TÉCNICO LISBOA

A New Language Server for the ECMAScript Specification Language

Ricky Xu

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. José Faustino Fragoso Femenin dos Santos

Examination Committee

Chairperson: Prof. Miguel Ângelo Marques de Matos
Supervisor: Prof. José Faustino Fragoso Femenin dos Santos
Member of the Committee: Prof. António José dos Reis Morgado

November 2024

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to start by thanking the teacher José Fragoso, his guidance and insights were very crucial for me to complete this thesis.

I would like to thank my family for always supporting me.

I would also like to thank my friends for all the experiences we shared together.

Abstract

JavaScript is one of the most popular programming languages today and has become the de facto standard for programming client-side web applications, supported by all browsers. To ensure that all browsers implement JavaScript consistently, the JavaScript committee established the ECMAScript Standard, a specification of the JavaScript syntax and semantics written in English. Over the years, the ECMAScript Standard has become increasingly complex and hard to modify. In order to support the management and extension of the ECMAScript Standard, the ECMA-SL project introduced a formal specification language that mirrors its informal meta-language. Using this language, the ECMA-SL team developed reference implementations of various versions of the ECMAScript Standard. However, its lack of adequate editor support makes using ECMA-SL extremely challenging. To address this issue, we developed ESL-LS, a language server for the ECMA-SL language that provides various features to support coding in ECMA-SL, including error diagnostics, go-to-definition, and automatic code completion. By using the Language Server Protocol, our language server remains independent of any specific IDE, thereby extending ECMA-SL support to many IDEs. Our evaluation demonstrates that the language server performs well, with response times that are imperceptible to users, and is useful, having received excellent feedback from the few existing ECMA-SL developers.

Keywords

ECMA-SL; Language Server; Features; Language Server Protocol

Resumo

O JavaScript é uma das linguagens de programação mais populares atualmente e tornou-se o padrão para programação de aplicações web no lado do cliente, sendo suportado por todos os navegadores. Para garantir que todos os navegadores implementem o JavaScript de forma consistente, o comité do JavaScript estabeleceu o ECMAScript Standard, uma especificação da sintaxe e semântica do JavaScript escrita em inglês. Ao longo dos anos, o ECMAScript Standard tornou-se cada vez mais complexo e difícil de modificar. Para apoiar a gestão e extensão do ECMAScript Standard, o projeto ECMA-SL introduziu uma linguagem de especificação formal que é igual à sua meta-linguagem informal. Usando esta linguagem, a equipa ECMA-SL desenvolveu implementações de referência de várias versões do ECMAScript Standard. No entanto, a falta de suporte adequado para editores torna o uso do ECMA-SL extremamente desafiador. Para resolver este problema, desenvolvemos o ESL-LS, um language server para a linguagem ECMA-SL que oferece várias funcionalidades para suportar a programação em ECMA-SL, incluindo diagnósticos de erros, navegação para a definição de elementos e preenchimento automático de código. Ao utilizar o Language Server Protocol, o nosso language server mantém-se independente de qualquer IDE específico, estendendo assim o suporte ao ECMA-SL a vários IDEs. A nossa avaliação demonstra que o language server tem um bom desempenho, com tempos de resposta impercetíveis para os utilizadores, e é útil, tendo recebido um excelente feedback dos poucos programadores de ECMA-SL existentes.

Palavras Chave

ECMA-SL; Language Server; Funcionalidades; Language Server Protocol

Contents

1 Introduction **1**

2 Background **5**

2.1 The ECMA-SL Language . . . 5

2.1.1 ECMAScript . . . 5

2.1.2 ECMA-SL Project . . . 7

2.1.2.A ECMA-SL Advantages . . . 7

2.1.2.B ECMA-SL Language . . . 7

2.1.2.C Execution Pipeline . . . 8

2.1.3 ECMA-SL Limitations . . . 9

2.2 An Overview of Language Servers . . . 10

2.2.1 Language Server Features . . . 10

2.2.2 Key Components of the Language Server . . . 12

2.2.3 Types of Messages . . . 12

2.2.4 Workflow . . . 14

3 Related Work **17**

3.1 Evolution of Language Servers: A Historical Overview . . . 17

3.2 Program Assisting Tools . . . 18

3.3 IDE Integration of Program Analysis Tools . . . 20

4 A Toy Language Server **23**

4.1 The Read/Write Language . . . 24

4.2 Syntax Highlighting for the Read/Write Language . . . 24

4.3 A Language Server for the Read/Write Language . . . 27

4.3.1 Language Server Architecture . . . 27

4.3.1.A Editor . . . 27

4.3.1.B Workspace . . . 28

4.3.1.C Language Server . . . 28

4.3.1.D Language Server Protocol . . . 29

4.3.2 Server-Side Program Representation . . . 29
4.3.2.A Text Documents . . . 29
4.3.2.B Function Declarations . . . 29
4.3.3 Supported Features . . . 30
4.3.3.A Code Completion . . . 31
4.3.3.B Go-to-definition . . . 31
4.3.3.C Diagnostics . . . 33
4.3.3.D Hover . . . 34

5 ECMA-SL Language Server **37**
5.1 Language Configuration and Syntax Highlighting . . . 38
5.1.1 Language Configuration . . . 38
5.1.2 Syntax Highlighting . . . 39
5.2 Language Server Architecture . . . 44
5.3 Server-Side Program Representation . . . 45
5.3.1 Variable Declarations . . . 45
5.4 Supported Features . . . 46
5.4.1 Code Completion . . . 46
5.4.2 Hover . . . 47
5.4.3 Go-to-definition . . . 49
5.4.4 Diagnostics . . . 50

6 Evaluation **53**
6.1 Performance Evaluation . . . 54
6.1.1 Experiment . . . 54
6.1.2 Results . . . 56
6.1.3 Analysis . . . 56
6.2 User Study . . . 57
6.2.1 Experiment . . . 57
6.2.2 Results . . . 58
6.2.2.A Specific Feedback . . . 58
6.2.2.B Overall Feedback . . . 59
6.2.3 Analysis . . . 60

7 Conclusions and Future Work **61**
7.1 Conclusions . . . 61
7.2 Future Work . . . 62

Bibliography 62

A Survey Answers 67

List of Figures

2.1 Internal Division of ECMAScript Standard 2024 6
2.2 ECMA-SL Project's Execution Pipeline 9
2.3 Without Language Server Protocol 10
2.4 With Language Server Protocol 11
2.5 Language Server Communication Example 15

4.1 RW Language Code Example 27
4.2 Toy Language Server Architecture 28
4.3 RW code completion example 31
4.4 RW go-to-definition example 32
4.5 RW diagnostics example 33
4.6 RW hover example 35
4.7 Pseudocode for getHoverInformation function 36

5.1 ECMA-SL Language Code Example 39
5.2 ECMA-SL Syntax Highlighting File 44
5.3 ESL-LS Architecture 45
5.4 ESL-LS code completion example 46
5.5 ESL-LS hover example 47
5.6 Pseudocode for computeHover function 48
5.7 ESL-LS go-to-definition example 49
5.8 ESL-LS diagnostics example 51
5.9 Pseudocode for addDiagnostic function 52

6.1 Familiarity and Satisfaction Charts 59
6.2 Features Usefulness Charts 60

List of Tables

6.1 Summary Statistics for the Interpreters 55
6.2 Summary statistics of the response times (in milliseconds) taken by ESL-LS features on each interpreter. 56

Acronyms

AI	Artificial Inteligence
IDEs	Integrated Development Environments
IST	Instituto Superior Técnico
JSON	JavaScript Object Notation
LSP	Language Server Protocol
RW	Read/Write
TC39	Technical Committee 39
UI	User Interface

1

Introduction

JavaScript [1] is one of the most popular programming languages today and has become the de facto standard for programming client-side web applications, with native support across all modern browsers. Given the variety of browsers used to access websites, it is crucial for each browser to consistently implement the same semantics of the language. Without this uniformity, web applications may function correctly in some browsers while encountering issues in others, leading to compatibility problems. Ensuring this consistency is vital for providing a seamless user experience across all browsers.

To ensure that all browsers implement JavaScript consistently, the JavaScript committee established the ECMAScript Standard [1], a specification of the language written in English. This standard acts as the pseudocode of a JavaScript interpreter, detailing step-by-step how each expression and statement should be processed. Over the years, the standard has become increasingly complex, spanning now almost one thousand pages. While direct modifications to the standard are required for evolving the language, these modifications are becoming increasingly difficult to implement, as even small changes to one feature can affect others, potentially introducing retrocompatibility issues and even bugs in the standard. As the JavaScript standard becomes more complex, we need automated tools for its management and validation.

To support the maintenance and management of the ECMAScript standard, a research team at Instituto Superior Técnico (IST) developed the ECMA-SL project [2]. This project introduces a specifi-

cation language that mirrors the meta-language used by the JavaScript committee to define the standard. Additionally, the ECMA-SL project provides several implementations of the standard, written in this specification language. These implementations include ECMARef6 [3, 4], an interpreter for ECMAScript 6 [5]; ECMARef5 [2, 6], an interpreter for ECMAScript 5 [7]; and ToyECMA [8], a simplified interpreter for JavaScript.

These versions of the standard written in ECMA-SL [2] are highly valuable because they are executable. This enables language maintainers to test new features by simply implementing them in ECMA-SL and running the official language tests, ensuring consistent behavior across implementations. However, implementing features in ECMA-SL poses a significant challenge, as the language lacks many of the features commonly found in well-known programming languages, such as adequate editor support, making development more difficult and time-consuming.

In this project, we aim to develop a language server [9] for the ECMA-SL language to ensure it has proper editor support. By employing the Language Server Protocol (LSP) [10], language servers facilitate IDE-independent language support, offering capabilities such as syntax highlighting and refactoring. The LSP standardizes communication between the language server and IDEs, enabling a single server to deliver uniform functionality across various development platforms, which simplifies maintenance and enhances developer productivity. The language server implemented in the context of this thesis offers the following capabilities:

- *Go-to-definition:* allows quick navigation to the definition of the function;
- *Hover:* displays type information when hovering over a variable or shows the function signature when hovering over a function;
- *Code Completion:* provides suggestions for completing words as the user types;
- *Diagnostics:* provides visual warnings displayed in the editor for code errors.

To evaluate our language server, we conducted two experiments:

- *Performance experiments:* we measured the response times of the language server for the four main implemented features while performing essential tasks within the existing JavaScript interpreters developed in ECMA-SL;
- *User study:* we invited a group of ECMA-SL developers from IST to use our tool and provide feedback on their experience with it.

The performance experiments demonstrate that the language server's features response times are low and unnoticeable, providing a smooth and efficient programming experience for ECMA-SL developers. This outcome was by no means guaranteed, given the size and complexity of the projects used for the evaluation.

The user study demonstrates that current ECMA-SL programmers consider the language server a valuable addition to the language ecosystem, finding most of its features extremely useful. However, some users suggested additional features, such as automatic code formatting and integration with the static type system of ECMA-SL [8]. We plan to incorporate these improvements into the language server in future iterations.

In summary, the primary contribution of this thesis is the development of a new language server for ECMA-SL, which was thoroughly evaluated both through experiments that measured its performance and user feedback.

This thesis is structured as follows: Chapter 2 provides the background for our thesis, including an introduction to the ECMA-SL language, for which we developed the language server, and an overview of language servers in general. Chapter 3 provides related works, exploring the historical evolution of language servers and program assisting tools. Chapter 4 describes the methodology used to develop a toy language server for a toy programming language. Chapter 5 describes the development process of the ECMA-SL language server, detailing the specific features implemented. Chapter 6 presents the evaluation, describing the experiments conducted to assess the performance of our language server, along with the user study and an analysis of the results. Chapter 7 concludes the thesis with a summary of the findings from our experiments and discusses future improvements that could be made.

2

Background

Contents

2.1 The ECMA-SL Language 5
2.2 An Overview of Language Servers 10

This chapter provides an overview of the essential background necessary to understand the key topics covered in this thesis: the ECMA-SL language (Section 2.1), for which the language server was developed, and language servers (Section 2.2).

2.1 The ECMA-SL Language

2.1.1 ECMAScript

The ECMAScript standard [1] is the document that defines the ECMAScript language, more commonly known as JavaScript. It is maintained and updated by the Technical Committee 39 (TC39), a group consisting of: JavaScript developers, implementers, academics, and other experts, operating under ECMA International. TC39 updates the language specification annually, introducing new features to ECMAScript. The latest version, the 15th edition, was released in June 2024. This specification outlines

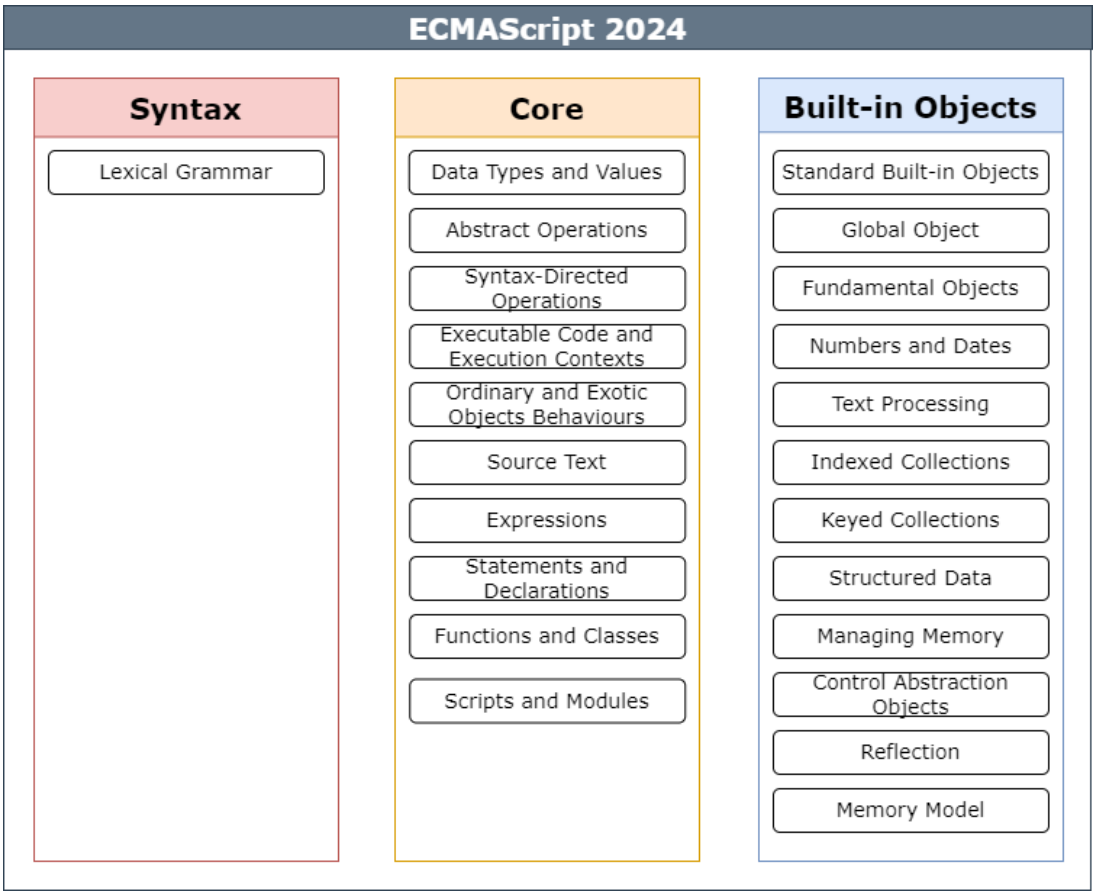


Figure 2.1: Internal Division of ECMAScript Standard 2024

key aspects of how ECMAScript operates, such as: lexical grammar, data types, values, functions, and objects. It can be thought of as the pseudocode of a JavaScript interpreter, outlining step by step how all the JavaScript expressions and statements should be processed.

