

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Draft 2

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

1	Introduction	5
1.1	Abstract	5
1.2	Structure of the Document	5
1.3	Motivation: why a new editor?	5
1.4	Editor Overview	6
1.4.1	Language Client	6
1.4.2	Syntax Highlighter	6
1.4.3	Language Server	6
2	Logical English	7
2.1	A brief overview	7
2.1.1	Atomic Formulas	7
2.1.2	Clauses	7
2.1.3	Templates	8
2.1.4	Meta Atomic Formulas	9
2.2	The structure of a Logical English program	9
2.2.1	Templates	10
2.2.2	Clauses	10
2.2.3	Scenarios	11
2.2.4	Queries	11
3	Editor Features	13
3.1	Highlighting	13
3.2	Code Completion	14
3.3	Error Diagnosis	14
3.3.1	Atomic formulas without templates	14
3.4	Clauses with misaligned connectives	14
3.5	Code Actions	15
3.6	Type Checking	16
3.6.1	Type Mismatch Errors	16
3.6.2	The Type Hierarchy	17

4	Literature Review	19
4.1	Language Servers	19
4.2	Editor Features	20
4.2.1	Code Generation	20
4.2.2	Code Completion	20
4.3	Type Systems	20
5	Project Requirements	22
5.1	The Language Client	22
5.2	The Syntax Highlighter	22
5.3	The Language Server	22
5.3.1	Code Completion	23
5.3.2	Error Diagnostics	23
5.3.3	Suggested Error Fixes	23
5.4	The Type System and Type Checker	23
5.4.1	Requirements of the Type System	23
6	Design	25
6.1	System Overview	25
6.2	Requirements Approach	26
6.2.1	Why a Language Server?	26
6.2.2	Why a Visual Studio Code client?	27
6.2.3	Why a separate Syntax Highlighter?	28
6.3	Development Framework	28
6.3.1	Language Server	28
6.3.2	The Syntax Highlighter	29
6.3.3	The Language Client	29
6.4	Design Methodology	30
6.4.1	Initial Design	30
6.4.2	Current Design Methodology	31
7	The Type System	33
7.1	Requirements	33
7.2	The User-based Type Hierarchy	33
7.2.1	The need for a hierarchy	33
7.2.2	The structure of the type hierarchy	34
8	Implementation	36
8.1	Data Representation	36
8.1.1	Description of Problem	36
8.1.2	Element Representation	37
8.1.3	Template Representation	37
8.1.4	Atomic Formula Representation	38

8.1.5	Section Representation	39
8.2	Semantic Highlighting	40
8.2.1	An Overview	40
8.2.2	Extracting the elements of an atomic formula	40
8.2.3	Matching a template to an atomic formula	42
8.2.4	Finding the template that best matches an atomic formula	42
8.3	Completion	43
8.3.1	Completing the remainder of an incomplete atomic formula	43
8.4	Error diagnosis	44
8.4.1	How a language server diagnoses errors	44
8.4.2	Diagnosing template-less atomic formulas	44
8.4.3	Diagnosing clauses that have misaligned connectives	45
8.4.4	Diagnosing type mismatches	45
8.5	Quick Fixes	47
8.5.1	How a language server provides quick fixes	47
8.5.2	Generating a new template that matches given atomic formulas	47
9	Evaluation	49
9.1	Testing	49
9.2	Feedback	49
10	Conclusions and Further Work	50
10.1	Conclusion	50
10.2	Challenge and Reflection	50
10.3	Further Work	50
11	User Guide	51
11.1	Installing the extension for Visual Studio Code	51
11.2	Running the language server manually	51
11.3	Reading the source code	51

Chapter 1: 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, predicts the completion of atomic formulas, identifies errors and generates “boilerplate” code to fix them. The language editor also extends the language of Logical English through type checking the use of terms by introducing a hierarchical type system. The extension is based on a language server that uses the Language Server Protocol. It is evaluated when connected to a Visual Studio Code front-end.

