University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Szymon Dziuda
Student no. 438422

Jakub Kołaczyński
Student no. 438520

Jan Szot
Student no. 430165

Jan Wangrat
Student no. 440012

Language server for Duckling programming language

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
Janina Mincer-Daszkiewicz Ph.D.
University of Warsaw, Institute of Informatics

Warsaw, June 2024

Abstract

Duckling is a new programming language, currently under development. It composes the speed of compiled languages with the ease of use of script languages. This thesis introduces a Language Server for Duckling, aiming to improve the development experience for programmers. It includes descriptions of the implementation of key features, such as code completion and diagnostics. Our solution uses the Language Server Protocol for wide IDE compatibility with a focus on Visual Studio Code.

Keywords

Language Server, Language Server Protocol (LSP), Duckling, Integrated Development Environment (IDE), syntax highlighting, code completion

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Computer Science

Subject classification

CCS
→ Software and its engineering
→ Software notations and tools
→ Development frameworks and environments
→ Integrated and visual development environments

Tytuł pracy w języku polskim

Serwer językowy dla języka programowania Duckling

Contents

Chapter 1

Introduction

Duckling is a new programming language being developed by DuckType sp.z.o.o., with assistance from the staff of the University of Warsaw's Faculty of Mathematics, Informatics, and Mechanics (MIM UW). More information about Duckling can be found on DuckType's website `https://ducktype.org/`.

Duckling aims to combine the best features and practices of existing programming languages with the goal of creating a competitive option for modern programming. One of the main objectives of Duckling is to combine the speed of compiled languages with the advantages of interpreted languages, such as a convenient and powerful debugger and profiler. Another key feature of Duckling is the virtual machine capable of deterministic memory leak detection. Programming languages with similar features are currently being worked on, but none of them is in the production-ready state.

Although Duckling is still in development, some features of the language are already implemented. This includes parts of the compiler, a prototype of the virtual machine, and the type system.

A crucial part of utilizing any programming language is the presence of specialized tools designed to assist programmers. Some of the most important of these tools are Integrated Development Environments (IDEs). They combine a multitude of functionalities, such as a code editor, project manager, and code compiler, to aid developers in their tasks.

The Language Server (LS) is a part of the IDE responsible for connecting a text editor with a language compiler or interpreter. LS implements various functionalities, for example code completion and error detection. It is responsible for real time reporting of programmers' mistakes and it helps them speed up their workflow.

Modern Language Servers are often implemented using Microsoft's Language Server Protocol (LSP) [Mic23b]. This protocol splits LS into two applications: a server and a client. The server is meant to handle all logic necessary to interpret the compiler's output. All clients act as interfaces to the given IDE. It is crucial to point out that the server for a given programming language has to be implemented only once and a client is meant to be a small, easily implementable application.

The goal of this thesis is to implement the Duckling Language Server (DucklingLS) for Visual Studio Code (VSCode). Adhering to LSP lays the groundwork for the potential integration of Duckling with other IDEs. Our solution will leverage existing parts of the Duckling language, such as lexer, interpreter, and compiler. We aim to implement key functionalities of LS, such as: syntax highlighting, error highlighting, code auto-completion, and go-to definition.

Duckling programming language features are introduced in Chapter 2. A detailed description of the inner workings of the protocol and the responsibilities of each application can be found in Chapter 3. An overview of existing language servers can be found in Chapter 4. Details and design choices of our language server are in Chapter 5. Features implemented as a part of this thesis are described in detail in Chapter 6. The results of our work are shown in Chapter 7. Conclusions from the whole project are presented in Chapter 8.

Chapter 2

Duckling

This chapter provides an overview on the Duckling programming language. We describe motivation behind the language's creation, highlight the most important goals and assumptions of the Duckling project, and provide some insights into the traits and features of the language. Later on, we show already existing parts of the Duckling language, such as lexer, parser, and virtual machine.

2.1. Motivation

The main inspiration for the Duckling language comes from the shortcomings of C++. Most issues of C++ originate from its age and heritage. C++ is a direct derivative of C, thus it inherits a lot of its features. The syntax of both of these languages is quite simple and well-known all around the world. Programs written in C and C++ come with little to no overhead. They perform well when it comes to speed and low-level programming. With that being said, both C and C++ suffer greatly from half-century-old design choices and backward compatibility requirements. One of the most glaring issues is memory safety.

On the other end of the modern programming spectrum stands Python. Coding in Python is quite intuitive and it has an expansive package library, which made it become an industry standard for startups. In some fields, such as machine learning, it has achieved a near monopoly-level in the market. Although Python originated as a script language, it is now being stretched far over its originally planned capabilities. Some of the most notorious problems with Python are the lack of native support for multi-processing, a lacking implementation of the object-oriented paradigm, and its non-satisfactory speed of code execution.

The creators of Duckling recognize the potential for improvement in the current programming language landscape. The market lacks a modern, fast, low-level language that is both easy to learn and use. Duckling is designed to fill this gap.

Rust is currently the greatest competitor for Duckling. It is an example of a modern programming language aiming to replace parts of C++ usage. It provides some interesting features related to memory and parallel execution safety. That being said, Rust is not the one and only true solution to C++ problems, and Duckling chose a different approach to many of those issues. There are also languages like Carbon which aim to fill a similar niche, but most of them are still in development stage.

2.2. Features

The main focus of Duckling is imperative programming with some support for the functional paradigm, with a high emphasis on memory safety and genericity. Exact features and solutions implemented in Duckling are described in the following subsections. Some of them might be subject to change, as Duckling is still in development. This section aims to provide the reader with general overview of what Duckling aims to achieve. Example code for calculating n-th element of Fibonacci sequence can be seen in Listing 2.1. Important feature of Duckling language is PondVM. It's omitted in this section and described in greater detail in Section 2.3.

```
/**
 * Program calculating n-th
 * fibonacci number in~O(n)
 */

include std.io;

fun main() {
    n := io.getint(); // n-th fibb number
    a := 0;
    b := 1;
    for (i : 0..n) {
        a, b = b, a + b;
    }
    io.output(b);
}
```

Listing 2.1: Duckling example (source: DuckType internal resources)

Imperative Programming

There exists a multitude of approaches that can be taken to implement an imperative language. One might look for differences between C and Python. Duckling's approach focuses on preventing programmers' mistakes.

Important class of features related to imperative programming is management of global variables. It is often difficult to manage where or when they are accessed or modified. Duckling proposes two mechanisms to improve this aspect of global variables. First one is explicit marking of which variables are accessible from a given function. Another is limiting global variables' scope to modules or name spaces.

Some other features that aim to make Duckling less bug prone are: explicit marking of functions, which are able to throw exceptions, pre-execution and post-execution conditions for functions, immutable variables and value constraints.

Functional Paradigm

Although Duckling is not designed as a functional language, it has some features which are usually associated with functional programming. One of those features is an idea of pure functions, which helps programmers determine which functions might have side effects. Apart from that, Duckling leverages its algebraic types to create pattern matching system similar to the one found in OCaml. Patterns might consist of both values and types of variables. An example of this kind of pattern matching can be seen in Listing 2.2. Inspirations from functional programming are meant to enable users to create generic code with relative ease.

```
val x: A|B = ...;
match (x) {
    case (a: A) {...}
    ...
}
```

Listing 2.2: Duckling pattern matching (source: DuckType internal resources)

Type System

One of the main advantages of Duckling is its type system and genericity. Duckling uses algebraic types known from Haskell. This feature is to be mixed with high genericity to achieve great power of expression. Genericity is inspired by C++, but redesigned to be overall easier to use.

Aside from that, Duckling contains some elements of object-oriented programming. This includes classes with hierarchical inheritance and interfaces with hybrid inheritance. Classes do not implement multiple inheritance.

Package System

Another important feature of Duckling is its package system. High modularity is one of the main reasons for Python's success. Duckling aims to create a good basic package system with package versioning. Such features are taken for granted nowadays but it is important to mark that there is no such system in C++. On top of that Duckling aims to leverage its high genericity to create parameterized packages.