The standard is extensive, comprising 23 sections. To simplify its structure, we have grouped these sections into three categories, as illustrated in Fig. 2.1.

- **Syntax:** This category contains the grammar rules of ECMAScript.
- **Core Language:** This category contains the language main building blocks, such as: statements, declarations, and expressions. It also contains algorithms that specify how these constructs should operate and behave.
- **Built-in Objects:** This category contains the definitions of the built-in libraries that are part of the JavaScript language. Each built-in library is associated with a special object, referred to as a built-in object, which exposes the methods and properties of that library. Examples of such objects are the String, Number, and Date built-in objects.

2.1.2 ECMA-SL Project

The ECMA-SL project [2] was initiated with the goal of developing an executable specification for the ECMAScript Standard. It introduced a formal specification language that mirrors the informal meta-language of the ECMAScript Standard. Using this language, the ECMA-SL team developed reference implementations of several versions of the ECMAScript Standard. These implementations include: ECMARef6 [3, 4], an interpreter for ECMAScript 6 [5]; ECMARef5 [2, 6], an interpreter for ECMAScript 5 [7]; and ToyECMA [8], a simplified interpreter for JavaScript. These interpreters execute JavaScript code by following the precise steps mandated by the ECMAScript Standard. This tight correspondence between the interpreters' code and the text of the standard is a well-accepted methodology for establishing trust in reference interpreters [11].

In this subsection, we present advantages to create an executable specification (Section 2.1.2.A), the ECMA-SL language (Section 2.1.2.B), and the execution pipeline used to run the reference interpreters written in ECMA-SL (Section 2.1.2.C).

2.1.2.A ECMA-SL Advantages

The interpreters developed in the ECMA-SL project [2], as executable specifications, offer several advantages over the textual specifications of the ECMAScript Standard. Below, we provide some of these benefits.

- Since these interpreters are executable and mirror the meta-language used in the ECMAScript Standard, errors in the specification can be identified more easily than by reviewing their original natural-language description. By testing new features directly in ECMA-SL, the ECMA-Script specification team can quickly detect any unexpected behavior.
- These interpreters can also be used to generate HTML descriptions of the ECMAScript Standard through ECMA-SL2English [2], a tool designed to produce English HTML documentation. The generated documentation is more consistent than manually written version of the standard, as the tool follows a structured and uniform method for explaining similar content.
- These interpreters can also be used as oracles against which to test industrial JavaScript implementations. If the results of the reference interpreter do not coincide with those of a JavaScript engine, such behavioral divergence may indicate a bug in the engine.

2.1.2.B ECMA-SL Language

A key contribution of the ECMA-SL project is the development of the ECMA Specification Language (ECMA-SL), a language that closely follows the pseudocode used in the ECMAScript Standard. It can

be organized in three main categories:

- **Values:** Include integers, floats, lists, tuples, booleans, strings, symbols, null, void, and object references, which are similar to JavaScript values.
- **Expressions:** Expressions can be divided into:
  - **Simple Expressions:** Include value expressions and variable accesses, which can be either local or global.
  - **Operator Expressions:** Include unary, binary, and n-ary operators.
  - **Object Expressions:** Include property lookups, the `in` operator to check if an object contains a property, and object literals.
  - **Call Expressions:** Include functions calls, with optional error handling using a catch clause.
- **Statements:** Statements can be divided into:
  - **Simple Statements:** Include variables assignments, which can be either local or global, and prints.
  - **Object Statements:** Include property assignments and deletions.
  - **Conditional Statements:** Include switch, match-with, and if-then-else statements.
  - **Loop Statements:** Include while, repeat and foreach statements.
  - **Return and Error Statements:** Include throw, fail, and return statements.

2.1.2.C Execution Pipeline

In Fig. 2.2, we present the execution pipeline of the ECMA-SL project for running JavaScript code using one of the reference interpreters written in ECMA-SL. This pipeline can be summarized in three steps:

1. JS2ECMA-SL parses the given JavaScript program (`input.js`) and convert its AST into a format compatible with the ECMA-SL language (`input.esl`).
2. ECMA-SL2Core compiles the reference interpreter written in ECMA-SL (ECMARef) along with the ECMA-SL representation of the JavaScript program to be interpreted (`input.esl`) into Core ECMA-SL, a simplified version of the ECMA-SL language.
3. Using the ECMA-SL Engine, we then run the Core ECMA-SL program obtained from the previous step (`input.cesl`), which captures the behaviour of the original JavaScript program.

Below we describe the main components of this execution pipeline.

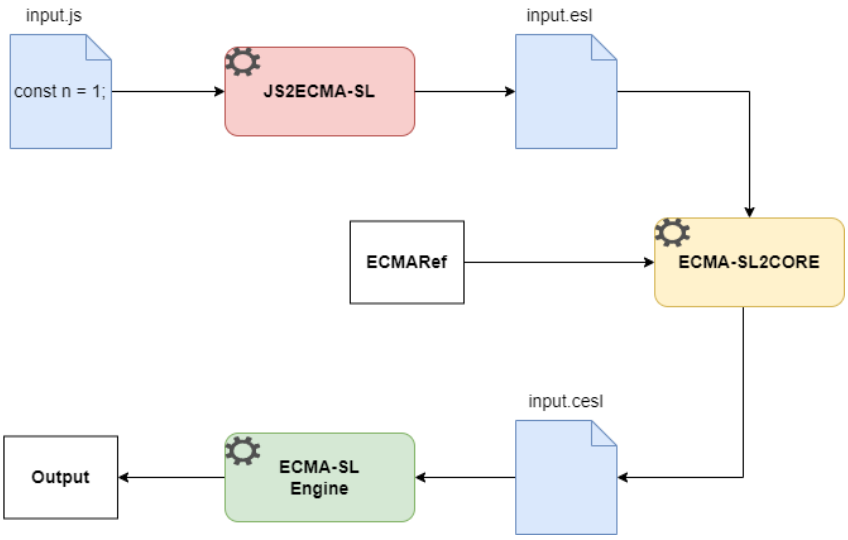


Figure 2.2: ECMA-SL Project's Execution Pipeline

- **JS2ECMA-SL:** Parses the given JavaScript program using Esprima [12], a standard-compliant ECMAScript parser, and generates an ECMA-SL compliant version of that AST.
- **ECMA-SL2Core:** This tool compiles an ECMA-SL program into Core ECMA-SL, a simplified version of the language that resolves all imports included in the program. As a result, the Core ECMA-SL file is self-contained, containing both the code of the used JavaScript interpreter and the ECMA-SL representation of the AST of the program to be interpreted.
- **ECMA-SL Engine:** This tool, implemented in OCaml [13], executes Core ECMA-SL programs. It operates in two modes: silent, which shows only the final output, and verbose, which provides detailed information about the program's execution.

2.1.3 ECMA-SL Limitations

Despite the numerous advantages offered by the ECMA-SL project, it also faces certain challenges. Below, we outline some of these challenges.

- **No Separate Compilation:** The ECMA-SL compiler lacks support for separate compilation, which allows large projects to be divided into independently compiled modules. As a result, developers must manually ensure that function names are unique across all program files, otherwise there can be unexpected behaviors when combining the code of multiple files.
- **Limited Editor Support:** The absence of good editor support hinders the development process. ECMA-SL programmers do not have access to the typical language-specific features found in more well-known languages, making it harder to program efficiently and reducing overall productivity.

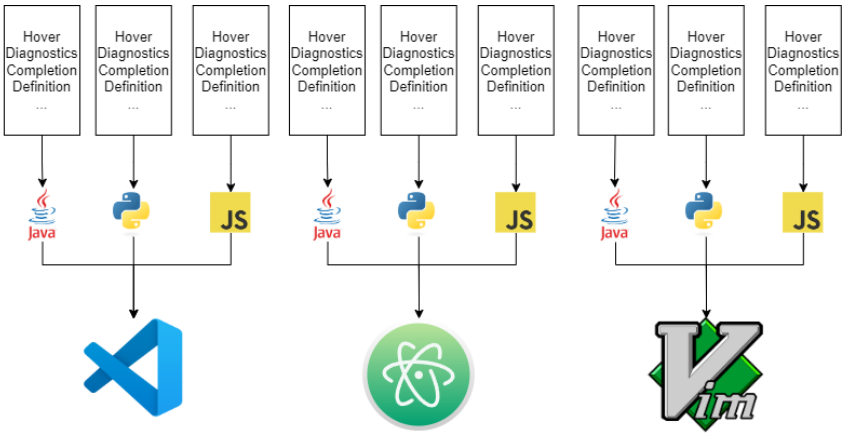


Figure 2.3: Without Language Server Protocol

2.2 An Overview of Language Servers

The increasing number of programming languages and the rapid growth in the number of programmers have made language servers an essential tool for the programming community. Language servers enhance code readability and boost productivity and efficiency by assisting programmers with language-specific features. These features include: code completion, go-to-definition, hover information, diagnostics, and more. In the past, these language server features had to be implemented separately for each code editor. This approach was inefficient and time-consuming, as developers needed to account for the specific requirements and interfaces of different editors when implementing features, as illustrated in Fig. 2.3. This challenge highlighted the need for standardization to streamline the process.

This standardization was achieved with the introduction of the Language Server Protocol (LSP) [10], developed by Microsoft in 2016. The LSP defines a standardized way for an editor to communicate with a language server. This communication protocol allows features developed for one programming language to be compatible across different editors, as illustrated in Fig. 2.4. Due to its effectiveness and versatility, major modern editors quickly adopted the LSP, including Visual Studio Code [14], Sublime Text [15], and more. The LSP enabled rapid development of language servers by allowing developers to focus on implementing language-specific features without needing to account for individual editor integrations.

2.2.1 Language Server Features

As previously mentioned, the features provided by language servers are highly valuable. They enhance the coding experience and increase developer productivity by offering helpful information during the coding process. Below, we outline some of the features offered by current language servers:

- **Code Completion:** Provides suggestions for completing the word or expression currently being typed, streamlining the coding process. This feature is especially helpful in large, complex projects

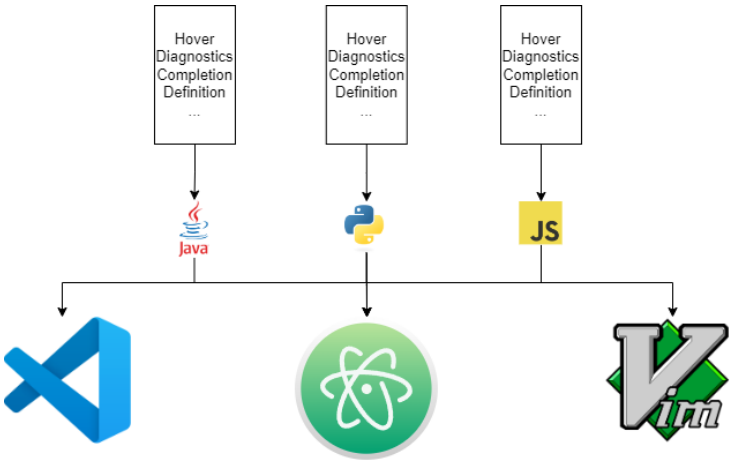


Figure 2.4: With Language Server Protocol

where names of functions or variables may be long or difficult to remember. It saves time and reduces errors, especially when working with code written by others.

- **Diagnostics:** Provides real-time feedback on errors and warnings in the code, allowing developers to identify and resolve issues quickly. By catching mistakes as the code is written, this feature eliminates the need to compile or run the program to detect errors, saving time and improving the overall efficiency of the coding process.
- **Go-to-definition:** Allows developers to instantly navigate to the location where a symbol is defined or declared. This is particularly valuable in large, complex projects, where remembering the exact definition, file, or location of a symbol can be challenging, especially when the code was not written by the user or when the definition is far from the current working area. By quickly navigating to the symbol's definition, this feature saves time and enhances productivity.
- **Hover:** Displays relevant details when hovering over a symbol, such as its type, definition, or a brief description of what it does. For instance, when hovering a function it can display the function's parameters, return type, and documentation. This instant access to useful information aids the coding process, allowing developers to work more efficiently.
- **Code Formatting:** Automatically formats the code. It improves the code's readability, by organizing the presentation of the code.
- **Rename:** Allows developers to rename symbols across the entire code base. This feature is particularly useful in large, complex projects where a symbol might appear in multiple locations, saving time and effort by automating the renaming process.

In the following, we explore the design and implementation of language servers. We begin by describing the key components of the language server (Section 2.2.2), the types of messages exchanged

between the client and server (Section 2.2.3), and the workflow of this communication (Section 2.2.4).

2.2.2 Key Components of the Language Server

The core components of a language server include: client, server, and Language Server Protocol (LSP). In the following, we examine each key component of the Language Server.

Client: Represents the IDE and communicates with the language server by sending requests to access language-specific features.

Server: Provides language-specific features to the client upon request. It processes each request and responds with the necessary information for the client to execute a feature. For instance, when the client requests code completion as a user types, the language server returns a list of relevant suggestions.

Language Server Protocol (LSP): The LSP enables communication between the client and server, allowing them to exchange information about language features. LSP uses JSON-RPC (JavaScript Object Notation Remote Procedure Call) as its communication protocol over channels such as standard input/output streams, named pipes, or sockets. JSON-RPC allows client-server communication by exchanging simple JSON-encoded messages that contain the required information for making requests and receiving responses.

2.2.3 Types of Messages

In this subsection, we examine the main types of messages exchanged during the communication between the client and the server: request messages, response messages, and notifications.

Request Message: The client sends a request message to the server to use a feature. Each request is assigned a unique ID, which is also included in the corresponding response, allowing the client to match the response with its original request. Below we present an example of a request message.

```
{
    "jsonrpc": "2.0",
    "id" : 1,
    "method": "textDocument/definition",
    "params": {
        "textDocument": {
            "uri": "file:///example.esl"
        },
        "position": {
            "line": 2,
            "character": 9
        }
    }
}
```

In this request message, the client is requesting to use the go-to-definition feature. It includes the path to the document and the specific location where the feature was invoked.

In the following, we describe common types of request messages:

- **Initialize:** Starts communication with the server, providing details about the client and its capabilities.
- **textDocument/completion:** Requests code completion suggestions at the current typing position.
- **textDocument/hover:** Requests hover information for the current location of the cursor.
- **textDocument/definition:** Requests the location of the definition of a symbol at the cursor's position when the feature is invoked.

Response Message: The server sends a response message to reply to the request message. This response includes the request's unique ID, allowing the client to associate it with the original request, as well as the necessary information for the client to execute the requested feature. Below we present an example of a response message for the go-to-definition request analysed above.

