users are directed to labeled structures within the FIT-Math dialect.

B.4 File Rename



■ **Figure B.5** Demonstration of the *Rename Files* feature, illustrating the process from initiation to review of proposed changes.

B.5 Folding Range

- ✓ **.Chapter Primitivní funkce a neurčitý integrál**
 Click to collapse the range.
- ✓ V této kapitole se od diferenciálního počtu reáln
 Jak dále uvidíme, "integrace".quoted je v jistém
 Na začátku kapitoly o derivování (podkapitola "Ry
 "Integrace".emphasize nám naopak umožní ze známé
 - 1:
| link: <https://courses.fit.cvut.cz/BI-MA1/text>
 - 2:
| link: <https://courses.fit.cvut.cz/BI-MA1>
- ✓ Vedle toho lze integraci také motivovat geometric
 Podrobněji se k této úloze dostaneme v kapitole .
 V našem případě ji dále využijeme k odhadování as

(a) Option to fold an entire document part by clicking the left arrow. Users can also choose to fold specific blocks selectively.

```

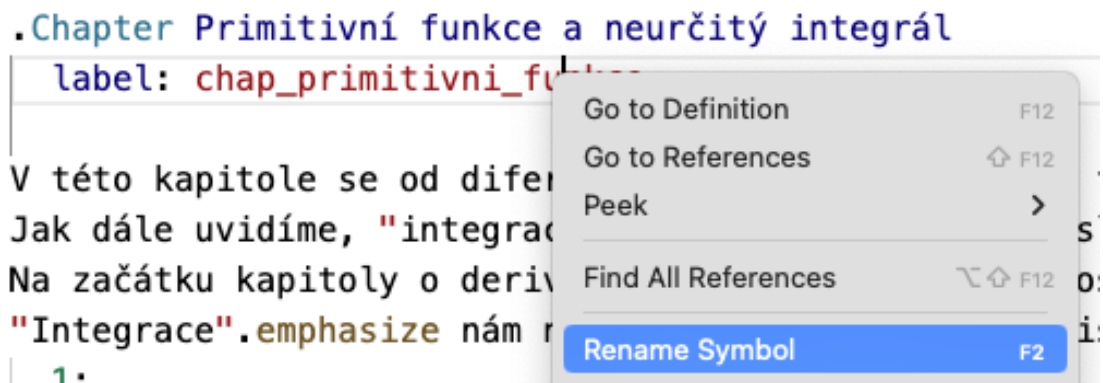
7 > .Chapter Primitivní funkce a neurčitý integrál ...
24
25 %%
26 %% Sekce: Neurčitý integrál
27 %%
28 > .Section Primitivní funkce ...
111 |
112 %%
113 %% Sekce
114 %%
115 > .Section Neurčitý integrál ...
381 |
382 %%
383 %% Sekce: Integrace per partes
384 %%
385 > .Section Integrace per partes ...

```

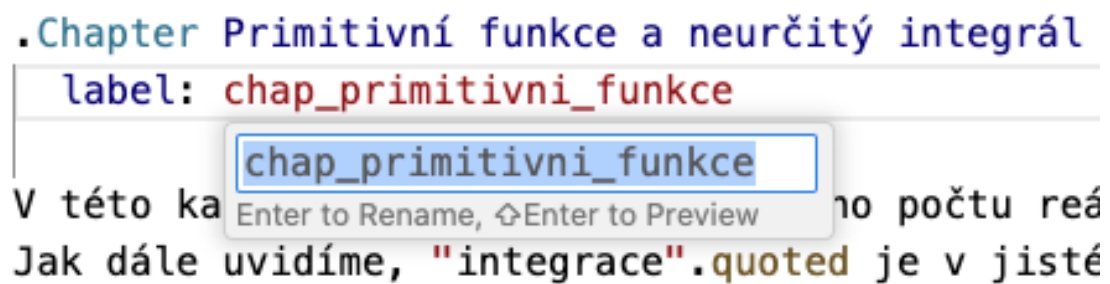
(b) Example of the folding feature applied to entire document parts.

■ **Figure B.6** Demonstration of the *Folding Range* feature, which allows users to fold document parts, blocks, and wobjects.

B.6 Rename Symbol



(a) User initiates the *Rename Symbol* feature on a referencable field within a metablock.



(b) The user is prompted to enter a new name for the symbol. Upon confirmation, a project-wide renaming is executed, refactoring the symbol and all associated references.

■ **Figure B.7** Demonstration of the *Rename Symbol* feature, enabling project-wide refactoring of references.

B.7 Semantic Tokens

The *Semantic Tokens* feature can be seen in action in Figures B.6a and B.7, as well as in other figures that depict *meta blocks*. The highlighting of these blocks is facilitated using this feature.

