Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

# Project Report (draft)

*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

September 2022

# Contents

# Introduction

## 1.1 Abstract

Language Extensions for code editors are a crucial tool in writing code quickly and without errors. In this project, I create a language extension for the logical, declarative programming language Logical English. The language extension highlights the syntactic and semantic features of Logical English, identifies errors, and generates "boilerplate" code to fix them. The language extension uses the Language Server Protocol and is therefore cross-editor. It is evaluated when connected to the Visual Studio Code and Codemirror IDEs.

## 1.2 Motivation

### 1.2.1 Why a new editor?

Logical English is a relatively new programming language, first introduced in late 2020 [8] **– is this where LE was first introduced? –**. Although Logical English has an online editor hosted on the SWISH platform [4], the editor is not user-friendly. SWISH is primarily a Prolog editor, and so any Logical English code has to be written in a long string that is, in the same file, passed to a Prolog function to be interpreted.

This has significant drawbacks: since Logical English code is written in a Prolog string, the Logical English content cannot be treated by the editor as a standalone program. This means that it can receive no syntax highlighting, error detection, or code completion features beyond that of a Prolog string. Since these features are essential for productivity **– find a source here about IDEs vs Notepad –**, a new editor was needed that was custom-built for writing Logical English.

### 1.2.2 Why a Language Server?

When deciding what type of language extension to create, we surveyed a variety of options. The SWISH platform in which Logical English is currently edited is built on Codemirror, a JavaScript framework for creating web-based code editors. Thus writing the language server entirely in Codemirror was an obvious choice. However, at the time, the SWISH platform was written in Codemirror 5, but the maintainers

were considering migrating to Codemirror 6. Prematurely writing the language extension in Codemirror 6 would be too much of a risk, as migration was uncertain, and users would not be able to test the extension until it was migrated. However, using Codemirror 5 would also have been a bad idea, as migration to Codemirror 6 would involve a large number of breaking changes [1], such as getting the position of text in a document, and making changes to the document, being entirely restructured.

This prompted us to then consider Monaco. **– why were we considering Monaco? –.**

In the end, we found that what we really needed was a language server. This language server would be editor-agnostic, meaning that one language server would be able to connect to any front-end that supports the Language Server Protocol. This includes online editors such as Codemirror 6 [11] and Monaco [12], along with desktop editors such as Visual Studio Code [5], Visual Studio [3] and IntelliJ [2] some of the most popular code editors [9]. I also found that the language server I produced was able to communicate error messages with Codemirror 5. Logical English is still in an early stage of development, and producing a language server would give us the flexibility needed to branch out to all kinds of coding environments.

# 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 [10], 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 [7] 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 [7], and the Visual Studio Code Language Server Extension documentation [5], 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 [13] 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 Requirements

In this section I talk about what LE is, and what my requirements are.

## 3.1 Logical English

Logical English is a logical and declarative programming language. It is written as a structured document, with a syntax that has few symbols and which closely resembles natural English. [8].

### 3.1.1 An overview

Logical English's main goal is to find which literals are true and answer a given question. A literal is a statement, which can be true or false, and cannot be broken down into any smaller statements. Examples include:

```
fred bloggs eats at cafe bleu.

emily smith eats at a cafe.

cafe bleu sells sandwitches.
```

Literals may have variables, such as `a cafe`: these will be discussed further.

A Logical English document is chiefly made up of clauses. Clauses are rules that start with a literal and determine when the literal is true. Examples include:

```
fred bloggs eats at cafe bleu if
    fred bloggs feels hungry.

emily smith eats at a cafe if
    emily smith feels hungry
    and the cafe sells sandwitches.

emily smith feels hungry.
```

In the second example, `a cafe` is a variable. This means that if we were later given `cafe jaune sells sandwitches`, then `emily smith eats at cafe jaune` would be true.

Templates are used for Logical English to understand which words in a literal correspond to terms (such as `emily smith` and `a cafe`), and which words are merely part of the statement (such as `eats at` or `feels hungry`). A literal's template is the literal with each of its terms replaced with placeholders. These placeholders start with `a` or `an` and are surrounded by asterisks. For example, the literals `fred bloggs eats at cafe bleu` and `emily smith eats at a cafe` both share the corresponding template

```
a person eats at a cafe.
```

In Logical English, each literal needs to have a corresponding a template.

### 3.1.2 The structure of a Logical English program

Now that literals, clauses and templates have been explained, we can examine a complete Logical English program. An example is provided in Listing 3.1.

**Listing 3.1:** A short Logical English program.

```
the templates are:
a person travels to a place.
a place has an amenity.

the knowledge base Travelling includes:
fred bloggs travels to a holiday resort if
    the holiday resort has swimming pools.

emily smith travels to a museum if
    the museum has statues
    and the museum has ancient coins.

scenario A is:
the blue lagoon has swimming pools.
the national history museum has statues.

query one is:
which person travels to which place.
```

#### Templates

The program starts with the template section, starting with `the templates are:`, in which the literals' templates are defined.