1.2 Structure of the Document

The first two chapters of this document introduce Logical English and the Logical English editor’s features. These chapters are intended for prospective users who are unfamiliar with either. These two chapters also set the scene for the remainder of the document: due to the complexity of the editor, and the language for which it is built, it may help to begin with a birds-eye view of the final product. In the remainder of the document, the project of building the Logical English editor is analysed. These chapters describe the requirements, research, design and implementation of the editor, evaluate the finished product, and lay out how this editor is connected to the wider body of surrounding literature.

1.3 Motivation: why a new editor?

Logical English is a relatively new programming language, first introduced under that name in late 2020 [24]. Although Logical English has an online editor hosted on the SWISH platform [13], the editor lacks features to edit Logical English that are common to most programming language editors. The SWISH editor is primarily a Prolog editor, so any Logical English code has to be written in a string that is an argument to a Prolog

function. 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.

Find a source about the productivity of IDEs vs a plaintext editor

1.4 Editor Overview

The editor documented in this report is a language extension for the popular Integrated Development Environment (IDE) ‘Visual Studio Code’. The language extension consists of three components: a language client, a syntax highlighter, and a language server.

1.4.1 Language Client

The language client is built for Visual Studio Code. The language client delivers the editor’s features to the user’s Visual Studio Code window.

1.4.2 Syntax Highlighter

The syntax highlighter is a document that contains the grammar of Logical English. The language client uses the syntax highlighter to highlight aspects of Logical English in the user’s document. The Syntax Highlighter is written according to the TextMate grammar specification, allowing it to be used by language clients for IDEs other than Visual Studio Code.

1.4.3 Language Server

The language server calculates the main features of the editor. When the user writes in the document, the language client communicates the change to the language server. The language client may request features such as:

- whether there are any errors in the document to diagnose
- whether what the user is currently writing can be auto-completed
- if the user is hovering over an error message, whether a ‘quick fix’ can be suggested

The language server calculates these features and communicates them to the language client. The language client ensures that these features are then displayed to the user.

The language server is written according to the Language Server Protocol. This allows the language server to be used by language clients for many IDEs other than Visual Studio Code.

Chapter 2: Logical English

Use screenshots of the editor.

Logical English is a logical and declarative programming language. Logical English is an example of a controlled natural language; it is written as a structured document, with a syntax that has few symbols and which resembles natural English. [24].

2.1 A brief overview

Logical English allows knowledge bases to be represented as logical rules. These logical rules can then be used to answer queries.

2.1.1 Atomic Formulas

A fundamental concept in Logical English is the atomic formula. An atomic formula is a statement which cannot be broken down into any smaller statements. Examples in Logical English include:

```
fred bloggs eats at cafe bleu.  
emily smith eats at cafe jaune.  
cafe bleu sells sandwiches.
```

Listing 2.1: An example of three atomic formulas in Logical English.

2.1.2 Clauses

Clauses are rules, starting with an atomic formula, that determine when the atomic formula is logically implied. 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 sandwiches.

emily smith feels hungry.
```

Listing 2.2: Examples of clauses in Logical English.

Clauses begin with a ‘head’, which is the atomic formula that is logically implied by the rest of the clause. If there are no other atomic formulas, then the head is taken to be logically implied in all cases. Otherwise, the head is followed by the keyword `if`, then a number of ‘body’ atomic formulas that logically imply the head. These body atomic formulas are separated by the connectives `and`, `or`, or it is not the case `that`.

A variable is introduced for the first time in a clause by beginning its name with a or an. In Listing 2.2, a cafe is a variable. Subsequent uses of the same variable start with the. This means that in Listing 2.2 if we were later given (or could later derive) that

```
cafe jaune sells sandwiches
```

then the atomic formula `emily smith eats at cafe jaune` would be logically implied by the second clause in the listing.

2.1.3 Templates

The words in an atomic formula can have one of two functions. A word could either be part of a *term* of the atomic formula, which is an object that the atomic formula describes. Otherwise, a word is part of the atomic formula’s *predicate*: the name of the assertion that is applied to the terms. Viewed this way, terms are also known as *predicate arguments*.

Templates are used in Logical English to clarify which parts of the atomic formula are terms, and which words are part of the predicate. An atomic formula’s template is written as the atomic formula with each of its terms replaced with placeholders. These placeholders start with a or an, and are surrounded by asterisks.

For example, the template