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "uri": "file:///example.esl",
        "range": {
            "start": {
                "line": 0,
                "character": 9
            },
            "end": {
                "line": 0,
                "character": 13
            }
        }
    }
}
```

This response message contains the same ID as the previously presented request message. The server returns the result after processing the request, providing the path to the document where the definition is located, along with the exact position within the document.

In the following, we describe common types of response messages:

- **Initialize:** Responds to the client's 'initialize' request, providing the server's capabilities and configuration settings.

- **textDocument/completion:** Returns a list of completion suggestions for the current cursor position.
- **textDocument/hover:** Provides information for the symbol under the cursor.
- **textDocument/definition:** Returns the location of the definition for a symbol at the specified position.

Notification Message: This type of messages are sent sent by either the client or the server and do not expect a response unlike the request message, these messages contain information about events. Since it does not have a response, notifications do not have an ID.

In the following, we analyse common types of notification messages:

- **Shutdown:** Notifies the server that the client is shutting down.
- **textDocument/didOpen:** Notifies the server that a document has been opened and includes the document's content.
- **textDocument/didChange:** Notifies the server about changes made to a document and contains the list of changes.
- **textDocument/didClose:** Notifies the server that a document has been closed.
- **textDocument/publishDiagnostics:** Sent from the server to the client to provide diagnostic information, such as a list of errors and warnings.

2.2.4 Workflow

The five main steps of the language servers' workflow are: initialization, request, processing, response, and notifications and events. In the following, we examine each step of the language servers' workflow.

Initialization: Client establishes connection with the language server and exchanges capabilities. This allows the client to understand which features the language server supports and enables the server to determine the capabilities of the client.

Request: The client sends a request to use the language server's features.

Processing: The server analyses the content of the currently open file and, based on the information it extracts, processes the request.

Response: The server sends a response message containing the necessary information for the client to use a feature. The client then updates its user User Interface (UI) to display the provided information. For

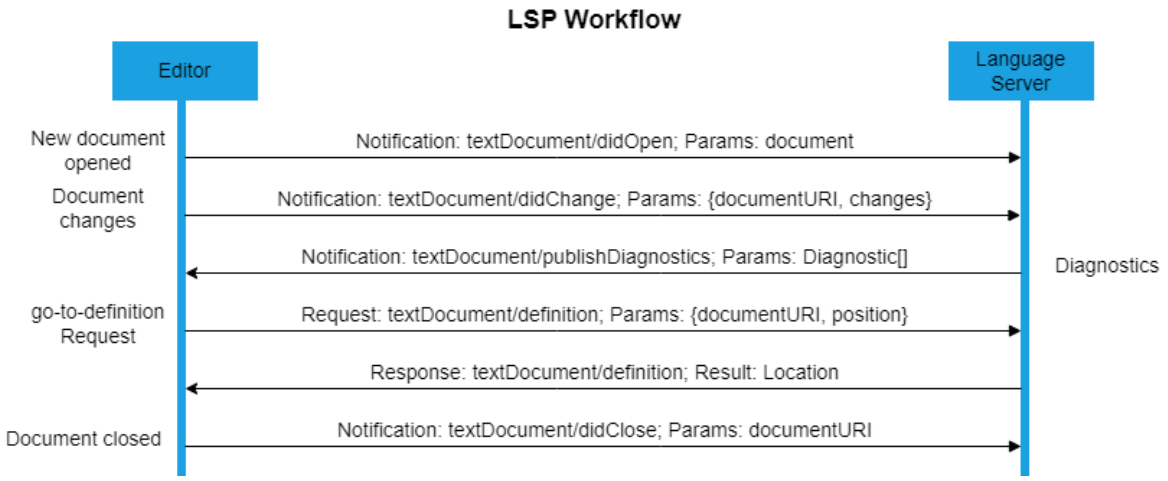


Figure 2.5: Language Server Communication Example

instance, in response to a code completion message, the UI would show a list of completion suggestions for the word currently being typed.

Notifications and Events: The client sends notifications to the server for events such as file changes and configuration updates. This allows the server to update its internal representation of the program, ensuring it provides the most accurate information through its features. The server can also send notifications to the client, such as providing diagnostics or warnings.

Fig. 2.5 illustrates an example of an interaction between Visual Studio Code and the Language Server using the LSP, representing its workflow. Below, we explain the main events of this interaction.

- **New document opened:** The client sends a 'textDocument/didOpen' notification to the language server to indicate that a document has been opened. This notification contains the document's content, which the server uses to create and maintain an internal representation of the program. It is crucial for the server and client to remain synchronized, as any mismatch could result in the server providing incorrect information. This notification can be summarized as:
  `Client Notification: textDocument/didOpen, Parameters: document.`
- **Document changes:** The client sends a 'textDocument/didChange' notification to the language server to indicate that the user has made changes to the code. This notification contains the document URI and the changes made. The server then updates its internal representation of the document based on these changes. This notification can be summarized as:
  `Client Notification: textDocument/didChange, Parameters: {documentURI, changes}.`
- **Diagnostics:** As the server processes the document changes, it analyses them and sends any detected errors or warnings back to the client through the 'textDocument/publishDiagnostics' noti-

fication. This enables the user to view real-time diagnostics, such as errors and warnings, directly displayed in the code editor's UI. This notification can be summarized as:
`Server Notification: textDocument/publishDiagnostics, Parameters: Diagnostic[].`

- **go-to-definition request:** The client sends a 'textDocument/definition' request to the language server, providing the document URI and the text position where the request was made , to determine where the symbol is defined. The language server responds with the URI and the position of the symbol's definition within the document, allowing the user to navigate directly to the symbol's definition within the code. This request can be summarized as:
  `Client Request: textDocument/definition, Parameters: {documentURI, position}.`
  The response can be summarized as:
  `Server Response: textDocument/definition, Result: Location.`

- **Document closed:** The client sends a 'textDocument/didClose' notification to the language server, indicating that the document is no longer needed on the server's internal representation. This notification can be summarized as:
  `Client Notification: textDocument/didClose, Parameters: documentURI.`

3

Related Work

Contents

3.1 Evolution of Language Servers: A Historical Overview 17
3.2 Program Assisting Tools 18
3.3 IDE Integration of Program Analysis Tools 20

3.1 Evolution of Language Servers: A Historical Overview

In the early days of software development, Integrated Development Environments (IDEs) like Turbo Pascal [16] began supporting language features that assisted users during the coding process. These early IDEs provided basic functionalities such as syntax highlighting, simple code completion, and basic diagnostics. However, these features were limited to the specific editor and language for which they were built, restricting their flexibility.

Because the implemented features were editor specific and implemented for a single programming language, it posed significant challenges for developers that worked with multiple languages. To address this issue, IDEs like Eclipse [17] introduced a plugin-based architecture, allowing users to extend the IDE's capabilities with language-specific plugins. While this approach provided more flexibility by

supporting different languages, it remained editor-specific and required substantial development time for each new feature.

As the number of programming languages grew, there was an increasing demand for editor-agnostic tools to offer a seamless coding experience across multiple environments. In response to this need, Microsoft introduced the Language Server Protocol (LSP) [10] in 2016. LSP standardized the communication between IDEs and language servers, enabling developers to build language servers that could be integrated with multiple editors without having to implement language features for each one.

Following the release of LSP, many popular IDEs, such as Eclipse [18] and IntelliJ IDEA [19], adopted the protocol to implement editor-agnostic language servers. The introduction of LSP significantly reduced the work for language server developers, allowing them to focus on creating more powerful and efficient language features rather than having to implement them for each editor. This accelerated the growth of language servers, improving the programming experience across different environments.

While the LSP has been instrumental in the growth of language servers, several challenges remain. One of the primary concerns in modern IDEs is performance. As IDEs incorporate increasingly complex features, the interaction between the IDE and the language server can experience delays in response times, which may adversely affect the user's coding experience. These response delays are the opposite of the purpose of a language server, which is designed to enhance productivity and streamline the coding process, not hinder it.

In the future, with the increasing development of Artificial Inteligence (AI), it is expected that language servers start incorporating AI to further enhance their language-specific features. A prominent example is GitHub Copilot [20], which, while not being a language server itself, demonstrates the potential of AI-powered language servers. GitHub Copilot uses AI to provide enhanced user support through real-time code suggestions, answering questions, and even generating code, showcasing what future AI-integrated language servers could achieve.

3.2 Program Assisting Tools

There are a variety of tools available to help programmers create correct and efficient software. These tools can be divided into two main categories:

- **Static tools:** These tools operate exclusively at the source code level, without requiring program execution. They can detect issues such as syntax errors and provide early feedback to developers, improving the efficiency of the development process by identifying potential problems before runtime.
- **Dynamic tools:** These tools work during the execution of the program, allowing developers to monitor the code's behavior in real-time. They are useful for detecting runtime or memory errors.

Both dynamic and static tools can be integrated with IDEs, each of them having different challenges.

Static Analysis Tools Static tools can be divided into various types:

- **Automatic program analysis tools [21, 22]:** These tools analyse code without the need for a specification. They ensure that the program is correct and does not generate runtime errors and can detect errors such as unitialized variables and memory leaks.
- **Verification tools [23, 24]:** These tools need a specification typically in the form of pre and post-conditions, to verify that the program behaves as expected.
- **Type systems [25, 26]:** These tools ensure that code respects the typing rules, ensuring that variables, functions, and expressions have the correct data types, avoiding type mismatches.
- **Linters [27, 28]:** These tools analyse the code's style, syntax, and possible bugs, using predefined rules. They help maintaining code quality and can detect issues such as unused variables and improper formatting.
- **Compilers [29, 30]:** These tools convert source code into machine code. While it is converting the code, it performs static analysis to detect errors such as syntax errors and type inconsistencies.

Dynamic Analysis Tools Dynamic tools operate by executing a program and observing its behavior. These tools can be classified into two categories: online and offline. Online tools function in parallel with the program during its execution and monitor the program's behavior. In contrast, offline tools analyse the program's execution after it has completed, by using data collected during the program's execution, such as logs and traces.

An example of dynamic tools are debuggers, which are typically classified as online tools. Some well-known debuggers include:

- **Valgrind [31]:** This tool automatically detect memory errors, such as memory leaks and use of unitialized variables. It helps developers ensure that their programs manage memory safely and efficiently.
- **Gdb [32]:** This tool allows the developer to observe the program's behavior during the program's execution. Users can set breakpoints to analyse the program in a specific part of the function.
- **Gprof [33]:** This tool allows developers identify which parts of the program are consuming the most execution time. It provides insights into performance bottlenecks, allowing for more efficient optimization of the code.

3.3 IDE Integration of Program Analysis Tools

In this section, we provide examples of static and dynamic analysis tools that have been integrated with IDEs, focusing on the challenges and lessons learned from these experiences. To structure the discussion, we first present examples related to static analysis tools, followed by those related to dynamic analysis tools.

Integration of Static Analysis Tools Since static tools do not require program execution to analyse and identify errors, they can be integrated with IDEs with relative ease. These tools can run in the background, continuously analysing the code even when the program is not being executed. This allows them to identify errors early in the development process, providing significant value to developers by catching issues before the code is run. Some well-known static tools integrated with IDEs are:

- **jStar-eclipse [34]:** jStar is a tool designed to verify JavaScript programs. It leverages separation logic [35], a formal logic that ensures if a program starts in a state satisfying a specified precondition and successfully completes, the resulting state will meet the corresponding postcondition. This logic enables reliable reasoning about program correctness.

- **Dafny-vscode [23]:** Dafny [36] is both a programming language and a verifier, designed for writing correct programs. It uses specifications like preconditions, postconditions, and loop invariants, which must hold true for the program to be considered correct. Dafny integrates with Visual Studio Code to assist developers in formally verifying their code.

- **Gillian [37–39]:** Gillian is a platform for compositional symbolic execution. It performs symbolic testing by running programs with symbolic inputs, generating execution traces that record the behavior of the program. Based on these traces, Gillian identifies errors and ensures the correctness of the program under test.

Integration of Dynamic Analysis Tools Integrating dynamic tools into IDEs can be more challenging than integrating static tools because dynamic tools require the actual execution of code to analyse a program's behavior during runtime, which can cause conflicts with the editor. Academic debuggers typically do not integrate with IDEs, instead, they often have their own custom visualizers built on HTML. These tools tend to be highly specialized, designed mainly for research purposes.

- **JSExplain [40]:** A JavaScript interpreter that closely adheres to the ECMAScript specification, producing detailed runtime traces. These traces help in identifying errors in the code by providing insights into the program's execution flow.

- **Bugloo [41]:** A source-level debugger designed for the Scheme programming language. Bugloo analyses execution traces to detect errors, providing a detailed view of the program's behavior at runtime.
- **Rexstepper [42]:** A debugging tool specifically designed for JavaScript regular expressions. Rexstepper allows users to step forward and backward through the execution of regular expressions, as well as set breakpoints, enabling a more precise and interactive debugging experience.

Industrial debuggers, unlike academic ones, are designed to integrate seamlessly with IDEs because they are more well-established, have higher demand, and are built with a focus on reliability and performance. Well-known examples of industrial debuggers were already covered earlier when discussing debuggers.

4

A Toy Language Server

Contents

4.1 The Read/Write Language 24

4.2 Syntax Highlighting for the Read/Write Language 24

4.3 A Language Server for the Read/Write Language 27

This chapter details the development of our toy language server. To familiarize ourselves with implementing language server features and understanding the Language Server Protocol (LSP), we began by creating a server for a simple custom programming language. In this initial version, we implemented key features such as syntax highlighting, code completion, diagnostics, hover, and go-to-definition in a straightforward manner, primarily as an exploratory exercise. Building this toy language server provided valuable experience with LSP, which facilitated a smoother transition when developing a more advanced server for the more complex programming language, ECMA-SL.

In the following sections, we discuss our toy programming language (Section 4.1), the implementation of syntax highlighting in the language server to enhance code readability (Section 4.2), and the development of the language server itself (Section 4.3). This includes an overview of its architecture, the server-side representation of programs, and the features that were implemented.

4.1 The Read/Write Language

To gain a deeper understanding of the Language Server Protocol and the process of implementing a language server, we developed a toy programming language called Read/Write (RW). This basic language supports only four types of instructions: 'function', 'call', 'read', and 'write'.

Below we explain the purpose of each instruction:

- **Function:** The 'function' instruction is used to define a new function. The syntax for this instruction consists of a function header followed by a function body. The function header consists of the keyword 'function', followed by the function name and an optional sequence of parameters separated by commas, wrapped in parentheses, and followed by a colon; for instance: `function example(parameter1, parameter2):`. The function body is simply a sequence of call, read, and write instructions.

- **Call:** The 'call' instruction is used to invoke a previously defined function. The syntax for this instruction requires the keyword 'call', followed by the function name and parentheses, which may optionally contain values separated by commas corresponding to the parameters defined in the function being invoked; for instance: `call example(1, 2)`. This way, we can execute a sequence of previously defined instructions.

- **Read:** The 'read' instruction is used to access the value stored in a variable. The syntax for this instruction requires the keyword 'read', followed by the name of the variable whose value is to be retrieved: `read x`. This instruction is essential for obtaining the necessary values to execute the program.

- **Write:** The 'write' instruction is used to assign a value to a variable. The syntax for this instruction requires the keyword 'write', followed by the name of the variable and the value to be assigned to it: `write x 1`. This enables us to store or modify a variable's value within the program, which can be useful for implementing future functions.

4.2 Syntax Highlighting for the Read/Write Language

Syntax highlighting plays a crucial role in providing a good programming experience for any programming language. Nowadays, developers are so accustomed to coding with syntax highlighting that working without it feels unusual. It has become such an integral part of programming that developers expect it to be available by default for all languages. Syntax highlighting enhances code readability and analysis, making it easier for developers to navigate and understand the structure of their programs more

efficiently. By visually differentiating elements such as functions, variables, and keywords, developers can quickly identify the different components of their code.

VSCode adopted the TextMate format for specifying syntax highlighting rules [43], which was initially developed for the TextMate editor [44]. With these rules, one can add syntax highlighting support for a new programming language by creating a simple JavaScript Object Notation (JSON) configuration file containing the corresponding rules. These rules define how different syntactic elements should be displayed within a document.

Syntax highlighting rules do not strictly define the colors and text styles used to display the various elements of the specified language, as this would be too intrusive for the developer. Nowadays, any reputable editor allows developers to choose or even define the display themes for their code. For instance, some developers prefer dark themes, while others prefer light themes. Therefore, the purpose of syntax display rules is not to strictly dictate which color and text style to apply to each syntactic element of the language, but rather to map the language's syntactic categories to the syntactic category classes supported by the editor. The JSON file presented below defines the syntax highlighting rules for the RW language, specifying how different parts of the language should be highlighted in the text editor.