Position-independent Code

In position-dependent code, the declaration and the definition often have to be split apart. It's rarely a useful feature and usually just an obstacle. As seen in Listing 2.3 in Duckling, both functions and constant variables are position-independent.

```
fun foo() {
    goo();
}

fun goo() {...}

const TWO := ONE + ONE;
const ONE := 1;
```

Listing 2.3: Position-independent code example in Duckling

2.3. Existing functionalities

Certain aspects of the language, like PondVM, are already implemented, at least in a prototype version. Some, like Type System, have well defined specification, but are awaiting implementation. Meanwhile, others such as the Package System and the compiler are currently in development. In this section, our focus will be on introducing the existing components of the compiler, especially those needed for communication with the Language Server.

PondVM

Duckling source code is compiled to Duckling Byte Code (DucklingBC — declarations and program overview) and DucklingASM (functions' bodies), which are then executed on custom Virtual Machine (PondVM [CPRW23], the name may change in the future). Main feature differentiating PondVM from similar virtual machines, such as Oracle JVM, is its capability to detect memory errors. Most solutions designed to tackle memory related issues, such as Valgrind, use combination of deterministic methods and heuristics. PondVM uses entirely deterministic approach. Being designed from the beginning with this functionality in mind, it's more capable than Valgrind. Memory safety is one of the key features of the Duckling language, and so the PondVM is a crucial part of the entire language ecosystem.

PondVM is also responsible for providing debugger functionalities. Debugger interface is implemented using HTTP protocol. This interface doubles as DucklingBC language server.

As of time of writing this thesis, PondVM's speed is comparable to Java Virtual Machine or C++ with Valgrind, see Figure 2.1. With that being said, optimization of PondVM is one of currently ongoing projects.

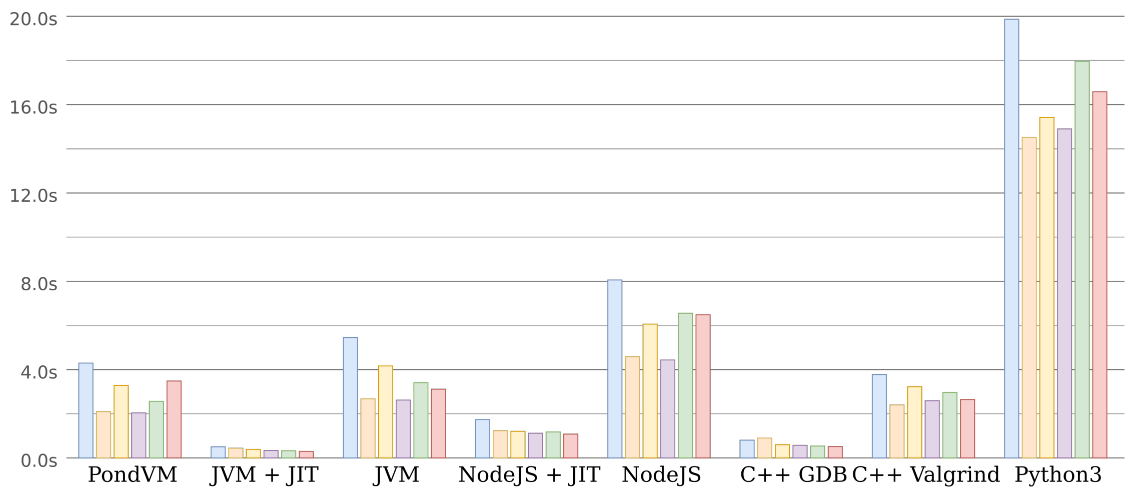


Figure 2.1: Average time of execution for different virtual machines (source: DuckType internal resources)

Lexer

A lexer is an essential component in the compilation process. It scans the source code, breaking it into tokens — meaningful units such as keywords, identifiers, and operators. The lexer's role is to recognize and categorize these tokens.

Duckling's lexer, distinguishing parenthesization, can generate a token tree. It is a rather non-standard solution, compared to the conventional list-based approach. It is inspired by Rust's compiler design.

Duckling's lexer has support for the Unicode Standard. It means that there will be no problem with naming a variable "α" or creating an operator denoted as "$\rightarrow$".

Parser

A parser is another crucial part of the compilation process. It takes the stream of tokens generated by the lexer and arranges them according to the rules of the Duckling's grammar. It then creates a parse syntax tree (PST) as a hierarchical representation of the program's structure. In Figure 2.2 you can see an example of a PST generated based on code from Listing 2.4.

```
@ATRR
fun c(){
    x = 3;
    y = 4;
}
```

Listing 2.4: Example code used for PST generation

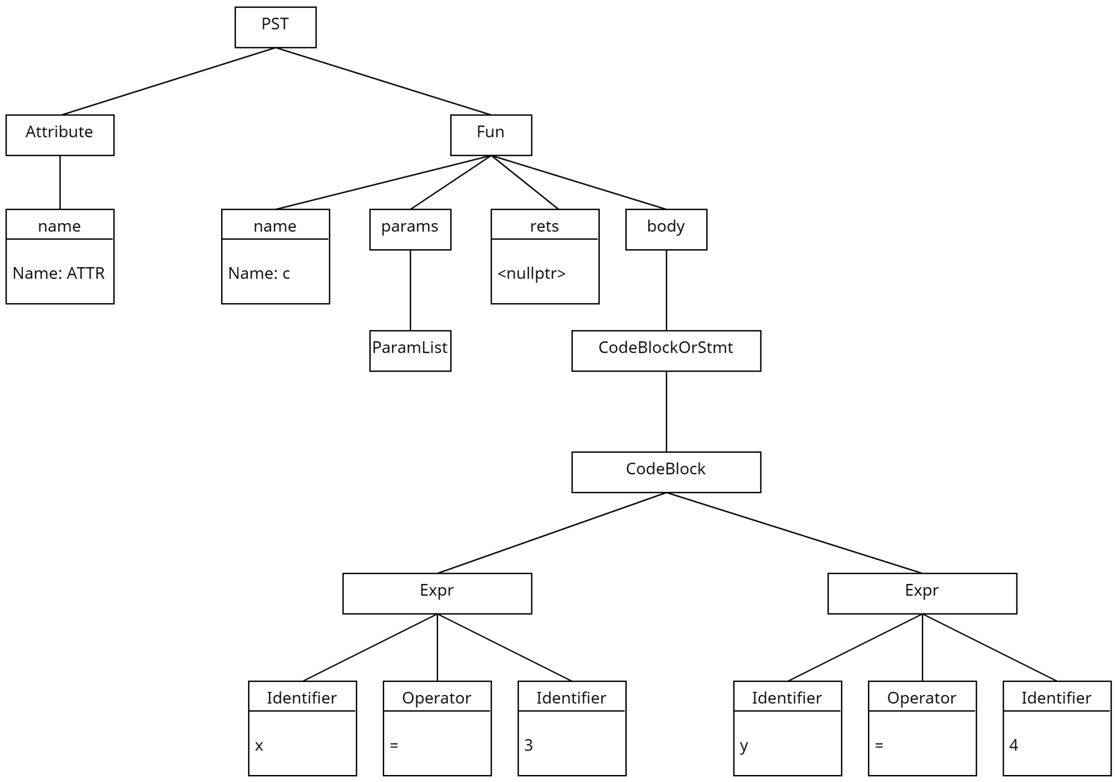


Figure 2.2: PST example (source: DuckType internal resources)

Duckling's parser is a Recursive Descent Parser written from scratch. It leverages a top-down parsing technique to analyze the syntax. It's called ”recursive descent” because the parsing process involves recursively calling functions to handle different parts of the grammar.

Duckling's goal is to make its language grammar as close as possible to a deterministic context-free grammar. This approach makes parsing significantly simpler.

Chapter 3

Language Server Protocol

In this chapter, we introduce the concept of the Language Server Protocol (LSP), highlighting the reasons that led to its inception and the objectives and principles guiding its development. We explain how communication between the server and the client works and take a closer look at possible features LSP can provide.

