Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Background Report

Author:
Nikolai Merritt

Supervisors:
Bob Kowalski, Jacinto Quintero, Galileo Sartor

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

September 2015

Abstract

Your abstract.

Acknowledgments

Comment this out if not needed.

Contents

Chapter 1

Introduction

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

Figure 1.1: Imperial College Logo. It's nice blue, and the font is quite stylish. But you can choose a different one if you don't like it.

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 [4], the Language Server Protocol, which language servers use to communicate with code editors, ”changed the field of IDEs” since one language server can communicate with any IDE that supports the protocol. 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.

Bour et al [3] and the Visual Studio Code Language Server Extension documentation [1] also describe two main challenges that Language Servers face, “incrementality and partiality”:

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

In building their language server for OCaml, Bour et al solved these two issues by additionally building their own parser, generated using an enhanced version of Menhir, that could handle incrementality and partiality. 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 it is reasonable for our language server to parse the document itself.

Wang et al [5] 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, we will likely find that a less complex representation can be favoured over AST.

no acheivements

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 in a way that the default themes of many of the most popular editors will recognise and style them.

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 user. It will allow the user to, at the click of a button, generate a template that matches the rules.

If there is time, we 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.

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, since this package has clear, thorough documentation and describes multiple example language servers.

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, we will prioritise solving these over working on type error detection.

6th June - 10th June	Learn Regular Expressions. Write a TextMate grammar for Logical English.
13th June - 17th June	Explore the process of creating a language server in TypeScript by creating a proof-of-concept language server with error highlighting, warning highlighting, and code generation using the Visual Studio Code documentation [2].
20th June - 25th June	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 Logical English Type representation in TypeScript. Use this type representation in giving types to the arguments of templates. Use this type representation to consider types in determining whether a rule conforms to a template.

Bibliography

[1] 2022. URL https://code.visualstudio.com/api/language-extensions/language-server-extension-guide. pages 2

[2] 2022. URL https://code.visualstudio.com/api/language-extensions/programmatic-language-features. pages 5

[3] B. et al. Merlin: A language server for ocaml (experience report), 2018. URL https://dl.acm.org/doi/pdf/10.1145/3236798. pages 2

[4] e. a. Rask. The specification language server protocol: A proposal for standardised lsp extensions, 2021. URL https://arxiv.org/abs/2108.02961. pages 2

[5] 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 2