LEVER: A FRAMEWORK FOR DSL EDITOR SUPPORT

LEVER: A FRAMEWORK FOR DSL EDITOR SUPPORT

By ALEXANDRE LACHANCE, BASc.

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree Master of Applied Science

McMaster University
MASTER OF APPLIED SCIENCE (2024)
Hamilton, Ontario, Canada (Computing and Software)

TITLE:	Lever: A Framework For DSL Editor Support
AUTHOR:	Alexandre Lachance BASc. (Computer Science and Software Engineering), UQAM, Québec, Canada
SUPERVISOR:	Dr. Sébastien Mosser
NUMBER OF PAGES:	xiii, 131

Lay Abstract

Domain-Specific Languages (DSLs) are specialized programming languages made to solve domain specific problems. They are powerful and efficient tools for developers working within those specific domains. However, creating editor support for DSLs (e.g., syntax highlighting, code completion), is challenging due to their smaller user bases and complexities involved in development. For this reason, developers using DSLs often lack the tooling they have become accustomed to with General-Purpose Languages (GPLs). This thesis addresses these challenges by proposing the *Lever* framework, a lightweight and adaptable solution for building language support tooling targeting DSLs. *Lever* leverages existing artifacts and a rule-based system to provide editor support, making DSLs more accessible and user-friendly. A case study using *Lever* to build editor support for an industry DSL demonstrates its practical application, while a comparison of implementing language support for the Protobuf DSL with *Lever*, Langium, and MPS highlights *Lever*'s superior ease of use and functionality.

Abstract

While Domain-Specific Languages (DSLs) offer greater expressiveness for domain-specific tasks than General-Purpose Languages (GPLs), they have smaller communities behind them and fewer resources available. This is often reflected in the quantity and quality of available tooling for DSLs as compared to GPLs. This disparity is particularly evident in the case of DSL in-editor support where developers have become accustomed to features such as syntax-highlighting, auto-completion, and context-aware renaming. Developing such tooling for DSLs is challenging due to the significant effort required to implement features from scratch. To address this problem, this thesis proposes *Lever*, a framework for building editor support for DSLs. To reduce implementation cost and stay lightweight, *Lever* utilizes existing language artifacts (the grammar and the existing tooling). It uses a rule-based system that adds the necessary semantics to map the Concrete Syntax Tree (CST) to a language agnostic Abstract Syntax Tree (AST) and Symbol Table (ST). *Lever* enables cross-editor compatibility through the usage of the Language Server Protocol (LSP). The practical use of *Lever* is demonstrated through a case study on building editor support for the P4 DSL. Furthermore, a comparison with Langium and MPS in implementing language support for the Protobuf DSL reveals that *Lever* offers greater ease of use and functionality for this use case.

Acknowledgements

I would like to thank my supervisor, Dr. Sébastien Mosser, for his guidance and support throughout my undergraduate and graduate studies. I am also grateful to Dr. Richard Paige and Dr. Jacques Carette for their valuable feedback as members of my supervisory committee. Finally, I want to thank my girlfriend and my family for their unwavering support, even from six hours away.

Table of Contents

List of Figures

Acronyms

API Application Programming Interface

AST Abstract Syntax Tree

BBDSL Black-Box DSL

CST Concrete Syntax Tree

DSEL Domain-Specific Embedded Language

DSL Domain-Specific Language

EBNF Extended Backus-Naur Form

GPL general-purpose programming language

IDE Integrated Development Environment

LDE Language-Driven Engineering

LS Language Server

LSP Language Server Protocol

ML Machine Learning

NLP Natural Language Processing

OSS Open-Source Software

SDN Software-Defined Network

ST Symbol Table

WASM WebAssembly

WBDSL White-Box DSL

Chapter 1

Introduction

This chapter introduces the key ideas and contributions of this thesis. It begins by exploring the challenges faced by developers working with Domain-Specific Languages (DSLs) with Section 1.1. These challenges highlight the need for a new solution. Section 1.2 introduces the *Lever* framework, designed to simplify the creation of DSL editor support by leveraging existing artifacts and a rule-based system. Section 1.3 outlines the key contributions of this work, including the framework's design, validation through a case study, and a comparative analysis. Finally, Section 1.4 provides an overview of the overall structure of the thesis.

1.1 Motivation

*Domain-Specific Language*s (DSLs) are becoming increasingly more recognized for their applicability within both industrial and academic applications (Voelter, 2013). As their name implies, DSLs are built with a single domain in mind. This allows them to be more expressive, powerful, and simple to use than *general-purpose programming*

*language*s (GPLs) in their target applications (Mernik et al., 2005). However, this specialization also leads to some disadvantages. Notably, DSLs often have smaller user bases and communities built around them, which often equates to less overall support for the language. This limited user base is often justified given that these languages are tailored to solve highly specific problems, making widespread adoption unnecessary (Fowler, 2005; Voelter, 2014). However, this means in-editor support (e.g., syntax highlighting, auto-completion, refactoring) for a DSL, if it exists, is not on par with the support that is usually expected for GPLs.

This challenge primarily applies to textual DSLs since graphical DSLs nearly always have editor support. This is because, the language workbenches graphical DSLs are built with provide an editing environment. Language workbenches are language engineering tools that support DSL creation (Fowler, 2005). In these tools, editor support and the interpretation of the language itself are tightly coupled, making it difficult and sometimes impossible to separate them. This design paradigm was carried over to textual DSL tools like Xtext and MPS. Therefore, textual DSLs developed in these tools often have support, but it is typically limited to that specific *Integrated Development Environment* (IDE). If developers want to use the DSLs in their preferred editor, they have to do it without any support.

Language workbenches are not perfect solutions. There are many reasons as to why some developers might choose to not use these tools when developing a DSL. They tend to require a very high initial time investment since these tools tend to be difficult to learn. Developers and teams that work in certain domains might want to use technologies that they are more familiar with to build their language. For instance, communities like P4, which focus on networking, are often composed of developers

more experienced with more traditional tools and compilers than modern language engineering tools. As a result, they tend focus on familiar but less advanced tooling rather than investing in complex workbenches. In some cases, they might not even be aware that this kind of tooling exists. Additionally, some languages require greater control and flexibility over their design that might not fit within the constraints of a language workbench. Moreover, they can significantly increase the complexity of the development process by introducing potentially unwanted additional layers of abstraction, configuration, and tooling. These are only a few reasons that might drive developers to develop a DSL outside of a language workbench.

For developers choosing this path, an added challenge is that they need to provide their own custom-built editor support. One solution would be to rebuild the language inside a language workbench, but this approach has drawbacks. First, one problem is the need for a full re-implementation of the language which, depending on the language itself, can be very complex and time-consuming. This problem is accentuated by the fact that none of the existing artifacts of the language can be reused to speed up the development. Re-implementation can also be the cause of some problems. For example, the grammar will need to be re-implemented in the style that the workbench uses. Different parsing methods can have different limitations which makes direct translation of the grammar very difficult. For instance, LALR (Look-Ahead Left-to-right, Rightmost derivation) parsers can handle left-recursive rules, while LL(k) (Left-to-right, Leftmost derivation with k tokens of lookahead) parsers require grammars to be right-recursive, necessitating substantial rewrites. Additionally, LL(k) parsers may need more lookahead tokens to resolve ambiguities and conflicts that an

LALR parser can handle more easily, which makes the translation even more difficult. Secondly, since language workbenches generally assume that the full language has been built using itself, there are no built-in ways for the editor support to interact with the existing tooling for the language. This is an issue because these DSLs often have existing tooling that could provide important insight if it was integrated in the editor support. For example, a DSL's compiler or static analyzer can provide valuable feedback such as errors or warnings. Integrating these tools directly into the editor would allow developers to receive real-time feedback. However, rebuilding these tools within the workbench is unnecessary and inefficient. This makes it essential to provide a way for the editor support to interact with existing tooling directly.

Another issue with DSLs is the lack of substantial source code datasets. Smaller user bases result in fewer accessible open-source repositories, which limits the available code for testing and development of language tooling. Making the problem even worse, a significant portion of the source code is often composed of duplicates (e.g., developers forking or reproducing a tutorial of the DSL). This lack of availability and low quality of data makes testing DSL editor support difficult.

1.2 Approach

This thesis describes the *Lever* framework: a lightweight and open-source approach to building featureful DSL editor support. It is named both as a play on words with "**L**anguage S**erver**" and as a reference to the concept of physical levers which are simple machines that make difficult tasks easier. The framework focuses on reusing existing artifacts, like the DSL's grammar and its existing tooling, to make the development of the editor support as simple as possible. To provide the editor support

features, the framework leverages the *Language Server Protocol* (LSP). This allows projects built with the *Lever* framework to be used across all compatible editors.

The challenge lies in creating a flexible system that can be compatible with diverse DSLs without requiring manual implementation. The framework addresses this by defining a rule language that is used as an annotation over the existing grammar of the target DSL. It fills-in the gaps in semantics necessary for language support while keeping the syntax of the language intact. Generative programming techniques are employed to validate and process these rules at compile time, making the system more efficient. The use of an off-the-shelf configuration language for these rules also reduces the learning curve and simplifies the implementation process.

A core feature of the framework is its plugin system. Plugins act as adapters from existing language tools to language server compatible data structures. They allow the integration of the DSL's existing tools, like a compiler or a static analyzer, into the framework. For example, this plugin interface would allow the user to see compiler errors or hints directly in their editor. The only requirement is for the developer that implements the framework to write a small script that translates the output of the tool into a specified format.

A process to build a good quality test dataset for the language server built using this framework is also presented. This process uses multi-forge collection in order to collect as much source code as possible written in the target DSL. It also tackles the duplicate issue through a multi-step search space reduction approach. The result is a simple and reusable collection pipeline adapted to the challenges linked to DSLs.

1.3 Contribution

The main contributions of this thesis are the following:

- *Lever*, a lightweight framework for building DSL editor support. This framework allows the streamlining of the creation of fully-featured language server (e.g., syntax highlighting, auto-completion, renaming) targeting a DSL. It is based on a simple rule-based system (in a JSON-like format) and a straight forward plugin system allowing integration with existing tooling.

- An implementation of a language server targeting the P4 DSL, built using the *Lever* framework, along with a Tree-sitter grammar for P4. This implementation demonstrates the adaptability and effectiveness of the framework by applying it to a complex DSL, providing much-needed editor support to the P4 community. The Tree-sitter grammar, developed due te the lack of an existing grammar in the required format, has been validated using the dataset creation pipeline and has been accepted as an official open-source contribution to the Tree-sitter project (see Tree-sitter's list of parsers[1]).

- A comparison of language support implementations targeting the Protobuf DSL built in the *Lever* framework, the Langium language engineering tool, and the MPS language workbench. These implementations serve as a point of comparison of the *Lever* framework to two of the main alternative approaches.

- A pipeline for deduplicated DSL source code dataset creation. This tackles the difficult problem of creating high-quality datasets of DSL code, especially in contexts where there is not a lot of data. The solution consists of a framework

[1] `https://github.com/tree-sitter/tree-sitter/wiki/List-of-parsers`

for building datasets from multiple Git forges, organized as a sequential six-step pipeline.

1.4 Outline

This thesis is structured into 7 main chapters:

- Chapter 1, Introduction, covers the motivation behind the work presented in this thesis, the approach that it motivated, and the contributions that this thesis brings to the field.
- Chapter 2, Background, gives an introduction to the main concepts and works behind the work presented in this thesis.
- Chapter 3, Designing *Lever*, presents the main challenges behind a language agnostic framework for editor support, and the techniques and concepts applied to solve them.
- Chapter 4, *Lever* through P4, presents a case study of the framework's application to P4[2], a DSL managed by the Linux Foundation for programming network devices.
- Chapter 5, Validation and Comparison, validates the *Lever* framework by comparing it to existing solutions (MPS and Langium) using Protobuf[3] (a Google DSL for serializing data) as an example target language.

[2] `https://p4.org/`
[3] `https://protobuf.dev/`

- Chapter 6, How to Validate Against an Existing DSL Ecosystem, goes into depth about what is needed to create a high-quality dataset of DSL source code to test and validate language tooling.
- Chapter 7, Conclusion, reiterates the main points of this thesis and gives an outline of possible future works.

Chapter 2

Background

This chapter serves as an introduction to the main subjects and concepts used in this thesis, and reviews the relevant literature on these topics. It begins by exploring DSLs, detailing the relevant taxonomy as well as how they are implemented. Next, it covers the topic Generative Programming and how it can be used in automating the development of language tooling. The chapter then examines the LSP as a standard for providing editor support across different text editors. Finally, language workbenches are introduced. These topics form the theoretical foundation for the development of the *Lever* framework discussed in next chapters.

2.1 Domain-Specific Languages

DSLs, as their name implies, are languages that focus on a single application domain (Mernik et al., 2005). As opposed to general-purpose programming languages (GPLs) like C#, Python, or Rust, which are designed to solve a wide array of problems across various domains, DSLs are designed to allow domain experts to express solutions to

domain specific problems more effectively and succinctly. By doing so, they often sacrifice the ability to represent any program, a DSL does not need to be Turing complete. In fact, some are not executable (Mernik et al., 2005; Voelter, 2013). For example, the Protobuf DSL is not executable since it is only used to define the structure of data for the serialization and deserialization of messages (Protobuf, 2024). This thesis will focus mainly on textual DSLs, but there also exists graphical based DSLs (Voelter, 2013).

In general, DSLs and GPLs are typically developed by different types of teams or groups. GPLs are usually built by groups or committees of computer scientists and software engineers to solve a broad range of programming problems. On the other hand, DSLs are often built by domain experts collaborating with a few engineers focussing on a specific problem domain (Mernik et al., 2005).

2.1.1 Taxonomy of DSLs

There are many ways to slice and dice DSLs into different categories. Understanding some of these taxonomies is important to effectively build a framework to support them. Due to the vast ecosystem of languages, it would be out of the scope of this thesis to attempt to cover all of them. Two key dimensions are most relevant when framing our approach: implementation strategy and level of openness.

Implementation Strategy

DSLs can be also be separated in two categories of implementation strategy: internal or external (Voelter, 2013). They each represent roughly half of DSLs (Kosar et al., 2016).

Internal DSLs are often called *Domain-Specific Embedded Language*s (DSELs) (Hudak, 1996), as they are implemented and embedded inside GPLs. This allows them to leverage the existing infrastructure of the host language like libraries, compiler, runtime, etc. While the development of internal DSLs can be faster and simpler due to this, this comes at the cost of flexibility and expressiveness. They are also easier to learn for user that already know the host language to learn. Common host languages for internal DSLs include Scala, Java, Ruby, with Haskell being the most popular (Kosar et al., 2016). However, they are often limited by the hosts languages limitations, especially in terms of syntax. Languages with more powerful metaprogramming constructs are less affected by this problem (i.e., the Rascal programming language[1]). They also most often do not have any editor support (Voelter, 2013) outside of what is provided for host language, because of the complexity this would imply. An example of such a DSL is Language Integrated Query (LINQ) language by Microsoft. This language, which has C# as a host language, is used to define database queries in a more adapted language than basic C#.

On the other hand, external DSLs are implemented independently of any other language (Voelter, 2013). Although they are more complex to design and develop, they offer greater expressiveness with syntax and semantics tailored specifically to their domain. This design independence facilitates easier development of specialized tools like IDE support and static analyzers. Examples of this type of DSL include HTML for web development and SQL for database management.

Given these advantages, this thesis focuses on external DSLs. Their flexibility and independent syntax make them ideal candidates for comprehensive tooling. The *Lever* framework aims to address the complexity of tool creation, by making the process of

[1] `https://www.rascal-mpl.org/`

developing support for these languages easier.

Level of Openness

In software engineering, the terms "white-box" and "black-box" are commonly used to describe different levels of visibility and control over a system's internal workings. Building on this established terminology, this thesis applies a similar concept to create a taxonomy for DSLs based on their openness or transparency to developers. Note that this taxonomy could also be applied to GPLs.

*White-Box DSL*s (WBDSLs), are similar to white-box testing, where the internal structure of the system is fully known and accessible. These languages give full control over to the user and developers. This openness is often achieved through a modern open-sourced codebase and a community-driven development model. It can also be helped by the use of Language Workbenches (see Section 2.4) which makes language development much more accessible and efficient. Tooling for WBDSLs is generally easier to build since developers can leverage the language's open implementation to create custom tools and integrate with existing systems seamlessly. An example of this type of DSL is *mbeddr*[2], which is built on the MPS platform, allowing developers to extend and customize the language according to their specific needs.

Like black-box testing, where only the inputs and outputs are visible, *Black-Box DSL*s (BBDSLs) offer limited visibility and control over their internal mechanisms. These languages are designed to be used as-is. This lack of control can either result from the language being legacy or from a deliberate decision by a corporation or entity that manages the DSL. Tooling for this kind of DSL is much more difficult to build since developers do not have access to the underlying language implementation. In

[2] `http://mbeddr.com/`

the case of a legacy language, the language's codebase might be overly complex or too outdated for it to be worth it to justify continued development. In these two cases, it means outside developers may not be able to easily modify or extend the language to create custom tools or integrate with existing systems. Examples of this type of DSL include LINQ (controlled by Microsoft), Protobuf (controlled by Google), and P4 (legacy codebase, and controlled by Intel and the Linux Foundation).

The approach presented in the following chapters focuses on BBDSLs since they present unique challenges that are not adequately addressed by existing tooling solutions. Unlike WBDSLs, which are often supported by robust community contributions and open-source ecosystems, BBDSLs typically have restricted access to their underlying implementations, making it difficult to develop comprehensive tools and editors. *Lever* aims to fill this gap by providing a framework that can integrate with existing systems, facilitating better tooling support without requiring access or integration with the language's source code.

2.1.2 State-of-Practice

This section focuses on the implementation of textual and external DSLs. This subject is discussed in the Background chapter due to the limited literature on the development of language support tools. Luckily, creating language support shares many principles with designing compiler front-ends, despite the differences in focus and objectives.

The implementation of these DSLs generally follows the same principles as the implementation of GPLs, however usually on a smaller scale. There are three main

concerns when implementing a language: concrete syntax, abstract syntax, and semantics (Harel and Rumpe, 2004). These concerns are not only foundational for language implementation but are also critical for developing language support tools. They are directly involved in the implementation of features like syntax highlighting and code completion. It is important to note that there are many ways to approach the implementation of DSLs and their editor support; the methods presented here are general practices that may not apply to all cases. Implementing languages is a vast discipline that cannot be fully covered in this context.

The concrete syntax of a language is a mapping of the language's concepts to text that can be manipulated by the user of the language (Méndez-Acuña et al., 2016). It defines how the different elements of the language are written. This includes keywords, operators, overall structure, etc. Implementing the concrete syntax is most often done through the definition of a parser (Méndez-Acuña et al., 2016). There are various tools that can be used for this purpose (e.g., ANTLR[3], Bison[4], Tree-sitter[5]). These are parser generators, they take a grammatical description of a language and create a syntax analyzer (Aho et al., 2014). This syntax analyzer then takes-in plain text programs and transforms them into a concrete syntax tree (*Concrete Syntax Tree* (CST)), also called a parse tree (Aho et al., 2014; Nystrom, 2021). This tree represents the structure of the program and contains all the syntactic details of the original text (see Figure 2.1). While in compiler, this type of structure is quickly dismissed for a more abstract one, in language support this close mapping must be maintained. Language support tools must be able to interact directly with the text of the program itself, this cannot be done with a more abstract structure. For example,

[3] `https://www.antlr.org/`
[4] `https://www.gnu.org/software/bison/`
[5] `https://tree-sitter.github.io/`

syntax highlighting and error diagnostics rely on the concrete syntax to accurately reflect the exact position and format of keywords, symbols, and expressions as they appear in the source code. This is even more the case for features that must edit the actual text like renaming symbols or refactoring code.

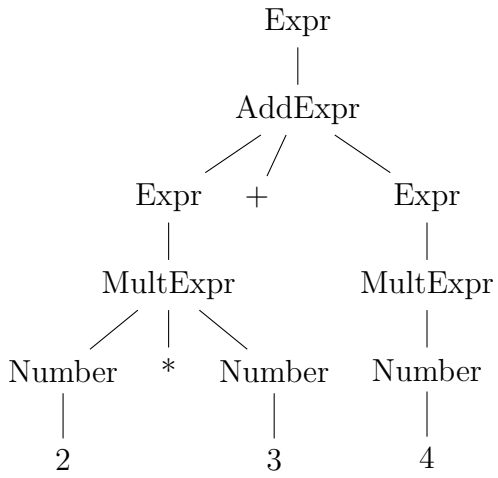


Figure 2.1: Concrete Syntax Tree (CST) for the expression `2 * 3 + 4`.

Once a CST is built, it can then be refined into an *Abstract Syntax Tree* (AST) (Wile, 1997). This structure removes unnecessary syntactic detail and focuses on the important structural and semantic elements of the code. The nodes of an AST represent high-level concepts such as expressions, statements, and declarations. This representation is meant to be an easier data structure to analyze and closer mapping to the domain-specific operations. It is just as valuable in a compiler as it is in a language support tool.

In parallel with the AST, a *Symbol Table* (ST) is often used (Aho et al., 2014; Nystrom, 2021). This structure keeps tracks of all symbols in a program (e.g., variables, constants, functions) and relevant information. This information includes names, usages, scoping, binding, etc. Sometimes this data can also be directly a part of the

AST, this is a design choice made by the language developer. For language support tools, a ST is used to enable features like going to the original definition of a function, finding all references of a variable, and accurate code completion.

Finally, we have semantic analysis, where the specific meanings of the domain-specific concepts are realized (Méndez-Acuña et al., 2016). There are two types of semantics, static and dynamic. Static semantics analyze the code without executing it, ensuring it follows the rules and constraints of the language. This typically includes type checking, scope resolution, etc. Dynamic semantics specify how a program written in the DSL is evaluated at runtime. This involves the actual interpretation or compilation of the code to produce the desired runtime behaviour. Understanding both types of semantics is essential for implementing effective language support tools. They both play an important role in providing error detection and feedback to developers.