```
*a cafe* serves *an item* from *a time* to *a time*.
```

Listing 2.3: A template in Logical English

is a template for the atomic formula

```
cafe jaune sells crepes from breakfast to noon.
```

This template allows the terms to be identified as `cafe jaune`, `crepes`, `breakfast` and `noon`. In Logical English, every atomic formula must have a corresponding a template.

2.1.4 Meta Atomic Formulas

A meta atomic formula contains another atomic formula as a term. For instance, meta atomic formulas could be used to describe events, obligations or actions.

A meta atomic formula must have the keyword `that` preface the atomic formula that it describes. An example is given in Listing 2.4.

```
the templates are:
*a bank* receives money.
*a person* pays money to *a bank*.
there is a requirement that *an action*.

the knowledge base Payments includes:
LE bank receives money if
    there is a requirement that bob pays money to LE bank.
```

Listing 2.4: An example of an atomic formula featuring in a meta atomic formula.

In Listing 2.4, the atomic formula `bob pays money to LE bank` features in the meta atomic formula `there is a requirement that _`.

2.2 The structure of a Logical English program

A complete Logical English program features atomic formulas, clauses and templates. An example is provided in Listing 2.5.

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

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.
```

Listing 2.5: A short Logical English program.

The rest of this section expands on the features shown in Listing 2.5, as well as other features of Logical English.

2.2.1 Templates

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

2.2.2 Clauses

The Knowledge Base

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

Connectives in Clauses

The body atomic formulas in a clause are separated by logical connectives. 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.
```

There is no default preference over `and` and `or`: it is an ambiguity error to write

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

2.2.3 Scenarios

Various scenarios can optionally be given. Scenarios contain clauses with no body atomic formulas that are used when running a query. Scenarios must have a name, and must start with `scenario <name> is:`.

2.2.4 Queries

The final section of a Logical English program are the queries. A query consists of one or more questions that the Logical English engine seeks to answer. These questions ask for which atomic formulas can be deduced by the Logical English engine. The terms that the query is to look for are written as placeholders that start with `which`.

For example, running query one with scenario A from 2.6

```
the templates are:
*a person* visits *a resort*.
*a resort* has *an amenity*.

the knowledge base Resorts includes:
fred bloggs visits a resort if
    the resort has swimming pools.

scenario A is:
the blue lagoon has swimming pools.

query one is:
```

2.2. THE STRUCTURE OF A LOGICAL ENGLISH PROGRAM Chapter 2. Logical English