B.8 Hover

For a demonstration of the *Hover* feature, see Figure 5.2 presented in Chapter 5.

B.9 Find All References

For a demonstration of the *Find All References* feature, see Figure 2.4 presented in Chapter 2.

Bibliography

1. FRNKA, Matěj. *Generator of Mind Maps from WooWoo Documents*. CTU, Faculty of Information Technology, 2022. Bachelor's Thesis.
2. STRAKA, David. *WooWoo Source Code Editor*. CTU, Faculty of Information Technology, 2021. Bachelor's Thesis.
3. PRETEXT PROJECT CONTRIBUTORS. *PreTeXt – Write Once, Read Anywhere* [online]. 2024. [visited on 2024-02-24]. Available from: <https://pretextbook.org/>.
4. PRETEXTBOOK CONTRIBUTORS. *PreTeXt* [online]. 2024. [visited on 2024-02-24]. Available from: <https://github.com/PreTeXtBook/pretext>.
5. KALVODA, Tomáš. *Catalogue of terms and statements* [online]. CTU, Faculty of Information Technology, 2024 [visited on 2024-03-23]. Available from: <https://woowoo.pages.fit/catalogue/>.
6. KALVODA, Tomáš. *WooWoo Specs* [online]. CTU, Faculty of Information Technology, 2021 [visited on 2024-02-24]. Available from: <https://kam.fit.cvut.cz/deploy/woowoo-specs/woowoo-specs.pdf>.
7. KALVODA, Tomáš. *Example WooWoo Document* [online]. CTU, Faculty of Information Technology, 2024 [visited on 2024-02-25]. Available from: <https://woowoo.pages.fit/sandbox/index.html>.
8. MICROSOFT. *Language Server Protocol Specification, Version 3.17* [online]. 2024. [visited on 2024-02-11]. Available from: <https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/>.
9. SOURCEGRAPH. *Langserver.org* [online]. 2024. [visited on 2024-03-03]. Available from: <https://langserver.org>.
10. MICROSOFT. *Language Server Extension Guide* [online]. 2024. [visited on 2024-03-11]. Available from: <https://code.visualstudio.com/api/language-extensions/language-server-extension-guide>.
11. STOCK, Vinicius. *Improving the Developer Experience with the Ruby LSP* [online]. 2023. [visited on 2024-03-05]. Available from: <https://shopify.engineering/improving-the-developer-experience-with-ruby-lsp>.
12. MICROSOFT. *Language Server Protocol Overview* [online]. 2024. [visited on 2024-02-11]. Available from: <https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/>.
13. HANDY, Alex. *Codenvy, Microsoft and Red Hat collaborate on Language Server Protocol*. *SD Times* [online]. 2016 [visited on 2024-03-03]. Available from: <https://sdtimes.com/che/codenvy-microsoft-red-hat-collaborate-language-server-protocol/>.