2.2 Generative Programming

Generative programming is focused on automating the creation of software through reusable components and domain-specific languages (DSLs). This philosophy is directly used in the definition of the approach presented in this thesis. It is important to note that generative programming is a vast and extensive field with many diverse techniques and methodologies discussed in multiple different communities. This section presents a selection of key concepts and practices, but it does not aim or pretend to cover the subject comprehensively. The focus is on the overview of fundamental concepts and ideas relevant to this text. Many other valuable perspectives and contributions are also part of the larger field but are not relevant in this context.

As Czarnecki (2005) defined, *"system family engineering seeks to exploit the commonalities among systems from a given problem domain while managing the variabilities among them in a systematic way."* Generative programming expands on the concepts of system family engineering by automizing the process through metaprogramming (Czarnecki et al., 2002). This paradigm relies on three core concepts: DSLs, generators, and components, which work together to create tailored software solutions efficiently (see Figure 2.2).

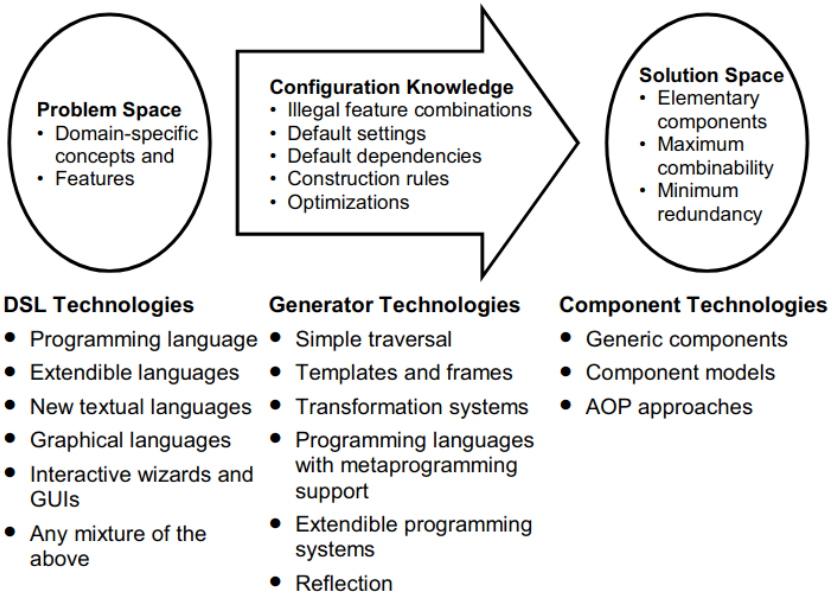


Figure 2.2: Generative Domain Model and Technology Projections (Czarnecki et al., 2002)

DSLs are essential in generative programming as they serve as the way to define the problem space (Czarnecki et al., 2002; Czarnecki, 2005). In other words, they allow developers to define the domain-specific concepts of a specific software variant. This specification can then be passed along to generators. These DSLs can come in many different forms as mentioned in Section 2.1.

Generators are implemented using of templated code, metaprogramming, transformation systems, partial evaluators, or extensible programming systems (Czarnecki et al., 2002; Czarnecki, 2005). They generate the code defined by the DSL by composing different components. They also contain configuration knowledge which allows them to identify illegal feature combinations or logical errors in the specification, and they can make optimizations. This ensures that the generated code is correctly oriented towards its specific target while remaining faithful to the design and architectural principles of the overall system family.

Components are the building blocks that are composed together by generators to create a full system (Czarnecki et al., 2002; Czarnecki, 2005). They are designed to be highly generic and reusable. To accomplish this, they are historically implemented using generic components, component models, or aspect-oriented programming approaches. However, aspect-oriented programming has received some criticism regarding its effectiveness and potential complexity, which can sometimes outweigh its benefits (Constantinides et al., 2004).

In summary, this paradigm relies on the creation of general and reusable components combined with specific rules and configurations that define the variations and features of each system in the family. It enables the quick and automated generation of software variants based around a system family based on a configuration written in a defined DSL. This approach will be used extensively in *Lever*, with its rule language as the DSL, to simplify the creation of DSL editor support tools.

2.3 The Language Server Protocol

The LSP is a protocol defined by Microsoft that defines the communication between a client, either a code editor or an IDE, with a *Language Server* (LS). While originally developed for Visual Studio Code, it is now available in a growing list of over 40 code editors, including popular ones such as Sublime Text, Emacs, or Vim (Microsoft, 2024). This effectively means that a single LS can be used across many editors as long as a client has been implemented. This allows developers of tooling for programming languages to focus their development on a single target instead of having to work on an extension/plugin for every editor that they want to support.

The *Lever* framework adopts the LSP as the foundation for its approach to DSL editor support. By doing so, it takes advantage of the LSP's wide and established ecosystem. It also allows developers to focus on creating language features without needing to address the specifics of different editors.

The LSP works by standardizing all interactions between the LS and the client. Messages are sent from the client to the LS when an action is performed by the user. These actions can range from typing code to renaming or hovering over a variable. The server then responds to these actions with the structure defined in the protocol. Most of the protocol is structured around this message and response mechanism, although there are some exceptions for messages from the server that expect no response. For example, diagnostics (Errors, Warnings, Hints, or Information) are sent from the server as notifications to the client. Most language servers do not implement the entirety of the LSP's features. The capabilities of the LS are requested by the client at initialization. All of this communication is done using JSON-RPC (JavaScript Object Notation-Remote Procedure Call). A high-level sequence diagram illustrating

this communication is shown in Figure 2.3.

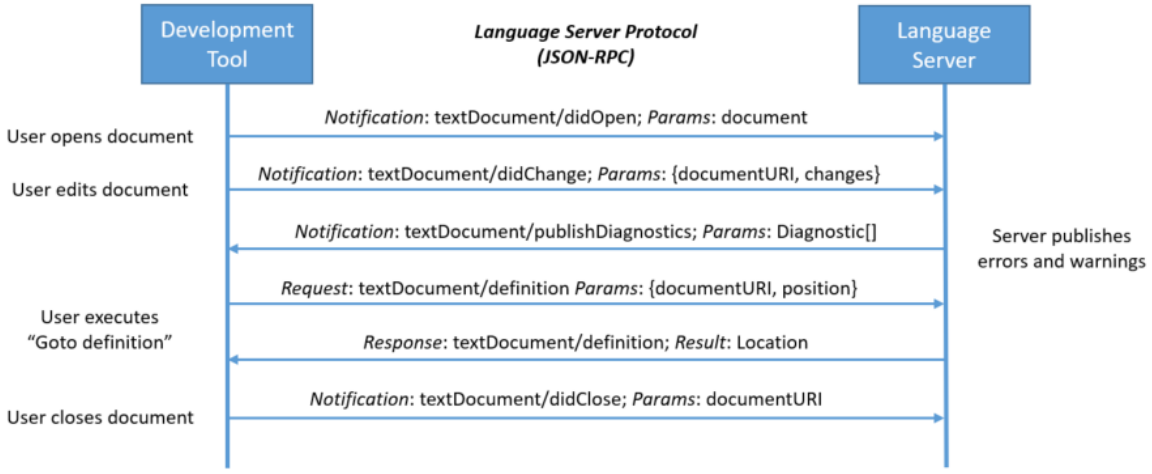


Figure 2.3: Language Server Sequence Diagram
Source: `https://microsoft.github.io/language-server-protocol/`

2.3.1 Supported Features

The LSP provides a broad range of features to support most common needs for language editor support. Key features include:

- **Syntax Highlighting:** Referred to as *Semantic Tokens* in the LSP, this feature allows the server to provide context-aware syntax highlighting.
- **Hover:** This feature aims to provide information when a user hovers their cursor over a symbol. This is usually used to display relevant information and documentation to help the user to better understand their code.
- **Renaming:** This feature allows the user to rename a symbol and for the change to be propagated to every one of its instances.

- **Go to Definition:** This feature allows the user to jump to the original instantiation of a symbol.
- **Formatting:** This feature formats an entire document.
- **Completion:** Also known as auto-completion, this features provides suggestions of completions for partially typed words. This helps the user by showing them what function, variable, and other elements are available for use.

The quality and functionality of these features fully depends on the implementation; not all language servers are created equal. The full list of features is available in the official documentation[6]. If a needed feature does not exist it can also be added as an LSP extension. For example, `rust-analyzer`[7], the main LSP for the Rust programming language, has over 30 extensions to the protocol (e.g., Join Lines, On Enter, Matching Brace). However, they aim to upstream non Rust-specific extensions if possible (Rust, 2024). This extensibility is important as the *Lever* framework, while based on this protocol, is not necessarily limited by it.

2.3.2 Limitations

While the LSP is a powerful framework for building language support, it has some major limitations that can sometimes make its usage difficult. Here are a few of them:

- Depending on the editor, the language client, necessary for the language server to interface with the editor, might need to be implemented. This is notable the case with Visual Studio Code, where the development of an extension is

[6] `https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/`

[7] `https://github.com/rust-lang/rust-analyzer`

necessary. Most other editors either provide a built-in client (e.g., Sublime Text, Neovim) or have easily installable extensions that provide a client (e.g., Vim, Emacs).

- The ability to extend the protocol is very powerful, it allows the tooling developers to not be limited in the feature set present in the base protocol. However, these extensions are not automatically supported by the different language clients. This means that the client of every editor must be extended manually, which goes against the fact that the LSP is editor-agnostic. Some work has been done to standardize LSP extensions (Kjær Rask et al., 2021); it has not yet been implemented officially.

- Implementations of the LSP can also vary across editors. Some editors do not support all the LSP features which can be problematic depending on the needs of the programming language or frustrating to the user. The quality of the implementations can also vary which can lead to bugs on certain editors that do not occur on others. Which again, undermines the main appeal of the LSP, which is the ability to develop once and have it automatically work across all compatible editors.

The protocol and its implementations keep evolving and some of these issues might be resolved in the future. However, despite these problems, the LSP remains a better alternative than traditional editor specific extensions for most use cases. This is why it will serve as a foundation of the approach presented in this thesis.

2.4 Language Workbenches

Language workbenches are tools built to enable the creation of DSLs by providing high-level abstractions and languages. They typically provide a way to define the syntax and semantics of your language, as well as provide an editor. By doing so, they make language creation much more accessible by greatly reducing the time and costs of development. For that reason, they are used more and more in both industry and academia (Erdweg et al., 2013).

There are a lot of different types and flavours or language workbenches. Some focus on textual DSLs by providing powerful editor tooling like language servers (e.g., Xtext, Langium) (Popov et al., 2024) or more editor-specific support. Others concentrate graphical DSLs and provide editors for that purpose (e.g., MetaEdit+ (Tolvanen et al., 2007)). Additionally, some workbenches are projectional (e.g., MPS (Voelter and Lisson, 2014a)), which means that there is no parsing as there would be for textual DSLs. Users interact with the language's AST directly. This enables greater flexibility in terms of language structure with arbitrary notation and easy composition of multiple languages.

We will go into more detail specifically on MPS and Langium in Chapter 5 where they will be used as points of comparison. The reason those two were chosen is that they represent the two main categories of workbenches relevant to this thesis (textual and projectional). They are also both modern technologies since version 1.0 of Langium was released in 2022 (Spönemann, 2022) and MPS, although initially released in 2011, still receives frequent feature updates. Langium is also chosen because it is the most comparable existing solution to the novel framework presented in the next chapter.

2.5 Conclusion

This chapter introduced the foundational concepts and literature that is important to the development of effective language editor support for DSLs. The nature of DSLs and how they are implemented was explored. Generative programming was presented as a valuable strategy for building solutions to support these languages. Additionally, the chapter examined the pros and cons of the LSP, which will be important to our approach by standardizing language support across various editors. Lastly, language workbenches were quickly discussed as powerful solutions to build DSLs and as closest existing solutions to the issue presented in the first chapter. This knowledge and understanding will serve as the foundation for the design and implementation of *Lever*, a framework that aims to simplify the creation of DSL editor support.

Chapter 3

Designing *Lever*

The development of effective and practical language editor support is crucial to assist developers using BBDSLs. This chapter delves into the requirements for such support, and introduces *Lever*, a framework specifically designed to meet these needs. Each section focuses on a key challenge that must be addressed to successfully design this framework. The challenges discussed in this chapter revolve around supporting diverse DSLs, simplifying implementation, and leveraging existing tooling to enhance efficiency. *Lever*'s approach is not just about creating new tools but also about integrating existing ones, ensuring they work seamlessly within the existing language's ecosystem. For each of the presented challenges, robust and adaptable solutions are then proposed and discussed.

The aim of this chapter is to demonstrate how *Lever* not only fulfills the identified requirements but also provides a flexible approach to language editor support. This flexibility makes *Lever* a valuable asset for both researchers and practitioners in the field of language engineering.

3.1 Distinguishing Language Support from Compiler Requirements

In Chapter 2, a lot of focus was put on the implementation aspects of DSLs. This emphasis was due to the abundance of literature and established methods in this area. However, the subject of language support was put on the back-burner due to an obvious gap in the available literature. While DSL editor support and DSL implementation share a lot of similarities and concepts, they are fundamentally distinct. Understanding these differences is essential for developing effective tooling.

To illustrate this difference, consider the issue of scoping in language support and in compilers. In language support, even with the same editing semantics of the same language, scoping rules can vary depending on the target feature. For instance, take the seemingly straightforward task of automatic code completion. In an editing environment, the requirements can be distinct from those during compilation. This is because, editing might involve searching for symbols outside the existing ST, such as when providing completion for symbols that have not yet been imported into the current scope (e.g., a symbol present in a different file). This scenario is typically irrelevant during compilation, where the focus is on interpreting what is already present, rather than providing the available options to the developer.

Language encompasses features and tools that help users to write, edit, and understand code. This includes syntax highlighting, code completion, syntax validation, and providing relevant validation. These features must correctly adapt to ongoing changes in the code as the user edits the file. The primary goal of language support is to improve the usability of the language, ensuring that users can efficiently and

effectively express their ideas without being hindered by the language's complexity or syntax.

On the other hand, DSL implementation focus on translating DSL code into executable programs or other output forms (e.g., intermediate representations, code in another language). Compilers or interpreters are designed to ensure that the code is not only correct, but also optimized for performance and resource use. The focus here is on accuracy, efficiency, and respect of the formal semantics of the language. This introduces significant complexity into the compiler/interpreter's design and implementation. Additionally, compilers, in most cases, operate on a static code base, meaning that they process a fixed set of source code without the need to account for changes during execution. This static nature simplifies certain aspects of the code interpretation, since there is no need to accommodate ongoing modifications or real-time updates.

This distinction becomes evident when examining tools like Langium, which highlight these differences and the issue with the current tooling landscape. Taking the scoping example again, when implementing a language in Langium, the user can only register a single scope provider. This is sufficient for compilation but is very limiting in an editing environment where, as shown, multiple scoping strategies can often be required.

Considering this important distinction, the *Lever* framework is intentionally designed with a primary focus on editing concerns rather than DSL compilation or interpretation. By separating the responsibilities of language support from those of a compiler/interpreter, the framework can therefore concentrate on providing helpful editor features without having to deal by the complexity of language compilation.

This means no unnecessary overhead is introduced by integrating complex compilation processes into the editing environment.

3.2 Supporting Diverse DSLs

When designing editor support for BBDSLs as a whole, one of the main challenges is handling the vast diversity of syntactic structure and editing semantics. For this reason, even with the right libraries, implementing every feature manually for a DSL would be both time-consuming and impractical. Therefore, something more encompassing and high-level than simple libraries is needed. To address this complexity, it is crucial to abstract away the language specifics. By doing so, it is possible to devise a way to have a single implementation of every feature that works across BBDSLs. This abstraction would enable the reuse of existing solutions and minimizing the need for custom development.

One effective approach is to adopt a universal internal representation that can accommodate the structural and syntactic variety across different DSLs. It is then possible to implement language editor support features by relying solely on this internal representation. In the case of *Lever*, ASTs and STs serve as this universal backbone. By using ASTs and STs, a layer of abstraction is created that allows the underlying implementation to remain consistent while adapting to the nuances of each DSL. This approach hints at the idea of an "abstract abstract syntax tree" and an "abstract symbol table" which serves as a higher-level representation that supports more than just individual language grammars. They provide a uniform and language-agnostic interface for implementation of editor support features.

These abstract concepts might seem counterintuitive, particularly from a compiler's perspective, where the syntax tree is inherently tied to the language's syntax. However, in the context of a generic framework for DSL support, this abstraction is not only logical but necessary. It enables the system to parse, analyze, and transform code in a language-agnostic manner, ensuring that common editor features like syntax highlighting, code completion, and syntax validation can be implemented uniformly across different DSLs.

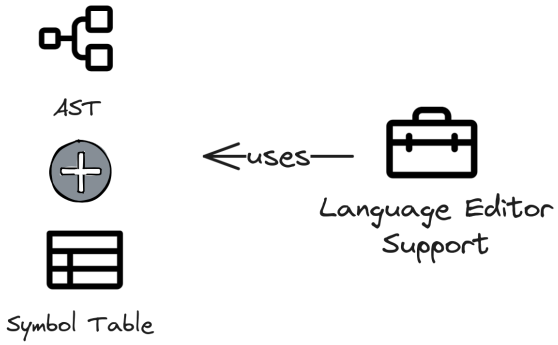


Figure 3.1: Overview of the structure of *Lever*'s internal representation.

For example, consider the implementation of code completion. This depends heavily on the specific syntax and editing semantics of the target DSL. However, if these specifics are abstracted, code completion simply consists of returning the relevant symbols to the user. Therefore, if by only relying on our internal representation, for every single DSL, code completion is simply a request of information into the AST and ST. If those structures are the same for every language, then the request is the same. This is how it is possible a single implementation of a language support feature that works across any BBDSL.

3.3 Defining Language Specifics

The previous section outlined how at the heart of *Lever* lie an abstract AST and ST in order to support diverse DSLs. These structures are the foundation of the framework. However, the challenge in this section is how to define language-specific details in a way that allow for the population of this internal representation. Different DSLs have diverse syntactic structures, making it a difficult problem to attempt to apply a single solution across various languages. Therefore, the main focus must be on enabling *Lever* to populate these abstract representations across DSLs while allowing the details of each language to be specified without compromising on the framework's generalizability. For this, an approach that leverages existing resources while minimizing the manual effort required for each new DSL is needed.

The solution proposed in this section lies in reusing existing artifacts by building upon the already existing syntax of BBDSLs. Rather than developing new parsers from scratch, *Lever* utilizes the existing DSL's grammar. In most cases it is already well-defined, either by the original language developers, or in some cases by the DSL's community for a variety of different purposes (e.g., syntax highlighting, static analysis). With the syntax already in place, the next step is to incorporate the necessary semantics to integrate the DSL into the universal internal representation. There are many ways this could be accomplished.

One approach could be to modify or extend the parser library to require additional syntactic information. This method involves enhancing the grammar language to capture more detailed syntactic constructs or adding new concepts as annotations over the existing grammar. Another could be using a graphical editor to add editing semantics to an existing grammar. A graphical editor would provide an intuitive

interface for defining relationships and rules within the language, simplifying the process of annotation. For example, a graphical editor could allow users to visually define scopes in the grammar by highlighting the boundaries of different scopes.

However, for the *Lever* framework, a simple textual DSL was chosen. This rule-based DSL acts as annotations over the existing grammar (see Figure 3.2). The rules do not in any way replace the grammar; instead, they supplement it by filling in semantic details that the original parser does not capture. This approach is not only very lightweight and simple to implement, it also allows keeping exact the structure of the DSL brought by the parser while extending the grammar to be able to interact with the language-agnostic framework's features. By reusing the grammar, it is ensured that the resulting language tooling respects the correct syntax of the DSL.

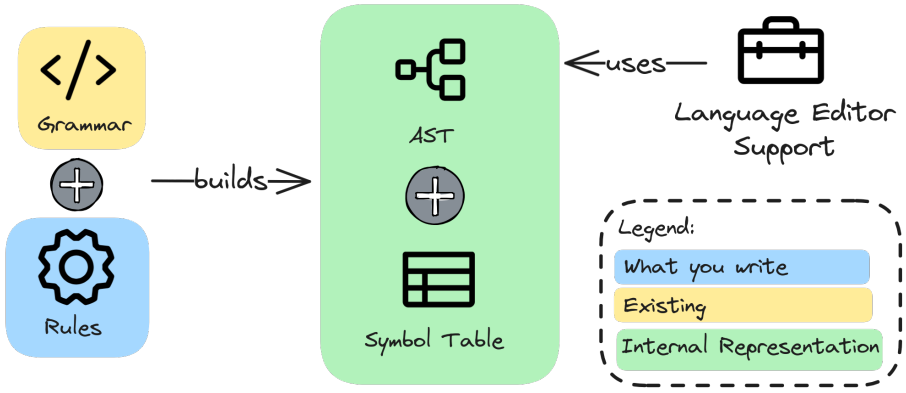


Figure 3.2: Overview of how *Lever*'s internal representation is populated.

To implement this approach, *Lever* specifically uses Tree-sitter grammars. However, this method is not necessarily limited to Tree-sitter. In fact, with some effort the framework could be reworked to be compatible with other popular parsing technologies. This was out of the scope of this thesis.

The structure of a *Lever* rules file is organized into several key sections, as shown

in Figure 3.4. All of these sections of the metamodel where identified through what information necessary to provide language editor features. This language definition includes details such as the language's name, file extensions, and library paths (Figure 3.3). As shown in Figure 3.3, the metamodel is kept simple, yet it contains all necessary information necessary for language editor support features. Library paths are necessary to ensure correct integration of the language ecosystem with correct imports for the DSL (Figure 3.5). The language definition provides a structured way to specify these core details so that *Lever* can do the rest.

```
LanguageDefinition (
    language: ...,
    keywords: ...,
    symbol_types: ...,
    ast_rules: ...,
    global_ast_rules: ...,
)
```

Figure 3.4: Example of *Lever* rule structure.

```
language: (
    name: "MyLanguage",
    file_extensions: [ "mylang" ],
    library_paths: (
        env_variables: [ "MYLANG_LIB_PATH" ],
        linux: [ "/usr/local/lib/mylang/", "~/local/lib/mylang/" ],
        windows: [ "C:\\Program Files\\MyLanguage\\lib" ],
        macos: [ "/usr/local/lib/mylang/" ],
    ),
),
```

Figure 3.5: Example *Lever* definition of language details.