```
which person travels to which resort.
```

Listing 2.6: Another short Logical English program.

gives the answer

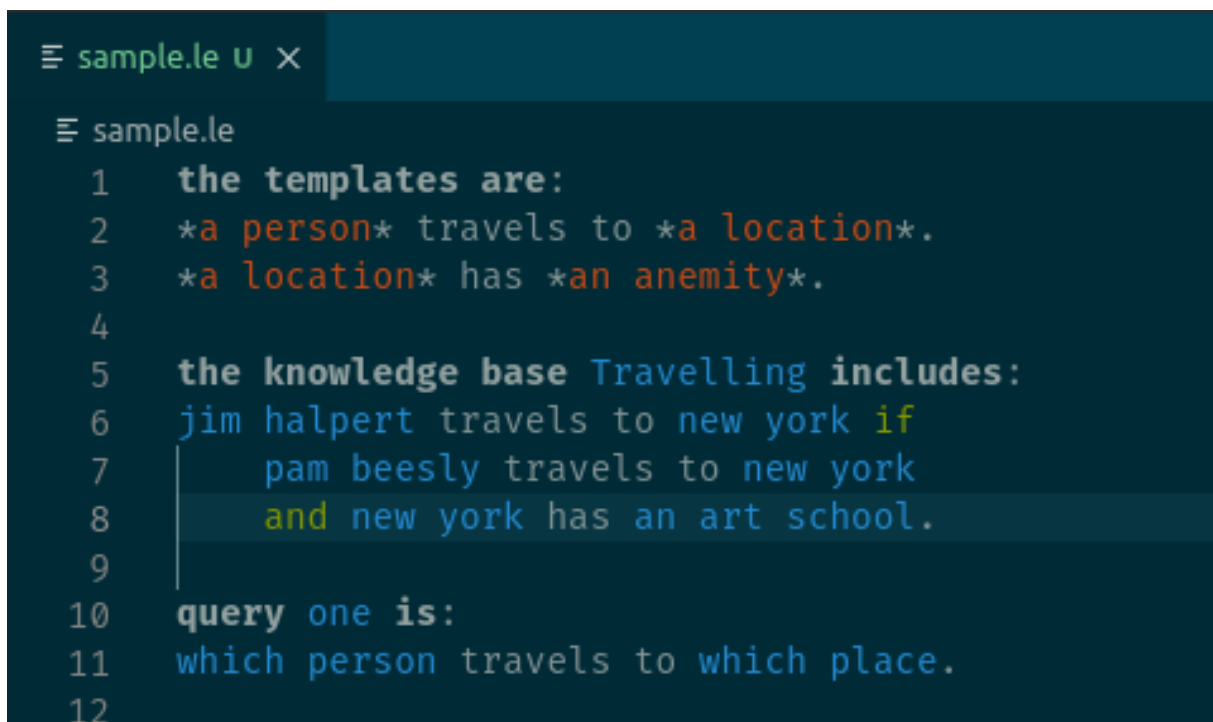
```
fred bloggs travels to the blue lagoon.
```

Query one could also be run with no scenario supplied, but doing so would yield no answer.

Chapter 3: Editor Features

The Logical English editor provides various features that help the user understand, write, and correct Logical English.

3.1 Highlighting