3.1. Historical solutions

Before the invention of the LSP, there was no standardized method for handling language intelligence — each IDE handled it on its own. Due to the huge amount of work required to develop a language-supporting plugin, only the most popular languages could offer wide IDE support. The plugins were not compatible between platforms, which meant that for each IDE we had to basically build language support from scratch. This lack of uniformity often meant that less common languages were left with limited or no support in many development environments. It all changed in 2016 when Microsoft introduced the LSP — a common protocol that language tooling could use to communicate with an IDE. Some of the historically used solutions are listed below.

Dedicated IDEs

Dedicated IDEs are very powerful and provide extensive language support. That being said, they are also hard and expensive to produce, so almost only huge projects like Java (IntelliJ IDEA, Eclipse) or C# (Microsoft Visual Studio) can afford to create one.

Dedicated plugins

Plugins created specifically for a given IDE provide decent language support and at the same time are not as time-consuming to develop as entire IDEs. Unfortunately, they are not very scalable and cannot be easily repurposed across many different IDEs. Also, when developing a plugin in a language different from the one used to develop the IDE, you could encounter many problems trying to combine the two languages (e.g. developers encountered some difficulties while adding Haskell support to Eclipse, which is written in Java).

Monto

Monto is a solution similar to LSP in its principles. Like LSP, it also aims to help to create a more scalable and extensive language support for different IDEs. In an overview, Monto consists of an IDE plugin (refered to as a Client in LSP) communicating with a Broker (Server in LSP), which delegates tasks to the language tools (like a lexer or parser). However, its protocol is less concise and more strict than that of LSP. It also lacks support from Microsoft, the provider of one of the currently most-used IDEs, VSCode.

3.2. Communication

The main assumption of a language support using LSP is to split the work required for providing language intelligence features into two parts. The first part is the server, which must understand the entire language, perform all the complex calculations, and provide all the necessary information. The second, much smaller part, the client, is typically an IDE plugin, which role is to communicate with the server via the Language Server Protocol using JSON-RPC.

JSON-RPC

JSON-RPC is a protocol that allows for remotely calling procedures. Its idea is to be as simple and universal as possible, it supports communication within the same process, over sockets, over HTTP, or in many other message-passing environments. There are two types of communications — **requests** (e.g., when a user calls ”Go to definition”, like in Listing 3.1), which require a response (example response can be seen in Listing 3.2), and **notifications** (e.g., when client wants to inform that a user has renamed a file), which do not receive a response. It uses the JSON format to exchange data. The message from the client to the server consists of the following fields:

- **jsonrpc** — its value must be exactly 2.0, as JSON-RPC 1.0 did not have this field.
- **id** — a unique ID of the request. It can be *null* (or omitted), in which case the message is a notification and cannot be responded to.
- **method** — a string informing which method is to be invoked.
- **params** — parameters required for the called method. Most often passed as a JSON object with names matching the ones in the function definition. It can be omitted if the function does not require any arguments.

The response from the server to the client has a similar structure:

- **jsonrpc** — its value must be exactly 2.0, same as previously.
- **id** — ID matching the one from the request. It must be *null* if there was an error while parsing the message from the client.
- **result** — result of the function invoked by the request. It must be omitted if an error occurred.
- **error** — object containing an error that happened either during parsing the request or the execution of the invoked function. It must be omitted if no errors occurred.

```
{
    "jsonrpc": "2.0",
    "id" : 1,
    "method": "textDocument/definition",
    "params": {
        "textDocument": {
            "uri": "file:///p%3A/mseng/VSCode/Playgrounds/cpp/use.cpp"
        },
        "position": {
            "line": 3,
            "character": 12
        }
    }
}
```

Listing 3.1: Example of JSON-RPC request (source: [Mic23b])

