Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Project report: rough 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?

his where was first oduced?

Logical English is a relatively new programming language, first introduced in late 2020 [8] . 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 , a new editor was needed that was custom-built for writing Logical English.

a source e about s vs epad

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? How would Monaco have worked with SWISH?

.

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 [13] and Monaco [14], along with desktop editors such as Visual Studio Code [5], Visual Studio [3] and IntelliJ [2] some of the most popular code editors [11]. 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.

1.2.3 Why a separate Syntax Highlighter?

Although language servers can mark code for highlighting (and, indeed, mine does), it is common to delegate all the syntactic highlighting to the client. This is done for efficiency reasons. Syntactic highlighting does not need a complex algorithm to parse the document, and it would be a waste of resources to do so: instead, it can be done through identifying parts of the document using regular expressions.

This is commonly done using a TextMate grammars [9] document. This is a JSON document that assigns certain standard labels to sections of the document that match regular expressions. Syntax highlighting using TextMate grammars is supported by all the IDEs mentioned above.

Talk about how the TextMate grammar marks words, and it is up to the IDEs colour scheme to colour them appropriately.

Throughout this report, the term “language extension” will refer to the language server and syntax highlighter together.

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 [12], 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 [15] 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. 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.

Implementation

4.1 The Technology Stack

4.1.1 The Language Server

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 libraries: the `Microsoft.VisualStudio.LanguageServer`, written for C# [3], or `vscode-languageserver`, written for TypeScript [5]. Since these libraries are were created by Microsoft [1], they are structured quite similarly.

In the end, the TypeScript library was chosen over the C# library. Both libraries being quite similar, this decision was made because TypeScript was better suited for the task than C#. The Language Server Protocol communicates using JSON, and TypeScript can handle JSON more fluidly than C# can. Useful features include destructuring JSON, giving JSON objects unique types based on their fields, and treating JSON objects as implementing interfaces that have the same fields. This

[1] This should not be surprising, as Microsoft also created the Language Server Protocol.

decision being made, the resulting technology stack was TypeScript run locally using `Node.JS`.

4.1.2 The Syntax Highlighter

Since the syntax highlighting is specified in a single JSON document, the technology stack required was minimal. Rather than writing the JSON document directly, I chose to write the TextMate grammar in a YAML document, from which the JSON document was then automatically generated using the command `yq` [10]. This was done because of the length of the JSON required: YAML documents are easier to read due to their less cluttered syntax, with the scope of objects being determined by whitespace rather than brackets. Writing regular expressions is also easier in a YAML document. Regular expressions feature many backslashes: in JSON, unlike in YAML, these backslashes need to be escaped with another backslash. Regular expressions are confusing enough to read as they are – I did not need any added confusion by having to parse escaped backslashes in my head!

4.1.3 The Language Client

Language servers written using the `vscode-languageserver` library connect seamlessly to visual studio code language clients written using the `vscode-languageclient` package. Thus, my main method of day-to-day testing was done using a local visual studio code client. I could be assured that all errors I found were due to the language server itself, not the connection, since the two libraries were built with each other in mind. Tests were also done using a Codemirror 5 client that ran locally in the browser, to see which features carried over to Codemirror 5.

4.2 The Syntax Highlighter

Using TextMate grammar, the syntax highlighter identified three tiers of Logical English features.

4.2.1 Sub-Line Features

The syntax highlighter marks keywords that are contained within a single line, such as `if`, `and`, `or`, `it is the case that` and `it is not the case that`. It also marks the pre-defined constant term `unknown`. This is done simply by recognising these keywords wherever they appear and are surrounded by word boundaries (i.e. spaces, tabs or other non-word characters).

A proposed feature is for the language extension to allow for such keywords to appear in template names, where they would have to be marked differently. This would not be a problem: the language server can override the syntax highlighter in its semantic highlighting.

The syntax highlighter also marks single-line comments that begin with `\%` and span until the end of the line. However, all the syntax highlighter does is mark them: it is a separate feature of the language server to ignore comments.

4.2.2 Single-Line features

The syntax highlighter also marks the headers of sections of a Logical English document. For the template section header, `the templates are:`, the characters from the initial `t` up to the final `e` is marked as a single header block. Beginning at the initial word character is an important point, since Logical English allows headers to be indented.

The knowledge base, scenario and query headers all support being named. Thus, their names are marked separately from the rest of the header.

It is interesting to note what TextMate grammar identifiers are used to mark the headers. Following the TextMate naming conventions [9] is essential to maximise the amount of colour schemes that colour Logical English documents correctly. However, the guidelines are quite vague, with `entity`, `meta` and `markup.heading` all being recommended for use in marking up section headers.