```
{
    "$schema": "https://raw.githubusercontent.com/martinring/tmlanguage/master/tmlanguage.json",
    "name": "RW",
    "scopeName": "source.rw",
    "patterns": [
        {
            "match": "\\b(read|write|function|call)\\b",
            "name": "keyword.control.rw"
        },
        {
            "match": "(?<=\\b(function|call)\\s)\\w+",
            "name": "entity.name.function.rw"
        },
        {
            "match": "(?<=\\b(read|write)\\s)\\w+",
            "name": "variable.name.rw"
        }
    ]
}
```

Below we explain how this syntax highlighting configuration file works.

- **$schema:** URL link to the JSON schema that defines the structure and rules for this grammar file. This link provides guidelines on how the grammar file should be constructed, essentially serving as a blueprint for creating a valid TextMate grammar.
- **name:** Name of the language.
- **scopeName:** Unique identifier of the language scope, used to identify files with this syntax. The file extension '.rw' is used for files written in the RW language, and 'source' indicates that the file

is a source code file.

- **patterns:** Contains matching rules for syntax highlighting along with the syntactic class to be associated with.

In the following, we further explain each pattern in detail and how it contributes to syntax highlighting.

RW Keywords: Below we define the pattern that represents RW keywords.

The rule consists of two fields:

- **match:** Regular expression that matches the keywords 'read', 'write', 'function', and 'call'. The boundary assertion `\\b` ensures that the keywords are matched as whole words.
- **name:** Tells the editor that the current pattern is to be associated with the syntactic class of keywords, `keyword.control.rw`.

Essentially, this rule instructs the editor on which elements are considered keywords in RW.

RW Functions: Below we define the pattern that represents RW function names.

The rule consists of two fields:

- **match:** Regular expression that matches function names, which are words that follow either the 'function' or 'call' instructions, separated by whitespace. The lookbehind assertion `(?<=...)` ensures the presence of the 'function' or 'call' instruction before the matched function name. It checks for these keywords followed by a whitespace (`\\s`), and then matches a sequence of alphanumeric characters (`\\w+`), which is the function name.
- **name:** Tells the editor that the current pattern is to be associated with the syntactic class of function names, `entity.name.function.rw`.

Essentially, this rule instructs the editor on which elements are considered function names in RW.

RW Variables: Below we define the pattern that represents RW variable names.

The rule consists of two fields:

- **match:** Regular expression that matches variable names, which are words that follow either the 'read' or 'write' instruction, separated by whitespace. The lookbehind assertion `(?<=...)` ensures the presence of the 'read' or 'write' instruction before the matched variable name. It checks for these keywords followed by a whitespace (`\\s`), and then matches a sequence of alphanumeric characters (`\\w+`), which is the variable name.

```
function initialize(x, y):
    write a x
    write b y

function sum():
    read a
    read b
    write result a + b
    read result

function main():
    call initialize(1, 2)
    call sum()
```

Figure 4.1: RW Language Code Example

- **name:** Tells the editor that the current pattern is to be associated with the syntactic class of variable names, `variable.name.rw`.

Essentially, this rule instructs the editor on which elements are considered variable names in RW.

In Fig. 4.1, we present a RW program example where variables are initialized with given values, the sum of both values is calculated, and then the resulting value is displayed. The figure also illustrates our syntax highlighting for the RW programming language, showing how keywords like 'function', 'call', 'read', and 'write' are each displayed in different colors. Additionally, function names and variable names are also highlighted with their own unique colors, as specified in the syntax highlighting JSON file. Keywords are displayed in purple, function names in yellow, and variables in blue.

4.3 A Language Server for the Read/Write Language

4.3.1 Language Server Architecture

The diagram presented in Fig. 4.2 showcases the architecture of the toy language server. It shows the interaction between the Editor and the Language Server. Below is an overview of the architecture.

4.3.1.A Editor

UI: User Interface (UI) of the editor, where users interact with the editor. Here, users can write and edit code, create new files, and navigate through existing files. The UI communicates with the language client to access the language server's functionalities, such as: hover information, diagnostics, go-to-definition, and code completion.

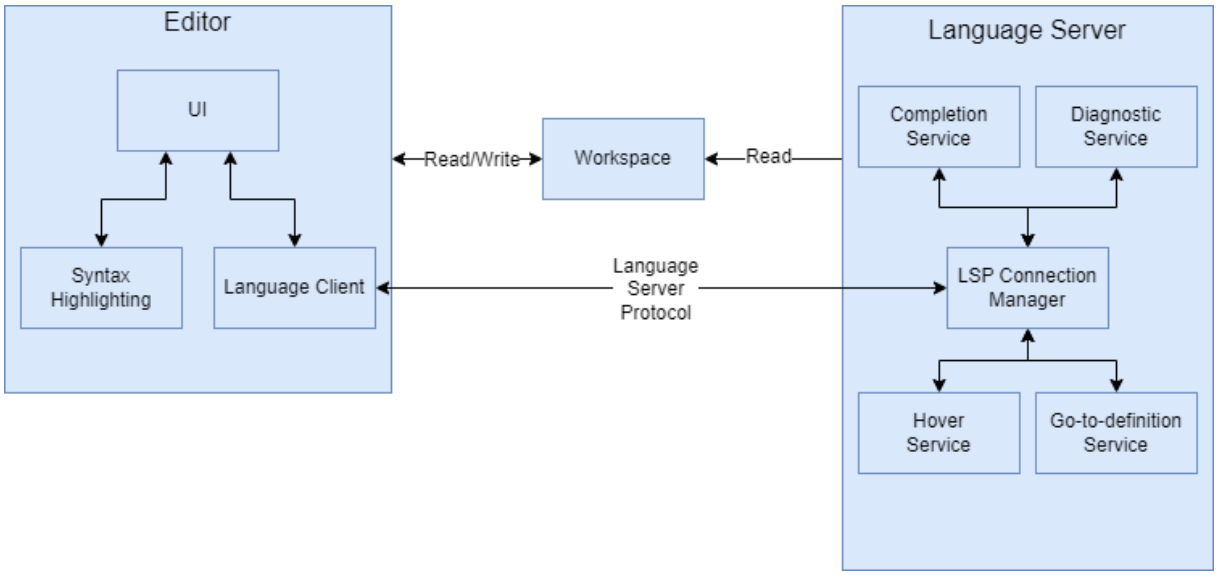


Figure 4.2: Toy Language Server Architecture

Syntax Highlighting: This component manages how text is highlighted within the editor's UI. It defines rules for highlighting various text elements, such as: keywords, function names, variables, and others, specifying the colors and text styles applied to each.

Language Client: This component serves as a bridge between the editor's UI and the language server. It sends requests to the server and processes the server's responses, allowing the user to access the language server's functionalities and the information it provides.

4.3.1.B Workspace

This component represents the files and projects opened in the editor. Both the editor and the language server have access to the workspace. The editor can read and write to the files and projects within the workspace, while the language server can read from it to gather important information needed for its functionalities.

4.3.1.C Language Server

- **Completion Service:** Provides code completion suggestions, such as functions and keywords.
- **Hover Service:** Provides type information for the word being hovered over.
- **Diagnostic Service:** Analyses the code to identify any issues and provides errors or warnings to the language client, which are displayed in the editor.
- **Go-to-definition Service:** Allows users to navigate to the definition of the function.

- **LSP Connection Manager:** Component that manages communication between the Language Client and the Language Server's services.

4.3.1.D Language Server Protocol

The Language Server Protocol (LSP) is the protocol that enables effective communication between the editor and the language server. This standardized protocol enables the editor to seamlessly access the language server's services.

4.3.2 Server-Side Program Representation

In the RW language server, information is organized and maintained using data structures that help manage text documents and function declarations efficiently. These structures support the language server's capabilities, such as diagnostics, code completion, hover, and go-to-definition. They provide the necessary information to implement the server's core functionalities. By working with text documents, we gain access to the user's code, allowing us to process it and extract key information for these services. Additionally, the RW server can handle files beyond those currently opened, as it checks all files for function declarations at the start. This enables immediate go-to-definition and code completion functionalities from the start, saving the user from having to locate where functions are declared and speeding up the process of typing existing functions. This approach is especially valuable in large, complex projects where users may not remember all the code written by themselves or others.

4.3.2.A Text Documents

The server monitors and manages all read-write files in the workspace using the 'TextDocuments' class, a text document manager. This class keeps the contents of the documents up to date for the server by tracking files opened by the user and synchronizing any changes. This means that every modification the user makes in the editor is also reflected in the server, ensuring it stays updated with the latest changes in the code.

4.3.2.B Function Declarations

To support the go-to-definition feature, we created a data structure to store function declarations, including their names and positions. This allows the server to quickly retrieve the necessary information when the user searches for its definition. The 'functionDeclarations' map is defined as follows:

```
Map<string, Map<string, { line: number; character: number }>>()
```

As seen in the map definition, this data structure is a nested map, where:

- The outer key represents the document URI (Uniform Resource Identifier), allowing us to identify the specific file from which we are retrieving information. This is crucial because the functions we are trying to use might be declared in different files.

- The inner key represents the function name, allowing us to identify and store the location of the function we want to track.

- The value represents the position of the function declaration, specifically indicating the line number and character offset where the function is defined in the file.

To support the code completion and hover features, we created a data structure to store the function signatures, function name and possible sequence of parameters, separated by commas and enclosed in parentheses. This allows the server to quickly retrieve the necessary information for when the user types a function declaration. The 'functionSignatures' map is defined as follows:

```
Map<string, Set<string>>()
```

As seen in the map definition, this data structure is a map that associates each file with the set of function signatures within that file. More specifically:

- The key represents the document URI (Uniform Resource Identifier), allowing us to identify the specific file from which we are retrieving information. This is crucial because the functions we are trying to use might be declared in different files.

- The value represents a set of function signatures, which includes the function name followed by the sequence of possible parameters, separated by commas, all enclosed in parentheses.

Both of these data structures store information about function declarations, but they serve different purposes. One is primarily used to store the file and position of each function declaration, while the other is used to store the function signature, including the function name and its parameters.

4.3.3 Supported Features

In this subsection, we describe the four main features implemented for the RW language server: code completion, diagnostics, hover, and go-to-definition. We begin with the simplest feature, code completion, and gradually progress to the most complex, hover.

Figure 4.3: RW code completion example

4.3.3.A Code Completion

The goal of the code completion feature in the RW language server is to enable the editor to automatically suggest completions for the current word being written, including both keywords and functions. For instance, as the programmer begins typing the keyword 'function', the editor will instantly suggest completing it. Similarly, when a programmer starts typing a function name, the editor will provide the full name along with its list of formal parameters. This assists the programmer by reducing the need to remember all function names and their definitions, speeding up the coding process, particularly when dealing with lengthy function names. Figure 4.3 illustrates an example of the code completion feature provided by the RW language server.

LSP provides an endpoint for implementing code completions: `connection.onCompletion`. This endpoint accepts a function that returns an array of all possible completion suggestions. Since the LSP already has the endpoint set up, we just need to implement the function that returns these suggestions and pass it to the `connection.onCompletion` endpoint. Below is the pseudocode for our implementation.

```
1 function computeCodeCompletion() {
2     const completionItems: CompletionItem[] = [];
3     addKeywords(completionItems); // add keywords to the list
4     addFunctionSignatures(completionItems); // add function signatures to the list
5     return completionItems;
6 }
7 connection.onCompletion(computeCodeCompletion);
```

We use a data structure to maintain the words to be used as code completions. Specifically, this data structure is a map that stores function signatures. This global data structure is updated with each change made to the project files, ensuring that any modifications or deletions of function declarations are reflected. This approach prevents the map from retaining outdated or removed function declarations. In our function implementation, we simply provide all possible suggestions (line 5), including keywords and function signatures (lines 3-4). The VSCode editor is responsible for determining which suggestions are appropriate.

4.3.3.B Go-to-definition

The goal of the go-to-definition feature in the RW language server is to enable the editor to navigate directly to the definition of a function upon the developer's request. For instance, when the programmer right-clicks and selects the 'Go to Definition' option, the editor will jump directly to the function's definition, even if it is in a different file. This feature simplifies navigation, allowing the developer to quickly find the

Figure 4.4: RW go-to-definition example

implementation of a function without needing to remember where or how it was written, thus speeding up the coding process. This is especially valuable in larger, more complex projects where numerous files and functions may have been created by different developers. Figure 4.4 illustrates an example of the go-to-definition feature provided by the RW language server.

LSP provides an endpoint for implementing go-to-definition: `connection.onDefinition`. This endpoint accepts a function that takes TextDocumentPositionParams as input and returns the file along with the position of the function declaration. Since the LSP already has the endpoint set up, we just need to implement the function that locates the file and position of the function declaration and pass it to the `connection.onDefinition` endpoint. Below is the pseudocode for our implementation.

```
1  function computeDefinition(params: TextDocumentPositionParams) {
2      const line = processText(params); // get line of text being hovered
3      for each word in line {
4          if (checkPosition(word)) { // check if cursor is within the current word
5              if (checkFunctionDeclarationsMap(word)) { // check if word is in the map
6                  return getDefinitionInformation(word); // return definition information
7              }
8          }
9      }
10     return null; // no matching definition found
11 }
12 connection.onDefinition(computecomputeDefinition);
```

We use a nested map data structure to maintain function declarations, including their file and position. This global data structure is updated with each change made to the file, ensuring that any modifications or deletions of function declarations are reflected. This approach prevents the map from retaining outdated or removed function declarations. In our function implementation, we retrieve the text from the document, split it into lines, and then extracts the text from the line where the cursor is positioned (line 2). From this line, we split the content into individual words (line 3). After identifying the word under the cursor (line 4), we check if this word exists in the map (line 5). If it does, the function returns the file and position of the corresponding function declaration (line 6). Below, we present pseudocode for the

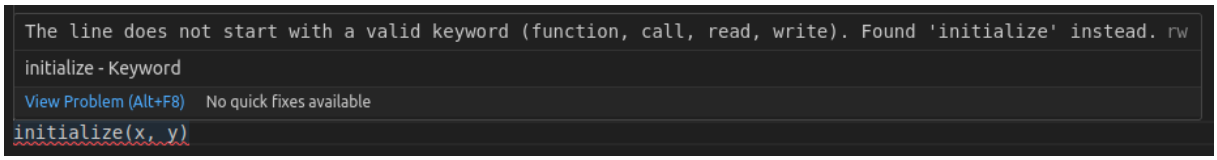


Figure 4.5: RW diagnostics example

implementation of the getDefinitionInformation function.

