Background

This thesis illustrates an approach of implementing a language server for the Nickel language which communicates with its clients, i.e. editors, over the open Language Server Protocol (in the following abbreviated as *LSP*). The current chapter provides the backgrounds on the technological details of the project. As the work presented aims to be transferable to other languages using the same methods, this chapter will provide the means to distinguish the nickel specific implementation details.

The primary technology built upon in this thesis is the language server protocol. The first part of this chapter introduces the LSP, its rationale and improvements over classical approaches, technical capabilities and protocol details. The second part is dedicated to Nickel, elaborating on the context and use-cases of the language followed by an inspection of the technical features Nickel is based on.

Language Server Protocol

Language servers are today's standard of integrating support for programming languages into code editors. Initially developed by Microsoft for the use with their polyglot editor Visual Studio Code[ˆhttps://code.visualstudio.com/] before being released to the public in 2016 by Microsoft, RedHat and Codeenvy, the LSP decouples language analysis and provision of IDE-like features from the environment used to write. Developed under open source license on GitHub[1], it allows developers of editors and languages to work independently on the support for new languages. If supported by both server and client, the LSP now supports more than 24 language features[ˆhttps://microsoft.github.io/language-server-protocol/specifications/specification-current/] including code completion, hover information, resolution of type and variable definitions, controlling document highlighting, formatting and more.

Rationale

Since its release, the LSP has grown to be supported by a multitude of languages and editors[@langservers [@lsp-website]]. In fact the LSP solves an important problem. Traditionally, language support has been a responsibility and selling point for IDE providers. The possible business value of language features incentivized specialized (proprietary) software [haskellformac, jetbrains, xcode, visual studio]. While this leads to deep integration of the supported languages, users often face a lock-in into certain platforms as availability of integration varies. Language integration in IDEs requires the development of separate extensions using the target platform's API and programming language of choice. In effect, reusing code for language integration is almost impossible.

This is especially difficult for emerging languages, with possibly limited development resources to be put towards the development of language tooling. Research

[1] https://github.com/microsoft/language-server-protocol/

that shows available tooling to be of great importance for the selection of a programming language [cite] suggests a similarly significant influence on the adoption of new languages.

Commands and Notifications

File Notification

Diagnostics

Hover

Completion

Go-To-*

Symbols

code lenses

Shortcomings

Infrastructure as Code

Software defined Networks

Data oriented languages

Nickel

Gradual typing

Row types

Contracts