#### Knowledge base

The program's clauses are then given in the knowledge base section. The knowledge base section can either start with `the knowledge base includes:`, or it can be given a name, in which case it starts with `the knowledge base <name> includes:`.

Clauses are written in order of dependecy: if clause A is referenced by clause B, then clause A must be written before clause B.

Clauses begin with exactly one head literal, which is the literal that is logically implied by the rest of the clause. If a clause consists of simply a single head, then the head is taken to be always true. Otherwise, the head literal be followed by an `if`, then a number of body literals, separated by the connectives `and`, `or`, or `it is not the case that`.

The precedence of these connectives is clarified by indentation: connectives that have higher precedence are indented further. For example, `(A and B) or C` is written

```
A
    and B
or C.
```

and `A and (B or C)` is written

```
A
and B
    or C.
```

The connective `it is not the case that` always takes highest precedence. However, there is no default preference over `and` and `or`: it is an ambiguity error to write

```
A
and B
or C.
```

In a clause, a variable is introduced for the first time by having its name preceed with `a` or `an`. Subsequent uses of the variable must then start with `the`.

### Scenarios

Various scenarios can optionally be given. Scenarios contain literals that are used when running a query. Scenarios must have a name, and must start with `scenario <name> is:`.

### Queries

The final sections of a Logical English program are the queries. Like scenarios, a query must have a name, and must start with `query <name> is:`. A question in a query corresponds to a template, with the terms to be found written as placeholders that start with `which`.

In listing 3.1, running query one with scenario A yields `fred bloggs travels to the blue lagoon.` Query one could also be run with no scenario supplied, but doing so would yield no answer.

## 3.2 Project Requirements

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.

### 3.2.1 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.

### 3.2.2 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.

## 3.3 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.

### 3.3.1 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 [6], 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.<br>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.<br>Use this type representation in to augment the template representation with types of the template's variables.<br>Consider types when determining whether a rule conforms to a template. |

# Implementation

## 4.1 Technology Stack

Before any work could be done, I needed to decide on which technology stack to use. This was dictated mainly by the programming language involved. The following features were needed:

**Linux, Windows and Mac OS support**
The language server had to be able to connect to offline editors, and therefore run on user's desktops. This meant that the most up-to-date editions of the three most popular operating systems – Linux, Windows and Mac OS X – had to be supported.

**Support for strong typing**
Since creating a language server is a large and complex project, I needed to be using a langauge with strong typing in order to both avoid mistakes and receive context-aware support from my IDE.

**A Language Server Protocol API**
Writing Language Server Protocol requests manually would be inefficient, time-consuming and a potential cause of errors. A library that abstracted away the exact layout and content of Language Server Protocol requests would aid productivity.

These last two requirements only left two options: the `Microsoft.VisualStudio.LanguageServer` API, written for C# [3], and the `vscode-languageserver` package, written for TypeScript [5]. Since these

# Evaluation

We haven't done any evaluation yet.

# Conclusions and Further Work

We haven't finished yet.

# Bibliography


[1] Codemirror: Migration guide from codemirror 5 to codemirror 6. URL https://codemirror.net/docs/migration/. pages 4

[2] Lsp4intellij - language server protocol support for the jetbrains plugins. URL https://github.com/ballerina-platform/lsp4intellij. pages 4

[3] Add a language server protocol extension. URL https://docs.microsoft.com/en-us/visualstudio/extensibility/adding-an-lsp-extension?view=vs-2022. pages 4, 11

[4] Logical english on the swish online editor, 2022. URL https://logicalenglish.logicalcontracts.com/. pages 3

[5] Language server extension guide, 2022. URL https://code.visualstudio.com/api/language-extensions/language-server-extension-guide. pages 4, 5, 11

[6] Programmatic language features, 2022. URL https://code.visualstudio.com/api/language-extensions/programmatic-language-features. pages 10

[7] B. et al. Merlin: A language server for ocaml (experience report), 2018. URL https://dl.acm.org/doi/pdf/10.1145/3236798. pages 5

[8] R. Kowalski. Logical english, 2020. URL https://www.doc.ic.ac.uk/~rak/papers/LPOP.pdf. pages 3, 6

[9] S. Overflow. 2021 developer survey, 2021. URL https://insights.stackoverflow.com/survey/2021/most-popular-technologies-new-collab-tools. pages 4

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

[11] M. Ridwan. Using language servers with codemirror 6. URL https://hjr265.me/blog/codemirror-lsp/. pages 4

[12] TypeFox. Monaco language client and vscode websocket json rpc. URL https://github.com/TypeFox/monaco-languageclient. pages 4

[13] 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 5