```
≡ sample.le u ×
≡ sample.le
1  the templates are:
2  *a person* travels to *a location*.
3  *a location* has *an anemity*.
4
5  the knowledge base Travelling includes:
6  jim halpert travels to new york if
7  |   pam beesly travels to new york
8  |   and new york has an art school.
9
10 query one is:
11 which person travels to which place.
12
```

Figure 3.1: The editor highlighting grammatical components of a short Logical English program.

As shown in Figure 3.1, the editor highlights grammatical features of Logical English. These include the argument names of templates, logical connectives between atomic formulas, headers and their names, as well as the terms of atomic formulas. Figure 3.1, as well as subsequent screenshots, are taken with Visual Studio Code using the built-in

‘Solarized Dark’ colour theme. Features of Logical English are highlighted regardless of the colour theme used, with the only change being the colours that features are given.

3.2 Code Completion

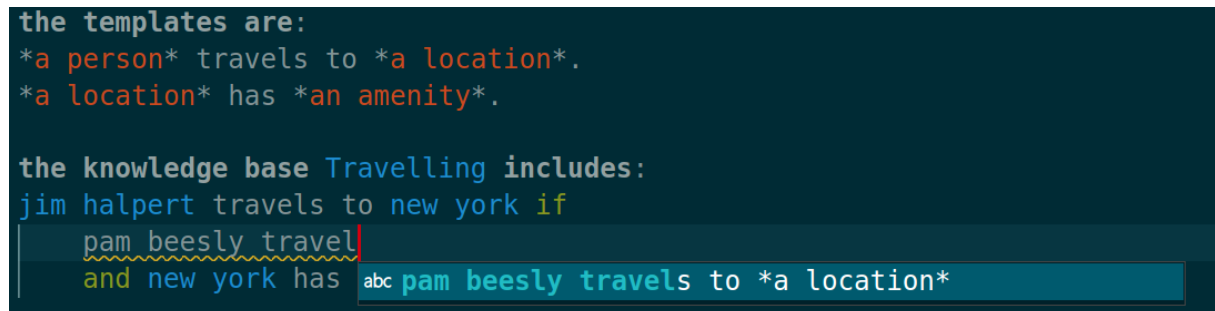


Figure 3.2: The editor suggesting the remainder of an incomplete atomic formula.

Figure 3.2 shows the editor suggesting the remainder of an incomplete atomic formula. This happens as the user is typing the atomic formula, as soon as the atomic formula begins to conform to a template. When selecting the suggested formula, done by clicking on the suggestion or pressing Tab, the suggested atomic formula is inserted with the remaining arguments (such as ‘a location’ in 3.2) being replaced by placeholders. The user can navigate across the placeholders by pressing Tab, allowing the user to fill in the placeholders efficiently.

3.3 Error Diagnosis

3.3.1 Atomic formulas without templates

If the document contains an atomic formula that does not conform to any template, the editor marks the atomic formula with an underline representing a ‘warning’. On hovering over the atomic formula, the editor produces an explanatory error message. This is shown in figure 3.3.

This feature occurs when an atomic formula does not conform to any template written in the document, nor any of Logical English’s pre-defined templates.

3.4 Clauses with misaligned connectives