```
1  function getDefinitionInformation(word) {
2      const declaration = getFromMap(word); // get declaration information from map
3      const result = {
4          uri: declaration.uri,
5          range: {
6              start: { line: declaration.line, character: declaration.character },
7              end: { line: declaration.line, character: declaration.character + word.length }
8          }
9      };
10     return result;
11 }
```

In this function, we retrieve the declaration information from the map (line 2). We then create a result object (lines 3-9) that contains the file in which the function was declared and the exact location of the declaration. Finally, the definition result is returned (line 10).

4.3.3.C Diagnostics

The goal of the diagnostics feature in the RW language server is to enable the editor to display errors or warnings whenever a line does not start with a keyword. For instance, as the programmer types, if the beginning of a line does not match any keyword, the editor will underline the line with a red wavy line to indicate an error. This immediate feedback helps the programmer identify syntax issues instantly, saving time by not needing to compile and run the program to find and correct such mistakes. Figure 4.5 illustrates an example of the diagnostics feature provided by the RW language server. Below is the pseudocode for our implementation.

```
1  function computeDiagnostic(textDocument: TextDocument) {
2      const diagnostics: Diagnostic[] = [];
3      const lines = processText(textDocument); // get text from the file and divide it by lines
4      for each line in lines {
5          if (line === '') {
6              return; // skip if line is empty
7          }
8          if (!checkFirstWord(line)) { // check if first word is a valid keyword
9              addDiagnostic(line, lineIndex, diagnostics); // create and add diagnostic for this
                   line
10         }
11     }
12     return diagnostics;
13 }
```

For this feature, we implemented a function that takes a TextDocument as input, representing the current file being modified by the programmer, and returns an array of diagnostics to the editor. This function is triggered every time the developer modifies the code. In our function implementation, we retrieve the text from the document and split it into lines (line 3). From these lines, we split them into individual lines (line 4). An exception is made for empty lines to prevent unnecessary warnings while the cursor is on such lines (lines 5-7). If a line does not start with any of the four keywords: 'function', 'call', 'read', or 'write' (line 8), we create a diagnostic and add it to be displayed as a warning to highlight the syntax error in that line (line 9). We implemented this basic diagnostic feature because the RW language we created is straightforward, consisting only of instructions that start with a keyword. Below, we present pseudocode for the implementation of the addDiagnostic function.

```
1  function addDiagnostic(line, diagnostics) {
2      const diagnostic = {
3          severity: DiagnosticSeverity.Error,
4          range: {
5              start: { line: lineIndex, character: 0 },
6              end: { line: lineIndex, character: line.length }
7          },
8          message: `The line does not start with a valid keyword (function, call, read, write).
           ↪  Found '${firstWord}' instead.`,
9          source: 'rw'
10     };
11     diagnostics.push(diagnostic); // add diagnostic to list
12 }
```

In this function, we create a diagnostic object (lines 2-10) that includes: the error severity, the range specifying the error's location, the error message to be displayed in the editor, and the source of the diagnostic. Once the diagnostic is created, it is added to the diagnostics list (line 11).

4.3.3.D Hover

The goal of the hover feature in the RW language server is to enable the editor display type information about the word being hovered over. For instance, if the developer hovers over the first word in the line: `call initialize(1, 2)`, the editor will show the word 'Keyword' in markdown format as a tooltip. Likewise, if the developer hovers over the second word in the same line, it will display a block of text with its function signature. This feature helps the programmer quickly understand the type information and the function signature of what they are writing, making the coding process more efficient. Figure 4.6 illustrates an example of the hover feature provided by the RW language server.

LSP provides an endpoint for implementing hover: `connection.onHover`. This endpoint accepts a function that takes TextDocumentPositionParams as input and returns type information. Since the LSP

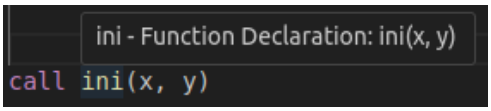


Figure 4.6: RW hover example

already has the endpoint set up, we just need to implement the function that returns the type information and pass it to the `connection.onHover` endpoint. Below is the pseudocode for our implementation.

```
1  function computeHover(params: TextDocumentPositionParams) {
2      const line = processText(params); // get line of text being hovered
3      for each word in line {
4          if (checkPosition(word)) { // check if cursor is within the current word
5              return getHoverInformation(word); // return hover information
6          }
7      }
8      return null; // no word found under cursor
9  }
10 connection.onHover(computeHover);
```

For this feature, we implemented the function presented in Fig. 4.7 that retrieves the text from the document, split it into lines, and then extracts the text from the line where the cursor is positioned (line 2). From this line, we split the content into individual words (line 3). Next, we check if the cursor is within any of these words (line 4), and once we identify the word based on its position, we compute its type information (line 5). Below, we present pseudocode for the implementation of the getHoverInformation function.

Given the simplicity of the RW language, determining type information is straightforward. If the cursor is on the first word, it is considered a keyword (line 4). The word following a 'read' or 'write' instruction is identified as a variable (lines 6-7), while the word following a 'function' or 'call' instruction is identified as a function declaration and its signature (lines 8-14). Additionally, if the first word is 'write' and the cursor is on the third word, it represents a value (lines 16-18). There is also a default case that labels the word as 'data' if it doesn't fit into any of these specific categories (line 19). After gathering the hover information, we return both the type information and the format in which it will be displayed (lines 22-27).

```
function getHoverInformation(word) {
    let description = '';
    switch (i) {
        case 0: description = 'Keyword'; break;
        case 1:
        if (firstWord === "write" || firstWord === "read") {
            description = 'Variable'; break;
        } else if (firstWord === "function" || firstWord === "call") {
            if (word in functionSignatures) {
                const signature = functionSignatures[word];
                description = `Function Declaration: ${signature}`;
            }
            break;
        }
        case 2:
        if (firstWord === "write") {
            description = 'Value'; break;
        }
        default: description = 'Data';
    }
    const hoverText = `${word} - ${description}`;
    return {
        contents: {
            kind: 'plaintext',
            value: hoverText
        }
    };
}
```

Figure 4.7: Pseudocode for getHoverInformation function

5

ECMA-SL Language Server

Contents

5.1 Language Configuration and Syntax Highlighting 38

5.2 Language Server Architecture 44

5.3 Server-Side Program Representation 45

5.4 Supported Features 46

This chapter details the development of ESL-LS, our ECMA-SL language server. Building on the insights gathered from creating the toy language server, we implemented the following key features for ESL-LS: language configuration, syntax highlighting, code completion, diagnostics, go-to-definition, and hover.

In the following sections, we cover the definition of the ESL-LS language configuration and the syntax highlighting (Section 5.1), the architecture of our language server (Section 5.2), the server-side representation of programs (Section 5.3) and the language server features that were implemented (Section 5.4).

5.1 Language Configuration and Syntax Highlighting

In this section, we describe the language configuration (Section 5.1.1) and the syntax highlighting features of ESL-LS (Section 5.1.2).

5.1.1 Language Configuration

VSCode allows us to define language-specific behaviors for any programming language using a file known as a language configuration [45]. These language-specific behaviors can include: comment toggling, bracket matching, auto-closing pairs, surrounding pairs, and autoidentation. For this purpose, we created a language configuration JSON file, which the VSCode editor uses to determine the rules for ECMA-SL's behaviors. Below, we present the language configuration file for ESL-LS.

```
{
    "comments": {
        "blockComment": ["/*", "*/"]
    },
    "brackets": [
        ["{", "}"],
        ["[", "]"],
        ["(", ")"]
    ],
    "autoClosingPairs": [
        ["{", "}"],
        ["[", "]"],
        ["(", ")"],
        ["\"", "\""],
        ["/*", "*/"]
    ],
    "surroundingPairs": [
        ["{", "}"],
        ["[", "]"],
        ["(", ")"],
        ["\"", "\""],
        ["/*", "*/"]
    ]
}
```

In the following, we explain how this language configuration file works:

- **Comments Configuration:** Defines symbol for block comments.
- **Brackets Configuration:** Defines pair of symbols considered brackets.
- **Auto-closing Pairs Configuration:** Defines pair of symbols that are completed automatically after being typed. For instance, when the programmer types (, the editor will instantly complete it with () symbols.

```
function maxF(v1, v2) {
  if (v1 > v2) {
    return v1;
  } else {
    return v2;
  }
}
```

Figure 5.1: ECMA-SL Language Code Example

- **Surrounding Pairs Configuration:** Defines pairs of symbols that can surround text. For instance, after selecting a portion of text and typing {, the editor will surround the selected portion of text with {} symbols.

5.1.2 Syntax Highlighting

As specified in Section 4.2, syntax highlighting is an essential part of the modern programming experience, enhancing code readability and making it easier for developers to navigate and understand code structure by visually differentiating components like functions, variables, and keywords.

VSCode uses the TextMate format for specifying syntax highlighting rules [43], originally developed for the TextMate editor [44]. With TextMate rules, adding syntax highlighting support for a new language is straightforward, developers can create a simple JSON configuration file containing the rules.

These highlighting rules are not strict regarding specific colors and text styles. Most modern editors, including VSCode, allow developers to choose their own display themes, making syntax highlighting adaptable to individual preferences, whether they prefer dark or light themes. The goal of these rules is to map the language's syntactic categories to the classes supported by the editor, leaving visual details to the developer's chosen theme.

In Fig. 5.1, we present an example of our syntax highlighting for the ECMA-SL programming language. The figure demonstrates how the keywords, function names, and variable names are all displayed with distinct colors, as specified in the syntax highlighting JSON file. Keywords are displayed in purple, function names in yellow, and variables and arguments in blue.

In the ECMA-SL syntax highlighting configuration file, we specify 14 syntactic elements, ranging from keywords and strings to function definitions and comments. Below, we explain the most relevant of those, concluding the section with an overview of the whole syntax highlighting configuration file.

ECMA-SL Keywords: Below we define the pattern `keyword` that represents ECMA-SL keywords.

```
"keywords": {
    "patterns": [
        {
            "name": "keyword.control.esl",
            "match": "\\b(if|else|elif|while|foreach|repeat|until|switch|case|sdefault|match|
            ↪  with|default|None|null|function|return|lambda|macro|catch|throw|fail|assume|a
            ↪  ssert|import|extern|print|delete|symbolic|typeof|gen_wrapper)\\b"
        }
    ]
}
```

The rule above consists of two fields:

- **name:** Tells the editor that the current pattern is to be associated with the syntactic class of keywords, `keyword.control.esl`.
- **match:** Regular expression that matches the strings corresponding to ECMA-SL keywords such as: 'if', 'else', 'while', and 'foreach'. The boundary assertion `\\b` inside the regular expression ensures that the keywords are matched as whole words.

Essentially, this rule instructs the editor on which elements are considered keywords in ECMA-SL.

ECMA-SL Strings: Below we define the pattern `strings` that represents ECMA-SL strings.

```
"strings": {
    "name": "string.quoted.double.esl",
    "begin": "\"",
    "beginCaptures": {
        "0": { "name": "punctuation.definition.string.begin.esl" }
    },
    "end": "(\")|((?:[^\\\\\\n])$)",
    "endCaptures": {
        "1": { "name": "punctuation.definition.string.end.esl" },
        "2": { "name": "invalid.illegal.newline.esl" }
    },
    "patterns": [
        {
            "name": "constant.character.escape.esl",
            "match": "\\\\(x[0-9A-Fa-f]{2}|u\\{[0-9A-Fa-f]{4,6}\\}|.|$)"
        }
    ]
}
```

The rule above consists of the following fields:

- **name:** Tells the editor that the current pattern is to be associated with the syntactic class of double quoted strings, `string.quoted.double.esl`.
- **begin:** Tells the editor that the current pattern begins with a double quote.

- **beginCaptures:** Captures the first character and tells the editor that the current character is to be associated with syntactic class of punctuation that marks the beginning of a string, `punctuation.definition.string.begin.esl`.
- **end:** Defines the end of the string pattern. The expression `(\")` captures a second double quote to mark the end of the string. The expression `((?:[^\\\\\\n])$)` matches the case when the string ends unexpectedly, such as when there is a newline or invalid escape character.
- **endcaptures:** Tells the editor that the first matched pattern is to be associated with the syntactic class of punctuation that marks the end of a string, `punctuation.definition.string.end.esl`. The second matched pattern is to be associated with the syntactic class of illegal newline to highlight an error,`invalid.illegal.newline.esl`.
- **patterns:** Defines additional pattern for escape sequences.
  - **name:** Tells the editor that the current pattern is to be associated with the syntactic class of escape sequence character, `constant.character.escape.esl`.
  - **match:** Regular expression that matches escape sequences within the string. The expression `\\\\` matches a backslash, beginning of an escape sequence. The expression `(x[0-9A-Fa-f]{2}` matches hexadecimal escape sequences (e.g.: '\\x41'). The expression `|u\\{[0-9A-Fa-f]{4,6}\\}` matches Unicode escape sequences (e.g.: '\\u0041'). The expression `|.` matches any single character after the backslash. The expression `|$` matches the end of the string if the backslash is not followed by any valid character.

Essentially, this rule instructs the editor on which elements are considered strings in ECMA-SL.

ECMA-SL Functions: Below we define the pattern `functions` that represents ECMA-SL function names.

```
"functions": {
    "patterns": [
        {
            "name": "entity.name.function.esl",
            "match": "\\b\\w+\\b\\s*(?=\\()"
        }
    ]
}
```

The rule above consists of two fields:

- **name:** Tells the editor that the current pattern is to be associated with the syntactic class of function names, `entity.name.function.esl`.

- **match:** Regular expression that matches function names. The boundary assertion `\\b` ensures the match starts at the beginning of a word. The expression `\\w+` matches a sequence of alphanumeric characters, the function name. The boundary assertion `\\b` ensures the function name ends at a word boundary. The expression `\\s*` matches any existing whitespaces. The lookahead `(?=\\(` ensures the match is followed by an opening parenthesis.

Essentially, this rule instructs the editor on which elements are considered strings in ECMA-SL.

ECMA-SL Elements Inside Brackets: Below we define the pattern `inside_brackets` that represents ECMA-SL elements inside brackets.

```
"inside_brackets": {
    "patterns": [
        {
            "begin": "\\(|\\{|\\[",
            "end": "\\)|\\}|\\]",
            "patterns": [
                { "include": "#comments" },
                { "all patterns were included" },
                {
                    "name": "variable.language.esl",
                    "match": "\\b[a-zA-Z_][a-zA-Z0-9_]*(?:\\.[a-zA-Z_][a-zA-Z0-9_]*)*\\b"
                },
                {
                    "name": "keyword.operator.esl",
                    "match": "[><=!&+\\-/*%^~.,]+"
                }
            ]
        }
    ]
}
```

The rule above consists of the following fields:

- **begin:** Indicates beginning of pattern.
- **end:** Indicates end of pattern.
- **patterns:** Includes references to all other patterns to ensure that elements inside the brackets respect previously defined rules. It also has a recursive inclusion to handle nested brackets properly.
- **name:** Tells the editor that the current pattern is to be associated with the syntactic class of language variables, `variable.language.esl`.
- **match:** Regular expression that matches variables within brackets. The expression `\\b[a-zA-Z_][a-zA-Z0-9_]*` matches a variable name that begins with a letter or underscore followed by alphanumeric characters or underscores. `(?:\\.[a-zA-Z_][a-zA-Z0-9_]*)*` matches

the dot notation for property access. Finally, the boundary assertion `\\b` marks the end of the word.

- **name:** Tells the editor that the current pattern is to be associated with the syntactic class of operators, `keyword.operator.esl`.
- **match:** Regular expression that matches operators within brackets.

Essentially, this rule instructs the editor on which elements are considered elements inside brackets in ECMA-SL.

ECMA-SL Variables: Below we define the pattern `variables` that represents ECMA-SL variables.