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "uri": "file:///p%3A/mseng/VSCode/Playgrounds/cpp/provide.cpp",
        "range": {
            "start": {
                "line": 0,
                "character": 4
            },
            "end": {
                "line": 0,
                "character": 11
            }
        }
    }
}
```

Listing 3.2: Example of JSON-RPC response (source: [Mic23b])

Client and Server

The communication between the Client (Development Tool) and the Server (Language Server) begins by exchanging information about capabilities implemented by both sides. As a result, the intersection of these two sets is taken as supported capabilities.

The Client can then start sending requests and notifications. The Server should respond to requests in roughly the same order as they are received, but this is not required (the Server can handle any level of concurrency; it will not be a problem because the responses are matched to requests by ID anyway). for an example of how LSP communication between the Client and the Server looks, please see Figure 3.1.

The shutdown procedures begin when the Client sends the shutdown request to the Server. The Server should then respond, acknowledging the request. The Client can then finally terminate the communication by sending the exit request.

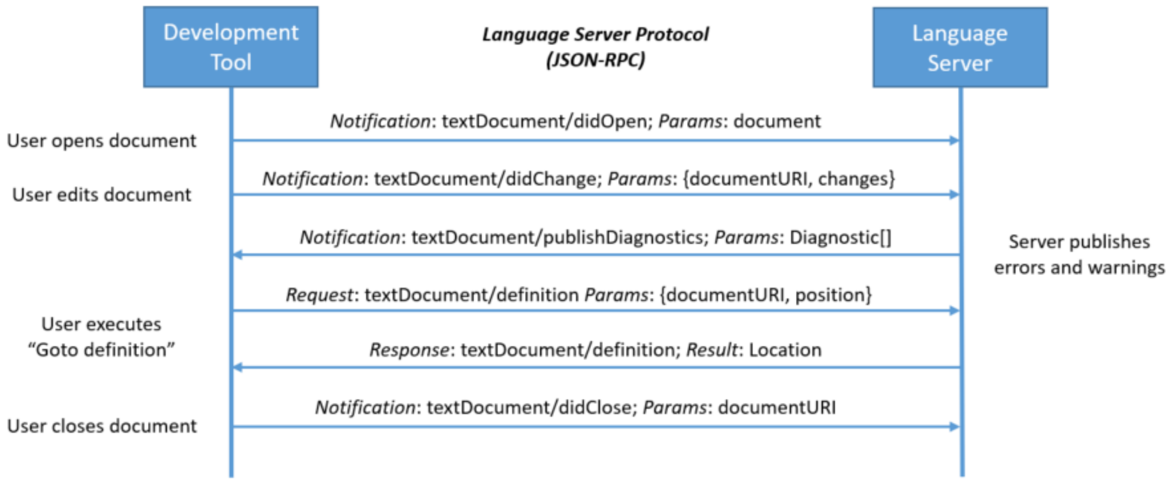

Development Tool
Language Server Protocol (JSON-RPC)
Language Server
User opens document
Notification: textDocument/didOpen; Params: document
User edits document
Notification: textDocument/didChange; Params: {documentURI, changes}
Notification: textDocument/publishDiagnostics; Params: Diagnostic[]
Server publishes errors and warnings
Request: textDocument/definition Params: {documentURI, position}
User executes "Goto definition"
Response: textDocument/definition; Result: Location
User closes document
Notification: textDocument/didClose; Params: documentURI

Figure 3.1: LSP communication (source: [Mic23b])

3.3. Capabilities

LSP supports a great number of capabilities. All of them are available in its specification. In this thesis, we focus on the following features:

- **Semantic Tokens** — syntax highlighting.
- **Document Diagnostics** — finding errors in files and highlighting them. Also useful for marking unused code.
- **Goto Definition** — jumping to the definition of a symbol.
- **Find References** — resolving all references to a symbol.
- **Completion** — providing auto-completion suggestions to facilitate writing.
- **Folding Range** — folding parts of the code (e.g. imports or code blocks).

Chapter 4

Existing Language Servers Overview

In this chapter, we closely examine the implementation of various existing language servers, highlighting aspects that are important when making decisions about the Duckling Language Server. We discuss key features and architectural choices in these servers that have influenced the development of the DucklingLS. This comparison is highly influenced by the one created by Yannik Sander during his work on implementing LSP for the Nickel Language [San22].

4.1. rust-analyzer

rust-analyzer [SMSf] is an implementation of LSP for the Rust programming language [Foub]. It is one of a few examples of a server written in the language it supports. Such an approach enables rust-analyzer developers to freely use all functionalities of the compiler without the need to use the foreign function interface or serialization of data, which are required when using functionalities written in languages different from the one used in the language server implementation.

This approach makes it impossible to use pre-existing tool kits, such as LSP4J [Cod] written in Java or VSCode Language Server [Mic] written in TypeScript. We cannot follow this path because, at the time of writing this thesis, Duckling does not exist yet.

Another interesting feature of rust-analyzer is its in-memory virtual file system. rust-analyzer does not compile the project based on the files stored in user's file system. This approach enables Rust Analyzer to store a copy of the project in memory and compile files without saving them. This approach requires less I/O operations, and makes the entire process quicker. Moreover, language server and parts of compiler responsible for code intelligence can be placed inside a single process. They can use a single instance of Rust's virtual file system and do not require additional synchronization.

4.2. rnix-lsp

rnix-lsp [Com] is a language server for Nix [Foua]. It is an example of a server loosely coupled with an interpreter. rnix-lsp implements its own version of a Nix parser written in Rust, instead of using parts of an existing interpreter written in C++.

A custom parser allows developers to easily generate and present all the information necessary to implement language server functionalities. With that being said, rnix-lsp does not implement the entire Nix interpreter, so information about the code is limited to static analysis.

The main problem of this approach comes from the duplication of parser logic. In case of bugs, the Nix interpreter and the rnix-lsp parser might interpret the same code in two different ways. This might lead to confusing errors, in which code is displayed in one way and computed in another. In DucklingLS's case, some parts of the language are still subject to change. Implementing any parsing server-side would force an additional workload onto the DuckType team later in the development.

4.3. CodeCompass

CodeCompass [Sou] presents an interesting use case for LSP. It is a code analyzer and diagnostic tool that acts as a plugin for an IDE. The server part in LSP architecture acts only as a proxy to an HTTP server on the internet. Code analysis and diagnostics are computed online and displayed in a web browser. CodeCompass shows that LSP is quite flexible in its specification and capabilities.

4.4. Ansible Language Server

The Ansible Language Server [Hat] is an example of a small project implementing LSP. It gets information about the code by running an executable and receiving the response through standard streams. This example was important for the project, as I/O was one of the candidates for the communication channel with the Duckling compiler.

4.5. Conclusions

Review of existing language servers highlighted a number of issues related to communication with the compiler.

The method of communication with the compiler is dictated by the server's architecture. It can range anywhere from incorporating parts of the compiler into the server, running binaries or CLI (Command Line Interface) based approaches, to re-implementing language logic entirely. The choice of method is closely related to the choice of language, since approaches involving a single process necessitate the use of a single language. Caching and incremental compilation are difficult in methods which do not rely on a single language.

One must address the fact that the file visible to a developer in the IDE, and file stored on the hard drive, are often different. This requires some kind of synchronization between IDE workspace and compiler. If the server and the compiler are written in the same language (therefore being able to work as a single application) the problem is trivial. On the other hand, when using binaries, state of the workspace visible to the user has to be somehow passed to the compiler on every request.

Chapter 5

Duckling Language Server

As pointed out in previous sections, LSP comes with many advantages over all the other solutions so we decided to implement language intelligence to the Duckling using this way. This chapter covers the main aspects and challenges associated with developing the DucklingLS, such as the selection of programming languages and the preferred IDE. It outlines the project's primary objectives, such as designing the architecture and developing various language features. Additionally, the chapter addresses the technical issues encountered during the implementation phase, such as communication with the parser and the reasons behind choosing specific technologies and methodologies for the DucklingLS's development.

5.1. TypeScript

TypeScript is by far the most common programming language used in Language Servers, especially because it comes with predefined functions and support provided by its creators, Microsoft, for implementing language servers in accordance with the LSP. It is further facilitated by its seamless integration with IDEs like VSCode. As of now, 21.63% of registered LS implementations are written in TypeScript, 9.39% in Java, and another 9.39% in Rust (as shown in Figure 5.1).

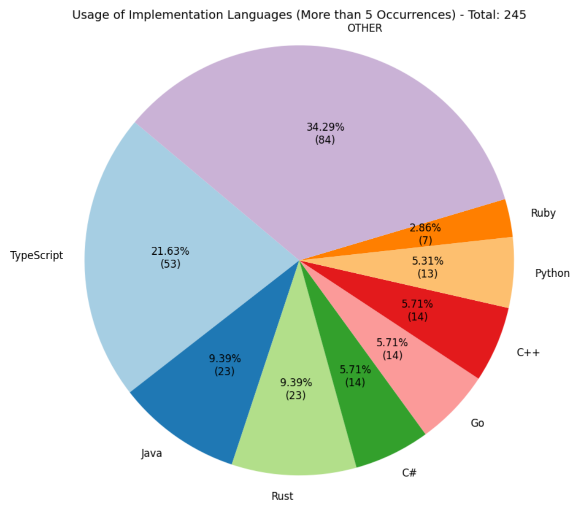


Figure 5.1: LSP languages (source: [Mic24d])

The variety of languages used to write LS is quite large because some developers decide to write the language servers in the same language as the programming language itself. We also considered C++, because Duckling's compiler is written in it.

5.1.1. TypeScript or C++

Both TypeScript and C++ are valid choices for the language in which the server is written, however we needed to settle on one.

TypeScript is user-friendly and integrates seamlessly with VSCode, making it the predominant choice for extensions. Its extensive pre-defined modules simplify the development of language servers. Despite being slower, TypeScript's performance is sufficient for language server tasks where slight delays in diagnostics are permissible.

Using C++ presents more challenges due to its low-level nature, and it provides limited support for language servers, lacking the range of pre-defined modules available in TypeScript. Although C++ is highly efficient, this performance advantage is not by any means crucial for language server development. Establishing communication between the Server written in C++ and the Client written in TypeScript is a complex task. It requires implementing the protocol layer of LSP in C++. Regardless of the language choice, Duckling's compiler is written in C++. Creating an interface to communicate with it, requires writing some C++ code anyways.

This comparison explains the reasons for choosing TypeScript over C++ for the development of the Language Server, especially due to our limited resources and because efficiency is not a significant challenge in Language Servers.

5.2. Visual Studio Code

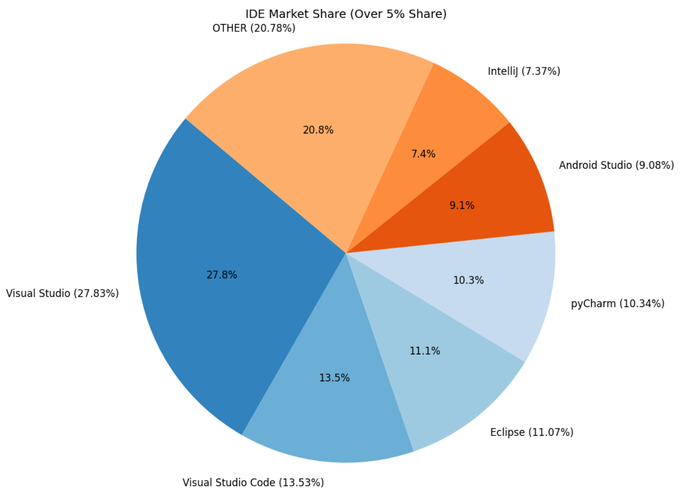


Figure 5.2: IDE usage (source: [PYP24])