If the document contains a clause which has connectives whose order of precedence is not stated through indentation (as discussed in section 2.2.2), the editor marks the

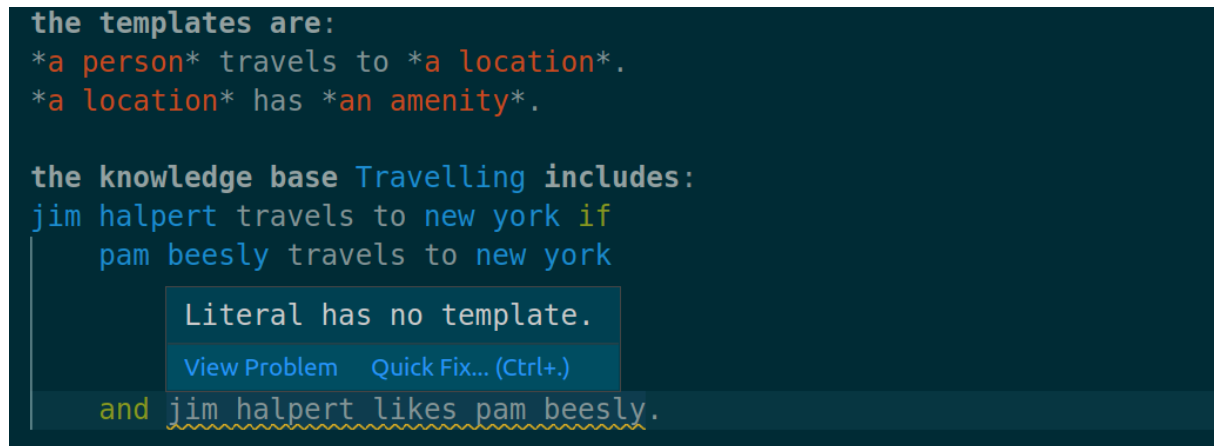


Figure 3.3: The editor identifying an atomic formula that does not match any of the templates.

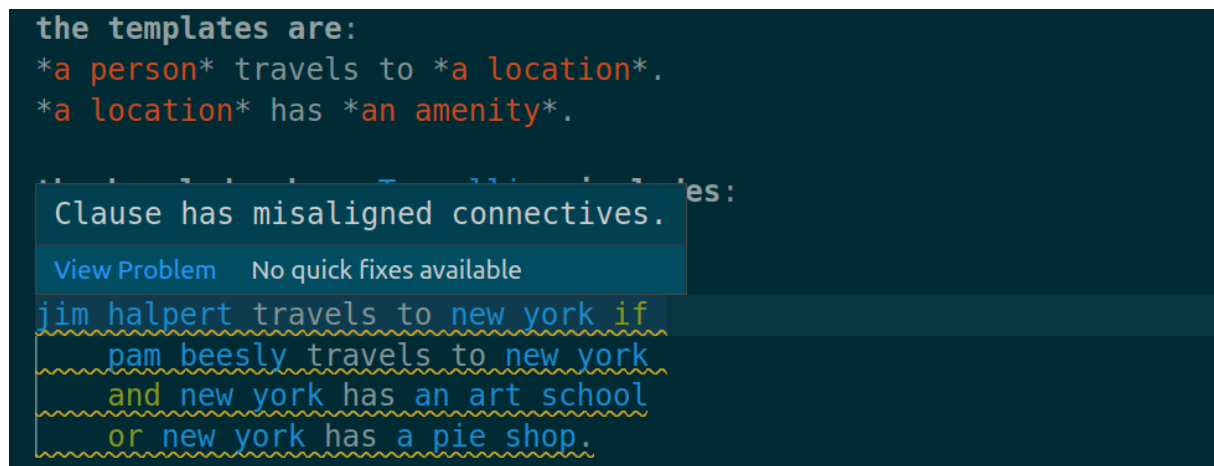


Figure 3.4: The editor identifying a clause with misaligned connectives.

clause with an underline representing a ‘warning’. On hovering over the clause, the editor produces the error message ‘Clause has misaligned connectives.’. This is shown in figure 3.4.

3.5 Code Actions

Figure 3.5 shows the process of generating a new template that conforms to atomic formulas. If a number of atomic formulas are marked as not conforming to any templates (discussed in section 3.3.1), hovering over any one of the atomic formulas and selecting ‘Quick Fix’ produces the ‘Generate a template’ option. When pressed, the editor generates a single template that conforms to every atomic formula that was marked with this error.

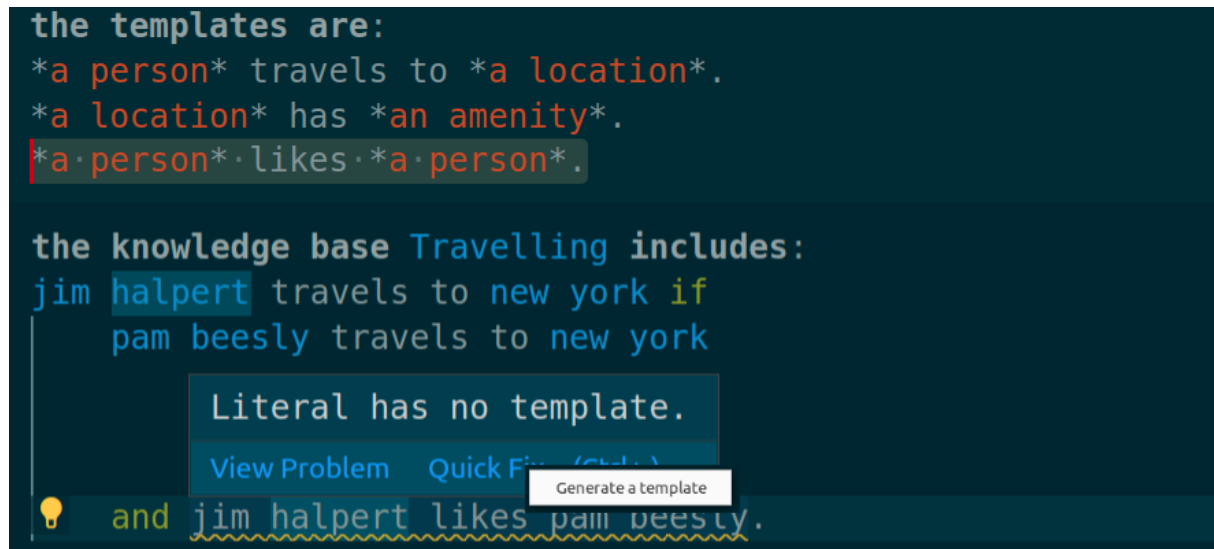


Figure 3.5: The editor generating a new template to match template-less atomic formulas.

If the marked atomic formulas contain terms which also feature in unmarked atomic formulas, then the types of those terms will feature in the template. This is shown in Figure 3.5: the terms `jim halpert` and `pam beesly` feature in unmarked atomic formulas and have type `a person`. This allows the generated template to feature the type `a person`.

3.6 Type Checking

3.6.1 Type Mismatch Errors

The comment (shown at the top of Figure 3.6) activates type checking features in the editor. If the following scenario occurs:

1. an atomic formula contains a term x , where x is assigned the type A
2. another atomic formula in the same clause contains the same term x , but x is assigned the type B
3. A and B do not have the same name, nor is one a sub-type of the other (discussed further in section 3.6.2).

then all the instances of the term are marked as warnings. If any one of the terms are hovered over, a message appears that explains that there is a type mismatch between the two stated types.

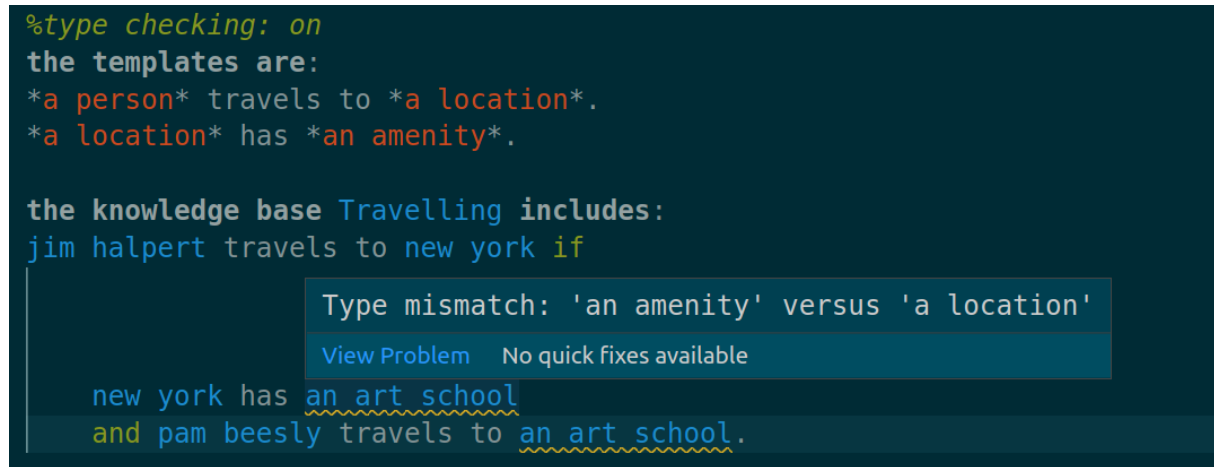