Next, the keywords section is defined which simply consists of a list of the target

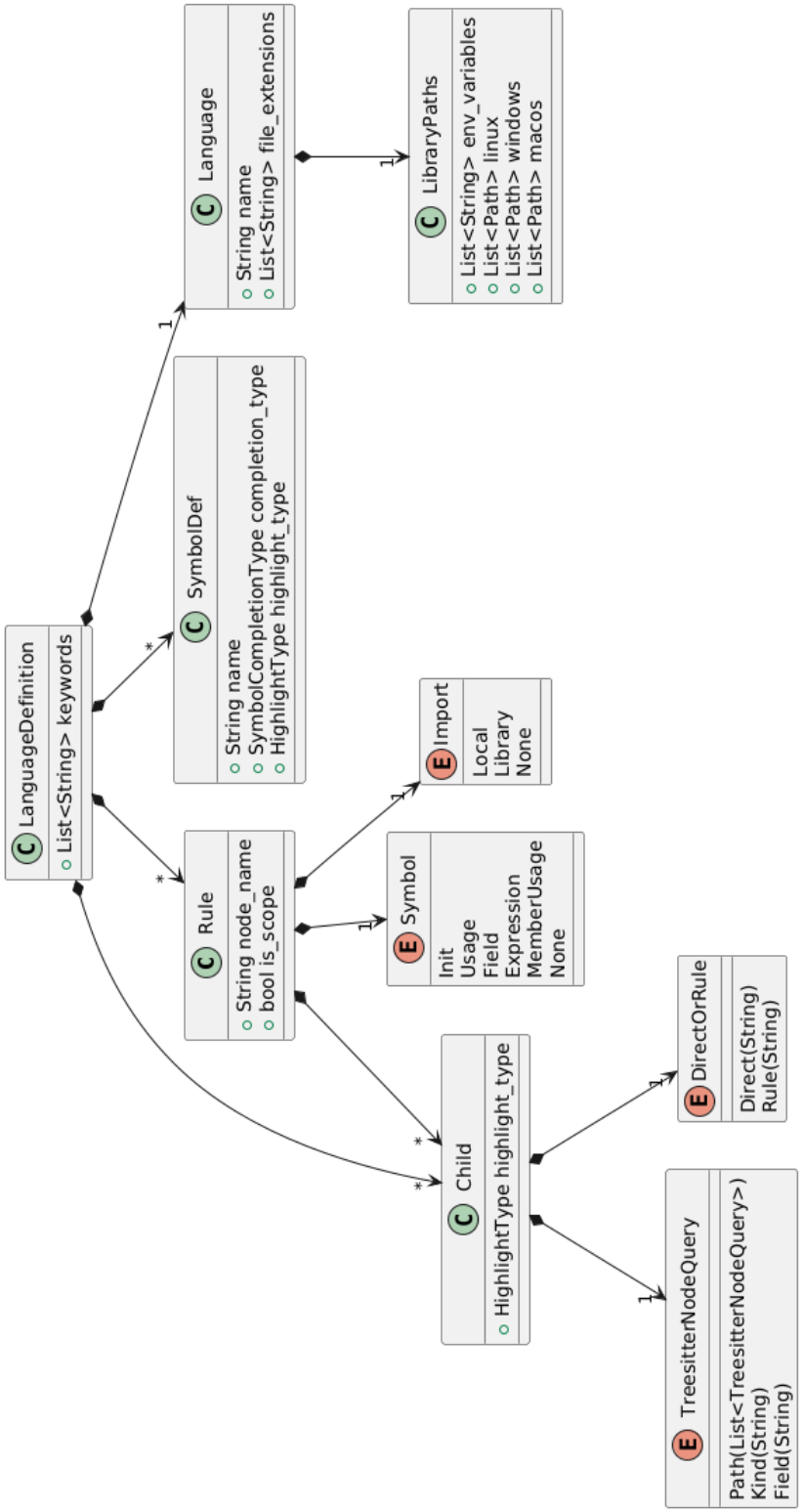


Figure 3.3: Structure of *Lever* rules.

language's keywords (Figure 3.6). Keywords are primarily used for syntax highlighting and code completion. By enumerating all keywords used in the DSL, the framework can highlight these words correctly as well as suggest them as completions while typing. This enhances the developer's experience by making the language's syntax more accessible and user-friendly.

```
keywords: [
    "function",
    "var",
    "example",
],
```

Figure 3.6: Example *Lever* definition of language keywords.

After the keywords, comes the `symbol_types` section. It defines the different types of symbols in the language, specifies how they should be displayed in the completion suggestions, and how they should be highlighted in the editor. For example, symbol type identifiers such as `Function` and `Variable` can be marked with different highlight types and completion types, allowing for clear differentiation between various symbols within the code (Figure 3.7). This helps developers by facilitating better code readability and maintainability. These symbol types are integrated into the abstract ST to ensure accurate handling of the language-specific handling of symbols types.

```
symbol_types: [
    (name: "Function", completion_type: Function, highlight_type: Function),
    (name: "Variable", completion_type: Class, highlight_type: Variable),
],
```

Figure 3.7: Example *Lever* definition of language symbol types.

The `ast_rules` section is where the more complex logic of the language's AST is defined. Each rule has a name and can specify several properties. It can declare a new scope, indicate whether it represents a symbol usage or initialization, and include child nodes. These rules map the CST to the abstract AST, determining how nodes are created in the AST and defining the overall structure of the AST through node children. Child nodes are defined with a query in the CST (the syntax tree generated using Tree-sitter) and the rule it is to be associated with. These queries can be done with simple node kind queries, queries looking for a specific tag on a node (field name), or by paths. Paths are used to query deeply nested syntax nodes without burdening the AST with too many unnecessary nodes. As for the associated rule, it can be specified as either as a direct node (bypassing the need to create an empty rule) or a reference to another other rules defined in the rule file (see Figure 3.8). This structured approach allows *Lever* to maintain the correct syntax tree for the DSL while enriching it with additional semantic information in the abstract internal representation.

```
ast_rules: [
    Rule(
        node_name: "Root", // Name of Rule (required)
        is_scope: true, // Defaults to false if not included
        children: [
            (query: Kind("function"), rule: Rule("Function")),
            (query: Kind("variable_def"), rule: Rule("Variable")),
        ]
    ),
    Rule(
        node_name: "Function",
        is_scope: true,
        symbol: Init(type: "Function", name_node: "Name"),
        children: [
            (query: Field("name"), rule: Direct("Name")),
            (query: Field("body"), rule: Rule("FunctionBody")),
        ]
    ),
    Rule(
        node_name: "Variable",
        symbol: Init(type: "Variable", name_node: "Name"),
        children: [
            (query: Field("name"), rule: Direct("Name")),
            (query: Field("value"), rule: Rule("Value")),
        ]
    ),
    ... // More rules
]
```

Figure 3.8: Example *Lever* definition of AST generation rules.

For elements that are present throughout the entire syntax tree, like comments, global rules can be implemented. These `global_ast_rules` are applied across the entire AST to identify and manage language constructs that can appear anywhere within a file (Figure 3.9). Another example of this kind of construct are preprocessor declarations since they are present outside the language's normal syntax. This capability ensures that universally applicable syntax elements can be handled, making the framework more robust and versatile.

```
global_ast_rules: [
    (query: Kind("comment"), rule: Direct("Comment"), highlight_type: Comment),
],
```

Figure 3.9: Example *Lever* definition of global AST generation rules.

The rules are designed to be lightweight and declarative, adding only the necessary semantics to bridge the gap between the DSL's syntax and abstract internal representation of the language-agnostic framework. This ensures that the DSL's core structure remains intact while enabling uniform support across varied and diverse languages. By reusing existing artifacts and adding targeted annotations, *Lever* can populate the abstract AST and ST efficiently and consistently. This approach simplifies the process of supporting new DSLs, maximizes reuse, and ensures that the resulting tooling remains accurate and aligned with the original grammar.

3.4 Lowering the Barrier to Entry

Having a low barrier to entry is crucial to ease adoption for any kind of software framework. If learning to use and integrate a framework is too complex or expensive of a task, the target demographic may decide that it is not worth the effort instead opt for alternative frameworks or develop their own solutions independently. The *Lever* framework addresses this challenge in order to make the development of language tooling as quick and painless as possible. This section explores the key ways used in Lever to lower the barriers to entry. This includes a straightforward project setup, providing the right level of abstraction for the rules, providing a quick feedback loop when writing rules, and for the tooling to be easily used with popular editors.

A critical aspect of the Lever framework is the simplicity of setting up a project from scratch. This challenge is tackled through the use of project templating, more specifically the usage of the `Cookiecutter`[1] tool. This tool allows the user to scaffold an entire *Lever* project by entering a single command and answering a few questions (e.g., name of target language, file extension of target language). More specifically, it is used to generate three main things: a language server project (where the rules are defined), a Tree-sitter grammar, and a VS Code extension. Template example code is also provided for each of these components is also generated. This automated setup not only saves time but also minimizes potential configuration mistakes during manual setup, allowing users to dive into development quickly and start seeing results immediately with *Lever*.

To maintain a low barrier to entry, the *Lever* framework incorporates a simple, lightweight rule language built upon the `Ron`[2] language (see Appendix A). `Ron`, which is essentially typed JSON, serves as the foundation for defining rules in *Lever*. Although `Ron` is not the most concise language, its simplicity and ease of use align with the goal of minimizing complexity in *Lever*. The rules defined in `Ron` are processed using Rust's powerful metaprogramming capabilities, allowing *Lever* to generate code at compile time. This approach ensures that rules are correctly interpreted and applied, avoiding complex and computationally expensive runtime processing. The combination of a straightforward rule language and compile-time validation helps lower the barrier to entry, enabling users to quickly learn and develop language tooling with *Lever*.

Additionally, Rust's metaprogramming provides a quick feedback loop during rule

[1] `https://github.com/cookiecutter/cookiecutter`
[2] `https://github.com/ron-rs/ron`

writing, which is essential for avoiding small mistakes. Previously, rules were bundled in the language server executable and interpreted at runtime, leaving developers without feedback until execution. This made the development process much more challenging. Now, with compile-time validation, any errors in the rules are caught early in the development process, ensuring they are valid if they compile. This immediate feedback allows developers to iterate rapidly, making corrections and adjustments as needed, resulting in a more seamless and productive development experience.

To further lower the barrier to entry, it is crucial that the tooling is easily integrable with existing editors. The *Lever* framework addresses this need by leveraging the LSP (see Figure 3.10). It is specifically designed for integration of the same executable across editors. By following the LSP specification, *Lever* eliminates limitation of building for a specific editor or the necessity of building a specific adapter for each editor the developers want to target. This lowers the barrier to entry by allowing developers to use *Lever* with their preferred editor and target a broad range of editors, all without the need for additional development.

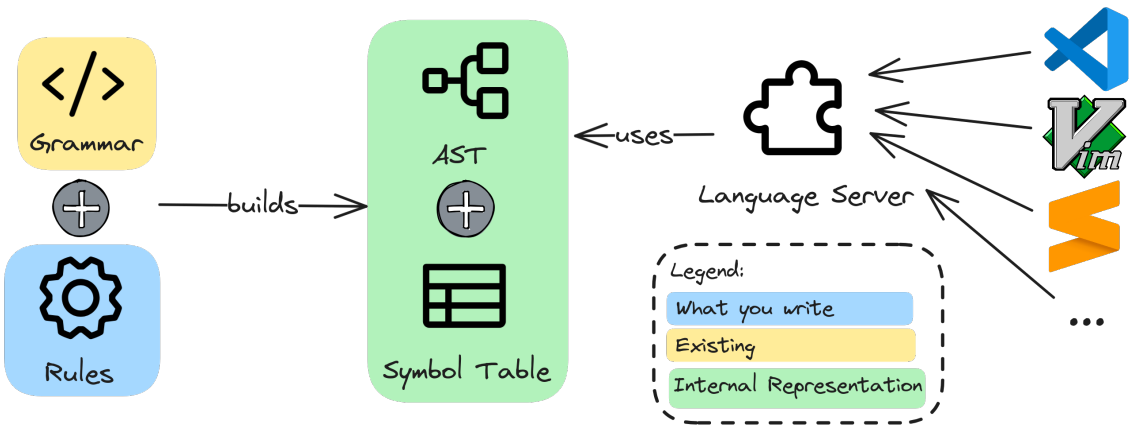


Figure 3.10: Overview of how the LSP is integrated in *Lever*.

In summary, by employing metaprogramming for quick feedback and simple abstraction, simplifying project setup, and integrating the LSP, the barriers to entry of *Lever* are significantly lowered. This makes the framework much more accessible and easy to pick up for developers that want to build editor language support for their BBDSL.

3.5 Leveraging Existing Tooling

Nearly every BBDSL has some sort of existing tooling that can provide some helpful feedback to the user on their code (e.g., compiler, static analyzer). However, this tooling is often not available directly in the editor and requires the user to manually execute it on the side. This separation can create inefficiencies and disrupt the workflow.

The goal of *Lever* is to maintain a lightweight solution therefore it does not make sense to attempt to emulate or reinvent existing tooling. In this case, it makes more sense to use these established tools by integrating them directly into the development environment. The *Lever* framework enables this integration through a simple plugin system designed to connect these external tools with the framework's internal data structures.

The plugin system in *Lever* was initially designed to use *WebAssembly* (WASM) to enable developers implementing plugins to develop for a single target. However, it quickly became evident that, although Rust has excellent WASM support, the technology still lacks maturity and support, and the big amount of added complexity was not worth the effort and results. For this reason, the plugin system was redesigned with a much simpler and straightforward approach: calling executables. That way

developers can simply write a simple adapter script in any language and point the framework to it. This adapter script just calls the target tooling and translates the results into a format that *Lever* can use to bring the output into the editor. For example, the system could invoke a compiler, capture the resulting error messages, translate them into a *Lever*-compatible format, and then *Lever* would display the errors directly in the editor, highlighting the relevant lines with red squiggly underlines. While it may not be the perfect solution, it does what is needed and follows the lightweight philosophy of the overall framework (see Figure 3.11).

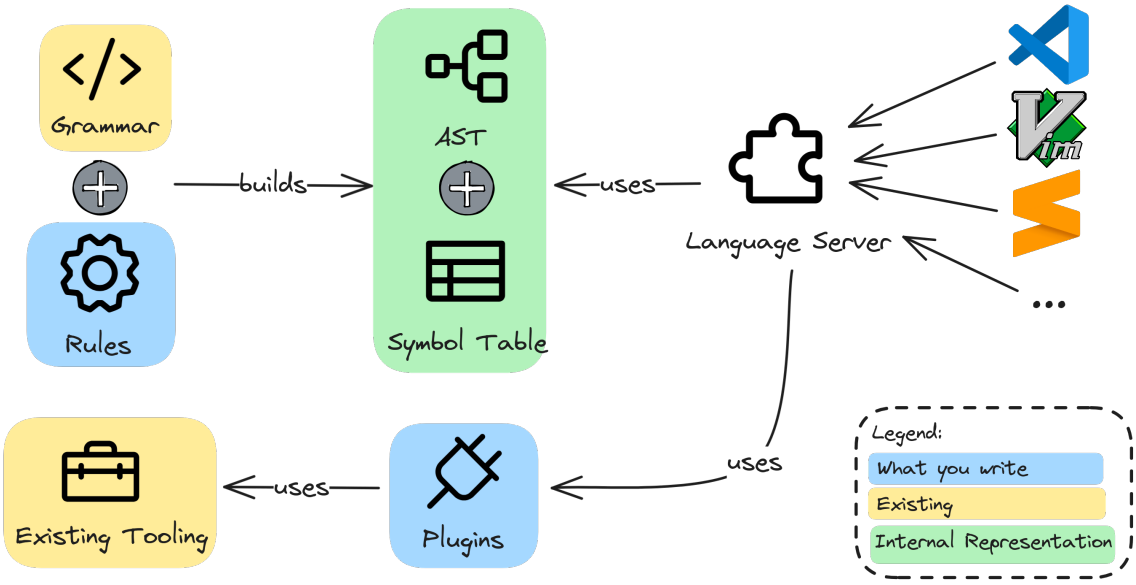


Figure 3.11: Overview of how plugins fit in the *Lever* architecture.

Unlike other *Lever* features that are triggered with each keystroke, these external tools are invoked on-save to stay efficient. Running a full static analysis or invoking a compiler after every keystroke would be computationally expensive and could slow down the user's editor. By triggering these tools only when saving, *Lever* ensures a responsive editing experience while still providing important feedback when needed.

This approach addresses a critical need for users: consistency between the errors reported by the language server and those reported by the compiler. By integrating the same tools used in the build process, *Lever* ensures that the feedback users receive during development aligns with what they will encounter during compilation. This consistency reduces potential confusion and frustration caused by inconsistent and conflicting feedback.

By leveraging existing tools, *Lever* provides more comprehensive feedback that aligns with the established tooling of the target BBDSL. Reusing these existing artifacts also greatly simplifies the implementation by reducing the amount of work needed to develop the editor support.

3.6 Conclusion

In this chapter, we presented the main design principles and decisions that lead to the creation of the *Lever* language editor support framework. It began by distinguishing language support from compiler requirements and thereby showing how *Lever* avoids the vast amount of complexity associated with compilers. Next, the chapter explored how by introducing a universal internal representation *Lever* can support diverse DSLs which allows for the language-agnostic implementation of its features. It was then demonstrated how the framework reuses the existing grammar of the target language by using a rule-based system to add language-specific editing semantics over it. To lower the barrier to entry, *Lever* simplifies project setup and provides immediate feedback with metaprogramming, thereby making it easier for developers to adopt and use. Finally, the chapter discussed how *Lever* integrates existing tools using a simple plugin system which in turn ensures consistency between the feedback

provided by the language server and external tools. The combination of these design decisions is what make *Lever* not only a practical framework to build language servers but also a lightweight one.

Chapter 4

Lever through P4

This chapter presents a descriptive case study demonstrating the development and implementation of a language server targeting the P4 language using the *Lever* framework. This implementation is open-source and is available to all on GitHub[1]. P4 is a DSL specifically designed for programming the data plane of network devices (e.g., switches, routers, NICs) (P4, 2024a). It allows the programming of how packets are parsed, modified, and forwarded by these devices. Originally, network functionality was entirely controlled by vendors. If developers wanted a new feature, it took years due to reliance on vendors needing to develop new hardware. P4 allows developers and network engineers to define network behaviour themselves, thus making changes much more feasible. The syntax of P4 is declarative, allowing developers to define packet structures, match-action tables, and control flow blocks using a C-like syntax tailored for specifying packet-processing behaviour in network devices.

The P4 language was chosen as this study's subject for a few reasons. Firstly, while P4 may not be a mainstream language, it is the leading choice for programming

[1] `https://github.com/ace-design/p4-lsp`

in the domain of *Software-Defined Network* (SDN). It is not just a toy language, it is a real-world language used in industry and supported by big organizations such as Intel and the Linux Foundation (P4, 2024a). Secondly, P4 is not a simple language, it has complex semantics and syntax. This makes it an ideal candidate to showcase the capabilities of the *Lever* framework in handling languages with complicated requirements. Finally, there is a lack of editor tooling available for P4 outside very basic regex-based syntax highlighting. This makes it an excellent example to demonstrate the *Lever* framework's ability to provide editor support when none is available.

4.1 Implementation

To start any *Lever* project, there are 3 main dependencies that need to be installed: Rust, Cookiecutter, and Tree-sitter. After obtaining those, scaffolding a new project through Cookiecutter is the first step in initializing the needed file structure (See Figure 4.1). While not technically necessary, this avoids the headache of manually setting up the project from scratch, ensuring a smooth start.

```
> cookiecutter gh:ace-design/lever-framework-cutter
  [1/5] language_name (): p4
  [2/5] language_slug (p4):
  [3/5] file_extension (p4):
  [4/5] author (): Ace Design
  [5/5] publisher (Ace Design):
```

Figure 4.1: Scaffolding a new *Lever* project using Cookiecutter.

The next step is to either clone an existing Tree-sitter grammar or implement a new one if none is available or suitable. For P4, no Tree-sitter grammar existed, therefore one needed to be defined. However, an official grammar is provided (P4,

2022) and can be translated. While most of this translation is straightforward, the official grammar omits certain details and the entirety of the preprocessor, which is a subset of the C preprocessor. This makes the translation more difficult and error-prone. For this reason, the grammar underwent rigorous testing on a representative dataset of P4 code to iron out the kinks (see Chapter 6). These efforts allowed the grammar to be adopted as the official P4 grammar for the Tree-sitter project (see Tree-sitter's list of parsers[2]). This is one of the contributions made in this thesis. In the ideal scenario for *Lever*, a grammar would already be available like the many already present in the list, which would allow the implementer to skip this step.

With the foundation now in place, the next step is to define the specific rules for the P4 language. The first part of the rules defines the overall language structure, including the language name, file extensions, and library paths. This information is critical as it ensures that the framework recognizes P4 files and integrates seamlessly with the system environment, locating necessary libraries and resources. The flexibility provided by environment variables allows developers to adapt the framework to different setups without modifying the core configuration.

```
language: (
    name: "P4",
    file_extensions: ["p4", "P4"],
    library_paths: (
        env_variables: ["P4_LIBRARY_PATH"],
        linux: [],
        windows: [],
        macos: [],
    ),
),
```

Figure 4.2: Language definition in *Lever* rules for P4.

[2] https://github.com/tree-sitter/tree-sitter/wiki/List-of-parsers

After defining the language metadata, the next step involves listing the keywords. These keywords serve multiple purposes, most notably for syntax highlighting and code completion. By enumerating all of P4's keywords (see Figure 4.3), the *Lever* framework can provide meaningful and context-aware editor features.

```
keywords: [
    "abstract",
    "action",
    "actions",
    "apply",
    "const",
    "control",
    "default",
    "define",
    "else",
    ...
],
```

Figure 4.3: Keyword definitions in *Lever* rules for P4 (truncated).

After defining the keywords, the symbol types specific to P4 are outlined (see Figure 4.4). Constants, variables, types, functions, parameters, fields, and tables are classified by completion type as well as highlighting type. These types are defined by the LSP. This ensures that the way completions are presented to the users is accurate to the language as well as being highlighted the right colour. Differentiating the different symbol types is also key to providing accurate completion and go-to-definition among other features. This approach within the *Lever* framework ensures that the editor support is both intuitive and functional for users of the resulting tooling when writing P4.