```
"variables": {
    "patterns": [
        {
            "name": "variable.name.esl",
            "match": "\\b\\w+(?:\\.\\w+)*\\b(?=\\s*:\\s*(=|\\w+|\\s*))|(?<=\\b(?:print|return
            ↪  |match)\\s+)\\b\\w+(?:\\.\\w+)*\\b|\\b\\w+(?:\\.\\w+)*\\b(?=\\[)"
        }
    ]
}
```

The rule above consists of two fields:

- **'name':** Tells the editor that the current pattern is to be associated with the syntactic class of variable names, `variable.name.esl`.
- **'match':** Regular expression that matches variables. The boundary assertion `\\b` ensures the match starts at the beginning of a word. The expression `\\w+` matches a sequence of alphanumeric characters, the variable name. The expression `(?:\\.\\w+)*` matches zero or more occurrences of dot notation for property access. The boundary assertion `\\b` marks the end of the word. The positive lookahead `(?=\\s*:\\s*(=|\\w+|\\s*))` ensures that the variable is followed by a colon and optionally an =, a type declaration, or some whitespace. The positive lookbehind `|(?<=\\b(?:print|return)\\s+)` matches a variable only if it follows the keywords 'print', 'return' or 'match'. The expression `\\b\\w+(?:\\.\\w+)*\\b` is identical to the previous one to extract the variable name. The expression `\\b\\w+(?:\\.\\w+)*\\b(?=\\[)` matches a variable name followed by a bracket.

Essentially, this rule instructs the editor on which elements are considered variables in ECMA-SL.

The syntax highlighting rules for the ECMA-SL language are presented in the JSON object presented in Fig. 5.2. These rules specify how different ECMA-SL syntactic elements should be highlighted in the

```
{
    "$schema":
    ↪  "https://raw.githubusercontent.com/martinring/tmlanguage/master/tmlanguage.json",
    "name": "ECMA-SL",
    "scopeName": "source.esl",
    "patterns": [
        { "include": "#comments" },
        { "include": "#breakpoints" },
        { "include": "#keywords" },
        { "include": "#constants" },
        { "include": "#builtin" },
        { "include": "#api" },
        { "include": "#numbers" },
        { "include": "#strings" },
        { "include": "#booleans" },
        { "include": "#symbols" },
        { "include": "#types" },
        { "include": "#functions" },
        { "include": "#inside_brackets" },
        { "include": "#variables" }
    ],
    "repository": {
        "..."
    }
}
```

Figure 5.2: ECMA-SL Syntax Highlighting File

text editor. This file is similar to the RW syntax highlighting file (Section 4.2), but in this case, the patterns object only includes references to predefined items. Each included item corresponds to a named set of display rules, with their definitions found in the repository object. The repository object is a dictionary, containing all the matching rules that can be referenced throughout the grammar, along with the syntactic class to be associated with.

5.2 Language Server Architecture

The diagram presented in Fig. 5.3 showcases the architecture of the ECMA-SL language server. It shows the interaction between the Editor and the Language Server. The architecture is essentially identical to that of the RW language server (Section 4.3.1), with the main difference being that the diagnostic service now communicates with the ECMA-SL compiler through the diagnostic file generated as a result of that interaction. Below we further explain how these components assist the diagnostic service.

ECMA-SL Compiler: Compiler for the ECMA-SL language. It compiles the code and generates error messages when issues are detected.

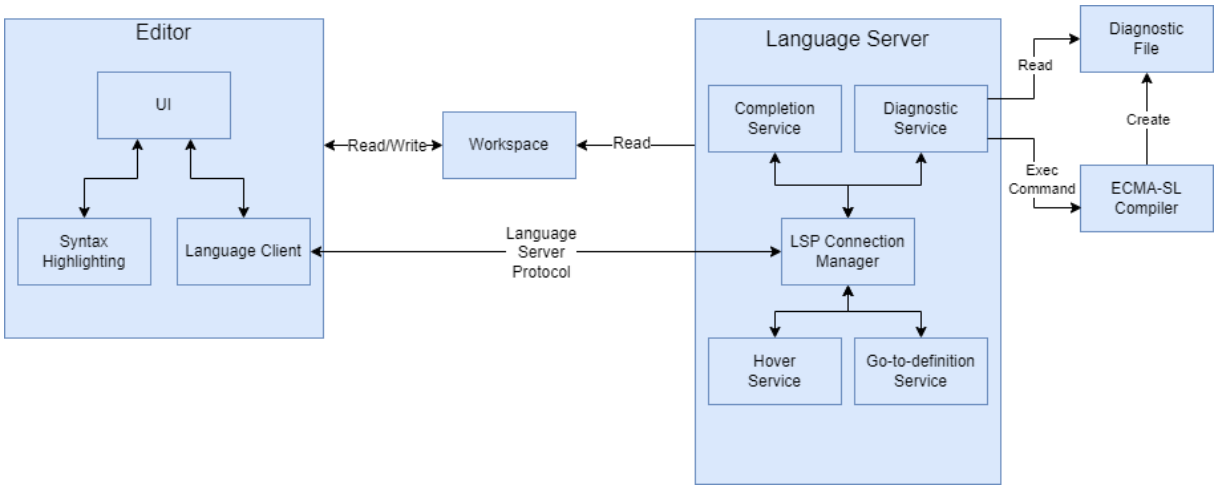


Figure 5.3: ESL-LS Architecture

Diagnostic File: File created to store the errors generated by the ECMA-SL compiler.

Diagnostic Service: Executes a command to compile the current file using the ECMA-SL compiler, generating a diagnostic file that contains any detected errors. The diagnostic service then reads this file, processes the information, and provides the identified errors or warnings to the language client, which are displayed in the editor.

5.3 Server-Side Program Representation

In ESL-LS, information is organized and maintained using data structures that help manage text documents, function declarations and variable declarations efficiently. These structures support the language server's capabilities, such as diagnostics, code completion, hover, and go-to-definition. The data structures remain essentially unchanged when compared against the corresponding data structures for the RW language (Section 4.3.2). The only significant difference is the addition of a new data structure for storing variable declarations.

5.3.1 Variable Declarations

To support the code completion and hover features, we also created a data structure to store the variable declarations, specifically capturing the word preceding `:=`. This allows the server to quickly retrieve the necessary information when the user types or hovers over a variable name. The 'variableDeclarations' map is defined as follows:

```
Map<string, Set<string>>()
```

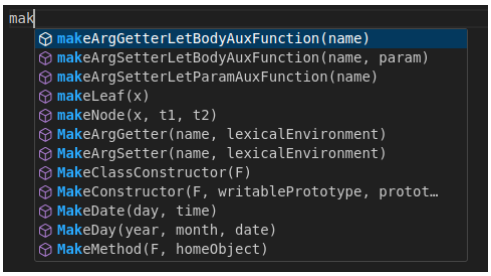


Figure 5.4: ESL-LS code completion example

As seen in the map definition, this data structure is a map that associates each file with the set of variables declared within that file. More specifically:

- The key represents the document URI, allowing us to identify the specific file from which we are retrieving information. This is crucial because the variables we are trying to use might be declared in different files.
- The value represents a set of variable names.

5.4 Supported Features

In this section, we describe the four main features implemented for the ECMA-SL language server: code completion, diagnostics, hover, and go-to-definition. We begin with the simplest feature, code completion, and gradually progress to the most complex, diagnostics.

5.4.1 Code Completion

The goal of the code completion feature in the ECMA-SL language server is to enable the editor to automatically suggest completions for the current word being written, including keywords, functions and variables. For instance, as the programmer begins typing the keyword 'function', the editor will instantly suggest completing it. Similarly, when a programmer starts typing a function name, the editor will provide the full name along with its list of formal parameters. Also, when a programmer starts typing a variable name, the editor will provide the full name. This assists the programmer by reducing the need to remember all variable names, function names and their definitions, significantly speeding up the coding process, particularly when dealing with lengthy function and variable names. Fig. 5.4 illustrates an example of the code completion feature provided by ESL-LS.

As explained in the previous chapter, LSP provides an endpoint for implementing code completions: `connection.onCompletion`. This endpoint accepts a function that takes a TextDocumentPosition as

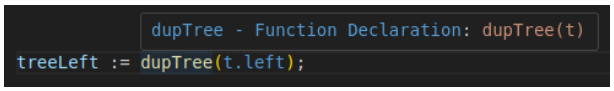


Figure 5.5: ESL-LS hover example

input and returns an array of all possible completion suggestions. Below is the pseudocode for our implementation.

```
1 function computeCodeCompletion() {
2     const completionItems: CompletionItem[] = [];
3     addKeywords(completionItems); // add keywords to the list
4     addFunctionSignatures(completionItems); // add function signatures to the list
5     addVariables(completionItems); // add variables to the list
6     return completionItems;
7 }
8 connection.onCompletion(computeCodeCompletion);
```

We use data structures to maintain the words intended for code completions. Specifically, these data structures are a map for storing function signatures and another map for storing variable declarations. These global data structures are updated with each change made to the file, ensuring that any modifications or deletions of function declarations and variable declarations are reflected. This approach prevents the maps from retaining outdated or removed function and variable declarations. In our implementation, we simply provide all possible suggestions (line 6), including keywords, function signatures and variables (lines 3-5). The VSCode editor is responsible for determining which suggestions are appropriate.

5.4.2 Hover

The goal of the hover feature in the ECMA-SL language server is to enable the editor display type information about the word being hovered over. For instance, if the developer hovers over the first word in the line: `treeLeft := dupTree(t.left);`, the editor will show the word 'Variable' in a tooltip. Likewise, if the developer hovers over the second word in the same line, it will display a block of text with the word 'Function Declaration' and its signature. This feature helps the programmer quickly understand the type information of what they are writing, making the coding process more efficient. Fig. 5.5 illustrates an example of the hover feature provided by ESL-LS.

As explained in the previous chapter, LSP provides an endpoint for implementing hover: `connection.onHover`. This endpoint accepts a function that takes TextDocumentPositionParams as input and returns type information. Fig. 5.6 gives the pseudocode of our implementation.

We use data structures to maintain the words intended for hover information. Specifically, these data structures are a map for storing function signatures and another map for storing variable declarations. These global data structures are updated with each change made to the file, ensuring that any mod-

```
function computeHover(params: TextDocumentPositionParams) {
    const line = processText(params); // get line of text being hovered
    for each word in line {
        if (checkPosition(word)) { // check if cursor is within the current word
            return getHoverInformation(word); // return hover information
        }
    }
    return null; // no word found under cursor
}
connection.onHover(computeHover);
```

Figure 5.6: Pseudocode for computeHover function

ifications or deletions of function declarations and variable declarations are reflected. This approach prevents the maps from retaining outdated or removed function declarations and variable declarations. For this feature, we implemented a function that retrieves the text from the document, split it into lines, and then extract the text from the line where the cursor is positioned (line 2). From this line, we split the content into individual words (line 3). Next, we check if the cursor is within any of these words (line 4), and once we identify the word based on searching in the data structures for storing function declarations and variable declarations, we display its type information (line 5). Below, we present pseudocode for the implementation of the getHoverInformation function.

```
function getHoverInformation(word) {
    let description = '';
    if (word in functionSignatures) {
        const signature = functionSignatures[word];
        description = `Function Declaration: ${signature}`;
    }
    if (word in variableDeclarations) {
        description = 'Variable';
    }
    const hoverText = `${word} - ${description}`;
    return {
        contents: {
            kind: 'markdown',
            value: `\`\`\`yaml\n${hoverText}\n\`\`\``
        }
    };
}
```

The function `getHoverInformation` works as follows. If the searched word matches a function name in the functionSignaturesMap, it is identified as a function declaration along with its signature (lines 3-6). If the searched word is found in the variable declarations map, it is identified as a variable (lines 7-9). Once the hover information is gathered, we return both the type information and the format for displaying it (lines 11-16).

Figure 5.7: ESL-LS go-to-definition example

5.4.3 Go-to-definition

The goal of the go-to-definition feature in the ECMA-SL language server is to enable the editor to navigate directly to the definition of a function upon the developer's request. For instance, when the programmer right-clicks and selects the 'Go to Definition' option, the editor will jump directly to the function's definition, even if it is in a different file. This feature simplifies navigation, allowing the developer to quickly find the implementation of a function without needing to remember where or how it was written, thus speeding up the coding process. This is especially valuable in larger, more complex projects where numerous files and functions may have been created by different developers. Fig. 5.7 illustrates an example of the go-to-definition feature provided by ESL-LS.

As explained in the previous chapter, LSP provides an endpoint for implementing go-to-definition: `connection.onDefinition`. This endpoint accepts a function that takes TextDocumentPositionParams as input and returns the file along with the position of the function declaration. Below is the pseudocode for our implementation.

```
function computeDefinition(params: TextDocumentPositionParams) {
    const line = processText(params); // get line of text being hovered
    for each word in line {
        if (checkPosition(word)) { // check if cursor is within the current word
            let results = [];
            results = getDefinitionInformation(word, results); // add definitions to list
            if (results.length > 1) {
                sortByMostSimilarPath(results); { // sort list
                return results[0];
            }
            if (results.length === 1) {
                return results[0];
            }
        }
    }
    return null; // no matching definition found
}
connection.onDefinition(computecomputeDefinition);
```

We use a nested map data structure to maintain function declarations, including their file and position.

This global data structure is updated with each change made to the file, ensuring that any modifications or deletions of function declarations are reflected. This approach prevents the map from retaining outdated or removed function declarations. In our function implementation, we retrieve the text from the document, split it into lines, and then extract the text from the line where the cursor is positioned (line 2). From this line, we split the content into individual words (line 3). After identifying the word under the cursor (line 4), we gather all possible definition information since multiple functions with the same name may exist across different files (line 6). If the results list contains more than one definition (line 7), we sort the list based on the most similar URI (line 8) and return the top result (line 9). If the list contains only one definition (line 11), we return it directly without sorting (line 12). Below, we present the pseudocode for the implementation of the getDefinitionInformation function.