Figure 3.6: The editor identifying a type mismatch scenario. The term an art school appears across two atomic formulas. The term is assigned the type a location in the first atomic formula, but is assigned the type an amenity in the second. These types do not have the same name, nor is one a sub-type of the other.

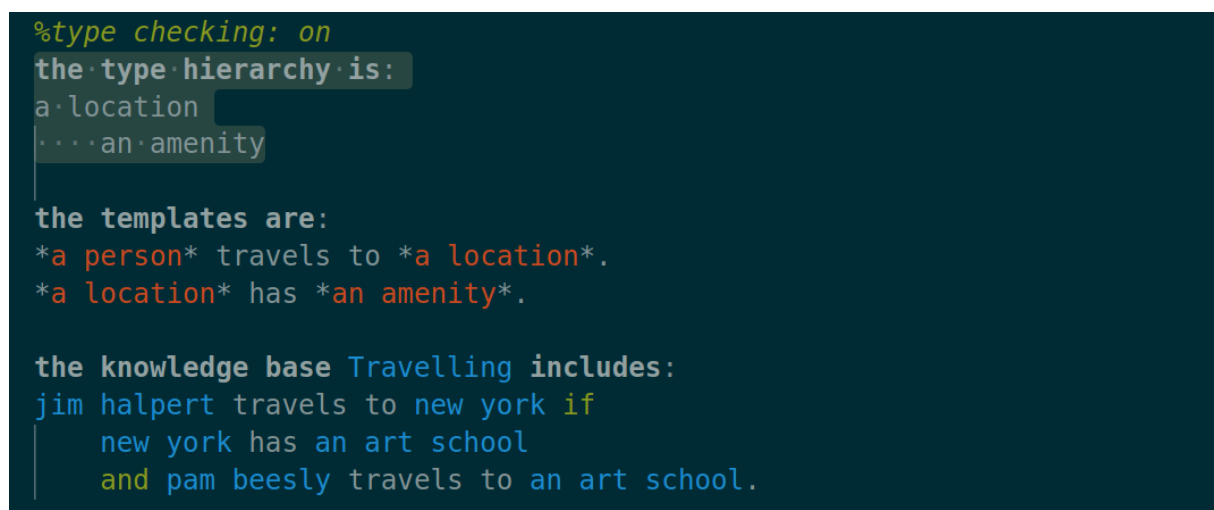


Figure 3.7: A type hierarchy that resolves the type mismatch error in Figure 3.6. The type a location is now a subtype of an amenity.

3.6.2 The Type Hierarchy

The header `the type hierarchy is:` marks the optional ‘type hierarchy’ section where a type hierarchy can be written. To write that the type B is a subtype of the type A , write the type name of B on a line below the type name of A , indented further than A ’s type name by a single tab. Continue this way with multiple types to form an indented list of the types.

The editor will use this type hierarchy in checking for type errors. For instance, in Figure

3.7, the type hierarchy states that type `an amenity` is a subtype of the type `a location`. This means that the usage of the term `art school` in the knowledge base does not result in a type mismatch error. Although the term is assigned two types, one type is now the subtype of the other.

Chapter 4: Literature Review

4.1 Language Servers

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 [30], 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 [21] 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 chosen editor that supports the Language Server Protocol, and begin programming in the editor.

However, building a language server does not come without difficulties. Bour et al [21], and the Visual Studio Code Language Server Extension documentation [14], describe two main challenges that Language Servers face, that of “incrementality” and “partiality”:

- Due to efficiency constraints, the IDE may only be able to send the portions of the document to the language server (incrementality)
- The language server has 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 simpler grammar; the only recursive feature that I will need to incorporate is that of modal atomic formulas. This makes it feasible for the language server to not need a separate, standalone parser. Instead, the language server will parse the document itself as and when needed.

4.2 Editor Features