The next section is where the rules for mapping the CST to the AST are defined. In other words, is where the additional semantics needed for language support are added onto to grammar. Rather than go through every rule, a single example will

```
symbol_types: [
    (name: "Constant", completion_type: Constant, highlight_type: Variable),
    (name: "Variable", completion_type: Variable, highlight_type: Variable),
    (name: "Type", completion_type: Class, highlight_type: Type),
    (name: "Function", completion_type: Function, highlight_type: Function),
    (name: "Parameter", completion_type: Variable, highlight_type: Parameter),
    (name: "Field", completion_type: Property, highlight_type: Property),
    (name: "Table", completion_type: Class, highlight_type: Class),
],
```

Figure 4.4: Keyword definitions in *Lever* rules for P4.

be presented to demonstrate the process. Note that the full rule file is available in the GitHub repository linked at the start of this chapter as well as in Appendix A. In Figure 4.5, a rule for defining a P4 struct declaration is shown. The rule defines the "StructDeclaration" node, specifying that it initiates a new scope and includes several child nodes (annotations, names, and fields). Each child node corresponds to a query in the CST that maps to either another rule or the direct creation node. For example, the rule identifies the name of the structure as a "Name" node, directly mapping to its corresponding CST query.

```
Rule(
    node_name: "StructDeclaration",
    symbol: Init(type: "Type", name_node: "Name"),
    is_scope: true,
    children: [
        (query: Field("annotation"), rule: Rule("Annotation")),
        (query: Field("name"), rule: Direct("Name")),
        (query: Field("field_list"), rule: Rule("FieldList"))
    ]
),
```

Figure 4.5: *Lever* rule defining a P4 struct declaration.

Completing the rules, global AST rules must be written to ensure that certain elements, such as comments and preprocessor directives, which can appear anywhere

in a file, are properly managed (see Figure 4.6). Due to their nature, they cannot be queried like the rest of the other language concepts as they are outside of the normal syntactic structure of the language. Without global rules, these elements could be ignored, which would result in an incomplete AST causing various problems. Handling them this way also reduces complexity in the rules and streamlines them, since otherwise these elements would need to be written as children of every single rule.

```
global_ast_rules: [
    (query: Kind("line_comment"), rule: Direct("Comment"), highlight_type:
    ↪  Comment),
    (query: Kind("block_comment"), rule: Direct("Comment"), highlight_type:
    ↪  Comment),
    (query: Kind("preproc_include_declaration"), rule: Rule("Import")),
],
```

Figure 4.6: *Lever* global AST rules for P4.

Since the *Lever* uses the Rust toolchain, to compile the rules into a functional language server, running `cargo build` command is required. In case of an error in the rules, a compilation error will be displayed. For example, in Figure 4.7, a comma was omitted in the rules when it was necessary. While the error message contains a lot of unnecessary information, it still contains the information of why and where the error occurred so that it can be easily fixed.

By defining these rules, the *Lever* framework is able to generate working language support for P4. The resulting language server provides auto-completion, hover information, go-to-definition, smart renaming, context-aware syntax highlighting, and syntax error detection, all of which can operate across a multi-file workspace. However, while this implementation is sufficient and could be deployed to developers as-is,

```
> cargo build
   Compiling lever_p4 v0.1.0 (lever-p4/language-server-p4)
error: proc macro panicked
 --> src/main.rs:3:1
  |
3 | start_server!(tree_sitter_p4::language());
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = help: message: called `Result::unwrap()` on an `Err` value: SpannedError {
  ↪  code: ExpectedComma, position: Position { line: 8, col: 9 } }
  = note: this error originates in the macro `start_server` (in Nightly builds,
  ↪  run with -Z macro-backtrace for more info)

error: could not compile `lever_p4` (bin "lever_p4") due to previous error
```

Figure 4.7: *Lever* error message.

this is not the extent of *Lever* capabilities. As mentioned in the previous chapter, *Lever* has a plugin system, allowing for the integration of the existing tooling of the target language. In P4's case, an obvious example of existing tooling is the `p4test` utility.

> "The P4Test Backend is a tool designed for testing and debugging P4 programs. [...] Additionally, it provides a syntax checker for P4 code, enabling the verification of the correctness of your P4 programs. [...] The P4test Backend can check the syntax of P4 programs without being restricted by any specific compiler back end. This is useful for ensuring that your P4 code is syntactically correct." (P4, 2024b)

Lever's plugin system makes it easy to integrate existing tools like this one. The process involves writing a simple adapter script that invokes the external tool, captures its output, and translates the result into a format that *Lever* can process. In this case, the plugin calls the P4Test executable, retrieves its code verification results,

and display those directly in the editor. This removes a lot of weight from the rules since the behaviour of P4Test does not need to be imitated by *Lever* and can be outsourced to the original tooling. The full Rust implementation of this plugin is included in Appendix B.

4.2 Conclusion

The development of P4 language support using the *Lever* framework showcases both its strengths and limitations. The rule-based language definition allows developers to quickly build a language server with features like auto-completion, syntax highlighting, and syntax error detection. The modularity of *Lever*, particularly through its plugin system, simplifies the integration of existing tools like P4Test. This reduces the need for reimplementing functionalities already supported by external utilities, demonstrating the framework's flexibility in adapting to various DSL ecosystems.

However, certain limitations were identified during this process. The reliance on pre-existing grammars is a double-edged sword. While *Lever* works well when a grammar is already available, the process becomes more time-consuming and error-prone when one needs to be developed from scratch. This was the case with P4. Moreover, while the rule system is effective for basic language features, it cannot effectively handle metaprogramming language concepts (e.g., macros, preprocessor elements). This is because, it relies on the existing syntax and grammar to function. This makes it difficult to adapt to more dynamic language elements that are generated or modified during compilation. Lastly, the error messages contain a lot of unnecessary information, making them less straightforward and understandable. These limitations show areas where potential future improvements could be done.

In conclusion, the P4 case study illustrates *Lever*'s potential to simplify the creation of language support tooling, but also exposes areas where some more work is needed. These results provide a foundation for future improvements and expansions of *Lever* 's functionality.

Chapter 5

Validation and Comparison

The simplest way to demonstrate the utility and effectiveness of this framework is to qualitatively compare it to the main available alternatives. To do so, Google's Protobuf DSL will be implemented in MPS, Langium, and finally *Lever*. The chapter will end with a discussion comparing the different implementations. It is important to note that there are no truly objective metrics to definitively compare the different frameworks. Instead, the comparison will rely on qualitative observations regarding development efforts, feature sets, ease of use, and overall development experience. All three implementations are open-sourced and available in this repository: `https://github.com/AlexandreLachanceGit/protobuf-editors`.

Protobuf is a good example for this exercise since it is a relatively simple BBD-SLs that is used in industry. It is a language used to define your data is structured, and it generates the code to read and write this data in one of the supported languages (Google, 2024). This structure is defined in `.proto` files (see Figure 5.1 for an example), which is what the target will be for the different implementations.

```
syntax = "proto3";

message Person {
    optional string name = 1;
    optional int32 id = 2;
    optional string email = 3;
}
```

Figure 5.1: The Protobuf definition for a Person message type.
Source: `https://protobuf.dev`

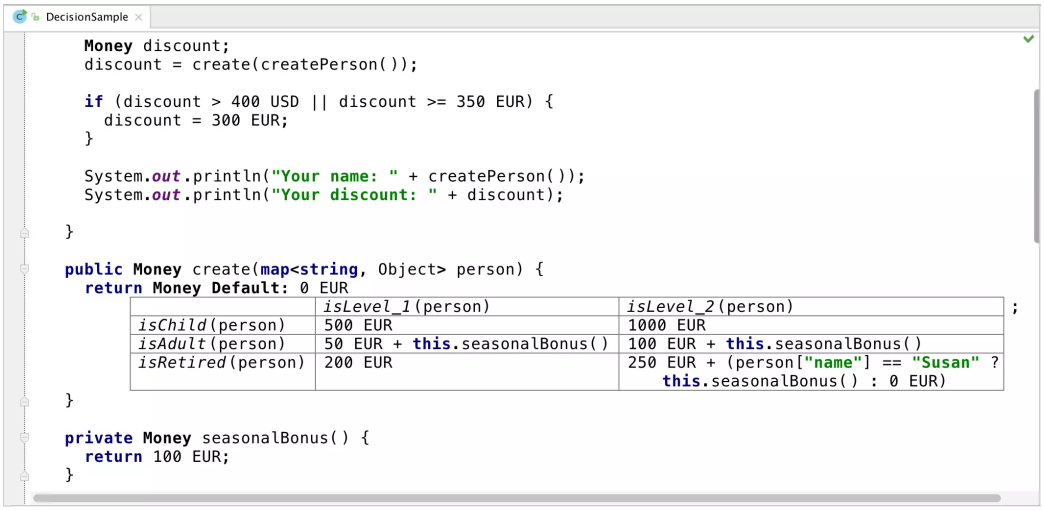
5.1 MPS

Meta Programming System, more commonly referred to as MPS, is an open-source language workbench developed by JetBrains (Voelter and Pech, 2012; JetBrains, 2024). Instead of generating traditional parsers, as most language workbenches do, MPS takes the projectional approach which consists of having the user interact with the AST of the program directly. This has several major benefits:

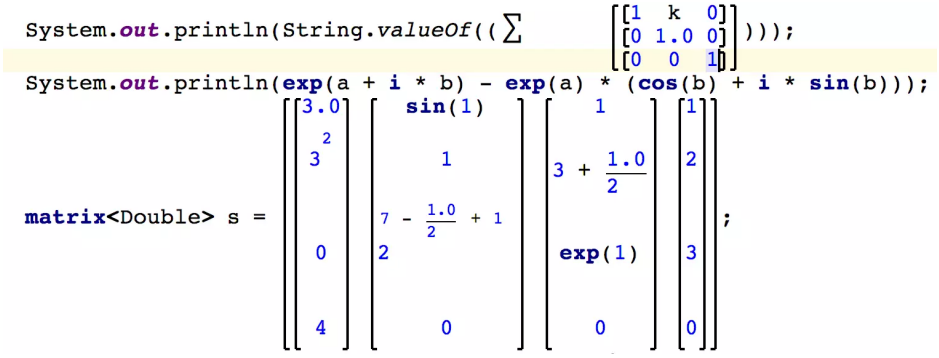
- It allows for the composition of multiple languages. This is not feasible in textual workbenches as it is not possible to compose different parsers. For example, mbeddr, one of the biggest projects built in MPS, is composed of 81 different languages (Voelter et al., 2019). This allows every different concern in large project to be dealt with using a specialized DSL. This paradigm is known as *Language-Driven Engineering* (LDE) (Steffen et al., 2019, 2024).

- It allows the editors of the different DSLs to be much more flexible. Using parsers you are limited to text, but with MPS the editor can take many forms like mathematical, graphical, or tabular notations (see Figure 5.2) (Voelter and Lisson, 2014b).

- Since the user is editing the AST directly, there are no syntax errors. In traditional editors, Syntax errors occur when code does not respect a language's grammar. However, MPS allows you to interact with the code through direct actions on the AST removing the need for a grammar, and thereby eliminating syntax errors.



(a) Tabular notation



(b) Mathematical notation

Figure 5.2: Examples of MPS editor notation.
Source: `https://www.jetbrains.com/mps/`

As for the output, language developers using MPS define generators that transform the MPS code to a chosen language (JetBrains, 2024). It is best when generating Java,

but it is also capable of generating code in various other languages. This code can then be fed to traditional compilers. For example, mbeddr generates C code that is then fed to a C compiler to target embedded systems (Voelter and Pech, 2012; Voelter et al., 2019). This allows developers using mbeddr to enjoy the power and tooling of the C language while developing using high-level abstractions with concern specific languages.

For in-editor language support, can generate both an extension for JetBrains' IntelliJ IDE or an independent IDE (Voelter and Pech, 2012; Voelter et al., 2019; JetBrains, 2024). This language support is very comprehensive, with most expected features (e.g., syntax highlight, renaming, refactoring), This makes MPS potent for the use case of using it as a tool to build language support for an existing language. The ability to design custom editors, closer to the actual semantics of the language than the original textual form of the DSL, by using different notations is very interesting and should be explored.

This section explores the implementation of a projectional editor for the Protobuf DSL using the MPS language workbench. It will detail the process of implementation, challenges encountered, and the resulting product.

5.1.1 Implementation

Creating a Protobuf editor using JetBrains' MPS involves multiple steps. This section outlines the key phases of the MPS implementation process, basing itself primarily on the official MPS documentation (MPS, 2024).

The first step is the creation of an MPS language project. As MPS is a full

graphical application, the project setup is very straightforward. The developer simply needs to enter basic information and confirm the creation (see Figure 5.3). After setting up the project, the next step is defining the language aspects. The different types of language aspects are Structure, Editor, Behavior, Constraints, Intentions, TextGen, Runtime, and TypeSystem. However, Behavior and Runtime are not relevant to the use case of supporting an existing language. Additionally, since Protobuf has a relatively simple type system, using the TypeSystem aspect to define it is not necessary.

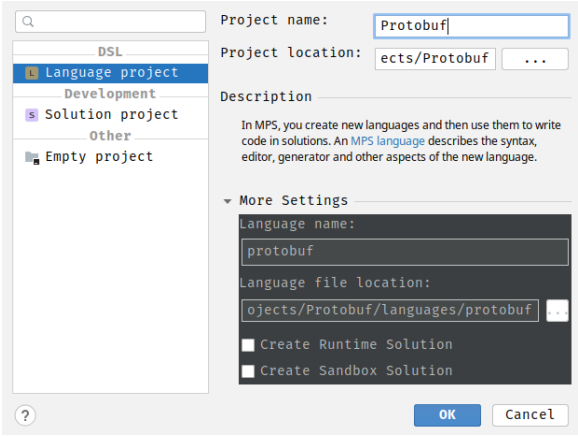


Figure 5.3: Initializing an MPS project.

To define Protobuf, the first aspect that needs to be defined is Structure. Since MPS is projectional, the Structure allows developers to define the elements that form the target language's AST without the need for a grammar. The Structure is defined through Concepts. These represent different types of nodes in the AST as well as define their properties, references, and child relationships. Concept interfaces are also possible to abstract common behaviours between different Concepts. All of this is completely separate and independent of how the different language elements will be displayed in the editor. For Protobuf, defining the Structure involves writing

Concepts for each of the language elements such as messages, fields, and enumerations (see 5.4). Each Concept contains the essential information necessary for building a valid Protobuf AST.

```
concept Message extends    Declaration
                implements INamedConcept
                           IMessageBody

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  << ... >>

  children:
  body : IMessageBody[0..n]

  references:
  << ... >>
```

Figure 5.4: MPS Concept for Protobuf messages.

For each of these Concepts an Editor must be defined to allow users of the resulting editor to interact with the AST. Editors consist of various cells (e.g., reference cells, collection cells, property cells, child cells) allowing for defining the strucutre of how a Concept is written. This is also where syntax highlighting is done by defining how each cell looks through colour, font, etc. For example, considering the Message element of Protobuf, its Editor must define where the name of a message (property cell) is written and where the different body elements go (collection cell) (see Figure 5.5).

With these two aspects completed, the editor is now mostly usable. However, the editing experience can be improved through Intentions, and Constraints. Intentions are context-based actions that simplify editing by automating common edits and allow modifications of existing elements. These actions are triggered based on the current state of a node and are only available when specific conditions are true. For example,

```
<default> editor for concept Message
  node cell layout:
    [-
    message { name } {
      (- % body % ↵ /empty cell: <default> -)
    }
    -]

  inspected cell layout:
    <choose cell model>
```

Figure 5.5: MPS Editor for Protobuf messages.

for Protobuf the `AddFieldOptions` intention (see Figure 5.6) is only available when field does not have options. It allows the user to quickly add them.

```
intention AddFieldOptions for concept IFieldOptions {
  error intention : false
  available in child nodes : false

  description(node, editorContext)->string {
    return "Add field options.";
  }

  isApplicable(node, editorContext)->boolean {
    return !node.hasOptions;
  }

  execute(node, editorContext)->void {
    node.hasOptions = true;
  }

  fields

  << ... >>

  additional methods

  << ... >>
}
```

Figure 5.6: MPS Intention to add field options.

On the other hand, Constraints are used to validate certain properties of a node. For example, for Protobuf an `EnumField` name is always in uppercase. A Constraint can be defined to prevent an invalid value to be entered (see Figure 5.7). This guarantees that the language follows predefined rules during editing.

Finally, for the projectional editor to serve a purpose, the TextGen aspect must be implemented. This is because, what the Editors allow to user to work with is the AST

```
concepts constraints EnumField {
  can be child <none>

  can be parent <none>

  can be ancestor <none>

  instance icon <none>

  property {name}
    get <default>
    set <default>
    is valid (propertyValue, node)->boolean {
      propertyValue.equals(propertyValue.toUpperCase());
    }


  <<referent constraints>>

  default scope <no default scope>

  additional methods

  << ... >>
}
```

Figure 5.7: MPS Editor for Protobuf messages.

directly, there is no text to be used by the compiler or interpreter of Protobuf DSL. The way the AST is translated into the textual representation must be defined. In the case of Protobuf, TextGen would specify how each concept (e.g., messages, fields) is serialized into Protobuf syntax, ensuring that the user's changes in the editor are correctly transformed into textual code (see Figure 5.8).

```
operation genWithBody(string nameBody, string name, nlist<> bodyChildren) {
  append indent ${nameBody} { } ${name} { } {{} \n;
  increase depth;
  append $list{bodyChildren with \n};
  decrease depth;
  append \n;
  append indent {}};
}
```

(a) Helper function for text generation of language elements with bodies.

```
text gen component for concept Message {
  (node)->void {
    append gen with body "message", node.name, node.body;
  }
}
```

(b) Text generation of Protobuf messages.

Figure 5.8: MPS TextGen for Protobuf messages with helper function.

5.1.2 Challenges and Limitations

Although MPS offers a powerful environment for language development through its projectional editing paradigm, it also comes with significant challenges and limitations. These issues prevent it from being a good solution to build language tooling for BBDSLs.

One of the main challenges of MPS is also its greatest strength, its projectional approach. Unlike other solutions, this limits it to a single editor (JetBrains' IntelliJ). This limitation can be a significant drawback for developers who prefer working in other environments or for teams with diverse editor preferences. It also makes adoption a lot more difficult. This is because, in general, developers are much more used to working in textual environments and have never touched a projectional editor. These have a significant learning curve that can scare potential users as well as potentially slow development.

Outside of this adoption issue, MPS also has some design problems for this use case. While MPS does facilitate better separation of concerns through its aspects approach, this fragmentation of concerns introduces significant overhead in terms of both initial development and ongoing updates. The Protobuf implementation alone required the creation of over 100 different files all aspects combined. For such a simple DSL this is very high. This not only slows down DSL but makes maintenance difficult as everything is scattered; a small change might require modifications of many different files.

Another major limitation is MPS's assumption of full language implementation within its environment. This makes it difficult to reuse existing DSL artifacts. If

a grammar or tool already exists, it must be rewritten entirely within MPS's projectional model, a process that is not only time-consuming but also prone to errors (certain features may not translate smoothly). For the same reason, MPS does not have built-in support for integrating with existing DSL tooling or library systems, forcing developers to manually implement these integrations. This further complicates the development process, as manual solutions for managing dependencies and libraries must be created and added to the already complex but open-source MPS workbench.

In summary, MPS provides a great workbench for language development. However, its projectional editing paradigm, steep learning curve, and limitations in reusing existing artifacts and integrating external tooling make it less adapted for the specific use case of building language support for BBDSLs.

5.2 Langium

Sold as the spiritual successor to Xtext[1], Langium is a textual language workbench with a focus on targeting VS Code (Spönemann, 2022). It is designed around a modern technology stack, as it is fully built in TypeScript and uses a modern parser called Chevrotain[2]. By utilizing a single language, TypeScript, for all development tasks, Langium simplifies both development and maintenance compared to Xtext, which requires knowledge of both Java and TypeScript (Petzold et al., 2023). The parser library itself is abstracted behind Langium's declarative language which generates the actual grammar. This language defines both the syntax and the semantics of the

[1] `https://eclipse.dev/Xtext/`
[2] `https://chevrotain.io/`

target language.

For in-editor language support, Langium can automatically generate a language server. This is more flexible than the editor-specific approach that projectional editors like MPS take, as it can be used with the editor chosen by the language user. It also allows a very relevant use-case for this thesis: using Langium solely in the goal of building language support for an existing language. In fact, this use-case is represented in one of the example projects built by the Langium team (TypeFox, 2024). However, there are some issues that point to it not being the best framework for this specific use-case. This is because, among other issues, Langium requires a complete and often complex translation of the existing DSL's grammar, lacks a built-in way to support the existing library system of the DSL, and does not have a built-in way to interface with existing tooling. All of these issues makes the process more difficult and error-prone. For a more in-depth breakdown of these issues and a case study of Langium, refer to Section 5.2.

This section explores the implementation of a language server for the Protobuf DSL using this tool. It will detail the process of implementation, challenges encountered, and the resulting product.

5.2.1 Implementation

The implementation of a language server using Langium involves several steps (see Figure 5.9). This subsection provides a high-level overview of this workflow, with most of the information being sourced from the official Langium documentation (Langium, 2024).

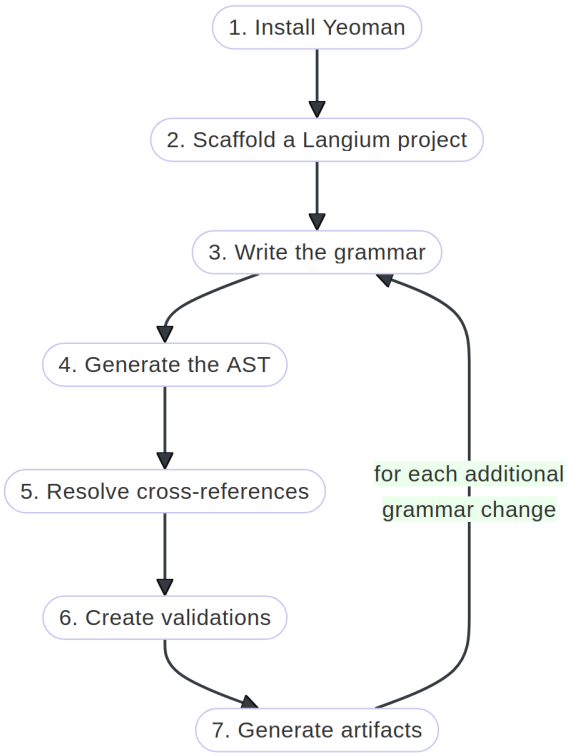


Figure 5.9: The Langium workflow.
Source: `https://langium.org/docs/learn/workflow/`

Although Langium requires a complex project structure to begin with, this complexity is alleviated through Yeoman[3]. It is a project scaffolding tool helps users quickly set up a projects structure and configuration. In the case of Langium, the user simply needs to execute Yeoman and answer a few questions to set up a project adapted to their needs (see Figure 5.10).