14. MICROSOFT. *Language Server Protocol Implementors* [online]. 2024. [visited on 2024-02-11]. Available from: <https://microsoft.github.io/language-server-protocol/implementors/servers/>.
15. JSON-RPC WORKING GROUP. *JSON-RPC 2.0 Specification* [online]. 2013. [visited on 2024-03-04]. Available from: <https://www.jsonrpc.org/specification>.
16. BIRRELL, Andrew D.; NELSON, Bruce Jay. Implementing remote procedure calls. *ACM Trans. Comput. Syst.* 1984, vol. 2, no. 1, pp. 39–59. ISSN 0734-2071. Available from DOI: 10.1145/2080.357392.
17. ALTEXSOFT. *Functional and Nonfunctional Requirements: Specification and Types* [online]. 2023. [visited on 2024-03-11]. Available from: <https://www.altexsoft.com/blog/functional-and-non-functional-requirements-specification-and-types/>.
18. QAT GLOBAL. *Guide to Functional Requirements* [online]. 2024. [visited on 2024-03-11]. Available from: <https://qat.com/guide-functional-requirements/>.
19. BRUNSFELD, Max. Enabling low-latency, syntax-aware editing using Tree-sitter. *Zed Industries Blog* [online]. 2023 [visited on 2024-03-21]. Available from: <https://zed.dev/blog/syntax-aware-editing>.
20. TYPESCRIPT LANGUAGE SERVER CONTRIBUTORS ET AL. *TypeScript Language Server* [online]. 2024. [visited on 2024-03-16]. Available from: <https://github.com/typescript-language-server/typescript-language-server>.
21. MICROSOFT. *Standalone Server (tsserver)* [online]. 2023. [visited on 2024-03-16]. Available from: <https://github.com/microsoft/TypeScript/wiki/Standalone-Server-%28tsserver%29>.
22. THE RUST DEV TOOLS TEAM. *RLS Deprecation* [online]. 2022. [visited on 2024-03-17]. Available from: <https://blog.rust-lang.org/2022/07/01/RLS-deprecation.html>.
23. FERROUS SYSTEMS & CONTRIBUTORS. *rust-analyzer* [online]. 2024. [visited on 2024-03-17]. Available from: <https://rust-analyzer.github.io/>.
24. BASH LANGUAGE SERVER CONTRIBUTORS ET AL. *bash-language-server: Bash Language Server* [online]. 2024. [visited on 2024-03-16]. Available from: <https://github.com/bash-lsp/bash-language-server>.
25. TREE-SITTER CONTRIBUTORS ET AL. *Tree-sitter: An incremental parsing system for programming tools* [online]. 2024. [visited on 2024-03-24]. Available from: <https://github.com/tree-sitter/tree-sitter/>.
26. KALVODA, Tomáš. *WooWoo library* [online]. 2024. [visited on 2024-03-18]. Available from: <https://gitlab.fit.cvut.cz/woowoo/woowoo>.
27. KSCHIESS et al. *parslet GitHub Repository* [online]. 2014. [visited on 2024-03-18]. Available from: <https://github.com/kschiess/parslet/blob/master/qed/accelerators.md>.
28. STRAKA, David. *Wootom GitHub repository* [online]. 2021. [visited on 2024-03-18]. Available from: <https://github.com/davidstraka2/wootom/>.
29. HUTCHISON, Luke A. D. Pika parsing: reformulating packrat parsing as a dynamic programming algorithm solves the left recursion and error recovery problems. *arXiv*. 2020. Available from eprint: 2005.06444v2.
30. MIT OPENCOURSEWARE. *Parser Generators - MIT OCW* [online]. 2016. [visited on 2024-03-20]. Available from: <https://ocw.mit.edu/ans7870/6/6.005/s16/classes/18-parser-generators/index.html>.
31. PARR, Terence; FISHER, Kathleen. LL(*): the foundation of the ANTLR parser generator. In: 2011, pp. 425–436. Available from DOI: 10.1145/1993498.1993548.

32. FREE SOFTWARE FOUNDATION, INC. *GNU Bison* [online]. 2014. [visited on 2024-03-24]. Available from: <https://www.gnu.org/software/bison/>.
33. KRILL, Paul. What's new in GitHub's Atom text editor. *InfoWorld* [online]. 2018 [visited on 2024-03-21]. Available from: <https://www.infoworld.com/article/3263904/whats-new-in-githubs-atom-text-editor.html>.
34. BRUNSFELD, Max et al. *Tree-sitter — Using Parsers* [online]. 2024. [visited on 2024-03-20]. Available from: <https://tree-sitter.github.io/tree-sitter/using-parsers>.
35. BRUNSFELD, Max. *Tree-sitter - a new parsing system for programming tools* [online]. 2018. [visited on 2024-03-24]. Available from: <https://youtu.be/Jes3bD6P0To?si=B0utnQbDgm00Kgjf&t=2076>.
36. NEOVIM CONTRIBUTORS. *neovim — Treesitter* [online]. 2024. [visited on 2024-03-24]. Available from: <https://neovim.io/doc/user/treesitter.html>.
37. BRUNSFELD, Max et al. *Tree-sitter — Syntax Highlighting* [online]. 2024. [visited on 2024-03-20]. Available from: <https://tree-sitter.github.io/tree-sitter/syntax-highlighting>.
38. GITHUB. *Why Tree-Sitter?* [online]. 2020. [visited on 2024-03-24]. Available from: <https://github.com/github/semantic/blob/main/docs/why-tree-sitter.md>.
39. MICROSOFT. *Authoring Python Extensions* [online]. 2022. [visited on 2024-03-25]. Available from: <https://code.visualstudio.com/api/advanced-topics/python-extension-template>.
40. OPEN LAW LIBRARY. *pygls: A Pythonic framework to build Language Servers* [online]. 2024. [visited on 2024-03-26]. Available from: <https://github.com/openlawlibrary/pygls>.
41. MARSH, Charlie. *Ruff: An extremely fast Python linter* [online]. 2024. [visited on 2024-03-25]. Available from: <https://github.com/astral-sh/ruff-lsp>.
42. AVAST. *yls: YARA Language Server* [online]. 2023. [visited on 2024-03-26]. Available from: <https://github.com/avast/yls>.
43. OPEN LAW LIBRARY. *pygls - pygls 1.3.1 documentation* [online]. 2024. [visited on 2024-04-13]. Available from: <https://pygls.readthedocs.io/en/latest/index.html>.
44. MICROSOFT. *Language Extensions Overview* [online]. 2024. [visited on 2024-04-08]. Available from: <https://code.visualstudio.com/api/language-extensions/overview>.
45. MICROSOFT. *Syntax Highlight Guide* [online]. 2024. [visited on 2024-04-08]. Available from: <https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide>.
46. MICROSOFT. *Semantic Highlight Guide* [online]. 2024. [visited on 2024-04-08]. Available from: <https://code.visualstudio.com/api/language-extensions/semantic-highlight-guide>.
47. BRUNSFELD, Max et al. *Tree-sitter — Using Parsers* [online]. 2024. [visited on 2024-03-20]. Available from: <https://tree-sitter.github.io/tree-sitter/creating-parsers>.
48. KITWARE, INC. AND CONTRIBUTORS. *CMake Documentation: FetchContent* [online]. 2024. [visited on 2024-04-09]. Available from: <https://cmake.org/cmake/help/latest/module/FetchContent.html>.
49. MICROSOFT. *Visual Studio Code — Extension Manifest* [online]. 2024. [visited on 2024-04-10]. Available from: <https://code.visualstudio.com/api/references/extension-manifest>.