As shown in Figure 5.2, VSCode (13,53% share) is the second most popular IDE, following Visual Studio (27,83% share). Popular as it is, Visual Studio is utilized for the development of .NET applications and does not support community extensions, which makes VSCode the most popular general usage IDE. Additionally, the Language Server Protocol itself was specifically developed for this IDE — it was created by Microsoft, which also owns VSCode. The broad support for LSP, its popularity, extensibility, and ease of development, along with it being suggested by the DuckType team, made VSCode an obvious choice for implementing the language client.

5.3. Communication with the client

This part of the communication is fully covered by the `VSCode-languageserver` [Mic] framework provided by Microsoft, therefore we will not go over in-depth details of this part. Our implementation is based on the *Language Server Extension Guide* [Mic23a]. In this segment, the key takeaway is the acquisition of the `connection` structure, which greatly simplifies the client communication.

5.4. Communication with the compiler

One of the main challenges encountered during the planning of the server architecture was integration with the language logic components, such as the parser or the compiler. As mentioned in Section 5.1.1, the server is written in TypeScript, and all other components are written in C++. We experimented with a number of different approaches.

5.4.1. Foreign Function Interface

Initially, we wanted to use a Foreign Function Interface (FFI) because it provides an elegant solution for calling C++ functions directly from TypeScript (it also does not require a lot of coding as all that needs to be done is to declare the functions' types in TypeScript).

However, this creates a need to compile C++ code into a shared library. It introduces a very specific issue for Linux users — when compiling C++ into a shared library, the compiler uses the standard library from `/usr/bin/`; therefore, if you have VSCode installed from Snap [Sna], it will not be able to access the standard library, thus breaking the program.

5.4.2. Executable compiler interface

This approach was used in the Ansible Language Server, as mentioned previously in Chapter 4. In simple applications, it can be a sufficient method of communication with a compiler. We used it in early stages of the project.

This method makes it difficult to implement any kind of caching system. The executable responsible for communication with the compiler can be treated as a pure function. It receives source code as an input, and provides language intelligence as an output. It does not store anything between calls, making it impossible to implement incremental compilation and forcing any caching to be done in the server.

In our initial implementation, the executable took file path as a parameter and returned `LSPTree` in `JSON` format and code errors. The `LSPTree`, which is further expanded upon in Section 5.5, enabled us to implement *Semantic Tokens* (responsible for syntax highlighting), partial *Completions* and *Folding Ranges* LSP functionalities. Code errors enabled us to implement *Diagnostics* functionality.

5.4.3. REST API

We explored the possibility of launching a REST API server daemon that would contain the language logic components. In this case, DucklingLS would be the client (of this server;

we need to remember that DucklingLS is also a server, with the IDE as its client). This architecture design is shown in Figure 5.3.

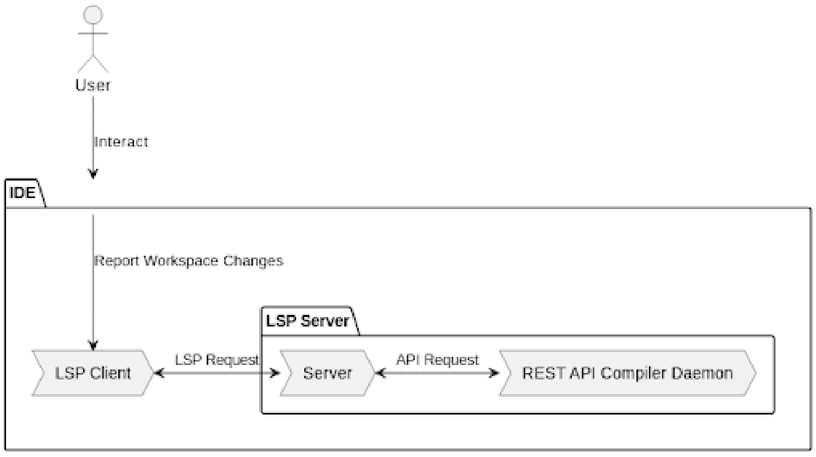


Figure 5.3: DucklingLS architecture design

This approach enables caching inside the language logic components, as the daemon is a standalone process that is running for the whole time in the background. Therefore, caching is not required inside the DucklingLS itself. The underlying communication between DucklingLS and the language logic components does not change much — the payload is still the same (as in the previous approach), only the method of communication changes.

However, as of the time of writing this thesis, the daemon communicates with DucklingLS using host's internal networking. This is not a perfect approach, as bootstrapping communication between two processes can sometimes be problematic. Choosing a free port and ensuring that no other processes write to it are the two main issues we encountered. It is possible to fix these problems by launching both DucklingLS and the Compiler Daemon inside a Docker [Inc] container.

5.5. LSPTree

Duckling's compiler goes through several steps in order to compile the input file. One of these steps results in what Duckling calls a PST. PST (implemented in C++) is a tree–like class structure, that represents the logical layout of the code. The nodes of the tree are called elements. During a meeting with Duckling's development team we decided on using the PST structure to generate a model of the parsed file in our server. An example of a PST generated from a snippet of code is shown in Figure 2.2.

Functionalities required to implement LSP features are centered around class structure called `LSPTree`. It is an enriched derivative of Duckling's PST. As seen in Figure 2.2, it is a tree-like structure, that represents logical layout of code. Compared to elements present in the PST, each `LSPTree` element contains information about its position in the file, and, in case of identifiers and declarations, some additional details required to differentiate between identifiers with the same name. `LSPTree` is JSON serializable, so it can be easily transferred between Compiler Daemon and LSP Server.

Effectively copying the entire structure of PST and enhancing it by adding some properties proved to be a challenge. PST was implemented by DuckType team in such a way that the first two attempts at implementing `LSPTree` failed. Main difficulties originated from the nature of C++ objectivity implementation, and its strict type system.

Let's take a `Function` declaration as an example. `Function` inherits from `Declaration`, which in turn inherits from `Statement`. There are examples of translation from PST to `LSPTree`, where `Function` is sometimes stored as an instance of `Declaration` and sometimes stored as an instance of `Statement`. Code standards in DuckLing compiler are such, that generators should return `std::unique_pointer` to an object. As `std::unique_pointer` is a template, and in C++ templates are not contra-variant, it turns out that `std::unique_pointer<Function>` does not inherit from `std::unique_pointer<Declaration>`. This means, that you cannot override method which returns `std::unique_pointer<Declaration>` with method which returns `std::unique_pointer<Function>`. You cannot just make `PSTFunction` always translate to `LSPDeclaration`, as there are instances, where class structure expects an `LSPStatement`.

Solution that we came up with was to create a ladder system of generators. This ladder system recursively calls implementations of more and more specific classes, and automatically casts them to less and less specific classes while returning the `std::unique_pointer`.

We decided against expanding PST to contain required data. Information required to fill this data is created in later stages of compilation. However, to get to the later stages of compilation, PST needs to be created. Backtracking in order to fill the gaps in required data seemed like a wrong design choice. If the parser was ever to be changed, this separation would ensure that the server's functionality was unaffected. The purpose of the `LSPTree` is, despite its similarity to the PST, quite different — the former enables the server to answer the user's (or VSCode's) LSP-oriented requests, while the latter is created as a way to tell the compiler how the code is structured.

5.6. Caching

Creating the `LSPTree` is time-consuming, so we aim to do it as infrequently as possible. For this reason, the `LSPTree` is only constructed after a change to files content is detected. `LSPTree` is then saved in a structure that maps the `documentURI` to the `LSPTree`. The same goes for code errors returned by the parser — a file is only diagnosed after a change, the results are then parsed and cached in a structure that maps `documentURI` to a list of `DucklingParserError` (this error structure is further described in Section 6.2). Consequently, all methods utilize this saved, ready-to-use structure, which not only reduces the number of calculations performed but also enables immediate responses to client's requests.

Chapter 6

Implemented Features

This chapter provides an overview of the features that have been successfully implemented in the DucklingLS. Each section within this chapter will cover functionalities that enhance the coding experience, explaining how these features operate, and the technological choices that made them possible.

6.1. Syntax Highlighting

One of the main features users expect from an IDE is code colorization, which not only improves readability but also aids in identifying variables, functions, and other code elements through visual differentiation. In developing DucklingLS, we chose to implement this feature first, mainly due to its complexity, as it needed communication with the parser and the whole PST structure to determine the exact position and classification of each token.

6.1.1. Semantic Tokens

The way of implementing syntax highlighting introduced by LSP is by using semantic tokens [Mic24f]. When a client requests semantic tokens for a specific file, these tokens are then used by the IDE to apply colorization. The Duckling Server employs a set of predefined Types (Listing 6.1) and Modifiers (Listing 6.2) to facilitate this process, but it is possible to add and modify them.

To establish communication between the server and the client, the server shares its `SemanticTokensLegend` (Listing 6.3), an index of token types and modifiers. This system uses numeric encoding for tokens to compress the transmitted data, particularly as responses for semantic tokens requests tend to be large.

Token numbering begins at zero; hence, `tokenTypes:0` corresponds to `namespace`. Multiple modifiers can apply to a single token and are represented using bitflags. Therefore, for example, `tokenModifiers:0` (00000_2) signifies the absence of any modifiers, whereas `tokenModifiers:3` (00011_2) indicates the application of the first two modifiers. The response for a semantic tokens request consists of a set of semantic tokens. Each token has its `deltaLine`, `startChar`, `length`, `tokenType`, and `tokenModifiers`. The `TokenFormat` setting provided by the server specifies whether the token positions (lines) are given in an absolute or relative manner, with the relative approach being the default due to its efficiency. The `VSCode-languageserver` package offers a `SemanticTokenBuilder`, which calculates relative positions and constructs the structure ready for the response.