After setting up the project, the next step is to define the language. This is done through Langium's grammar language which allows the developer to define both the syntax and part of the semantics of the DSL. It is a very simple and straight forward language to use.

[3] `https://yeoman.io/`

Figure 5.10: Creating a Langium project using Yeoman.

The Langium grammar language is then used to generate code. This includes both a Chevrotain (a JavaScript parsing library) grammar and an AST data structure. All the generated code is available for review to the user in a subdirectory. This transparency allows the user to inspect and debug the parser in cases of errors if needed. Although, the code cannot be manually edited as all changes would be overwritten by the generator at the next compilation.

When writing a Langium grammar, the first step is to implement the terminal rules (see Figure 5.11). These rules are used in conjunction with the keywords to perform the lexical tokenization of a document. They are written using either regular expressions (regex) or *Extended Backus-Naur Form* (EBNF). Although, EBNF compatibility is only present to make it easier to port Xtext grammars and the use regex is strongly recommended. Terminal rules can also be marked as hidden. This allows them to be both global and ignored during lexing which is useful for ignoring

elements like comments or white space.

```
terminal ID: /[\._a-zA-Z][\w_\.]*/;
terminal INT returns number: /[0-9]+/;
terminal STRING: /"(\\.|[^"\\])*"|'(\\.|[^'\\])*'/;
terminal BOOL: "true" | "false";

hidden terminal WS: /\s+/;
hidden terminal ML_COMMENT: /\/\*[\s\S]*?\*\//;
hidden terminal SL_COMMENT: /\/\/[^\n\r]*/;
```

Figure 5.11: Langium terminal rules for Protobuf.

After the terminal rules, the parser rules need to be written. These are used to validate and parse sequences of tokens to build an AST. Each parser rule represents an AST node kind in the generated data structure. They are declared with a name, followed by a semicolon, and finally with the parser definition.

Parser rules use EBNF-like expressions to define the structure and cardinality of elements. Cardinalities specify the number of elements, ranging from exactly one to zero or many. Expressions are ordered sequences, but the pipe operator (or) can be used to offer alternative valid expressions. Keywords, written as strings, match specific character sequences; they cannot be empty or contain white space.

Assignments in parser rules set properties for the resulting AST node. Single values, multiple values, and boolean properties can be assigned using the different provided operators. Cross-references can also be used to allow rules to reference objects of a specific type directly. This is done by linking a property of a rule to an instance of another rule. An example of this can be seen in the reference to a message ID in the "Rpc" rule in Figure 5.12.

The starting point of parsing is declared using the entry rule. This rule is simply

preceded with the keyword `entry`. Then the parsing continues with the children of this rule.

```
entry Model:
    (syntax=Syntax)?
    (
        empty+=';'
        | packages=Package
        | imports+=Import
        | messages+=Message
        | enums+=Enum
        | options+=Option
        | services+=Service
    )*;

Message:
    'message' name=ID '{' MessageBody '}';

Rpc:
    'rpc' name=ID
    '(' stream?='stream' message=[Message:ID] ')'
    'returns'
    '(' stream?='stream' message=[Message:ID] ')'
    ( ('{' (options+=Option | ';')* '}') | ';');
```

Figure 5.12: Partial Langium parser rules for Protobuf.

After implementing the grammar, a basic Language Server can be generated. This server provides essential language features such as auto-completion, go-to-definition, and type definition, enhancing the development process significantly. It also offers detailed and helpful syntax error messages. The Langium language server also includes automatic workspace management capabilities that works by detecting and processing all files written in the target DSL present in the current workspace. By being written in JavaScript/TypeScript, this language server is very simple to deploy. This is because, it can be packaged in a VS Code extension that can be directly published

in the marketplace without the need to ship binaries.

From this base, it is possible to push Langium further to obtain a smarter and more complete tool. This is accomplished through a service oriented architecture where different features can be implemented manually in TypeScript. New services simply need to be registered through dependency injection. It is also possible to overwrite default behaviours through the same mechanism. One of the best examples of this is the implementation of validators. This allows the user to write custom TypeScript functions to validate a specific node type. For example, we could validate that the identifier of an enumerator field is written in upper case (see Figure 5.13). Other examples of services that could be implemented through the same mechanism include a formatter, hover information (documentation) and workspace management.

```
checkEnumFieldIdIsAllCaps(enumField: EnumField,
                          accept: ValidationAcceptor): void {
    if (enumField.name &&
        enumField.name != enumField.name.toUpperCase()) {
        accept('warning',
               'Enum field identifier should be ALL CAPS.',
               { node: enumField, property: 'name' })
    }
}
```

Figure 5.13: Example of a validator that validates that an identifier is written in upper case.

Langium also generates a Monarch and a TextMate syntax. These are used for syntax highlighting in different editors, with Monarch mainly being used in VS Code while TextMate is used in editors like Sublime Text and Atom. This allows faster syntax highlighting than what can be accomplished directly through the generated language server. However, the generated syntaxes are very basic with only support

for keywords, comments, strings, and symbols (e.g., brackets, parentheses, operators). These also cannot be manually edited without being overwritten at the next compilation.

5.2.2 Challenges and Limitations

While Langium provides a lot of value, the implementation of a Language Server for the Protobuf language has also revealed some significant problems. These can significantly affect the development experience of the tool for building language support. The section discusses the main challenges associated with development using Langium.

One of the biggest problems with Langium is the difficulty associated with debugging issues. When a problem occurs with your grammar or one of your services, it is very difficult to identify the source. This is because, there are many possible sources of errors; it could be an error in the grammar, a problem in the implementation of a service, or even a bug with Langium itself. Langium does not help the user with this outside of very basic parsing errors. Additionally, the fact that there is too much responsibility placed on the grammar, with the merging of different concerns such as parsing and some aspects of static semantics, further complicates development and debugging. This conflation of concerns makes it challenging to isolate and resolve issues effectively.

Additionally, Langium lacks a built-in way to interface with existing tooling. This absence can be a drawback, especially when trying to use existing tools to streamline development. Developers need to invest extra effort to create custom solutions for

integrating with other tools, adding complexity and detracting from the overall efficiency of using Langium for language development. However, this is a very difficult system to implement manually.

Another significant issue of Langium is the available documentation. It is mainly composed of guides, tutorials, and examples (both in text and code). These are very useful to get started with this framework as they outline the basics perfectly. However, they fall short when it comes to in-depth concepts and advanced use cases. There is no *Application Programming Interface* (API) documentation available which leaves the user to hunt for what they need either directly in the open-source code base or in existing open-source projects. Such additional documentation would fill this gap in knowledge and Langium a better platform for more advanced projects.

As mentioned quickly in the first point, stability and reliability is another area of concern for Langium. As a prime example, the VS Code extension for Langium and its associated Language Server, which are built in Langium, have a tendency to crash frequently during the writing of a grammar. This not only impacts the development of tooling using Langium, it outlines a stability issue with products built using it. This issue might be due to the fact that this tool is relatively new and in constant development, however it still impacts its user and make its usage more difficult.

Moreover, there is a feature that is clearly missing from Langium. This is an easy way to implement a library system for your target language. Out of the box, Langium provides services completion or renaming across files that are in the same directory. Which effectively means that Langium's representation of a code base has automatically imported all the other files. For most DSLs, this is not the intended behaviour. Instead, they often require a more controlled and explicit management

of dependencies and libraries. This means developers have to manually implement a system to handle libraries and dependencies correctly which is not be a trivial task.

Lastly, even if a grammar for the target language already exists and is open-sourced, it needs to be fully translated and rewritten in the Langium grammar language. Depending on the complexity of the grammar and the original parser technology, this can be a very difficult and time-consuming process. Certain features of the original grammar may not have direct equivalents in Langium, which would necessitate complex workarounds or adjustments. This translation effort can introduce new errors and inconsistencies, which would further complicate the development process.

Together, these issues highlight the areas where Langium, despite its modern approach and capabilities, falls short for the use case of this thesis. Addressing these challenges would be crucial to improve the development experience and make Langium a more robust and versatile tool, especially for implementing language support targeting BBDSLs.

5.3 Lever

There is no point in covering *Lever* like MPS and Langium in this section as its implementation and faults have already been covered in the previous chapters. However, it is important to point out how the implementation process differed from the one of P4. The main difference in this case was that the Tree-sitter grammar for Protobuf already existed[4]. Therefore, the process was simply to initialize the project with Cookiecutter, clone the grammar, and write the rules. There was only one slight hiccup, the fact that the Tree-sitter version of the grammar didn't match the one of the

[4] `https://github.com/treywood/tree-sitter-proto`

Lever framework, blocking compilation. The grammar then add to be forked to simply change the version number. This fixed the only issue that occurred in this use of *Lever*. A plugin was then implemented using the `protoc` compiler[5] as the target this. This furthered validation by providing feedback and validation outside of the simple syntax checking provided by default with *Lever*. The full Python implementation of this plugin is included in Appendix C.

5.4 Discussion

This section presents a comparative discussion of the three approaches to building BBDSL editor support presented in this chapter: MPS, Langium, and *Lever*. As shown, each tool presented has its strengths and weaknesses when it comes to this use case.

One of the most main differences between the three tools is the development effort required. The projectional editing approach of MPS causes a very time time-consuming development. This is because, the fragmented concerns require developers to write and maintain hundreds of small files. On the opposite of the spectrum, *Lever* and Langium both lower the barrier to entry with straightforward textual languages used to define behaviour. However, Langium still requires a complete and sometimes complex translation of the existing target DSL's grammar. This is not always necessary in *Lever* as existing grammars can be reused, as it was the cases for Protobuf, because of the integrative and lightweight philosophy of the framework.

Ease of use is another major area of comparison. MPS, due to its projectional

[5] `https://github.com/protocolbuffers/protobuf`

approach, is very difficult to learn, especially for developers used to traditional text-based languages and editors. This not only affects the adoptability for developers that write the tooling but also for the users of this tooling. Langium was built with modern technologies that a lot of developers are used to. More specifically, outside of its simple grammar language, everything else is written in TypeScript and the development is built around Visual Studio Code. This helps with adoption as well as ease of use. *Lever*'s rule language is also simple, but, in cases where necessary, translating a grammar to a Tree-sitter grammar can be difficult. All of these solutions simplify project initialization through scaffolding, however they all suffer for a lack of availability of quality documentation and tutorials. This can lead to developers to avoid these solutions and attempt fully manual implementations.

The capabilities of each framework also differ significantly in terms of features. MPS is, without a doubt, the most powerful form of editor support out of the three. Due to its projectional nature, it allows developers to define how editors for each language element are designed independently of the target DSL. Custom notations like tables or graphical editors can be defined as long as they contain the necessary information for text generation of the DSL. This flexibility is valuable when building complex language supports. However, it means that it can only be used in the JetBrains IDE as it is the only one that supports it. *Lever* and Langium, on the other hand, automate the generation of a language server which can be used across popular text editors like Visual Studio Code. While their depth of language support feature is lesser than MPS, they still provide the features that developers have become accustomed to when working with GPLs like syntax highlighting and syntax error detection. Where both MPS and Langium fall short, is their support for multi-file

workspaces and libraries. As both were intended to be language creation tools, they implement their own approach. Their respective approaches do not function with the vast majority of DSLs, requiring manual implementation of these systems. *Lever* has an integrative approach to library system and attempts to be broad enough to support most common library systems.

One key strength of *Lever* compared to the other two approaches is its warning and error diagnostics. While, when using MPS and Langium, developers attempt to simulate the diagnostics of the language through Constraints and Validators respectively, *Lever* can directly integrate the existing compiler's or static-analyzer's into the editor. This not only simplifies implementation, it also avoids the case where your editor and your compiler give you different or even contradicting feedback.

In conclusion, all three solutions allow for the development of DSL editor support. It is clear that MPS and Langium are much more featureful and powerful solutions for most use cases. However, as shown, providing editor support for BBDSLs is not their use case of predilection. Through specific design decisions as well as its integrative and lightweight approach, *Lever* shows it lends itself more to this use case.

Chapter 6

How to Validate Against an Existing DSL Ecosystem

This chapter delves into the process of constructing a high-quality validation dataset for a target DSL. Given the complexity of DSLs, having a robust dataset is essential for accuratly validating coverage of the language server built using *Lever*. This coverage can be calculated by comparing what the original compiler of the DSL can analyze to what the language server can. However, for this analysis to be effective, the dataset must be wide-ranging and representative of the overall ecosystem.

In the software engineering world, Git is the industrial standard for version control. It enables development teams across the world to collaborate on software projects. A lot of these projects are open-sourced and available to everyone through *Git forges* like *GitHub*, *GitLab*, and *Bitbucket* to name a few. Many of these *forges* provide APIs to query projects, authors, issues, etc. This provides researchers with easy access to a vast amount of data. For example, on *GitHub* alone, 52 million new public repositories were created in Woodward (2022). These projects can provide

incredibly deep insights into the software field, as advocated by the Mining Software Repositories research community.

Extracting information from projects can be complex since many pitfalls must be avoided. The reason for that is the fact that software projects are inherently very complex structures. For example, inconsistent coding practices, the evolution of projects over time, the large size of some projects, the lack of documentation, and duplicate projects can all be challenges when analyzing *Git* projects.

Interestingly, DSLs trigger new challenges when mining repositories. This is mainly due to their domain-specific aspect, which makes them less mainstream and, as such, different from mainstream applications using fashionable languages or frameworks. In this chapter, the focus will be on the challenge of deduplication, a prototypical example of a situation that exists in classical mining but is amplified by the specificities of working with the evolution of DSLs. The presence of duplicates affects the overall quality of the dataset, leading to skewed results and inaccurate validation of the language server's capabilities.

Project duplicates can be a very serious issue when trying to build a useful dataset of software repositories from forges. Not only do duplicates lead to unnecessary redundancy, consuming and wasting valuable storage and computational resources, but they also compromise the result of the validation based on the dataset. In this case, empirical results could be biased due to the disproportionate influence of some projects that were duplicated a lot. This would skew the results of the accuracy of our grammar.

This might seem like an easy problem to solve when encountering it for the first time; it is however far from trivial. There is a multitude of things that complicate the

problem. Firstly, in a perfect world, duplicates should be easy to find. Most, if not all, major *Git* forges provide ways to *fork* a project (duplicate it). These relationships are tracked by the platform and can be referenced in the data from the different APIs. However, some major changes might have been made since the fork, therefore, the project might no longer be considered a duplicate for some use cases. On the other hand, some projects might be duplicates even if they were not forked (see Fig. 6.1). Some projects are *cloned* or downloaded, and then pushed as new projects, losing the relationship between the two projects in the process.

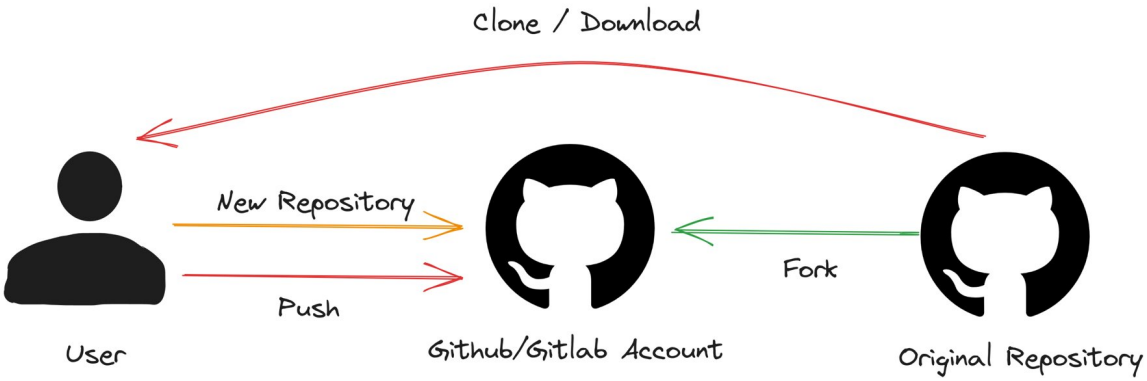


Figure 6.1: Diagram of the duplication causes.

The naive solution to this problem would be to compare every single one of the projects in the dataset to every single other one. Due to the complex nature of code repositories, this is computationally unfeasible. We can however exploit the different types of relationships outlined in the previous paragraph to greatly reduce the search space of these operations and make deduplication achievable. This is what the approach defined in this chapter aims to accomplish.

This chapter is organized as follows. Section 6.1 focuses on the related work regarding mining and deduplication. Section 6.2 describes the solution, as well as its

implementation. In Section 6.3, the proposed solution to the P4 language. Finally, Section 6.4 concludes the chapter.

6.1 Related Work

In exploring the subject of deduplication of software repositories, it became evident that little work had been done on the subject yet. There is, however, a lot of previous work on code duplication detection. This section will discuss the few truly related works and how the tangentially related work can help us with this problem.

> **Note:** This chapter, while still connected to the overall theme, addresses a distinct problem from the rest of the thesis, which is why it has its own related work section. Including this literature review in Chapter 2 would not have been appropriate.

6.1.1 Code duplication

Some work has been done to identify the issue of duplicate code. For example, a study found that 70% of files on *GitHub* were duplicates of other files Lopes et al. (2017). More relevant to this chapter, the same study also found that between 9% and 31% of projects were made up of at least 80% of duplicate files Lopes et al. (2017). While this study Lopes et al. (2017) doesn't address the problem of complete project duplication, it may hint at it. More focus studies are required to understand the scope of the issue better.

Furthermore, the impact of code duplication cannot be ignored. There has also

been work on analyzing the problem of duplicate code when training *Machine Learning* (ML) models. A study found that some results could be inflated by up to 100% when models are trained on a dataset containing duplicates as opposed to a deduplicated one Allamanis (2019). These results show the importance of addressing the duplication issue when working with code data-trained ML models. It is essential to know that duplicates only affect training in some fields and contexts. A study on malware detection models found that duplicates had little effect on the efficiency of the final system Zhao et al. (2021). Further study on whether duplicate impact ML models trained on projects (or repositories) might be necessary. However, the impact of empirical tool testing on duplicates cannot be denied due to the bias introduced in the results.

6.1.2 Repository similarity

Another essential part of this chapter and the proposed approach is how the similarity between different projects is calculated. Some projects might not be 100% duplicated and still be considered duplicates depending on the application of the dataset. It is then essential to find a set of metrics to evaluate similarity and set a threshold for these metrics on what is considered a duplicate. The state-of-the-art tool for this is *CrossSim* Nguyen et al. (2020); it is a tool that enables computing the similarity of different *Open-Source Software* (OSS) projects. However, the focus is not on finding duplicates. It was made to make it easier to find similar projects, and it is not suitable for this use case. A major limitation of this tool, and similar ones, is its impracticality for general deduplication. Most, if not all, approaches focus on projects with a singular programming language: *Java* Nguyen et al. (2020). Modern-day projects use, on

average, five programming languages per project and *Java* is only a part of it Mayer and Bauer (2015). Therefore, there is a need to find a quick and language-agnostic way to compare projects for this approach.

There is, however, another approach that is interesting for this use case. This approach vectorizes repositories and all associated data into comparable embeddings. In *Natural Language Processing* (NLP), embeddings represent words or phrases as vectors that capture semantics and enable more accessible computational analysis. This has been used successfully for the deduplication of complex texts Gyawali et al. (2020). The same principle can be used for repositories. It has already been implemented by Rokon *et al.* Rokon et al. (2021) and has proven very effective. Their approach captures the semantics of the associated metadata, the structure of the repository, and the entire source code Rokon et al. (2021). If the generated vectors genuinely represent the full semantics of a repository, as the authors propose, the distance between vectors would be an ideal metric for repository similarity. However, their code has yet to be available and is, therefore, unusable in this case. The reproduction of such an approach falls far from the scope of this chapter in terms of complexity and the size of the work.

6.1.3 Repository deduplication

There has been some work on the specific subject of *Git* repository deduplication. It mainly consists of a single paper by *Spinellis et al. (2020)* Spinellis et al. (2020), in which the authors found 30 thousand project duplicates out of 1.8 million projects. The approach consisted of geometric mean-based grouping and denoising of project clumps Spinellis et al. (2020). They used various metadata points like stars, *Git*

history, fork information, etc. However, their approach has a lot of limitations and problems. First of all, the authors only focused on *GitHub* when there are a lot of different forges available that could provide a variety of projects. This might be fine on a large scale, but the extra data provided by other forges could help make the final dataset more versatile and helpful when working on a smaller scale. Secondly, their approach could have been clearer and depended on much manual work to denoise and clean the data. While this led to better results, the approach took time to reproduce, even on a smaller scale. Also, the liberal approach to denoising is not applicable in smaller-scale scenarios since it would remove many potential candidates. Thirdly, their method only examined surface-level metadata about each project and never examined the projects' actual content. For this reason, the authors only found duplicates with common *Git* histories and fork relationships. Because of this, many duplicates were likely missed and are still present in the final dataset. Metadata is also unreliable; for example, they used stars as a metric to determine attractor projects. One study looked at the correlation between *GitHub* stars and code quality, and the authors were unsuccessful in linking the two Naveed (2022). Lastly, their choice of technology could have been more optimal. They used a relational database to represent graph data, leading to convoluted SQL queries. However, it is possible to draw some inspiration from this paper to build a more adapted approach, notably using *Git* history and fork data.

6.2 Proposed Solution

This section describes the proposed approach in detail, including its strengths and weaknesses. The approach is structured as a pipeline, composed of six sequential

steps, spanning from the creation of the initial dataset to the final deduplicated dataset (you can refer to Fig. 6.2 for an overview of the entire solution pipeline).