```
1  function getDefinitionInformation(word, results) {
2      for [uri, Declarations] in functionDeclarationMap {
3          const declaration = Declarations.get(word);
4          if (declaration) {
5              results.add({
6                  uri: uri,
7                  range: {
8                      start: { line: declaration.line, character: declaration.character },
9                      end: { line: declaration.line, character: declaration.character +
                       ↪  word.length }
10                 }
11             });
12         }
13     }
14     return results;
15 }
```

In this function, we iterate over each file in the functionDeclarationsMap (line 2) and obtain the declaration for the provided word from the Declarations map, if it exists (line 3). If a declaration is found (line 4), we create a result object that includes the file where the function was declared and the precise location of the declaration, adding it to the results list (lines 5-11) and return the results list (line 14).

5.4.4 Diagnostics

The goal of the diagnostics feature in the ECMA-SL language server is to enable the editor to display errors or warnings whenever there are errors in the code. For instance, as the programmer types on the function body and the line does not finish with ;, the editor will underline the part with a red wavy line to indicate an error. This immediate feedback helps the programmer identify syntax issues instantly, saving time by not needing to compile and run the program to find and correct such mistakes. Fig. 5.8 illustrates an example of the diagnostics feature provided by ESL-LS. Below is the pseudocode for our implementation.

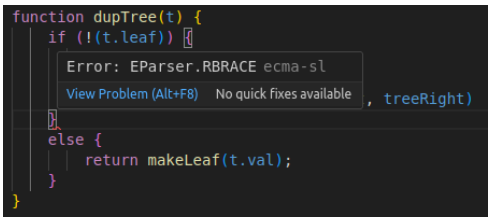


Figure 5.8: ESL-LS diagnostics example

```
1  function computeDiagnostic(textDocument: TextDocument) {
2      const diagnostics: Diagnostic[] = [];
3      const data = readErrorFile(textDocument);     // create and get text of errors file
4      sendDiagnostic(data, diagnostics, textDocument); // create and send diagnostics
5      return;
6  }
```

For this feature, we implemented a function that takes a TextDocument as input, representing the current file being modified by the programmer, and returns an array of diagnostics to the editor. This function is triggered every time the developer saves the code. For this function we used the existing ECMA-SL compiler to give us the errors and warnings of the code. We compile and run the code and create a diagnostic file where we save the errors thrown by the compiler (line 3). We process the information from the file and return the diagnostics from it (line 5). In Fig. 5.9, we present pseudocode for the implementation of the addDiagnostic function.

The addDiagnostic function can process two types of errors from the ECMA-SL compiler: runtime errors (lines 2-15) and error messages (lines 16-43). Both these errors normally provide the line and character offset, but sometimes error messages may come without line and character offset information due to the error being originated in an imported file (lines 29-40). In that case we underline the first character of the file and show the first line of the error message found in the file (lines 30-38). For every type of error we create a diagnostic object (lines 3-11, 18-26 and 30-38) that includes: the error severity, the range specifying the error's location, the error message to be displayed in the editor, and the source of the diagnostic. Once the diagnostic is created, it is added to the diagnostics list (lines 12, 27 and 39) and sent to be displayed on the editor (lines 13, 41 and 44).