```
export enum SemanticTokenTypes {
    namespace = 'namespace',
    type = 'type',
    class = 'class',
    enum = 'enum',
    interface = 'interface',
    struct = 'struct',
    typeParameter = 'typeParameter',
    parameter = 'parameter',
    variable = 'variable',
    property = 'property',
    enumMember = 'enumMember',
    event = 'event',
    function = 'function',
    method = 'method',
    macro = 'macro',
    keyword = 'keyword',
    modifier = 'modifier',
    comment = 'comment',
    string = 'string',
    number = 'number',
    regexp = 'regexp',
    operator = 'operator',
    decorator = 'decorator'
}
```

Listing 6.1: Predefined semantic token types [Mic24f]

```
export enum SemanticTokenModifiers {
    declaration = 'declaration',
    definition = 'definition',
    readonly = 'readonly',
    static = 'static',
    deprecated = 'deprecated',
    abstract = 'abstract',
    async = 'async',
    modification = 'modification',
    documentation = 'documentation',
    defaultLibrary = 'defaultLibrary'
}
```

Listing 6.2: Predefined semantic token modifiers [Mic24f]

```
const semanticTokensLegend = {
    tokenTypes: Object.values(SemanticTokenTypes),
    tokenModifiers: Object.values(SemanticTokenModifiers)
};
```

Listing 6.3: `SemanticTokensLegend` for Duckling

6.1.2. Implementation within `LSPTree`

Obtaining Semantic Tokens from an existing `LSPTree` is trivial, all we need to do is call `getSemanticTokens()` in the root of the tree. This function will go over the tree and flatten it to a list containing all of the tokens. The tokens are of type `Token` (as in Listing 6.4).

```
export interface Token {
    line: number;
    startCharacter: number;
    length: number;
    tokenType: number;
    tokenModifiers: number;
}
```

Listing 6.4: Token

There are two problems, though: as of the time of writing this thesis, Duckling's parser does not generate tokens for comments. This part is handled by us in DucklingLS itself. We manually detect comments by analyzing the source code in the server, generate tokens for them, and add these tokens to the result list.

The second problem arises from tokens consisting of two keywords, such as `do {...} while (...);` or `import lib as l;`. As of the time of writing, the Duckling compiler only stores information about the first keyword of a token. Coloring both keywords would require additional code analysis.

6.1.3. Sending Semantic Tokens to the Client

As shown in Listing 6.5, the `data` field is not really a list of `Token` elements. That's where the `VSCode-languageserver` package comes in handy once again. It provides an easy-to-use class, `SemanticTokensBuilder`, where we can push all our tokens and get them in the format required for the response.

```
export interface SemanticTokens {
    resultId?: string;
    data: uinteger[];
}
```

Listing 6.5: Semantic Tokens Response [Mic24f]

6.2. Diagnostics

Diagnostics [Mic24b] are a crucial feature of a Language Server as they provide real-time feedback to developers on potential errors, warnings, or other issues in their code.

6.2.1. DucklingParserError

The `DucklingParserError` structure (see Listing 6.6) is similar to the `Diagnostic` structure described in the LSP specification, albeit without some features that are not supported by the Duckling language at the time of writing this thesis. Each `DucklingParserError`, like each `Diagnostic`, contains a `DiagnosticSeverity` — a simple enum (described in Listing 6.7).

```
export class DucklingParserError {
    public readonly line: number;
    public readonly column: number;
    public readonly message: string;
    public readonly severity: DiagnosticSeverity;
    public readonly type: string;
}
```

Listing 6.6: Duckling Parser Error

```
export namespace DiagnosticSeverity {
    export const Error: 1 = 1;
    export const Warning: 2 = 2;
    export const Information: 3 = 3;
    export const Hint: 4 = 4;
}
```

Listing 6.7: Diagnostic Severity [Mic24b]

6.2.2. Publishing Diagnostics

Whenever the file is changed, all we need to do is get the list of `DucklingParserError`, convert it to a list of `Diagnostic` (which is trivial, since these two structures are nearly identical), and send it to the client. The last part is done by invoking `connection.sendDiagnostics` with a structure that follows the LSP specification (as described in Listing 6.8). In our implementation, `version` is omitted since this feature is not supported by the Duckling language.

```
interface PublishDiagnosticsParams {
    uri: DocumentUri;
    version?: integer;
    diagnostics: Diagnostic[];
}
```

Listing 6.8: Publish Diagnositcs Params [Mic24e]

6.3. Folding Ranges

The logic behind custom folding ranges consists of two parts. The first part is responsible for folding code blocks, which essentially include every part of code enclosed by brackets. The second part primarily handles comment blocks, strings, and imports.

6.3.1. Communication with `LSPTree`

To facilitate the folding of code blocks, we have added a new method to the `LSPTree` — `getFoldingRanges`. This method returns the folding ranges of subelements for all objects except the code blocks; for code blocks, it returns both their own folding range and the folding ranges of their subelements. This approach allows us to fold all the code blocks effectively.

6.3.2. Token Families

The second folding logic is based on semantic tokens. We define `tokenFamilies` (Listing 6.9) to determine which token types should be folded together, and then we iterate through the semantic tokens to fold consistent segments of the same family. This method enables us to handle comments, imports, and strings, as well as long sequences of declarations. This approach is highly modifiable, allowing us to easily adjust the logic by merely changing the token families.

```
const tokenFamilies: { [key: number]: number } = {
    [getTokenTypeIndex('keyword')]: 1,
    [getTokenTypeIndex('variable')]: 1,
    [getTokenTypeIndex('comment')]: 2,
    [getTokenTypeIndex('string')]: 3,
};
```

Listing 6.9: Folding tokenFamilies

6.3.3. Returning the result

The folding ranges should be sent to the client in the form of a list of `FoldingRange` (Listing 6.10) elements.

```
export interface FoldingRange {
    startLine: uinteger;
    startCharacter?: uinteger;
    endLine: uinteger;
    endCharacter?: uinteger;
    kind?: FoldingRangeKind;
    collapsedText?: string;
}
```

Listing 6.10: Folding Range [Mic24c]

6.4. Code Completion

The current state of Duckling language development does not yet allow us to fully implement the code completion feature. At present, we provide completion suggestions for keywords and identifiers.

6.4.1. Communication with Compiler and `LSPTree`

To retrieve the keywords, we make a request to the Compiler Daemon when the server is initialized, which then sends us all the keywords directly from the Duckling grammar used in the compiler. for identifiers, we iterate through all the elements in the `LSPTree` and add all the identifiers to a list whenever there is a change.

This setup ensures that despite the limitations in the language's development stage, users still receive essential support for code writing, improving efficiency and reducing syntactical errors. As the Duckling language evolves, this feature will be expanded to include more complex completion suggestions, such as context-aware, type-specific recommendations, and intelligent handling of imports.

6.4.2. Returning the result

The result is returned as a list of `CompletionItem` elements (this interface is shown in Listing 6.11; optional fields not used by us were omitted).