The approach relies on creating a relationship graph to reduce duplicate search space at each step. That is the sole focus of the three steps that follow the initialization of the dataset. The final two steps are used to delete (or flag) duplicates. The *Simple Duplicate Deletion* step uses metadata to delete projects that are obvious duplicates, further reducing the number of project pairs that need to be compared in the final step. The final step (*Full Similarity Metrics Deletion*) is a very computationally intensive step because of the intrinsic complexity of software repositories. This is why the whole approach focuses on reducing the search space before reaching it. It involves a complete comparison of project pairs.

The rest of the section will focus on describing the process behind each step and the thought process behind all those decisions.

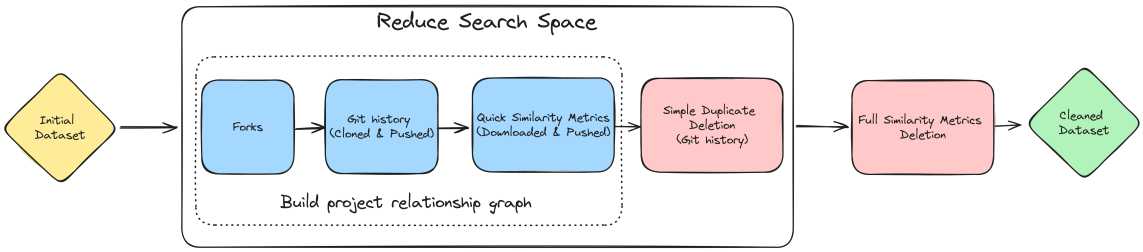


Figure 6.2: Diagram of the full solution pipeline.

6.2.1 Initial Dataset

To start the construction of the dataset, the initial repositories must first be acquired. To do so, the different REST APIs that the different forges provide can be used. The two main *Git* forges that provide APIs are *GitHub* and *GitLab*. Using their query

features, a dataset can be built that focuses on any desired characteristic of the repositories. This could be the programming language used, the number of stars, the license used, the description of the project, etc. However, since this approach relies on the existing relationships between the different projects, it is better to target a known community. For example, targeting the community that uses a specific programming language would be better since there will be clearer links between the different projects.

There are some limitations with the different APIs that make collecting repositories difficult. For example, the *GitHub* API only allows up to 1000 repositories per query. This can be circumvented by splitting the query into multiple queries. The easiest way to do so is to split the whole span of the query into multiple date ranges. Splitting can be implemented to be done automatically. The *GitLab* API is also very unstable, often returning errors. This can make collecting data more strenuous. To prevent missing relationships in the following steps, if the metadata of a repository indicates that it is a fork but the parent is absent from the collected data (for various reasons), the missing projects are added to the dataset. These APIs only return the metadata of the repositories that match the query.

The schema of this metadata is very similar across forges, but some terminology varies. Therefore, some schema matching must be performed before proceeding. Some fields do not serve any purpose for deduplication and are therefore removed. To keep track of the origin of each repository, a field is added for this information (see Fig. 6.3).

The approach requires the actual code and full *Git* history of each project. To achieve this, all of the repositories must be cloned. This can be a problem since,

- **forge**: The project's forge.
- **id**: The project id.
- **name**: The project name.
- **full_name**: The project name with name space.
- **description**: The project description.
- **created_at**: The project creation date.
- **updated_at**: The last time the project was updated.
- **allow_forking**: If the project allows forking.
- **forks_count**: The number of times this project was forked.
- **stars**: The project's amount of stars.
- **owner_id**: The id of the project's owner.
- **owner_username**: The username of the project's owner.
- **fork**: If the project is a fork.
- **parent_id**: If fork, the project's parent id.
- **parent_name**: If fork, the project's parent name.
- **parent_full_name**: If fork, the project's parent full name.
- **parent_creator_id**: If fork, the project's parent creator id.

Figure 6.3: The metadata fields for repository nodes.

depending on the number of repositories, it can take up a lot of storage space. Luckily, because of the nature of code (text), most repositories are only a few megabytes. In case of storage constraint, this could be optimized by deleting irrelevant files for each project, such as *PDF* documents or *ZIP* archives.

After completing these steps, a unified multi-forge dataset is ready for deduplication.

6.2.2 Forks

This is the first step of the pipeline that starts building the relationship graph and, therefore, starts reducing the final search space. It addresses the first type of duplicate, forks (see Fig. 6.1).

The process begins by loading all the metadata obtained from the first step as nodes in a graph. The different types of relationships will be represented as different edge types between nodes (repositories/projects). In this case, it is better to use a graph database. It will make the data more accessible to work with since the database will be adapted to the structure.

This step involves connecting the projects that have been forked to their parent. It is simple since the different repositories already link to their parent in their fields. After linking the nodes, a graph begins to take shape. It is also possible to identify which projects are "source" or central projects (projects that are parents to others and not children of any other). These projects will be crucial to the next steps.

6.2.3 Git History

This step aims to add to the relationship graph by using the information given by *Git*, more specifically, the *Git* history. It addresses the second type of duplicates, projects that were cloned and then pushed as new projects (see Fig. 6.1).

Projects that are cloned from online forges keep their *Git* history. This information can then be used to link them to the original repository. This requires comparing a project's first commit ID to the first commit ID of the central projects identified in the previous step. It is only necessary to compare projects to central projects, as those forked from them will share the same starting commit history. Commit IDs are designed to be as globally unique as possible Git (2023), there is then a very low chance of false positive. However, some projects could have diverted significantly after the first commit and would no longer be considered duplicates. This is why full project comparison is still conducted at the last step within the relationship graph.

6.2.4 Quick Similarity Metric

This step aims to finalize the relationship graph using additional surface-level data about the project. In this case, the name of the project and the project structure are used. It addresses the second type of duplicates, projects that were downloaded/copy-pasted and then pushed as new projects (see Fig. 6.1). In this case, the information provided by the *Git* forge or *Git* itself can no longer be relied upon. This is because the projects are downloaded directly from a forge, they lose all *Git* related information and are downloaded as an archive (typically a *ZIP* file). Therefore, this step can only be based on the information contained within the project itself. To identify the project pairs for comparison, central projects are compared to disconnected projects.

This approach carries the risk of missing duplicates of other projects already present in the relationship graph. However, it is less likely that a project was duplicated from a child project than from a more popular central project. This limitation can be partly mitigated by setting a lower similarity threshold during this step, relying more on the final step to identify true duplicates.

It would be too computationally expensive to compare all central and disconnected projects using their file content. This is the reason only surface-level information is used for this step. Two metrics have been identified to quickly calculate the similarity between project pairs: the similarity between the names and the similarity between the file trees. If better ones are identified (in a framework approach), these metrics could easily be swapped out for different ones.

The name of a project represents the purpose of it. It then makes sense that the name would be similar if the code is the same. There are multiple ways to calculate the similarity between two project names. The most accurate is to use NLP techniques, like word embeddings, that capture the semantics of the words. That would allow completely different names with similar meanings to have a high similarity score. However, incorporating this into the solution involves adding a lot more complexity and computations. Instead, the edit distance between the names, specifically the Levenshtein distance, was chosen. The distance was normalized between 0.0 and 1.0 to ensure comparability across project pairs with varying project name lengths. This technique is much more straightforward to implement and requires no additional computational resources. It, however, does not capture semantics and is less accurate.

The file tree structure of a project usually reflects the composition of it. Projects with a similar structure, while the content of the files may vary, are more likely to be

duplicated. Therefore, by comparing the file trees, the surface-level project similarity is captured without diving into the actual file content and code, making this operation very fast and efficient. The best way to compare trees is to calculate the distance between trees. Tree edit distance is the minimum number of operations (add, delete, replace) needed to transform one tree structure into another. The state-of-the-art way to calculate this is using the *APTED* algorithm Pawlik and Augsten (2015)Pawlik and Augsten (2016). To get a similarity score out of this, the distance is normalized between 0.0 and 1.0.

To combine these two scores into a single similarity score, both of the values are weighed and then added up ($\omega_0 \cdot D_{tree} + \omega_1 \cdot D_{name}$). If this score is over a configurable threshold, they are added to the relationship graph.

A heuristic is added to reduce the amount of computations and time needed by a very significant amount. When project trees contain a lot of files and are structurally very different from each other, calculating the *tree edit distance* becomes very expensive. The similarity is calculated only if the file count ratio between the two projects ($\frac{max_(n_a,n_b)}{min_(n_a,n_b)}$, n being the number of files in a project) is below a specified threshold (θ). The threshold ensures that the similarity computation is only performed for project pairs with a reasonable file count ratio, preventing unnecessary and computationally expensive comparisons for projects significantly different in size. The entire quick similarity algorithm can be seen in equation 6.2.1.

6.2.5 Simple Duplicate Deletion

This step is the first of two that deletes (or flags) duplicates, but it also reduces the search space of the next and last steps. This deletion is done using the *Git* history of

$$\text{Similarity}(n_a, n_b, D_{tree}, D_{name}, \theta) = \begin{cases} \omega_0 \cdot D_{tree} + \omega_1 \cdot D_{name} & \text{if } \frac{\max(n_a,n_b)}{\min(n_a,n_b)} < \theta \\ 0.0 & \text{otherwise} \end{cases} \tag{6.2.1}$$

where:

ω_0 = weight of the tree edit distance
D_{tree} = normalized tree edit distance
ω_1 = weight of the name edit distance
D_{name} = normalized name edit distance
n_a = number of files in project a
n_b = number of files in project b
θ = maximum project file ratio

a project. It compares a repository with its direct parent in the relationship graph. The only thing that it looks at is the commit history of both. If the latest commit ID of the child is present in the parent's history, then no changes have been made since the project was forked or cloned and pushed as a new project. It is then a duplicate and is deleted (or flagged), reducing the search space one last time.

6.2.6 Full Similarity Metrics Deletion

With the search space reduced as much as possible, a complete project comparison can now be conducted. The relationship graph is analyzed to identify pairs that require evaluation for duplicate identification. This graph contains all projects that come from others and, therefore, contains all potential duplicates. The pairs are then picked by looking at every child-parent pair in the graph.

The approach needs to be language-agnostic and fast. This rules out state-of-the-art techniques for software project comparison. Most of them are either language-specific (mainly *Java*) and/or not made to be used at this scale. The next best

technique that could be considered is an NLP approach. However, because the number of file comparisons that need to be done per project can be in the hundreds or thousands, depending on the project pair, this is not computationally feasible and adds a lot of complexity. It is then necessary to look at simpler methods to compare projects.

To simplify the problem, rather than comparing the entire project as a monolith, it is divided by comparing the similarity of each pair of files that share the same path in the two different projects. To avoid unnecessary computations, there is an option to compare only specific types of files. Then, the average of all the file pairs similarity scores can be calculated (see Eq. 6.2.2). If a file exists in one project and not the other, the similarity score for that file pair is 0.

$$Similarity = \frac{S_1 + S_2 + ... + S_n}{n} \tag{6.2.2}$$

where:

$$S_1 + S_2 + ... + S_n = \text{Sum of scores of } n \text{ files}$$

$$n = \text{Number of files}$$

A sequence matching algorithm is used to evaluate the similarity between the content of two files. More specifically, the Python standard library's "difflib.SequenceMatcher" is used; it is an improved version of the Gestalt Approach Pat (2023) by Ratcliff and Obershelp. This algorithm compares two sequences by recursively finding the longest common subsequence and assigning similarity scores based on the lengths of the common subsequences between the sequences. It has a linear

best-case complexity and a quadratic complexity for the worst-case. There are better algorithms for the use case, but given its speed and simplicity, it is a good compromise. Given better resources, an NLP approach could be considered and swapped, given the framework approach.

The source code implementing the approach described in this chapter is publicly available on GitHub[1].

6.3 Experiments

To validate this approach, this approach is applied unto the P4 ecosystem. This section is divided into the experimental setup (software and hardware), results, and analysis.

6.3.1 Experimental Setup

In terms of hardware, experiments were done on a computer running Linux, equipped with a 9th-gen Intel i5-9600K (6 cores, 6 threads) CPU and 32 GB of RAM. This is consumer hardware, and the results could be enhanced by improving the hardware to a more professional level. However, the impact would only be in terms of computation time.

For software, multiple choices were made. As the general implementation programming language, *Python* was used for its simplicity and useful standard library. *Javascript* was also used to query the *GitHub* since the language has libraries developed by *GitHub* themselves. As recommended, a graph database was chosen for the database. More specifically, *Neo4J* was chosen and this is for multiple reasons. Firstly,

[1] `https://github.com/AlexandreLachanceGit/git-deduplication`

it has a free-to-use community version that implements every feature needed. Secondly, it also implements many algorithms by default, like the Levenshtein distance, that would have required manual work to implement otherwise. Thirdly, the query language, *Cypher*, is straightforward to use and adapted to graph queries, making working with the graph simple. Lastly, it implements some visualization technologies by default, which are very useful for better understanding the data and showing how the approach is working.

6.3.2 Results and Analysis

Building the dataset

The dataset comprises 2610 repositories sourced from both *GitHub* and *GitLab*. For *GitHub*, 2529 repositories were found with a query for *P4* programming language projects. It was then discovered that these projects referenced 33 repositories that were absent as their parent, so those repositories were added. For *GitLab*, the API identified 72 repositories, but only a maximum of 48 could be retrieved before encountering an internal error that was difficult to debug. While *GitHub* projects vastly outnumber *GitLab* projects (see Fig. 6.4), this isn't a problem since they are both represented with the same schema in the dataset and won't be processed differently.

Fork Relationships

The *Forks* step in the pipeline created a total of 1600 relationships in the relationship graph, resulting in it containing 1864 connected nodes. This means that a total of 71.41% of the repository nodes were connected. It highlights how interconnected the *P4* programming community projects are.

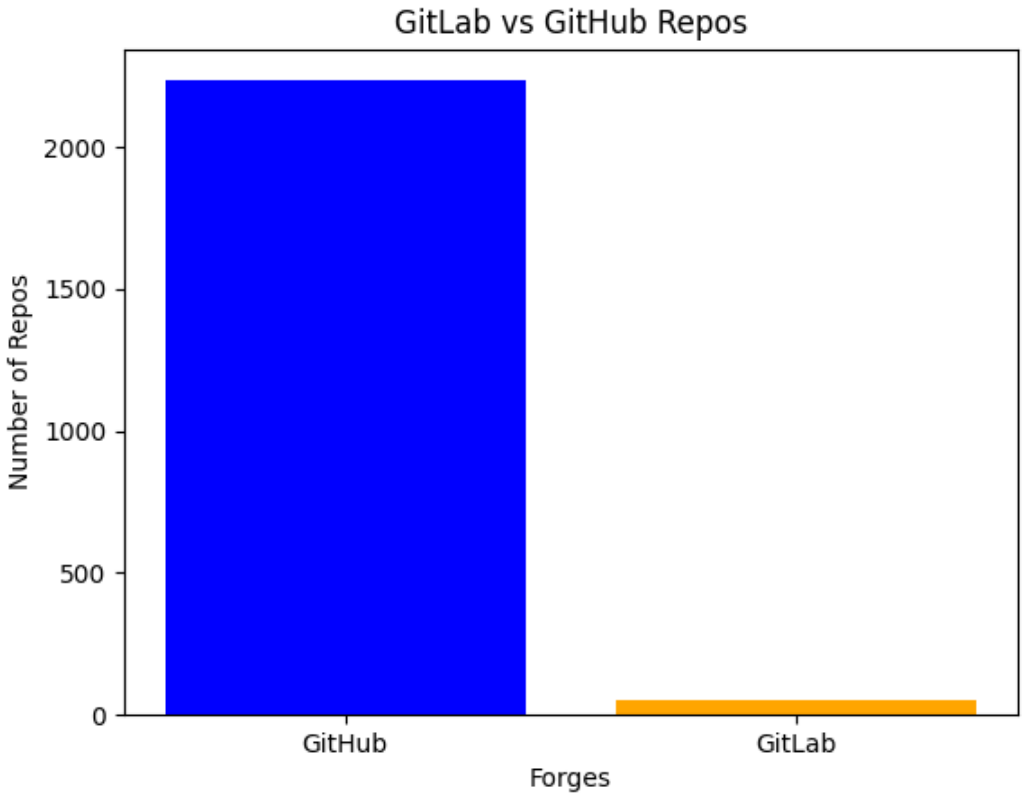


Figure 6.4: Graph of repository distribution across forges.

Git History Relationships

The *Git History* step in the pipeline added 71 new relationships in the graph, resulting in a total of 1904 connected nodes (72.95%). This step's execution time was only 7.5 seconds, which highlights its effectiveness and efficiency.

Quick Similarity Relationships

The *Quick Similarity* step in the pipeline created 9 new relationships, resulting in a total of 1913 connected nodes (73.30%). This step involved a substantial amount of computations. As shown in Fig. 6.5, the results follow something resembling an inverse-square relationship. This is in line with expectations. Given that most project pairs are likely to differ unless the dataset is saturated with duplicates, it is logical for the similarity scores to show a trend toward 0. The threshold at which a project was considered "similar" or a potential duplicate was set at 0.7, which resulted in 9 new relationships.

The problem with this step is the return on investment in computation time, as opposed to new relationships. Most relationships were already found efficiently using *Git* metadata in the previous steps. This step of the pipeline looks for outliers that do not have metadata linking them. This step, depending on the use case, could be skipped. It could also be improved by parallelizing the algorithm since this method of doing independent pair comparisons lends very well to parallelization.

Simple Duplicate Deletion

The *Simple Duplicate Deletion* step in the pipeline resulted in a total of 1412 projects flagged as duplicates. It involved comparing 1680 pairs *Git* history and the whole

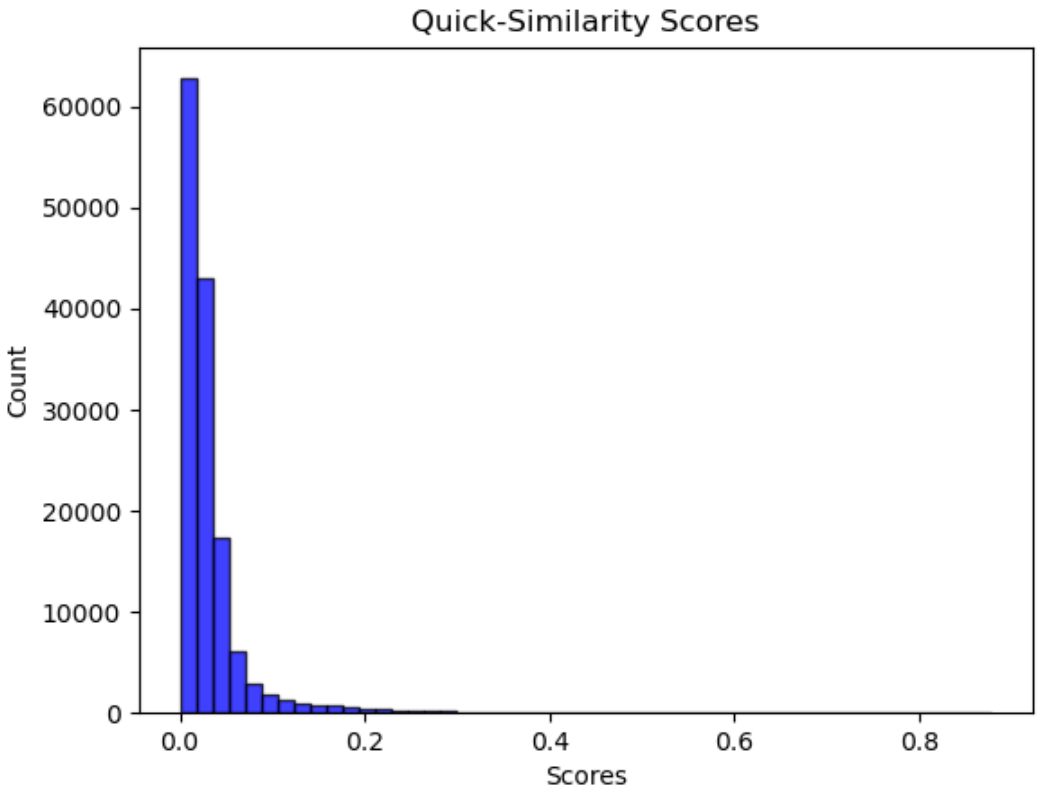


Figure 6.5: Graph of quick-similarity score distribution.

operation, a single *Cypher* query in the *Neo4j* database, was completed in only 87 milliseconds. This took down the final search space for the next step to 256 pairs.

Full Similarity Metrics Deletion

The final step in the pipeline found 115 new duplicate repositories in the dataset. This step's computation time is notably higher: 22 minutes to compare the 256 pairs left. It initially took a lot longer, but parallelizing the algorithm was straightforward, reducing the time to what it is now. The similarity score threshold was set at 0.75. This could be adapted based on the use case for the dataset (see Fig. 6.6). The system only compared files of type *Python*, *Bash*, and *P4* since those were the relevant plain text files. This saved computation by preventing the comparison of other file types like *PDF*s and *ZIP* archives.

Total

After all the steps of the pipeline, 1527 duplicates were found. This represents 58.51% of the original dataset, showing the prevalence of duplicate projects on *Git* and the need for this solution. Overall, the deduplication took around 25 minutes on the previously specified hardware for this relatively small dataset, ignoring the initial dataset collection. Depending on parameters and the use case, this approach could prove difficult to justify for larger datasets. The number of project pairs to compare will increase exponentially, even if the pipeline aims to reduce the search space as much as possible.