50. MICROSOFT. *Language Server Protocol - Visual Studio* [online]. 2022. [visited on 2024-03-30]. Available from: <https://learn.microsoft.com/en-us/visualstudio/extensibility/language-server-protocol?view=vs-2022>.
51. PYTHON PACKAGING AUTHORITY. *Binary Distribution Format* [online]. 2024. [visited on 2024-03-30]. Available from: <https://packaging.python.org/en/latest/specifications/binary-distribution-format/>.
52. DENTON, Charlie. *Python Wheels* [online]. 2024. [visited on 2024-03-30]. Available from: <https://pythonwheels.com/>.
53. PYPY. *cibuildwheel: Build Python wheels on CI with minimal configuration* [online]. 2024. [visited on 2024-03-30]. Available from: <https://github.com/pypa/cibuildwheel>.
54. MICROSOFT. *Publishing Extensions - Visual Studio Code* [online]. 2024. [visited on 2024-03-30]. Available from: <https://code.visualstudio.com/api/working-with-extensions/publishing-extension>.
55. STACK OVERFLOW. *Stack Overflow 2023 Developer Survey* [online]. 2024. [visited on 2024-03-31]. Available from: <https://survey.stackoverflow.co/2023/#overview>.
56. KÖHNLEIN, Jan. *The Open VSX Registry at the Eclipse Foundation: What You Need to Know* [online]. 2021. [visited on 2024-03-31]. Available from: https://www.eclipse.org/community/eclipse_newsletter/2021/january/1.php.
57. ARCH LINUX COMMUNITY. *Visual Studio Code - ArchWiki* [online]. 2024. [visited on 2024-03-31]. Available from: https://wiki.archlinux.org/title/Visual_Studio_Code.
58. TRONÍČEK, Filip. *Open VSX - Publishing Extensions* [online]. 2022. [visited on 2024-03-31]. Available from: <https://github.com/eclipse/openvsx/wiki/Publishing-Extensions>.
59. OSHEROVE, Roy. *The Art of Unit Testing: with Examples in C#*. 2nd ed. Manning Publications, 2013. ISBN 978-1617290893.

Contents of the Enclosed Medium

The attachment contains four Git repositories corresponding to the projects developed and described in this thesis. In addition, the `woowoo-language-server` repository includes the performance tests. For clarity, only the core files and folders are listed in the directory tree below. Additionally, each project folder contains a `README.md` file that provides further information.

```
/ ..... Root directory containing all project repositories and files
├── tree-sitter-woowoo ..... Repository for WooWoo Tree-sitter grammar
│   ├── corpus ..... Structured tests for tree-sitter-woowoo
│   ├── grammar.js ..... Grammar definition file
│   └── src ..... Source code for the parser
│       ├── parser.c ..... Parser generated by Tree-sitter
│       └── scanner.cc ..... Custom external scanner implementation
├── wuff ..... Repository for the wuff analysis engine
│   ├── .github ..... GitHub Actions pipelines for wheels distribution
│   ├── src ..... Entire wuff C++ codebase, including Python bindings
│   ├── tests ..... Unit tests for the Python module
│   └── setup.py ..... Script for building and installing the module
├── woowoo-language-server ..... Repository for the WooWoo Language Server
│   ├── server.py ..... Script to start the server and manage feature registration
│   ├── woowoo_language_server.py ..... Core implementation of the WooWooLanguageServer
│   ├── dialects ..... Directory containing the default dialect, fit-math.yaml
│   └── test ..... Performance tests, scripts, data, and results
├── vscode-woowoo-lsp ..... Repository for the WooWoo VSCode extension
│   ├── src ..... TypeScript codebase of the extension
│   ├── syntaxes ..... Directory containing woo.json, the TextMate grammar for WooWoo
│   ├── bundled ..... Directory containing the tool and launcher files
│   └── package.json ..... Manifest file for the extension
```