```
export interface CompletionItem {
    label: string;
    kind?: CompletionItemKind;
    detail?: string;
    documentation?: string | MarkupContent;
    insertText?: string;
}
```

Listing 6.11: Completion Item [Mic24a]

6.5. Conclusions

These are all of the key features (as listed in Section 3.3) that we were able to implement with the current state of work on the Duckling language. The lack of more complex logical code analysis in the compiler unfortunately prevented us from implementing some features (like Goto Definition or Find References).

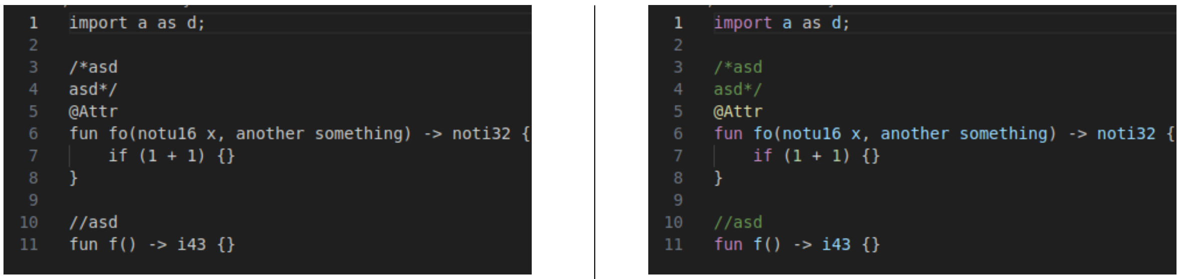
Chapter 7

User Experience

In this chapter, we want to showcase all of the user experience improvements delivered by our language server. We would like to present how DucklingLS contributes to an overall better and more enjoyable coding process for Duckling users.

7.1. Code Colorization

Semantic tokens returned to the client provides IDEs with enough information to color programmers' code. We only provide the structural knowledge (e.g. whether a token is an identifier, a keyword, or a literal) about a file to an IDE which allows the user to choose their own color scheme through themes. An example of code before and after colorization is shown in Figure 7.1.



```
import a as d;

/*asd
asd*/
@Attr
fun fo(notu16 x, another something) -> noti32 {
    if (1 + 1) {}
}

//asd
fun f() -> i43 {}
```

(a) Before

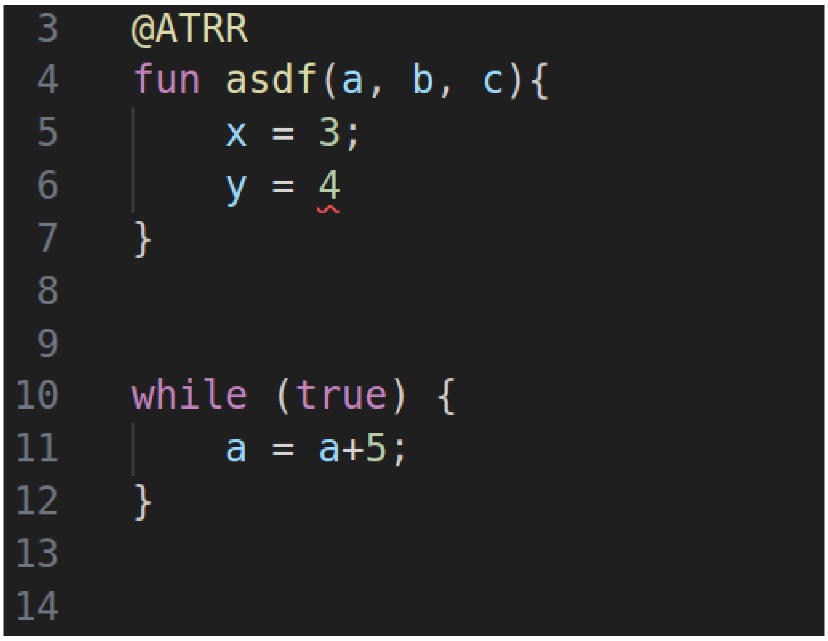
```
import a as d;

/*asd
asd*/
@Attr
fun fo(notu16 x, another something) -> noti32 {
    if (1 + 1) {}
}

//asd
fun f() -> i43 {}
```

(b) After

Figure 7.1: Code Colorization example

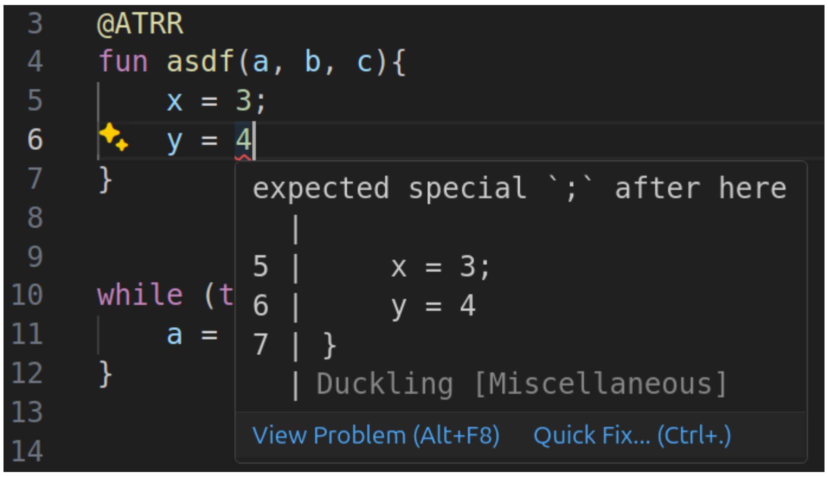
7.2. Error Marking

Marking and providing context for errors is possible thanks to the Diagnostics from DucklingLS. As shown in Figure 7.2, our server detects and returns information about errors to the IDE resulting in a squiggly red underline (in this case because of the missing semicolon at the end of the line). More context is shown after hovering the mouse over the underline (Figure 7.3) and (in VSCode) in the Problems tab (Figure 7.4).



```
@ATRR
fun asdf(a, b, c){
    x = 3;
    y = 4
}


while (true) {
    a = a+5;
}
```

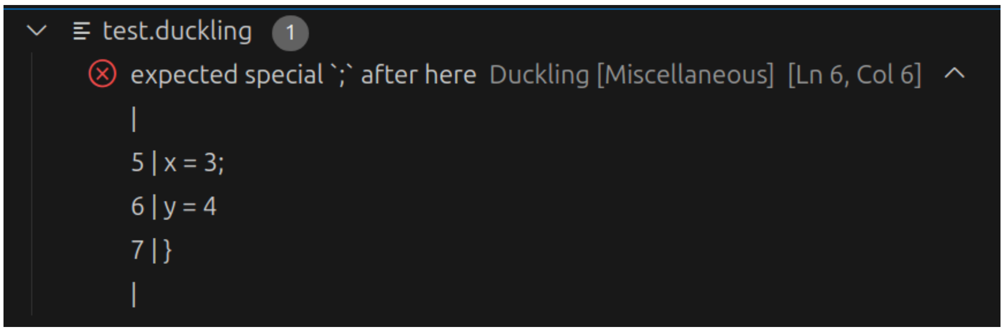
Figure 7.2: Error Marking example



```
@ATRR
fun asdf(a, b, c){
    x = 3;
    y = 4
}


while (t
    a =
}
```

```
expected special `;` after here
  |
5 |     x = 3;
6 |     y = 4
7 | }
  | Duckling [Miscellaneous]
```

Figure 7.3: Error context after hovering over the underline

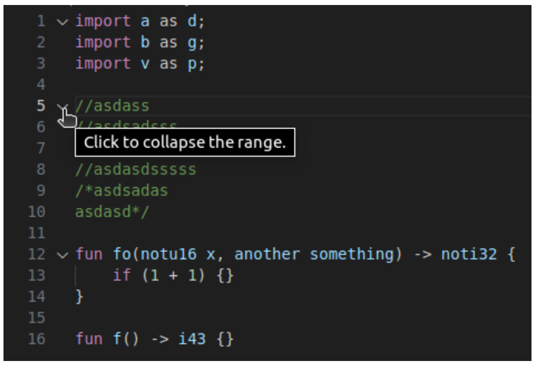


```
expected special `;` after here Duckling [Miscellaneous] [Ln 6, Col 6]
|
5 | x = 3;
6 | y = 4
7 | }
|
```

Figure 7.4: Error context in the Problems tab

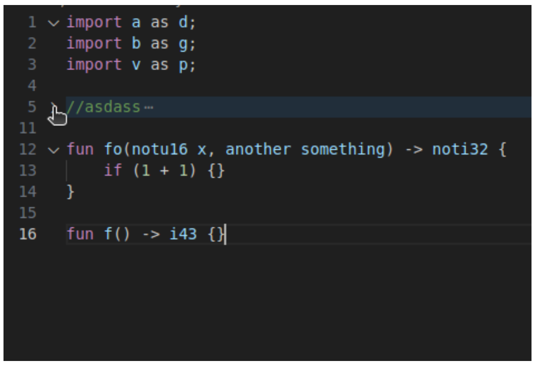
7.3. Folding

After providing Folding Ranges to the client, arrow icons next to the line numbers appear. They allow the user to fold code with consistent content (e.g. imports, comments, or code blocks). An example of code before and after folding is shown in Figure 7.5.