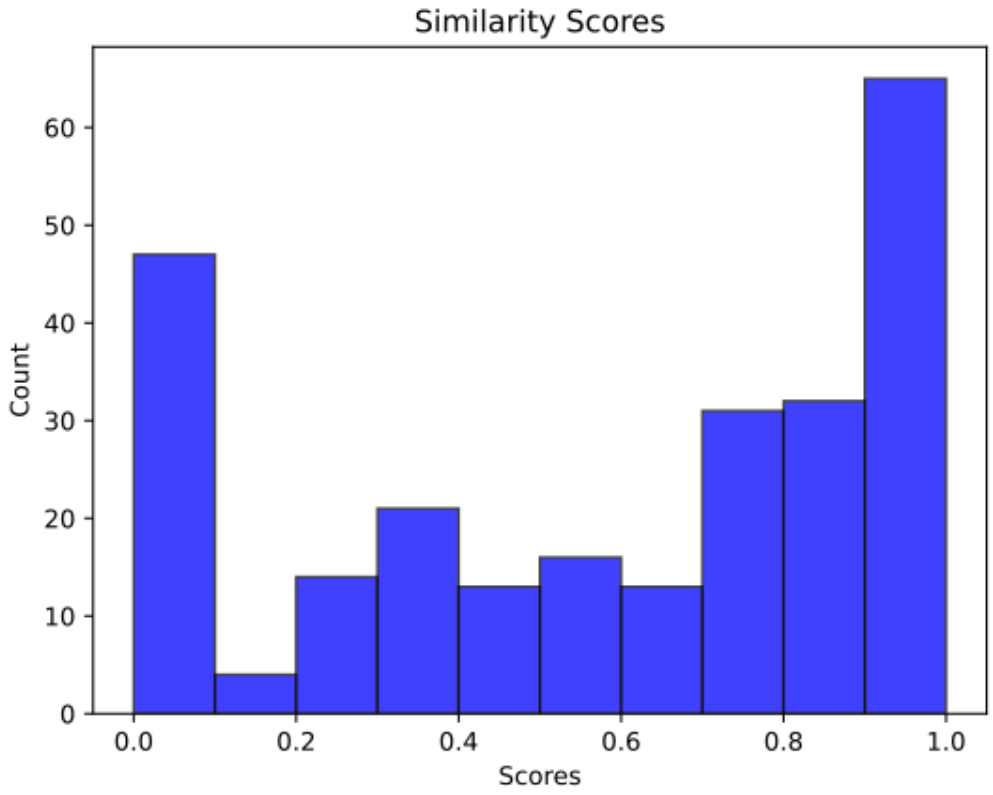


Figure 6.6: Graph of full-similarity score distribution.

6.3.3 Threats to validity

The validation of our results presents several challenges that may affect the validity of our findings.

Internal Validity

One significant issue lies in the scale and thoroughness of our approach compared to previous studies. Earlier research worked with datasets containing millions of repositories, while our analysis was limited to thousands. They were also not as thorough. For one reason or another, the datasets used in these earlier works are no longer available online, as all copies seem to have been deleted. This prevents us from directly comparing our findings against a well-established ground truth.

Construct Validity

Manually verifying false negatives or identifying duplicate projects that were missed is an overly time-consuming task to complete within a reasonable timeframe. This is because there would be a total of 6,812,100 pairs to verify, and even with random sampling, it would be tough to have a representative sample that a person can evaluate. However, while looking at the data, we found that projects that were nested inside a repository (i.e. `tutorial/` vs `mytutorial/tutorial/`), even if the content of both directories are the same, were not found as duplicates. Some false negatives are then to be expected. The next step involved focusing on identifying false positives, or repositories incorrectly flagged as duplicates. To achieve this, random samples of duplicate and original project pairs were taken, followed by a qualitative assessment to determine whether the projects were genuinely duplicates. A total of 50 project

pairs, out of 286, were evaluated qualitatively by examining project structure and file content. No false positives were found within the sample.

The similarity scores were found to accurately reflect the degree of similarity between the projects. Projects with scores near the threshold (0.75) exhibited more differences, while those closer to 1.0 were nearly identical. However, files present in only one project were overly weighted in the similarity results based on the current equation. This could be improved by weighting each similarity score according to the number of lines in the file before calculating the overall average.

External Validity

The external validity of our findings may be limited by the specific focus on the P4 programming language ecosystem. The high duplication rate observed in this context may not generalize to other domains or programming languages. Future studies involving a broader set of programming languages are necessary to evaluate the generalizability of our approach.

Conclusion Validity

Although our qualitative assessment found no false positives, the potential for undetected false negatives and the bias introduced by unverified project pairs could influence the overall reliability of our findings. Future work should explore more robust sampling methods and develop automated tools to improve the efficiency and accuracy of duplicate detection across larger datasets.

6.4 Conclusion

In this chapter, a multi-forge Git repository dataset deduplication framework was presented, designed to support the assessment of DSL tooling as they evolve. The key finding reveals that by leveraging various types of relationships between projects, such as forks, cloned and pushed projects, and downloaded and pushed projects, computational requirements for identifying duplicates can be significantly reduced. Specifically, the approach reduced the number of full project comparisons needed from over 6 million to just 256, ultimately identifying 1,527 duplicates, representing 58.51% of the original dataset.

This result underscores the high prevalence of duplicate projects in software repositories and highlights the importance of addressing this issue to ensure the accuracy of empirical research. Furthermore, the approach demonstrates that a scalable and efficient deduplication process is achievable and adaptable to various programming languages and contexts.

Chapter 7

Conclusion

This thesis addressed the significant challenges faced in developing editor support for DSLs, particularly BBDSLs, which often lack adequate tooling due to their specialized nature and limited resources. To overcome these challenges, the *Lever* framework was presented. *Lever* abstracts the complexity of building tooling for DSLs by leveraging the target DSL's existing artifacts and tools. Its rule-based system allows developers to annotate existing grammars with language-specific semantics, significantly reducing development effort while maintaining syntactic accuracy. Additionally, the plugin system facilitates seamless integration with existing tools, ensuring consistency between editor feedback and external outputs. This eliminates the need for manual reimplementation, further reducing the development effort. The applicability of the framework was demonstrated through a case study which applied it to a complex DSL called P4. Comparisons with existing solutions like Langium and MPS using the Protobuf DSL also highlighted *Lever* simplicity, flexibility, and ease of integration into a DSLs's existing ecosystem.

Furthermore, this thesis introduced a pipeline for creating high-quality, deduplicated datasets of DSL source code. This addressed the issue of code duplication in open-source repositories that could affect the results of empirical analysis conducted on them. By implementing a multi-forge collection and deduplication approach, the pipeline effectively reduces duplicates, thereby ensuring that the validation of language tooling is based on diverse and representative data. Applying this method to the P4 programming language ecosystem revealed a significant presence of duplicate repositories, which justified the need for this approach.

7.1 Future Work

The works of this thesis present many opportunities for future work. One immediate opportunity is to enhance the *Lever* framework itself. Currently, it is limited to the use of Tree-sitter grammars for parsing. Expanding the framework to support other parsing technologies (e.g., ANTLR, Bison) would increase its applicability to a broader range of DSLs without unnecessary translations. Additionally, while *Lever* implements a lot of the important features of the LSP, improving the automatic coverage of protocol would allow the framework to provide more value for the same development effort. It would also reduce the need for the manual implementation of features. Furthermore, developing a more expressive rule language outside of an off-the-shelf configuration language could be an avenue of improvement. Creating an independent and dedicated DSL for rule definition, if designed correctly, would enhance the readability of the rules and improve the ease of use of the framework, among other benefits.

Outside of these other opportunities, one of the biggest limitations of *Lever* is the

handling of metaprogramming. During compilation, metaprogramming can change the syntactic and semantic structure of the language dynamically. This poses a challenge for *Lever*, as it relies on static rules and existing grammar definitions. Consequently, handling features such as macros or preprocessor directives difficult. This limitation restricts its ability to fully support languages that make extensive use of metaprogramming features. It would be interesting to explore how this limitation could be lessened or even avoided in future works.

There are also many ways to further the work done on deduplication in Chapter 6. Firstly, the solution proposed used a very bare-bones technique to calculate the similarity of different projects as its last step. This isn't in line with the state-of-the-art on the subject as mentioned in the chapter's related works section. In future work and experiments, it would be interesting to attempt to deduplicate a dataset of *Java* repositories instead of a *P4* one. It would allow us to look at how the approach functions when better language tooling is available. Because of the framework approach and the fact that the rest of the deduplication pipeline is language-agnostic, it should be painless to swap out the current final similarity calculation for *CrossSim* (Nguyen et al., 2020) for example. Additionally, there is interesting new research being done on the subject of vectorizing repositories (Rokon et al., 2021). These recent advancements could be very helpful in coming up with a new language-agnostic method to evaluate the similarity between repositories. This approach takes the full semantics of a repository and represents it using a vector; it takes into account the associated metadata, the structure of the repository, and the entire source code (Rokon et al., 2021). Vector distance calculations would be a lot faster than the distance between raw file structures and file contents. It would however introduce more complexity to

the solution since these embeddings need to be trained. Depending on the computational overhead of vectorizing different repositories, the approach of significantly reducing the search space presented in this chapter could still be very relevant in this case. Our approach allows us to reduce the number of projects that need to be compared, therefore we would only need to vectorize these projects.

Appendix A

P4 *Lever* Rules

The *Lever* rules for P4.

```
LanguageDefinition (
    language: (
        name: "P4",
        file_extensions: ["p4", "P4"],
        library_paths: (
            env_variables: ["P4_LIBRARY_PATH"],
            linux: [],
            windows: [],
            macos: [],
        ),
    ),
    keywords: [
        "abstract",
        "action",
        "actions",
        "apply",
        "const",
        "control",
        "default",
        "define",
        "else",
        "entries",
        "enum",
        "error",
        "exit",
        "extern",
        "header",
```

```
        "header_union",
        "if",
        "include",
        "key",
        "match_kind",
        "type",
        "parser",
        "package",
        "pragma",
        "return",
        "select",
        "state",
        "struct",
        "switch",
        "table",
        "transition",
        "typedef",
        "varbit",
        "valueset",
    ],
    symbol_types: [
        (name: "Constant", completion_type: Constant, highlight_type: Variable),
        (name: "Variable", completion_type: Variable, highlight_type: Variable),
        (name: "Type", completion_type: Class, highlight_type: Type),
        (name: "Function", completion_type: Function, highlight_type: Function),
        (name: "Parameter", completion_type: Variable, highlight_type:
        ↪  Parameter),
        (name: "Field", completion_type: Property, highlight_type: Property),
        (name: "Table", completion_type: Class, highlight_type: Class),
    ],
    global_ast_rules: [
        (query: Kind("line_comment"), rule: Direct("Comment"), highlight_type:
        ↪  Comment),
        (query: Kind("block_comment"), rule: Direct("Comment"), highlight_type:
        ↪  Comment),
        (query: Kind("preproc_include_declaration"), rule: Rule("Import")),
    ],
    ast_rules: [
        Rule(
            node_name: "Root", // Name of Rule (required)
            is_scope: true, // defaults to false
            children: [
                (query: Kind("constant_declaration"), rule:
                ↪  Rule("ConstantDeclaration")),
                (query: Kind("parser_declaration"), rule: Rule("Parser")),
                (query: Kind("control_declaration"), rule: Rule("Control")),
                (query: Kind("instantiation"), rule: Rule("Instantiation")),
                (query: Kind("type_declaration"), rule: Rule("TypeDeclaration")),
```

```
            (query: Kind("action_declaration"), rule:
            ↪  Rule("ActionDeclaration")),
            (query: Kind("function_declaration"), rule:
            ↪  Rule("FunctionDeclaration")),
            (query: Kind("error_declaration"), rule:
            ↪  Rule("ErrorDeclaration")),
            (query: Kind("extern_declaration"), rule:
            ↪  Rule("ExternDeclaration")),
        ]
    ),
    Rule(
        node_name: "Import",
        children: [
            (query: Field("local_file"), rule: Rule("LocalImport")),
            (query: Field("library_file"), rule: Rule("LibraryImport")),
        ]
    ),
    Rule(
        node_name: "LibraryImport",
        import: Library
    ),
    Rule(
        node_name: "LocalImport",
        import: Local
    ),
    Rule(
        node_name: "ConstantDeclaration",
        symbol: Init(type: "Constant", name_node: "Name", type_node: "Type"),
        children: [
            (query: Field("annotation"), rule: Rule("Annotation")),
            (query: Field("type"), rule: Rule("Type")),
            (query: Field("name"), rule: Direct("Name")),
            (query: Field("value"), rule: Rule("Expression")),
        ]
    ),
    Rule(
        node_name: "Parser",
        symbol: Init(type: "Function", name_node: "Name"),
        is_scope: true,
        children: [
            (query: Field("annotation"), rule: Rule("Annotation")),
            (query: Path([Field("declaration"), Field("name")]), rule:
            ↪  Direct("Name")),
            (query: Path([Field("declaration"), Field("parameters")]), rule:
            ↪  Rule("Parameters")),
            (query: Field("body"), rule: Rule("Body")),
        ]
    ),
```

```
        Rule(
            node_name: "Control",
            symbol: Init(type: "Function", name_node: "Name"),
            is_scope: true,
            children: [
                (query: Field("annotation"), rule: Rule("Annotation")),
                (query: Path([Field("declaration"), Field("name")]), rule:
                ↪  Direct("Name")),
                (query: Path([Field("declaration"), Field("parameters")]), rule:
                ↪  Rule("Parameters")),
                (query: Field("body"), rule: Rule("Body")),
            ]
        ),
        Rule(
            node_name: "Instantiation",
            is_scope: true,
            children: [
                (query: Kind("annotation"), rule: Rule("Annotation")),
                (query: Kind("type_ref"), rule: Rule("Type")),
                (query: Kind("name"), rule: Direct("Name")),
                (query: Kind("argument_list"), rule: Rule("Args")),
            ]
        ),
        Rule(
            node_name: "TypeDeclaration",
            children: [
                (query: Kind("typedef_declaration"), rule:
                ↪  Rule("TypeDefDeclaration")),
                (query: Kind("header_type_declaration"), rule:
                ↪  Rule("HeaderTypeDeclaration")),
                (query: Kind("header_union_declaration"), rule:
                ↪  Rule("HeaderUnionDec")),
                (query: Kind("struct_type_declaration"), rule:
                ↪  Rule("StructDeclaration")),
                (query: Kind("enum_declaration"), rule: Rule("EnumDeclaration")),
                (query: Kind("parser_type_declaration"), rule:
                ↪  Rule("ParserTypeDeclaration")),
                (query: Kind("control_type_declaration"), rule:
                ↪  Rule("ControlTypeDeclaration")),
                (query: Kind("package_type_declaration"), rule:
                ↪  Rule("PackageTypeDeclaration")),
            ]
        ),
        Rule(
            node_name: "TypeDefDeclaration",
            symbol: Init(type: "Type", name_node: "Name", type_node: "Type"),
            is_scope: true,
            children: [
```

```
            (query: Field("annotation"), rule: Rule("Annotation")),
            (query: Field("type"), rule: Rule("Type")),
            (query: Field("name"), rule: Direct("Name")),
        ]
    ),
    Rule(
        node_name: "HeaderTypeDeclaration",
        symbol: Init(type: "Type", name_node: "Name"),
        is_scope: true,
        children: [
            (query: Field("annotation"), rule: Rule("Annotation")),
            (query: Field("name"), rule: Direct("Name")),
            (query: Field("field_list"), rule: Rule("FieldList"))
        ]
    ),
    Rule(
        node_name: "HeaderUnionDeclaration",
        symbol: Init(type: "Type", name_node: "Name"),
        is_scope: true,
        children: [
            (query: Field("annotation"), rule: Rule("Annotation")),
            (query: Field("name"), rule: Direct("Name")),
            (query: Field("field_list"), rule: Rule("FieldList"))
        ]
    ),
    Rule(
        node_name: "StructDeclaration",
        symbol: Init(type: "Type", name_node: "Name"),
        is_scope: true,
        children: [
            (query: Field("annotation"), rule: Rule("Annotation")),
            (query: Field("name"), rule: Direct("Name")),
            (query: Field("field_list"), rule: Rule("FieldList"))
        ]
    ),
    Rule(
        node_name: "EnumDeclaration",
        symbol: Init(type: "Type", name_node: "Name", type_node: "Type"),
        is_scope: true,
        children: [
            (query: Field("annotation"), rule: Rule("Annotation")),
            (query: Field("type"), rule: Rule("Type")),
            (query: Field("name"), rule: Direct("Name")),
            (query: Kind("identifier_list"), rule: Rule("OptionList")),
            (query: Kind("specified_identifier_list"), rule:
            ↪  Rule("SpecifiedOptionList"))
        ]
    ),
```

```
        Rule(
            node_name: "ParserTypeDeclaration",
            symbol: Init(type: "Function", name_node: "Name"),
            is_scope: true,
            children: [
                (query: Field("annotation"), rule: Rule("Annotation")),
                (query: Field("name"), rule: Direct("Name")),
                (query: Field("parameters"), rule: Rule("Parameters"))
            ]
        ),
        Rule(
            node_name: "ControlTypeDeclaration",
            symbol: Init(type: "Function", name_node: "Name"),
            is_scope: true,
            children: [
                (query: Field("annotation"), rule: Rule("Annotation")),
                (query: Field("name"), rule: Direct("Name")),
                (query: Field("parameters"), rule: Rule("Parameters"))
            ]
        ),
        Rule(
            node_name: "PackageTypeDeclaration",
            symbol: Init(type: "Function", name_node: "Name"),
            is_scope: true,
            children: [
                (query: Field("annotation"), rule: Rule("Annotation")),
                (query: Field("name"), rule: Direct("Name")),
                (query: Field("parameters"), rule: Rule("Parameters"))
            ]
        ),
        Rule(
            node_name: "FieldList",
            children: [
                (query: Kind("struct_field"), rule: Rule("Field"))
            ]
        ),
        Rule(
            node_name: "Field",
            symbol: Init(type: "Field", name_node: "Name", type_node: "Type"),
            children: [
                (query: Field("annotation"), rule: Rule("Annotation")),
                (query: Field("type"), rule: Rule("Type")),
                (query: Field("name"), rule: Direct("Name")),
            ]
        ),
        Rule(
            node_name: "OptionList",
            children: [
```

```
            (query: Kind("name"), rule: Direct("Option")),
        ]
    ),
    Rule(
        node_name: "SpecifiedOptionList",
        children: [
            (query: Kind("name"), rule: Direct("Option")),
        ]
    ),
    Rule(
        node_name: "SpecifiedOption",
        children: [
            (query: Kind("name"), rule: Direct("Option")),
            (query: Kind("initializer"), rule: Rule("Expression")),
        ]
    ),
    Rule(
        node_name: "ActionDeclaration",
        symbol: Init(type: "Function", name_node: "Name"),
        is_scope: true,
        children: [
            (query: Field("annotation"), rule: Rule("Annotation")),
            (query: Field("name"), rule: Direct("Name")),
            (query: Field("parameters"), rule: Rule("Parameters")),
            (query: Field("block"), rule: Rule("Block")),
        ],
    ),
    Rule(
        node_name: "FunctionDeclaration",
        symbol: Init(type: "Function", name_node: ["FunctionPrototype",
        ↪  "Name"]),
        is_scope: true,
        children: [
            (query: Kind("function_prototype"), rule:
            ↪  Rule("FunctionPrototype")),
            (query: Kind("block_statement"), rule: Rule("Block")),
        ]
    ),
    Rule(
        node_name: "ErrorDeclaration",
        is_scope: true,
        children: [
            (query: Field("option_list"), rule: Rule("Options")),
        ]
    ),
    Rule(
        node_name: "ExternDeclaration",
        symbol: Init(type: "Function", name_node: "Name"),
```

```
            is_scope: true,
            children: [
                (query: Field("annotation"), rule: Rule("Annotation")),
                (query: Kind("non_type_name"), rule: Direct("Name")),
                (query: Field("function"), rule: Rule("FunctionPrototype")),
                (query: Field("method"), rule: Rule("MethodList")),
            ]
        ),
        Rule(
            node_name: "FunctionPrototype",
            is_scope: true,
            children: [
                (query: Kind("type_or_void"), rule: Rule("Type")),
                (query: Kind("name"), rule: Direct("Name")),
                (query: Kind("parameter_list"), rule: Rule("Parameters")),
            ]
        ),
        Rule(
            node_name: "MethodList",
            children: [
                (query: Kind("method_prototype"), rule: Rule("MethodPrototype")),
            ]
        ),
        Rule(
            node_name: "MethodPrototype",
            is_scope: true,
            children: [
                (query: Field("annotation"), rule: Rule("Annotation")),
                (query: Field("function"), rule: Rule("FunctionPrototype")),
                (query: Field("type"), rule: Rule("Type")),
                (query: Field("parameters"), rule: Rule("Parameters")),
            ]
        ),
        Rule(
            node_name: "Options",
            children: [
                (query: Kind("name"), rule: Direct("Option")),
            ]
        ),
        Rule(
            node_name: "Args",
            children: [
                (query: Kind("argument"), rule: Rule("Arg")),
            ]
        ),
        Rule(
            node_name: "Arg",
            children: [
```

```
            (query: Kind("expression"), rule: Rule("Expression")),
        ]
    ),
    Rule(
        node_name: "Parameters",
        children: [
            (query: Kind("parameter"), rule: Rule("Parameter")),
        ]
    ),
    Rule(
        node_name: "Parameter",
        symbol: Init(type: "Parameter", name_node: "Name", type_node:
        ↪  "Type"),
        children: [
            (query: Field("direction"), rule: Direct("Direction"),
            ↪  highlight_type: EnumMember),
            (query: Field("type"), rule: Rule("Type")),
            (query: Field("name"), rule: Direct("Name")),
        ]
    ),
    Rule(
        node_name: "Body",
        is_scope: true,
        children: [
            (query: Kind("constant_declaration"), rule:
            ↪  Rule("ConstantDeclaration")),
            (query: Kind("variable_declaration"), rule:
            ↪  Rule("VariableDeclaration")),
            (query: Kind("instantiation"), rule: Rule("Instantiation")),
            (query: Kind("value_set_declaration"), rule:
            ↪  Rule("ValueSetDeclaration")),

            (query: Kind("parser_state"), rule: Rule("ParserState")),
            (query: Kind("action_declaration"), rule:
            ↪  Rule("ActionDeclaration")),
            (query: Kind("table_declaration"), rule: Rule("ControlTable")),
            (query: Kind("block_statement"), rule: Rule("Block")),
        ]
    ),
    Rule(
        node_name: "Block",
        is_scope: true,
        children: [
            (query: Kind("constant_declaration"), rule:
            ↪  Rule("ConstantDeclaration")),
            (query: Kind("variable_declaration"), rule:
            ↪  Rule("VariableDeclaration")),
```

```
            (query: Kind("assignment_or_method_call_statement"), rule:
            ↪  Rule("AssignmentOrMethodCall")),
            (query: Kind("direct_application"), rule:
            ↪  Rule("DirectApplication")),
            (query: Kind("conditional_statement"), rule:
            ↪  Rule("Conditional")),
            (query: Kind("empty_statement"), rule: Direct("EmptyStatement")),
            (query: Kind("block_statement"), rule: Rule("Block")),
            (query: Kind("parser_block_statement"), rule: Rule("Block")),
            (query: Kind("exit_statement"), rule: Direct("ExitStatement")),
            (query: Kind("return_statement"), rule: Rule("Return")),
            (query: Kind("switch_statement"), rule: Rule("Switch")),
            (query: Kind("transition_statement"), rule: Rule("Transition")),
        ]
    ),
    Rule(
        node_name: "Transition",
        children: [
            (query: Kind("select_expression"), rule: Rule("Select")),
        ]
    ),
    Rule(
        node_name: "Select",
        children: [
            (query: Path([Kind("select_expression_params"),
            ↪  Kind("expression_list")]), rule: Rule("ValueList")),
            (query: Path([Kind("select_expression_body"),
            ↪  Kind("select_case_list")]), rule: Rule("SelectCaseList")),
        ]
    ),
    Rule(
        node_name: "SelectCaseList",
        children: [
            (query: Kind("select_case"), rule: Rule("SelectCase")),
        ]
    ),
    Rule(
        node_name: "SelectCase",
        children: [
            (query: Path([Kind("simple_keyset_expression"),
            ↪  Kind("expression")]), rule: Rule("Expression")),
            (query: Field("name"), rule: Rule("NameUsage")),
        ]
    ),
    Rule(
        node_name: "VariableDeclaration",
        symbol: Init(type: "Variable", name_node: "Name", type_node: "Type"),
        children: [
```

```
            (query: Field("annotation"), rule: Rule("Annotation")),
            (query: Field("type"), rule: Rule("Type")),
            (query: Field("name"), rule: Direct("Name")),
            (query: Field("value"), rule: Rule("Expression")),
        ]
    ),
    Rule(
        node_name: "Conditional",
        children: [
            (query: Field("expression"), rule: Rule("Expression")),
            (query: Field("bodyIf"), rule: Rule("Block")),
            (query: Field("bodyElse"), rule: Rule("Block")),
        ]
    ),
    Rule(
        node_name: "Switch",
        children: [
            (query: Field("expression"), rule: Rule("Expression")),
            (query: Path([Field("body"), Field("switch_case")]), rule:
            ↪  Rule("SwitchCase")),
        ]
    ),
    Rule(
        node_name: "SwitchCase",
        children: [
            (query: Field("name"), rule: Rule("Expression")),
            (query: Field("value"), rule: Rule("Block")),
        ]
    ),
    Rule(
        node_name: "AssignmentOrMethodCall",
        children: [
            (query: Field("name"), rule: Direct("Name")),
            (query: Field("expression"), rule: Rule("Expression")),
            (query: Field("parameters"), rule: Rule("Args")),
        ]
    ),
    Rule(
        node_name: "ControlTable",
        symbol: Init(type: "Table", name_node: "Name"),
        children: [
            (query: Field("annotation"), rule: Rule("Annotation")),
            (query: Field("name"), rule: Direct("Name")),
            (query: Field("table"), rule: Rule("Table")),
        ]
    ),
    Rule(
        node_name: "Table",
```

```
            children: [
                (query: Path([Kind("keys_table"), Field("keys")]), rule:
                ↪  Rule("KeyElementList")),
                (query: Path([Kind("action_table"), Field("actions")]), rule:
                ↪  Rule("ActionList")),
                (query: Kind("name_table"), rule: Rule("NameTable")),
            ]
        ),
        Rule(
            node_name: "KeyElementList",
            children: [
                (query: Kind("key_element"), rule: Rule("KeyElement")),
            ]
        ),
        Rule(
            node_name: "KeyElement",
            children: [
                (query: Field("expression"), rule: Rule("Expression")),
                (query: Field("name"), rule: Direct("Name")),
            ]
        ),
        Rule(
            node_name: "ActionList",
            children: [
                (query: Kind("action"), rule: Rule("Action")),
            ]
        ),
        Rule(
            node_name: "Action",
            children: [
                (query: Kind("prefixed_non_type_name"), rule: Rule("NameUsage")),
            ]
        ),
        Rule(
            node_name: "NameTable",
            children: [
                (query: Field("name"), rule: Direct("Name")),
                (query: Field("expression"), rule: Rule("Expression")),
            ]
        ),
        Rule(
            node_name: "ParserState",
            children: [
                (query: Field("annotation"), rule: Rule("Annotation")),
                (query: Field("name"), rule: Direct("Name")),
                (query: Field("body"), rule: Rule("Block")),
            ]
        ),
```

```
        Rule(
            node_name: "ValueSetDeclaration",
            children: [
                (query: Field("annotation"), rule: Rule("Annotation")),
                (query: Field("type"), rule: Rule("Type")),
                (query: Field("name"), rule: Direct("Name")),
                (query: Field("expression"), rule: Rule("Expression")),
            ]
        ),
        Rule(
            node_name: "DirectApplication",
            children: [
                (query: Field("name"), rule: Rule("NameUsage")),
                (query: Field("specialized"), rule: Direct("Name")),

                (query: Field("args"), rule: Rule("Args")),
            ]
        ),
        Rule(
            node_name: "Return",
            children: [
                (query: Field("value"), rule: Rule("Expression")),
            ]
        ),
        Rule(
            node_name: "TypeArgumentList",
            children: [
                (query: Kind("type_arg"), rule: Rule("Type")),
            ]
        ),
        Rule(
            node_name: "Type",
            symbol: Usage,
        ),
        Rule(
            node_name: "NameUsage",
            symbol: Usage
        ),
        Rule(
            node_name: "Member",
            symbol: MemberUsage,
        ),
        Rule(
            node_name: "ValueList",
            children: [
                (query: Kind("expression"), rule: Rule("Expression")),
            ]
        ),
```

```
        Rule(
            node_name: "Expression",
            symbol: Expression,
            children: [
                (query: Kind("integer"), rule: Direct("Integer"), highlight_type:
                ↪  Number),
                (query: Kind("string"), rule: Direct("String"), highlight_type:
                ↪  String),
                (query: Kind("bool"), rule: Direct("Bool"), highlight_type:
                ↪  EnumMember),
                (query: Kind("non_type_name"), rule: Rule("NameUsage")),
                (query: Kind("named_type"), rule: Rule("NameUsage")),
                (query: Kind("type_name"), rule: Rule("NameUsage")),
                (query: Kind("expression"), rule: Rule("Expression")),
                (query: Kind("member"), rule: Rule("Member")),
                (query: Field("operator"), rule: Direct("Operator")),
            ]
        ),
        Rule(
            node_name: "Annotation",
        )
    ]
)
```