The most pragmatic course of action was to survey a variety of popular TextMate grammars. Finding analogies of section headers in popular languages was difficult. In C-style languages, sections of code are either labelled with single-word keywords such as `if`, `for` or `while`, or also name a type of data structure, such as class or interface names. These are semantically quite different from Logical English section headers. In the end, I settled on a function name, used in a function definition or declaration, as being the closest analogy. Although the function name may reappear in the document, it does not represent a type and rarely represents data (the data instead usually being the return value, at the end of a function call). This was usually labelled with `entity.name` prefix. The fact that the names of HTML tags are also labelled with `entity.name` was further evidence that `entity.name` was the best choice.

Put some references to popular IDE's default TextMate schemes here. Also maybe talk about why the name of a query / section / knowledge base is given variable.

4.3 The Language Server

The language server is a complex tool and has multiple features. Each category of feature corresponds, in the most part, been extracted into a single source code file.

4.3.1 General Design choices

Research design methodologies, evaluate them for this problem. Cite which methodology corresponds to mine.

4.3.2 Parsing the document

Talk about how different parts of the document: sections, templates, clauses, literals are extracted. Talk about ContentRanges.

4.3.3 Template Functionality

Rewrite this to be about templates only.

In parsing a Logical English document, the language server is heavily reliant on "helper functions". A common class of problem is, given a pattern or structure, to extract parts of a block of code according to the pattern. For instance, is done in (but is not limited to):

1. extracting the template section or knowledge base section from the document
2. extracting clauses from the knowledge base section, and templates from the template section
3. extracting literals from clauses
4. extracting terms from a literal, assuming that a literal matches a given template
5. generating a template from a literal, given the literal's terms
6. determining whether a literal matches a given template

These core functionalities are used throughout the language server, so it is worth spending some time to understand how these functionalities work. The first three can be done syntactically: these are essentially done through simple regular expressions. The last three are more complex.

Representing a template

Before I could solve the two problems, I first had to design a suitable data structure for representing a template. The more effectively the structure is designed, the easier the solutions will be; in fact, the Template class proved to be the backbone of the language server. Such a design must be close enough to the conceptual notion of a template to be powerful, whilst also being close enough to the Logical English syntax of a template for it to be applicable to the problem.

Conceptually, a template is a list consisting of either variables, or text that makes up

Talk about my intial design and why I changed it.

think of a better word

the template's name. Thus a natural representation is a wrapper class around an array of `TemplateElement` items, where a `TemplateElement` is either a `TemplateVariable` or a `PredicateWord`. Expressed in TypeScript:

```
type TemplateElement = TemplateVariable | PredicateWord;

class Template {
    private readonly elements: TemplateElement[];
    ...
}
```

The `TemplateVariable` and `PredicateWord` types are classes that contain their text label. Both also contain a `type` field, that specifies which of the two types the object is. This way, any `TemplateElement` can have its `type` field queried.
Expressed in TypeScript:

```
enum TemplateElementKind {
    Variable,
    Word
}

class TemplateVariable {
    public readonly name: string;
    public readonly type = TemplateElementKind.Variable;
    ...
}

class PredicateWord {
    public readonly word: string;
    public readonly type = TemplateElementKind.Word;
    ...
}
```

This design choice allows `TemplateVariable` to have extra functionality added without breaking the existing code. The creators of Logical English are considering to give types to template variables and perform type checking at runtime to find mistakes in user's Logical English code. This could easily be accommodated by this current design by simply adding extra fields and methods to the `TemplateVariable` class.

Talk a bit more about how an actual string is converted using regex.

Using a template to extract terms from a literal

Extracting the terms from a literal, assuming that the literal matches a given template, is done using a state engine. The (space-separated) words in the literal are iterated over. Each word `w` is compared to the template's first predicate word.
If they do not match, `w` must be part of a term, so `w` is appended to a string buffer.

When `w` matches the first predicate word, we have reached the end of the term, so the buffer is appended to a list of terms and is cleared.

Make this into a diagram or pseudocode.

Research into state engine.

Generating a template from a literal

Given a literal and a list of the literal's terms, it is fairly straightforward to generalise the literal into a corresponding template. This is done by leveraging the fact that a template's elements alternate between being a variable and being a predicate word. In other words, the literal's predicate words are each separated by exactly one term. This means that we can obtain the literals elements by splitting the literal, with each of the literal's terms as delimeters. Doing so, using TypeScript's powerful regular expressions system, yields a list of the template's elements, in string form. From this, the corresponding `TemplateElement[]` is generated by checking which elements are terms and which must be predicate words. The template is then built from this array.