```
import a as d;
import b as g;
import v as p;

//asdass
//asdsadsss
Click to collapse the range.
//asdasdsssss
/*asdsadas
asdasd*/

fun fo(notu16 x, another something) -> noti32 {
    if (1 + 1) {}
}

fun f() -> i43 {}
```

(a) Before



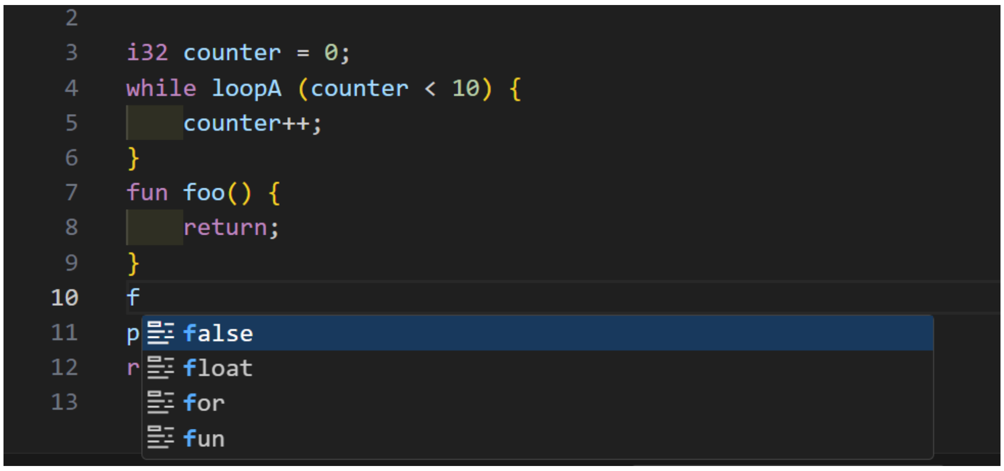
```
import a as d;
import b as g;
import v as p;

//asdass⋯

fun fo(notu16 x, another something) -> noti32 {
    if (1 + 1) {}
}

fun f() -> i43 {}
```

(b) After

Figure 7.5: Folding example

7.4. Code Completion

Upon a change in code, our server provides a list of elements (e.g. identifiers, keywords) which is later used for Code Completion by the client. An example list of completions is shown in Figure 7.6.



```

i32 counter = 0;
while loopA (counter < 10) {
    counter++;
}
fun foo() {
    return;
}
f
p  false
r  float
   for
   fun
```

Figure 7.6: Code Completion example

Chapter 8

Conclusions

In this chapter, we present the conclusions and final remarks from the entire project. We also propose areas for future expansion.

8.1. Work Organization

Throughout the entire project we held weekly meetings. During these meetings we worked together, brainstormed ideas, and made design decisions. We were often accompanied by a member of the Duckling team who helped us with any problems and checked on our progress.

We also participated in monthly meetings with the entire team working on Duckling. These meetings were a great opportunity to acquire knowledge about Duckling and on how to approach designing a new programming language. We could also share insights of our project, explaining how language servers work and what we were working on at the moment.

The code of the project can be found in Duckling's repository on GitHub. As of the time of writing this thesis, the repository remains private.

Within the team, we all worked together on most of the features. The most notable contributions from each team member are listed below:

- **Szymon Dziuda** — *Diagnostics*; help with `LSPTree`, Compiler Daemon, and *Semantic Tokens*; pull reviews; writing this thesis.
- **Jakub Kołaczyński** — architecture of the solution; Compiler Daemon; *Semantic Tokens*; help with `LSPTree`; writing this thesis.
- **Jan Szot** — `LSPTree`; *Code Completion*; writing this thesis.
- **Jan Wangrat** — *Folding Ranges*; help with *Code Completion* and *Semantic Tokens*; writing this thesis.

8.2. Difficulties

Unfortunately, some of the Duckling language logic components (especially the compiler) are not yet complete. This prevented us from implementing a few features we initially believed to be within our reach. Without the compiler, we are only able to provide knowledge about the structural composition of the code.

Moreover, as Duckling is currently undergoing dynamic development, we had to adapt to changes within the compiler during our work. We had to stay in touch with the Duckling development team and ensure our git project was in sync with the appropriate version.

When starting this project, nobody had had any previous experience with TypeScript, therefore we all had to learn a new programming language. It took us some time to get to know it well, but in the end it still proved to be beneficial because of the great support for writing language servers in TypeScript.

8.3. Final result

The most important outcome of this thesis and project is the server architecture. The REST API Compiler Daemon allows for the easy incorporation of new features of the Duckling language as they are added in the future.

We are also satisfied with the already implemented features in our language server. We believe that they are able to improve coding experience and provide help for people writing in Duckling.

8.4. Future work

As stated in the previous section, adding new features to the server should not be a problem once the language logic components are completed.

Some of the current features can also be improved upon once Duckling is more complete. This includes code completion, which as of time of writing this thesis, doesn't provide suggestions for class members (such as parameters of methods of a given class).

As stated in Section 5.4.3, using Docker [Inc] to start both the server and the compiler daemon could result in an overall greater experience for the user and better stability.

Bibliography

[Cod] Kichwa Coders. Lsp4j. https://github.com/eclipse-lsp4j/lsp4j.

[Com] Nix Community. rnix-lsp. https://github.com/nix-community/rnix-lsp.

[CPRW23] Kacper Chętkowski, Jakub Piotrowicz, Andrzej Radzimiński, and Antoni Wiśniewski. Prototyp maszyny wirtualnej i kodu pośredniego dla języków kompilowanych. Bachelor's thesis, University of Warsaw, Faculty of Mathematics, Informatics and Mechanics, 2023.

[Foua] NixOS Foundation. Nix. https://nixos.org/.

[Foub] Rust Foundation. Rust. https://www.rust-lang.org/.

[Hat] Red Hat. Ansible language server. https://ansible.readthedocs.io/projects/vscode-ansible/.

[Inc] Docker Inc. Docker. https://www.docker.com/.

[Mic] Microsoft. vscode-languageserver. https://github.com/microsoft/vscode-languageserver-node.

[Mic23a] Microsoft. Language server extension guide. https://code.visualstudio.com/api/language-extensions/language-server-extension-guide, 2023.

[Mic23b] Microsoft. Language server protocol. https://microsoft.github.io/language-server-protocol/, 2023.

[Mic24a] Microsoft. Completion specification. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_completion, 2024.

[Mic24b] Microsoft. Diagnostics specification. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#diagnostic, 2024.

[Mic24c] Microsoft. Folding range specification. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_foldingRange, 2024.

[Mic24d] Microsoft. Language server implementations. https://microsoft.github.io/language-server-protocol/implementors/servers/, 2024.

[Mic24e] Microsoft. Publish diagnostics specification. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_publishDiagnostics, 2024.

[Mic24f] Microsoft. Semantic tokens specification. `https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_semanticTokens`, 2024.

[PYP24] PYPL. Pypl popularity of programming language index. `https://pypl.github.io/IDE.html`, 2024.

[San22] Yannik Sander. Design and implementation of the language server protocol for the nickel language. Master's thesis, KTH, School of Electrical Engineering and Computer Science, 2022.

[SMSf] Ferrous Systems, Mozilla, Embark Studios, and freiheit.com. rust-analyzer. `https://rust-analyzer.github.io/`.

[Sna] Snapcraft. Snapcraft. `https://snapcraft.io/`.

[Sou] Ericsson's Open Source. Codecompass. `https://codecompass.net/`.

All references accessed 01.06.2024.