```
function addDiagnostic(data, diagnostics, textDocument) {
    if(runtimeErrorMatch(data)) {
        const diagnostic = {
            severity: DiagnosticSeverity.Error,
            range: {
                start: { line: data.line, character: data.charStart },
                end: { line: data.line, character: data.charEnd }
            },
            message: `RuntimeError: ${data.message}`,
            source: 'ecma-sl'
        };
        diagnostics.push(diagnostic); // add diagnostics to the list
        connection.sendDiagnostics({ uri: textDocument.uri, diagnostics });
        return;
    }
    if(errorMatch(data)) {
        if (data.file == textDocument.uri) {
            const diagnostic = {
                severity: DiagnosticSeverity.Error,
                range: {
                    start: { line: data.line, character: data.charStart },
                    end: { line: data.line, character: data.charEnd }
                },
                message: `Error: ${data.message}`,
                source: 'ecma-sl'
            };
            diagnostics.push(diagnostic);
        }
        else {
            const diagnostic: Diagnostic = {
                severity: DiagnosticSeverity.Error,
                range: {
                    start: { line: 0, character: 0 },
                    end: { line: 0, character: 1 }
                },
                message: data.firstLine(),
                source: 'ecma-sl'
            };
              diagnostics.push(diagnostic);
        }
        connection.sendDiagnostics({ uri: textDocument.uri, diagnostics });
        return;
    }
    connection.sendDiagnostics({ uri: textDocument.uri, diagnostics });
    return;
}
```

Figure 5.9: Pseudocode for addDiagnostic function

6

Evaluation

Contents

6.1 Performance Evaluation 54
6.2 User Study 57

Identifying the main evaluation questions is essential for evaluating ESL-LS, the tool developed for our project. We defined two key evaluation areas: the usefulness of ESL-LS for programming in ECMA-SL, and its performance during the programming experience. The evaluation questions will help us evaluate both the language server's performance and the user's experience. This is vital as it will help us determine the strengths and weaknesses of ESL-LS. The feedback gained from this evaluation will be used in future for improving the language server, ultimately enhancing the user experience and boosting productivity during programming and debugging. Based on these evaluation areas, we created the following two main evaluation questions.

EQ1: Is ESL-LS performant enough to avoid degrading the programming experience?
The performance of the language server is crucial to ensure it does not introduce noticeable delays during development. It is essential that ESL-LS operates seamlessly, without negatively impacting the developer's experience or productivity. We obtain the answer for this evaluation question through performance experiments (Section 6.1).

EQ2: Is ESL-LS useful for programming in ESL?
ESL-LS was developed with the goal of enhancing the developer's programming experience. The features are designed to support and assist developers effectively throughout the development of ECMA-SL code. Therefore, it is crucial that these features are genuinely beneficial to users. We obtain the answer for this evaluation question through a user study (Section 6.2).

In this chapter, we describe both the experiments for evaluating ESL-LS's performance (Section 6.1) and the user study for evaluating its usefulness (Section 6.2).

6.1 Performance Evaluation

6.1.1 Experiment

We conducted experiments on ESL-LS to assess its performance, ensuring that the server operates efficiently without compromising the user experience during programming. We examine the response time of the language server when processing client requests for various features. Specifically, we measured the time elapsed from when the server receives a request until it completes processing the necessary information and returns a response to the client. We analysed the performance of the following features: code completion, diagnostics, hover, and go-to-definition.

Experimental Setup These performance evaluations were done on a machine with a 16-core Intel i7-13700HX CPU, 32GB of RAM and running Ubuntu 22.04.4 LTS.

Datasets As stated before, ECMA-SL was primarily designed as a language for implementing the JavaScript standard [1]. Hence, ECMA-SL developers typically use the language for either implementing JavaScript interpreters that closely follow the official JavaScript specification or implementing runtime libraries for those interpreters. To this day, the three largest ECMA-SL projects correspond to three distinct JavaScript interpreters:

- **ECMARef5** [2, 6] is an implementation of the 5th version of the ECMAScript standard (ES5) [7]. It has full support for the core syntax of the language and for almost all of its built-in libraries, including the Regular Expression and JSON libraries. It passes a total of 12,026 out of 12,074 tests from the ES5 tests in Test262 [46], the official JavaScript conformance testsuite.
- **ECMARef6** [3, 4] is an implementation of the 6th version of the ECMAScript standard (ES6) [5]. It supports all of the syntax of the language with the exception of function generators and all of its built-in libraries, including the Promises, ArrayBuffer, and TypeArray libraries. It passes a total of 19,009 out of 21,662 tests from the ES6 tests in Test262.

Table 6.1: Summary Statistics for the Interpreters

	Total	ECMARef5	ECMARef6	ToyECMA
Total Files	138	47	81	10
Total Lines	65,752	23,351	41,542	859
Max Characters per Line	814	814	814	111
Average Characters per Line	43.76	50.84	40.17	24.97
Median Characters per Line	37.0	43	36.0	24
Max Lines per File	3,690	3,024	3,690	377
Average Lines per File	476.46	496.83	512.86	85.90
Median Lines per File	149.5	150	166	25.0

- **ToyECMA** [8] is an implementation of a simplified JavaScript interpreter to be used for teaching purposes. Even though, ToyECMA is not meant to cover a large subset of the language, it does support most syntactic constructs, including literals, expressions, statements and declarations.

We evaluated the performance of ESL-LS in the JavaScript interpreters described above. Table 6.1 provides statistics about these interpreters, as well as global statistics. In particular, it includes the total number of files and lines of code, along with the maximum, average, and median values for characters per line, and the maximum, average, and median values for lines of code per file. Analysing the number of characters per line and lines per file is essential, as our feature implementations rely on processing elements at specific line positions and parsing lines of code within the editor. From these statistics we can conclude the following:

- The dataset used for the experiments is notably large and complex. The three interpreters total 138 files and 65,752 lines of code, making them ideal for performance testing as larger projects are more likely to introduce response delays than smaller ones. Among the interpreters, ECMARef6 has the largest number of files at 81, while ToyECMA has the fewest with only 10 files. This is expected as ToyECMA was only created for pedagogical purposes.
- Both ECMARef5 and ECMARef6 have higher average lines per file and characters per line compared to ToyECMA. This indicates that experiments involving files from these interpreters may experience longer response times for certain features, as processing larger and more complex files tends to be more resource-intensive.

Experiments Below, we explain how the experiments for each feature were conducted:

- **Go-to-Definition:** For the go-to-definition feature, we opened 10 random files in each interpreter and requested the definition of 1 function in each file, effectively performing 10 go-to-definition requests per project.

Table 6.2: Summary statistics of the response times (in milliseconds) taken by ESL-LS features on each interpreter.

Interpreter	Features	n	max	min	avg	med
ECMARef5	**Go-to-definition**	10	0.463 ms	0.078 ms	0.221 ms	0.200 ms
	Hover	10	0.731 ms	0.124 ms	0.303 ms	0.181 ms
	Code Completion	10	0.725 ms	0.195 ms	0.345 ms	0.272 ms
	Diagnostics	20	41.039 ms	22.477 ms	31.325 ms	31.501 ms
ECMARef6	**Go-to-definition**	10	0.731 ms	0.124 ms	0.303 ms	0.181 ms
	Hover	10	1.395 ms	0.631 ms	1.004 ms	0.964 ms
	Code Completion	10	1.039 ms	0.295 ms	0.589 ms	0.443 ms
	Diagnostics	20	63.139 ms	30.291 ms	40.248 ms	39.490 ms
ToyECMA	**Go-to-definition**	10	0.167 ms	0.035 ms	0.072 ms	0.056 ms
	Hover	10	0.387 ms	0.060 ms	0.165 ms	0.122 ms
	Code Completion	10	0.109 ms	0.038 ms	0.058 ms	0.053 ms
	Diagnostics	20	48.458 ms	20.782 ms	37.932 ms	38.564 ms

- **Hover:** For the hover feature, we opened 10 random files in each interpreter and hovered over different words in the code to retrieve hover information, effectively performing 10 hover requests per project.
- **Code Completion:** For the code completion feature, we opened 10 random files in each interpreter and typed random letters to request completion suggestions, effectively performing 10 code completion requests per project.
- **Diagnostics:** For the diagnostics feature, we opened 10 random files in each interpreter and ran diagnostics both with and without errors in the code, effectively performing 20 diagnostics requests per project.

6.1.2 Results

Table 6.2 summarizes the performance results for ECMARef5, ECMARef6, and ToyECMA. The table presents for each tested feature in each interpreter: the number of conducted experiments (n), as well as the maximum (max), minimum (min), average (avg), and median (med) response times.

6.1.3 Analysis

Based on the obtained results, we can conclude the following:

- The average response times for ECMARef5 and ECMARef6 are higher than those for ToyECMA. This was expected, as ECMARef5 and ECMARef6 manage a larger number of files, lines, and

characters per line. However, the difference in response times is negligible, with delays only in the range of milliseconds, making them unnoticeable.

- Our language server performs efficiently enough to maintain a smooth ECMA-SL programming experience. The longest response time across all features was 63.139 milliseconds for the diagnostics feature, a duration too short for users to perceive any noticeable impact on the editor's performance. In fact, in UI literature, response times lower than 100 milliseconds are considered to be perceived as instantaneous by users [47].

We measured the response times on the server because that is where we control the code. It is important to note that both the client and the server are running on the same machine and the various IDEs are highly optimized to process LSP requests and responses very quickly. Therefore, the majority of the time associated with these requests will necessarily be on the server side, allowing us to confidently disregard any client-side latencies.

6.2 User Study

6.2.1 Experiment

To gather valuable feedback from users about ESL-LS, we created a survey using Google Forms. This survey's objective is to provide feedback on the user's experience with the language server's features and the overall experience of using it. This feedback allows us to assess the actual usefulness of the features from the users' perspective and identify areas for improvement in future iterations of the language server. This study was conducted with a small sample size of six students, which reflects the limited number of individuals currently familiar with ECMA-SL. Although the group is small, it includes both MSc and PhD students who are or were part of the ECMA-SL development team (two PhD students and four MSc students).

Through the survey, we were able to gather valuable user feedback on ESL-LS. To start the survey, we asked a question to gauge the users' familiarity with ECMA-SL. While we know all participants can program in the language, their comfort levels may vary. The question provided was:

- How familiar are you with ECMA-SL?

This question was answered using a scale from one to four, where one represents ”Not familiar” and four represents ”Very comfortable”.

The survey also included both range-based questions, designed to measure users' satisfaction with each specific feature and overall experience with ESL-LS, as well as open-ended questions, aimed at understanding what users feel is missing and what could be added or improved in future updates. The

range-based questions were answered using a scale from one to four, where one represented ”Not useful” for individual features and ”Not satisfied” for ESL-LS overall, while four represented ”Very useful” for individual features and ”Very satisfied” for the general experience with ESL-LS. The range-based questions are as follows:

- Did you find using ESL-LS enjoyable?
- How useful is ESL-LS's hover feature for programming in ESL?
- How useful is ESL-LS's diagnostics feature for programming in ESL?
- How useful is ESL-LS's code completion feature for programming in ESL?
- How useful is ESL-LS's go-to-definition feature for programming in ESL?

The open-ended questions are the following:

- Would you change this feature? If yes, why?
- What features can be added to ESL-LS to improve the user experience?

The first open-ended question was asked individually for each feature to gather more specific insights on potential improvements.

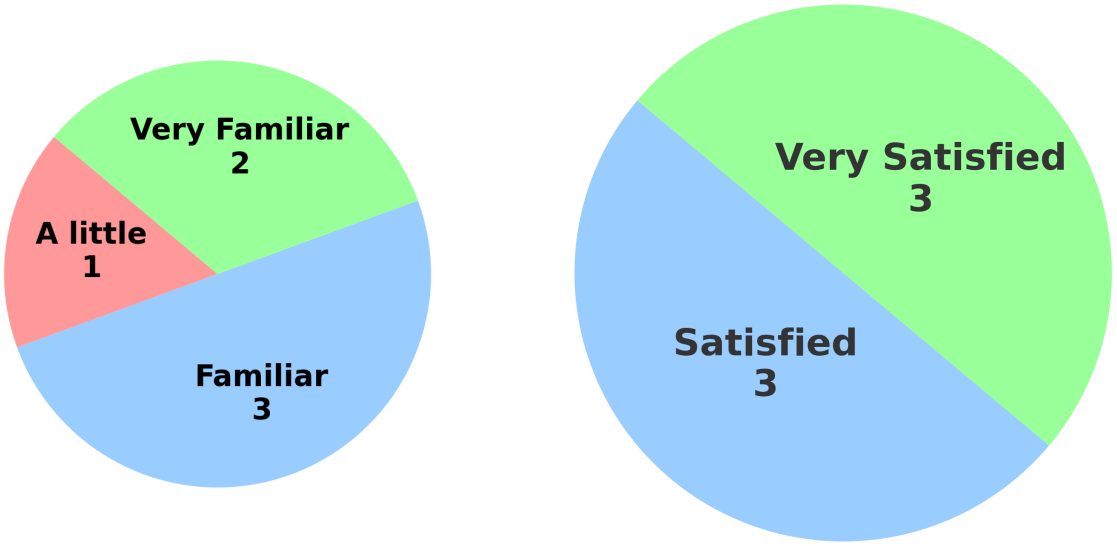
6.2.2 Results

As shown in Fig.6.1(a), most users are either familiar or very familiar with the ECMA-SL language, with only one user reporting little familiarity. Specifically, one user is a little familiar, three are familiar, and two are very familiar. This outcome aligns with our expectations, as the users have prior experience with the language, though their levels of comfort vary. Fig.6.1(b) indicates that three users were satisfied and three were very satisfied with ELS-LS, demonstrating strong appreciation for the language server.

6.2.2.A Specific Feedback

In the following, we present the specific feedback received for each feature.

- **Hover Feature:** Fig. 6.2(a) indicates positive feedback for this feature, with two users finding it useful, three considering it very useful, and one rating it as acceptable. One suggestion for improvement was to integrate the server ECMA-SL's static type system, which is currently not supporting the full syntax of the language;



(a) ECMA-SL Familiarity

(b) ESL-LS General Satisfaction

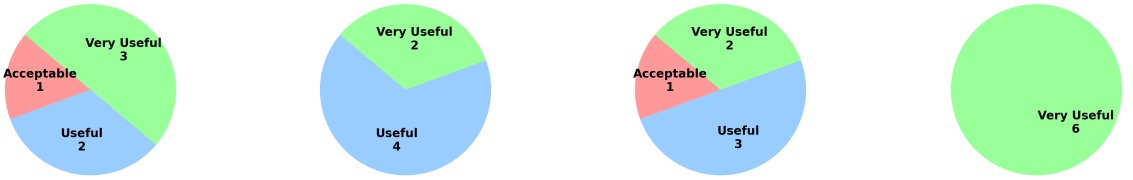
Figure 6.1: Familiarity and Satisfaction Charts

- **Diagnostics Feature:** Fig. 6.2(b) indicates positive feedback for this feature, with four users finding it useful and two considering it very useful. However, users suggested improving the feature by displaying more detailed error messages through integration with the static type system for ECMA-SL. Additionally, they recommended providing examples or hints to resolve common issues, such as missing semicolons;
- **Code Completion Feature:** Fig. 6.2(c) indicates positive feedback for this feature, with two users finding it useful, three considering it very useful, and one rating it as acceptable. Suggestions for improvement included adding context-awareness for the current word during code completion, adding completions for word properties, and a more subjective recommendation to avoid automatically completing function names with their signatures;
- **Go-to-definition Feature:** Fig. 6.2(d) indicates overwhelmingly positive feedback, with all the users finding it very useful. No suggestions for changes were provided.

6.2.2.B Overall Feedback

New features users suggested for future iterations of the language server to include:

- Integration with the static type system for the language; this improvement would enhance the functionality of existing features, allowing the language server to provide more accurate information



(a) Hover **(b)** Diagnostics **(c)** Code Completion **(d)** Go-to-definition

Figure 6.2: Features Usefulness Charts

and display better error messages within the code;

- Automatically generating code snippets for common statements, such as switch statements, to speed up development;
- Implement a feature that allows users to easily locate all references to a specific function throughout the code base, improving code navigation;
- Implement automatic code formatting, allowing consistent formatting style for every file;
- Implement real-time diagnostics. Unfortunately, due to the current implementation of our diagnostics feature, this is not feasible. The server requires the current file to be saved before executing the command to compile the file. Otherwise, it will only provide information based on the last saved version of the file, not the most recent changes.

The original feedback can be found in Appendix A.

6.2.3 Analysis

We can conclude that the users are satisfied with ESL-LS and find its features useful, being a valuable addition to the ECMA-SL language. This confirms one of our goals that was for our language server to be useful and improve efficiency while programming in ECMA-SL. Despite the positive feedback, the users shared various excellent suggestions to improve it, such as:

- Integration with the static type system;
- Autocompletion for common default statements;
- Implement find references feature;
- Implement automatic code formatting.

7

Conclusions and Future Work

Contents

7.1 Conclusions 61
7.2 Future Work 62

7.1 Conclusions

In this thesis, we introduced ESL-LS, a new language server for the ECMA-SL language, which supports the following features:

- *Language Configuration:* defines the behaviors of the language server, such as comment toggling and auto-closing pairs;
- *Syntax Highlighting:* defines how different code elements are visually distinguished by color and text style;
- *Go-to-definition:* allows quick navigation to the definition of the requested function;
- *Hover:* displays type information when hovering over a variable or shows the function signature when hovering over a function;

- *Code Completion:* provides suggestions for completing words as the user types;
- *Diagnostics:* provides visual warnings displayed in the editor for code errors.

We evaluated ESL-LS in the following two ways:

- *Performance Experiments:* tested the performance of ESL-LS by measuring its response times to all types of supported requests on three large complex ESL projects.
- *User Study:* conducted a survey with six ECMA-SL programmers to gather their feedback on the language server and its features.

Our performance experiments demonstrate that the server operates efficiently, with response times that are extremely low and unnoticeable to programmers. This is crucial, as the primary goal of this language server was to enhance the coding experience and boost productivity. Hence, introducing any noticeable delays would undermine this objective.

The user study gathered mostly positive feedback, with developers finding the language-specific features useful and considering ESL-LS an extremely valuable addition to the ECMA-SL ecosystem.

However, despite the good performance results and positive feedback, there is still room for improvement. Users suggested several enhancements for future iterations of the language server, which we plan to address to further improve the user's experience.

7.2 Future Work

In the future, we plan to enhance our language server by adding the following:

- **Integration with the static type system for the ECMA-SL language [8]:** This improvement would enhance the functionality of existing features, allowing the language server to provide more accurate information and display better error messages within the code. This is currently not possible because the system does not yet support the full syntax of the language.
- **Autocompletion for common default statements:** This feature would automatically complete default statements, streamlining the coding process.
- **Find references feature:** This feature would allow developers to locate all references to a function in the code, improving code navigation.
- **Automatic code formatting:** This feature would automatically format all code belonging to the same project to make sure the formatting style remains consistent across multiple files.

Bibliography

[1] E. Ecma, "262: Ecmascript language specification," *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,*, 1999.

[2] L. M. A. Loureiro, "ECMA-SL - A Platform for Specifying and Running the ECMAScript Standard," Master's thesis, Sep. 2021.

[3] R. R. Rahal, "ECMARef6: a reference interpreter for modern JavaScript ," Master's thesis, Jun. 2023.

[4] J. P. C. Brown, "A Reference Implementation of ES6 Built-in Libraries," Master's thesis, Nov. 2022.

[5] Ecma International, "ECMAScript Language Specification 6.0," 2015, accessed: November 26, 2024. [Online]. Available: https://262.ecma-international.org/6.0/

[6] D. M. S. Gonçalves, "A Reference Implementation of ECMAScript Built-in Objects," Master's thesis, Nov. 2021.

[7] Ecma International, "ECMAScript Language Specification 5.1," 2011, accessed: November 26, 2024. [Online]. Available: https://262.ecma-international.org/5.1/

[8] A. F. F. do Nascimento, "A Typed Intermediate Language for Specifying the ECMAScript Standard," Master's thesis, Nov. 2023.

[9] N. Gunasinghe and N. Marcus, *Implementing a Language Server*. Berkeley, CA: Apress, 2022, pp. 23–33. [Online]. Available: https://doi.org/10.1007/978-1-4842-7792-8_3

[10] Microsoft, "Language server protocol," https://microsoft.github.io/language-server-protocol/, accessed: November 26, 2024.

[11] M. Bodin, A. Charguéraud, D. Filaretti *et al.*, "Jscert: A mechanized specification of javascript," https://jscert.org/, 2023, accessed: November 26, 2024.

[12] A. Hidayat, "Esprima: Ecmascript parsing infrastructure for multipurpose analysis," https://esprima.org/, 2024, accessed: November 26, 2024.

[13] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and et al., “The ocaml system release 5.1: Documentation and user’s manual,” Inria, Intern report, 2023. [Online]. Available: https://hal.inria.fr/hal-00930213v10

[14] Microsoft, “Visual studio code,” https://code.visualstudio.com/, accessed: November 26, 2024.

[15] Sublime, “Sublime text,” https://www.sublimetext.com/, accessed: November 26, 2024.

[16] M. D. Nguyen, “The ides we had 30 years ago and what they taught us,” 2023, accessed: November 26, 2024. [Online]. Available: https://blogsystem5.substack.com/p/the-ides-we-had-30-years-ago-and

[17] B. Computing, “Notes on the eclipse plug-in architecture,” https://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html, 2003, accessed: November 26, 2024.

[18] E. Foundation, “Eclipse,” https://eclipseide.org/, accessed: November 26, 2024.

[19] JetBrains, “Intellij idea,” https://www.jetbrains.com/idea/, accessed: November 26, 2024.

[20] GitHub, “Github copilot: Your ai pair programmer,” https://github.com/features/copilot, accessed: November 26, 2024.

[21] C. Calcagno and D. Distefano, “Infer: An Automatic Program Verifier for Memory Safety of C Programs,” in *NASA Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 459–465.

[22] Q. L. Le, A. Raad, J. Villard, J. Berdine, D. Dreyer, and P. W. O’Hearn, “Finding real bugs in big programs with incorrectness logic,” *Proceedings of the ACM on programming languages*, vol. 6, no. OOPSLA1, apr 2022. [Online]. Available: https://doi.org/10.1145/3527325

[23] Pyrolitic, “Dafny vscode extension,” https://github.com/pyrolitic/dafny-vscode, accessed: November 26, 2024.

[24] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, “Frama-c: A software analysis perspective,” *Formal Aspects Comput.*, vol. 27, no. 3, pp. 573–609, 2015. [Online]. Available: https://doi.org/10.1007/s00165-014-0326-7

[25] Python, “Python programming language,” https://www.python.org/, accessed: November 26, 2024.

[26] Rust, “The rust programming language,” https://www.rust-lang.org/, accessed: November 26, 2024.

[27] ESLint, “Eslint: Find and fix problems in your javascript code,” https://eslint.org/, accessed: November 26, 2024.

[28] Pylint, “Pylint: Python static code analysis,” https://pypi.org/project/pylint/, accessed: November 26, 2024.

[29] GCC, “The gnu compiler collection (gcc),” https://gcc.gnu.org/, accessed: November 26, 2024.

[30] llvm, “The llvm compiler infrastructure,” https://llvm.org/, accessed: November 26, 2024.

[31] V. Developers, “Valgrind: Tool suite for debugging and profiling,” https://valgrind.org/, accessed: November 26, 2024.

[32] G. Developers, “Gdb: The gnu project debugger,” https://www.sourceware.org/gdb/, accessed: November 26, 2024.

[33] J. Fenlason and R. Stallman, “Gprof: Gnu profiler manual,” https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html, accessed: 2024-10-13.

[34] D. Naudziuniene, M. Botincan, D. Distefano, M. Dodds, R. Grigore, and M. J. Parkinson, “jstar-eclipse: an ide for automated verification of java programs,” 2011. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/ESEC-FSE.11.jstar-eclipse.pdf

[35] P. W. O’Hearn, J. C. Reynolds, and H. Yang, “Local reasoning about programs that alter data structures,” in *Proceedings of the 15th International Workshop on Computer Science Logic*, ser. CSL ’01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 1–19.

[36] K. R. M. Leino, “Dafny: An automatic program verifier for functional correctness,” Microsoft Research, Tech. Rep., 2008. [Online]. Available: https://www.microsoft.com/en-us/research/uploads/prod/2008/12/dafny_krml203.pdf

[37] J. Fragoso Santos, P. Maksimović, S.-E. Ayoun, and P. Gardner, “Gillian, part i: A multi-language platform for symbolic execution,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. Association for Computing Machinery, 2020.

[38] P. Maksimovic, S. Ayoun, J. F. Santos, and P. Gardner, “Gillian, part II: real-world verification for javascript and C,” in *Computer Aided Verification - 33rd International Conference, CAV 2021*, ser. Lecture Notes in Computer Science, vol. 12760. Springer, 2021, pp. 827–850.

[39] N. Karmios, S. Élie Ayoun, and P. Gardner, “Symbolic debugging with gillian,” in *Proceedings of the 1st ACM International Workshop on Future Debugging Techniques (DEBT 2023)*. New York, NY, USA: ACM, 2023, pp. 1–2. [Online]. Available: https://doi.org/10.1145/3605155.3605861

[40] A. Charguéraud, A. Schmitt, and T. Wood, “Jsexplain: A double debugger for javascript,” in *Companion Proceedings of the The Web Conference 2018*, ser. WWW ’18. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2018, p. 691–699. [Online]. Available: https://doi.org/10.1145/3184558.3185969

[41] D. Ciabrini and M. Serrano, “Bugloo: A source-level debugger for the scheme programming language,” INRIA, Tech. Rep., 2003, accessed: November 26, 2024. [Online]. Available: https://www-sop.inria.fr/mimosa/fp/Bugloo/doc/cia03.pdf

[42] L. Almeida, M. Gonzaga, J. F. Santos, and R. Abreu, “Rexstepper: A reference debugger for javascript regular expressions,” in *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE ’23. IEEE Press, 2023, p. 41–45. [Online]. Available: https://doi.org/10.1109/ICSE-Companion58688.2023.00021

[43] MacroMates. TextMate Language Grammar. Accessed: November 26, 2024. [Online]. Available: https://macromates.com/manual/en/language_grammars

[44] ——. TextMate Editor. Accessed: November 26, 2024. [Online]. Available: https://macromates.com/

[45] Microsoft. Language Configuration. Accessed: November 26, 2024. [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/extensibility/language-configuration?view=vs-2022

[46] “Test262: Official ecmascript conformance test suite,” https://github.com/tc39/test262, accessed: November 26, 2024.

[47] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994.

A

Survey Answers

The survey results are presented as follows: we begin with the specific feedback, followed by the overall feedback.

The specific feedback is the following:

- **Hover:** The feature currently lacks the ability to show important information about the code, in particular the types of variables/arguments. It will become incredibly useful once it's fully integrated with the static type system for ESL.
- **Diagnostics:**
  - Once again, there are lots of errors that we could show, were we able to integrate the LSP with the static type system. Other than that, everything looks good.
  - Maybe having some examples of how to fix the issue would be good. For example, maybe saying: Maybe a semi-colon is missing ';' would help the programmer find the issue quickly.
- **Code Completion:**
  - I fell like the code completion would not always give me the desired completions, specially when it came to variables previously defined on the same scope. For example, I could never get it to autocomplete the name defined within the 'foreach' statement guard.

  - The autocomplete of some functions could be better. For instance, choosing the function ”to_char_code” (defined in the stdlib) for completion, it results in ”to_char_code(s : string)”. It would be better to write ”to_char_code()” or just ”to_char_code”.
  - Maybe also allow code completion for known object properties. Although, this is potentially hard to.

The overall feedback is the following:

- The major thing missing from the LSP is integration with the static type system for the language (although this is not currently possible because that system is not working properly). Such improvement would better inform all the features above, as well as allow the LSP to show more and better errors within the code.
- Automatically generate code snippets for default statements, such as a switch statement.
- Find all references to a specific function.
- Would be good to have some form of type inference, so the developers could easily look at the argument types using the lsp’s code lenses to make sure the types match with the developer’s expectation.
- An option to find all references made to a function (highlight every line of code where that function is called).
- Automatic code formatting (like ocamlformat for example). Given a configuration file, automatically format all code to make sure the formatting style remains consistent across a multiple files.
- real-time syntax, type, or style error warnings while coding.