Think of a more succint name than "predicate word".

Rewrite this using a diagram or pseudocode.

Determining whether a literal matches a template

The above two functionalities are put together to determine whether a literal L matches a given template T. I first assume that L does indeed match T, and extract L's terms. Using L's terms, I generalise L into a new template T'. It then suffices to check whether T and T' are isomorphic: that is, whether T's variables could be renamed to give T'.

Write this as a formal proof.

4.3.4 Genreralising two or more literals into a shared template

Talk about Least General Generalisation.

4.3.5 Connection to a Language Client

Research more about how this works.

4.3.6 Semantic Highlighting

The language server augments the syntax highlighter by highlighting the terms that appear in literals. This is a step beyond the regular expressions of the syntax highlighter. To do this, the server must understand the structure of each template, which template each literal is an instance of, and how to use this matching to identify which

substrings of a literal are terms. Each of these functionalities are described in the previous section: here, I will explain how they are put together.

The server first extracts the template and literal strings in the document. Each template string is converted into an instance of the `Template` class. The semantic highlighter finds the first template that matches each literal.

This could lead to bugs. Instead, find the most specific template that matches the literal. Code this up then talk about why this is necessary, and how specificity is determined.

It uses this template to extract the literal's terms. The position of each term in the document is calculated using the `ContentRange` of each literal, offset by the position of each term in the literal. At first, I naively used the `String.indexOf` function to find the character positions of each term. When the same term occured more than once, the position of the first term was given both times, which lead to subsequent repitions of the term not being highlithed. Instead, the line position of the previous term was added to the...

Needlessly complicated. Rewrite code to use indexOf(element, startingIndex)instead.

4.3.7 Code Completion

Here I'll explain how half-written literals are matched against templates. I'll explain Template.matchScore, how half-written literals can have their existing terms extracted and substituted into their template.

4.3.8 Error Diagnostics

Here I'll explain how literals are iterated over and checked whether they match any template. ContentRange is used to work out where the error diagnostic should go. I'll also talk about how clauses are checked for misaligned connectives. ContentRange is used to work out where error diagnostic should go.

4.3.9 Error Fixes

Explain how the literals without any template are re-calculated, and talk about how this is inefficient but necessary by my stateless design. May optimise later.

Talk about how the least general generalisation of all template-less literals is calculated. Why it is a good idea in practice to generate a template from all such literals – users will start with code where all literals match and introduce new literals that dont match.

Here I should also talk about doing more than just LGG. If Im in a clause where literal A has T as a term, and I write a literal B that uses T, then when generalising B I should replace T with a varaible.

4.4 The Visual Studio Code Language Client

Research how connection works

4.5 The Codemirror Language Client

Talk about how it connects to language server via a stdio to websocket proxy.

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 5

[2] Lsp4intellij - language server protocol support for the jetbrains plugins. URL https://github.com/ballerina-platform/lsp4intellij. pages 5

[3] Add a language server protocol extension. URL https://docs.microsoft.com/en-us/visualstudio/extensibility/adding-an-lsp-extension?view=vs-2022. pages 5, 12

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

[5] Language server extension guide, 2022. URL https://code.visualstudio.com/api/language-extensions/language-server-extension-guide. pages 5, 6, 12

[6] Programmatic language features, 2022. URL https://code.visualstudio.com/api/language-extensions/programmatic-language-features. pages 11

[7] B. et al. Merlin: A language server for ocaml (experience report), 2018. URL https://dl.acm.org/doi/pdf/10.1145/3236798. pages 6

[8] R. Kowalski. Logical english, 2020. URL https://www.doc.ic.ac.uk/~rak/papers/LPOP.pdf. pages 4, 7

[9] macromates.com. Textmate grammars specification, 2021. URL https://macromates.com/manual/en/language_grammars. pages 5, 14

[10] mikefarah. yq. URL https://github.com/mikefarah/yq. pages 13

[11] S. Overflow. 2021 developer survey, 2021. URL https://insights.stackoverflow.com/survey/2021/most-popular-technologies-new-collab-tools. pages 5

[12] e. a. Rask. The specification language server protocol: A proposal for standardised lsp extensions, 2021. URL https://arxiv.org/abs/2108.02961. pages 6

[13] M. Ridwan. Using language servers with codemirror 6. URL `https://hjr265.me/blog/codemirror-lsp/`. pages 5

[14] TypeFox. Monaco language client and vscode websocket json rpc. URL `https://github.com/TypeFox/monaco-languageclient`. pages 5

[15] 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 6