Appendix B

P4Test Plugin

The Rust code for the plugin using the P4Test utility.

```
use lsp_types::{Diagnostic, DiagnosticSeverity, Position, Range};
use regex::Regex;
use serde::Serialize;
use std::env;
use std::process::Command;

#[derive(Serialize)]
struct Output {
    output_type: String,
    data: String,
}

fn main() {
    let args: Vec<String> = env::args().collect();

    let out = if let Some(output) = get_result(args[1].to_string()) {
        serde_json::to_string(&output).unwrap()
    } else {
        serde_json::to_string(&Output {
            output_type: "Nothing".to_string(),
            data: String::new(),
        })
        .unwrap()
    };

    println!("{out}")
}

fn get_result(path: String) -> Option<Output> {
    let p4test_output = run_executable(&path)?;
```

```
    let diags = parse_output(p4test_output, path)?;

    Some(Output {
        output_type: "Diagnostic".to_string(),
        data: serde_json::to_string(&diags).ok()?,
    })
}

fn parse_output(message: String, file_path: String) -> Option<Vec<Diagnostic>> {
    // Parse and remove line number
    let line_nb_re = Regex::new(format!(r"{}\((\d+)\):?",
    ↪  file_path).as_str()).unwrap();
    let captures = line_nb_re.captures(&message)?;
    let line_nb = captures.get(1)?.as_str().parse::<u32>().ok()? - 1;
    let current_msg = line_nb_re.replace(&message, "");

    let kind_re = Regex::new(r"\[--W(.*)=(.*)\]").unwrap();
    let captures = kind_re.captures(&current_msg);

    // Parse and remove severity and kind
    let (severity, kind) = if let Some(captures) = captures {
        let severity_capture = captures.get(1);
        let severity = if let Some(cap) = severity_capture {
            match cap.as_str() {
                "error" => DiagnosticSeverity::ERROR,
                "warn" => DiagnosticSeverity::WARNING,
                _ => DiagnosticSeverity::ERROR,
            }
        } else {
            DiagnosticSeverity::ERROR
        };

        let kind_cap = captures.get(2);
        let kind = if let Some(cap) = kind_cap {
            cap.as_str()
        } else {
            ""
        };

        (severity, kind)
    } else {
        (DiagnosticSeverity::ERROR, "")
    };
    let current_msg = kind_re.replace(&current_msg, "");

    // Make and return diagnostic
    let lines: Vec<&str> = current_msg.trim().lines().collect();
```

```
    let diag_msg = lines[0].replace("error:", "").replace("warning:", "");
    let diag_range = get_range(line_nb, lines[2]);

    Some(vec![Diagnostic::new(
        diag_range,
        Some(severity),
        Some(lsp_types::NumberOrString::String(kind.to_string())),
        Some("p4test".to_string()),
        diag_msg.trim().to_string(),
        None,
        None,
    )])
}

fn get_range(line_nb: u32, arrows: &str) -> Range {
    let mut start: u32 = 0;

    for char in arrows.chars() {
        if char == ' ' {
            start += 1;
        } else {
            break;
        }
    }

    Range::new(
        Position::new(line_nb, start),
        Position::new(line_nb, arrows.len() as u32),
    )
}

fn run_executable(path: &str) -> Option<String> {
    let output = Command::new("p4test").args(vec![path]).output().ok()?;

    if !output.status.success() {
        Some(String::from_utf8(output.stderr).ok()?)
    } else {
        None
    }
}
```

Appendix C

Protobuf Protoc Plugin

The Python code for the plugin using the `protoc` compiler.

```
#!/bin/python3

import sys
import subprocess
import json
import re

PARSING_REGEX = r"^(.+):(\d+):(\d+): (.+)$"


def run_protoc(file_path):
    result = subprocess.run(
        ["protoc", "-o", "/dev/null", file_path], capture_output=True, text=True
    )

    if result.returncode != 0:
        errors_str = result.stderr.split("\n")[:-1]

    return errors_str


def parse_error(error_str):
    s = re.search(PARSING_REGEX, error_str)
    line = int(s.group(2))
    return {
        "source": "protoc",
        "range": {
            "start": {
                "line": line,
                "character": 0,
```

```
            },
            "end": {
                "line": line + 1,
                "character": 0,
            },
        },
        "severity": 1,
        "message": s.group(4),
    }


def main():
    errors_str = run_protoc(sys.argv[1])
    errors = list(map(parse_error, errors_str))
    print(json.dumps({"output_type": "Diagnostic", "data": errors}))


if __name__ == "__main__":
    main()
```

Bibliography

Git - Git Objects, Dec. 2023. URL https://git-scm.com/book/en/v2/Git-Internals-Git-Objects. [Online; accessed 14. Dec. 2023].

Pattern Matching: the Gestalt Approach, Dec. 2023. URL https://www.drdobbs.com/database/pattern-matching-the-gestalt-approach/184407970?pgno=5. [Online; accessed 14. Dec. 2023].

A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Always Learning. Pearson, Harlow, second edition, pearson new international edition edition, 2014. ISBN 978-1-292-02434-9.

M. Allamanis. The adverse effects of code duplication in machine learning models of code. In *Onward! 2019: Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153. Association for Computing Machinery, New York, NY, USA, Oct. 2019. ISBN 978-1-45036995-4. doi: 10.1145/3359591.3359735.

C. Constantinides, T. Skotiniotis, and M. Störzer. AOP considered harmful. In *European Interactive Workshop on Aspects in Software*, Berlin, Germany, Sept. 2004.

K. Czarnecki. Overview of Generative Software Development. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, editors, *Unconventional Programming Paradigms*, volume 3566, pages 326–341. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-27884-9 978-3-540-31482-0. doi: 10.1007/11527800_25.

K. Czarnecki, K. Østerbye, and M. Völter. Generative Programming. In G. Goos, J. Hartmanis, J. van Leeuwen, J. Hernández, and A. Moreira, editors, *Object-Oriented Technology ECOOP 2002 Workshop Reader*, volume 2548, pages 15–29. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-00233-8 978-3-540-36208-1. doi: 10.1007/3-540-36208-8_2.

S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning. The State of the Art in Language Workbenches. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, M. Erwig, R. F. Paige, and E. Van Wyk, editors, *Software Language Engineering*, volume 8225, pages 197–217. Springer International Publishing, Cham, 2013. ISBN 978-3-319-02653-4 978-3-319-02654-1. doi: 10.1007/978-3-319-02654-1_11.

M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages?

https://martinfowler.com/articles/languageWorkbench.html, Dec. 2005.

Google. Protocol Buffers Documentation. https://protobuf.dev/, 2024.

B. Gyawali, L. Anastasiou, and P. Knoth. Deduplication of Scholarly Documents using Locality Sensitive Hashing and Word Embeddings. *European Language Resources Association*, May 2020. URL `https://oro.open.ac.uk/70519`.

D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10):64–72, Oct. 2004. ISSN 0018-9162. doi: 10.1109/MC.2004.172.

P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, Dec. 1996. ISSN 0360-0300, 1557-7341. doi: 10.1145/242224.242477.

JetBrains. How Does MPS Work? - Concepts | MPS by JetBrains, July 2024. URL `https://www.jetbrains.com/mps/concepts`. [Online; accessed 5. Jul. 2024].

J. Kjær Rask, F. Palludan Madsen, N. Battle, H. Daniel Macedo, and P. Gorm Larsen. The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions. *Electronic Proceedings in Theoretical Computer Science*, 338:3–18, Aug. 2021. ISSN 2075-2180. doi: 10.4204/EPTCS.338.3.

T. Kosar, S. Bohra, and M. Mernik. Domain-Specific Languages: A Systematic Mapping Study. *Information and Software Technology*, 71:77–91, Mar. 2016. ISSN 09505849. doi: 10.1016/j.infsof.2015.11.001.

Langium. Langium Reference Documentation, July 2024. URL `https://langium.org/docs/reference`. [Online; accessed 2. Aug. 2024].

C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. DéjàVu: a map of code duplicates on GitHub. *Proc. ACM Program. Lang.*, 1 (OOPSLA):1–28, Oct. 2017. ISSN 2475-1421. doi: 10.1145/3133908.

P. Mayer and A. Bauer. An empirical analysis of the utilization of multiple programming languages in open source projects. In *EASE '15: Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–10. Association for Computing Machinery, New York, NY, USA, Apr. 2015. ISBN 978-1-45033350-4. doi: 10.1145/2745802.2745805.

D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, and B. Baudry. Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review. *Computer Languages, Systems & Structures*, 46: 206–235, Nov. 2016. ISSN 14778424. doi: 10.1016/j.cl.2016.09.004.

M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, Dec. 2005. ISSN 0360-0300, 1557-7341. doi: 10.1145/1118890.1118892.

Microsoft. Tools supporting the LSP, Aug. 2024. URL `https://microsoft.github.io/language-server-protocol/implementors/tools`. [Online; accessed 29. Aug. 2024].

MPS. MPS User's Guide, July 2024. URL `https://www.jetbrains.com/help/mps/mps-user-s-guide.html`. [Online; accessed 19. Sep. 2024].

M. S. Naveed. Correlation Between GitHub Stars and Code Vulnerabilities. *JCBI*, 4 (01):141–151, Dec. 2022. ISSN 2710-1614. doi: 10.56979/401/2022/111.

P. T. Nguyen, J. Di Rocco, R. Rubei, and D. Di Ruscio. An automated approach to assess the similarity of GitHub repositories. *Software Qual. J.*, 28(2):595–631, June 2020. ISSN 1573-1367. doi: 10.1007/s11219-019-09483-0.

R. Nystrom. *Crafting Interpreters*. Genever Benning, 2021.

P4. P4˜16˜ Language Specification - Appendix: P4 grammar, Sept. 2022. URL `https://p4.org/p4-spec/docs/P4-16-v-1.2.3.html#sec-grammar`. [Online; accessed 13. Sep. 2024].

P4. P4 – Language Consortium, Sept. 2024a. URL `https://p4.org`. [Online; accessed 5. Sep. 2024].

P4. p4c/backends/p4test/README.md at main · p4lang/p4c, Sept. 2024b. URL `https://github.com/p4lang/p4c/blob/main/backends/p4test/README.md`. [Online; accessed 17. Sep. 2024].

M. Pawlik and N. Augsten. Efficient computation of the tree edit distance. *ACM Transactions on Database Systems (TODS)*, pages 3:1–3:40, 2015. doi: 10.1145/2699485.

M. Pawlik and N. Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, 2016. ISSN 0306-4379. doi: 10.1016/j.is.2015.08.004.

J. Petzold, J. Kreiß, and R. von Hanxleden. PASTA: Pragmatic Automated System-Theoretic Process Analysis. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 559–567, Porto, Portugal, June 2023. IEEE. ISBN 9798350347937. doi: 10.1109/DSN58367.2023.00058.

G. Popov, J. Lu, and V. Vishnyakov. Toward Extensible Low-Code Development Platforms. In *Advances in Emerging Information and Communication Technology*, pages 487–497. Springer, Cham, Switzerland, May 2024. ISBN 978-3-031-53237-5. doi: 10.1007/978-3-031-53237-5_29.

Protobuf. Protocol Buffers, Aug. 2024. URL `https://protobuf.dev`. [Online; accessed 27. Aug. 2024].

M. O. F. Rokon, P. Yan, R. Islam, and M. Faloutsos. Repo2vec: A comprehensive embedding approach for determining repository similarity. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 355–365, 2021. doi: 10.1109/ICSME52107.2021.00038.

Rust. LSP Extensions, July 2024. URL `https://github.com/rust-lang/rust-analyzer/blob/master/docs/dev/lsp-extensions.md`. [Online; accessed 16. Jul. 2024].

D. Spinellis, Z. Kotti, and A. Mockus. A Dataset for GitHub Repository Deduplication. In *MSR '20: Proceedings of the 17th International Conference on Mining Software Repositories*, pages 523–527. Association for Computing Machinery, New York, NY, USA, June 2020. ISBN 978-1-45037517-7. doi: 10.1145/3379597.3387496.

M. Spönemann. Langium 1.0: A mature language toolkit, Dec. 2022. URL `https://www.typefox.io/blog/langium-1.0-a-mature-language-toolkit`. [Online; accessed 8. Jul. 2024].

B. Steffen, F. Gossen, S. Naujokat, and T. Margaria. Language-Driven Engineering:

From General-Purpose to Purpose-Specific Languages. In B. Steffen and G. Woeginger, editors, *Computing and Software Science*, volume 10000, pages 311–344. Springer International Publishing, Cham, 2019. ISBN 978-3-319-91907-2 978-3-319-91908-9. doi: 10.1007/978-3-319-91908-9_17.

B. Steffen, T. Margaria, A. Bainczyk, S. Boßelmann, D. Busch, M. Driessen, M. Frohme, F. Howar, S. Jörges, M. Krause, M. Krumrey, A.-L. Lamprecht, M. Lybecait, A. Murtovi, S. Naujokat, J. Neubauer, A. Schieweck, J. Schürmann, S. Smyth, B. Steffen, F. Storek, T. Tegeler, S. Teumert, D. Wirkner, and P. Zweihoff. Language-Driven Engineering An Interdisciplinary Software Development Paradigm, Feb. 2024.

J.-P. Tolvanen, R. Pohjonen, and S. Kelly. Advanced tooling for domain-specific modeling: MetaEdit+. *ResearchGate*, Jan. 2007. URL `https://www.researchgate.net/publication/229021354_Advanced_tooling_for_domain-specific_modeling_MetaEdit`.

TypeFox. Langium SQL, July 2024. URL `https://github.com/TypeFox/langium-sql`. [Online; accessed 31. Jul. 2024].

M. Voelter. *Designing, Implementing and Using Domain-Specific Languages*. n/a, 2013.

M. Voelter. *Generic Tools, Specific Languages*. s.n., S.l., 2014. ISBN 978-94-6203-586-7.

M. Voelter and S. Lisson. Supporting diverse notations in MPS' projectional editor. *ResearchGate*, 1236:7–16, Jan. 2014a. URL `https://www.researchgate.net/pub`

lication/287089192_Supporting_diverse_notations_in_MPS%27_projectional_editor.

M. Voelter and S. Lisson. Supporting Diverse Notations in MPS' Projectional Editor. *CEUR Workshop Proceedings*, 2014b.

M. Voelter and V. Pech. Language modularity with the MPS language workbench. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1449–1450, Zurich, June 2012. IEEE. ISBN 978-1-4673-1066-6 978-1-4673-1067-3. doi: 10.1109/ICSE.2012.6227070.

M. Voelter, B. Kolb, T. Szabó, D. Ratiu, and A. van Deursen. Lessons learned from developing mbeddr: A case study in language engineering with MPS. *Software & Systems Modeling*, 18(1):585–630, Feb. 2019. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-016-0575-4.

D. S. Wile. Abstract Syntax from Concrete Syntax. In *Proceedings of the 19th International Conference on Software Engineering*, pages 472–480, 1997. doi: 10.1145/253228.253388.

M. Woodward. Octoverse 2022: 10 years of tracking open source. *GitHub Blog*, Nov. 2022. URL https://github.blog/2022-11-17-octoverse-2022-10-years-of-tracking-open-source.

Y. Zhao, L. Li, H. Wang, H. Cai, T. F. Bissyandé, J. Klein, and J. Grundy. On the Impact of Sample Duplication in Machine-Learning-Based Android Malware Detection. *ACM Trans. Software Eng. Method.*, 30(3):1–38, May 2021. ISSN 1049-331X. doi: 10.1145/3446905.