IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Background Report

Supervisor:
Dr. Fariba Sadri

Second Marker:
Prof. Robert Kowalski

Author:
Nikolai Merritt

Advisors:
Prof. Robert Kowalski
Dr. Jacinto Quintero
Mr. Galileo Sartor

Submitted in partial fulfillment of the requirements for the MSc degree in MSc Computing Science of Imperial College London

June 2022

Introduction

Language Extensions for code editors are a crucial tool in writing code quickly and without errors. In this project I create a Syntax Highlighter and Language Server for the logical and declarative programming language Logical English. I will give the Language Server the feature to auto-generate “boilerplate” template code from rules. I also aim to allow the Language Server to identify type mismatch errors that occur between rules and templates.

Literature Review

Language Servers have proven to be a powerful tool in creating cross-editor support for a wide variety of programming languages. As noted by Rask et al [5], the Language Server Protocol, which language servers use to communicate with code editors, “changed the field of IDEs”. This is because a language server can easily communicate with any IDE that supports the protocol, thus allowing IDEs to easily support a new language. Further, In surveying the effectiveness of language servers when building a language server for OCaml, Bour et al [3] note that “adding support for a new editor to a language server requires no language-specific logic”. This allows people who are not yet familiar with a given language to link a language server to their IDE of choice and begin programming.

However, building a language server does not come without difficulties. Bour et al [3], and the Visual Studio Code Language Server Extension documentation [1], describe two main challenges that Language Servers face, that of “incrementality and partiality”:

- Due to effiency constraints, the IDE only being able to send the portions of the document to the language server (incrementality)
- The language server having to parse incomplete portions of code that the user is writing (partiality)

In building their language server for OCaml, Bour et al solved these two issues by building their own parser, generated using an enhanced version of Menhir. This was needed because OCaml has a complex, recursive grammar, which made parsing incomplete portions of code a highly complex task. Logical English, however, has a very simple, non-recursive grammar, and our language server only concerns itself with parsing certain aspects of the language. Thus I expect it to be feasible for our language server to parse Logical English documents itself.

There is also existing literature on boilerplate generation from existing code. Wang et al [6] created a powerful compilation agent that auto-generates Java boilerplate code from more succint, annotated Java. The boilerplate code is generated at the Abstract Syntax Tree (AST) level: the code generator starts with the AST representing the annotated code and, using the Lombok compilation agent, produces an AST corresponding to non-annotated, boilerplate Java. Since neither templates nor heads of rules are recursive, I will likely find that a less complex representation can be favoured over AST.

Project Specification

The project will consist of developing two tools for Logical English: a Syntax Highlighter and a Language Server. These two tools will be cross-editor, meaning that they can be used with many of the most popular programming editors with minimal configuration.

Syntax Highlighter

The Syntax Highlighter will identify both micro-features of Logical English such as keywords and variable names, and macro-features such as section headers. It will identify these features using TextMate grammar. This way, the features identified by the grammar can be recognised and styled by the default themes of many popular code editors.

Language Server

The Language Server will allow the user to generate new templates from rules. If a set of rules do not match any existing templates, the Language Server will communicate this to the editor. It will allow the user to, at the click of a button, generate a template that matches the rules.

If there is time, I will give the language server the feature to alert the user of certain type mismatch errors. The user will be notified of errors where a rule is supplied in the knowledge base with a type that conficts with the corresponding type in the rule's template. To determine whether the one type conficts with the other, the Language Server would consider type inheritance as supported by Logical English.

The language server will communicate with potential language clients using the Langauge Server Protocol. This way, many popular code editors will be able to easily communicate with the language server.

Project Implementation Plan

The syntax highlighter will be implemented using TextMate grammar, since this has the widest range of editor support. The Language Server will be implemented in TypeScript using the `vscode-languageserver` NPM package. This package has clear, thorough documentation which describes multiple example language servers. In testing, both the language server and syntax highlighter will be tested on a Visual Studio Code language client. This choice is made due to Visual Studio Code's powerful debugging features for language plugins.

Project Timeline

The timeline for developing and testing these two tools is below. This plan has us completing both the template generation and type error detection features of the language server. However, if any large problems arise, I will prioritise solving these over working on type error detection.

6th June - 10th June	Write a TextMate grammar for Logical English.
13th June - 17th June	Using the Visual Studio Code documentation [2], create a proof-of-concept language server with dummy error highlighting, warning highlighting, and code generation.
20th June - 25th June	In the language server, convert Logical English templates to a suitable TypeScript representation. Using this representation, determine whether a Logical English rule conforms to a template.
27th June - 8th July	Create a template from first two, then arbitrarily many, rules.
11th July - 23rd July	Create a TypeScript representation for Logical English types, to be used by the language server. Use this type representation in to augment the template representation with types of the template's variables. Consider types when determining whether a rule conforms to a template.

Technical Acheivements

Before fully beginning on this project, I have built and tested a rudimentary syntax highlighter and language server.

I have written a syntax highlighter, using TextMate, for a small subset of the Logical English grammar [1]. This highlighter successfully detected the micro-features `if` and `then` as keywords. It also detects the macro-features `The templates are:` and `The knowledge base subset includes:` as headers for the template and knowledge base sections, respectively. Once detected, these features were highlighted by the default light and dark theme of Visual Studio Code.

I also followed Jeremy Greer's language server tutorial [4] to create a basic language server. This language server "blacklists" certain words by marking their usages as errors. [2] The language server was tested both by manually writing language server requests through standard console input, and by connecting with a Visual Studio Code language client that was taken from the same tutorial. In both tests the server behaved as expected (that is, as described in Greer's tutorial) when ran on Windows Subsystem for Linux.

The purposes of these trial runs were twofold. Firstly, completing these projects allowed us to familiarised ourselves with the tools and languages that I will be working with. Secondly, I have confirmed that a syntax highlighter and language server will run as intended using our choice of platform, tools and testing environment.

[1] The syntax highlighter can be found at https://github.com/nikolaimerritt/LogicalEnglish/tree/main/language-server/logical-english

[2] The server can be found at https://github.com/nikolaimerritt/LogicalEnglish/tree/main/language-server/blacklist-server. However, the code is entirely taken from the tutorial.

Bibliography

[1] 2022. URL https://code.visualstudio.com/api/language-extensions/language-server-extension-guide. pages ii

[2] 2022. URL https://code.visualstudio.com/api/language-extensions/programmatic-language-features. pages iv

[3] B. et al. Merlin: A language server for ocaml (experience report), 2018. URL https://dl.acm.org/doi/pdf/10.1145/3236798. pages ii

[4] J. Greer, 2022. URL https://www.toptal.com/javascript/language-server-protocol-tutorial. pages v

[5] e. a. Rask. The specification language server protocol: A proposal for standardised lsp extensions, 2021. URL https://arxiv.org/abs/2108.02961. pages ii

[6] Y. Wang, H. Zhang, B. C. d. S. Oliveira, and M. Servetto. Classless java. *SIGPLAN Not.*, 52(3), oct 2016. ISSN 0362-1340. doi: 10.1145/3093335.2993238. URL https://doi.org/10.1145/3093335.2993238. pages ii