4.2.1 Code Generation

There is also existing literature on boilerplate generation from existing code. Wang et al [33] created a powerful compilation agent that auto-generates Java boilerplate code from more succinct, 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. However, the recursive Logical English features from which I generate boilerplate code – that of modal atomic formulas – are much more constrained than the recursive features of Java. Thus I can favour a less complex representation over an AST.

4.2.2 Code Completion

Bruch et al [16] create the ‘Best Matching Neighbours’ algorithm for suggesting method calls in object-oriented code. The Best Matching Neighbours algorithm suggests code completions of a code library’s methods by learning from the patterns in large, pre-existing codebases that use the library. This does not have an analogue in Logical English: there are no code libraries that many documents draw from; each Logical English document defines all the templates and atomic formulas that it uses. This means that, in suggesting the completion an atomic formula, the Logical English editor must use a simpler, rule-based approach rather than a learning algorithm.

However, in their study, Bruch et al hypothesise that useful code completion [16, p. 214]:

- filters irrelevant suggestions, and
- presents suggestions in order of relevance

Bruch et al applied the two principles in their design of the Best Matching Neighbours algorithm and support the principles through their algorithm’s evaluation. These principles are general enough to be applied to the Logical English editor, and are used when suggesting completions for an atomic formula.

4.3 Type Systems

As noted by R. Davies, type systems are a central feature to programming languages: when used, they express the program’s fundamental structure [17, p. 1]. Partly because of this, the field of type systems is thoroughly researched and has a large body of literature.

A key function of the type system developed for Logical English is to assign types to atomic formulas.

Many popular programming languages that assign types to propositions do so based on the proposition's syntax. For instance, the programming language C++ assigns a type to an atomic formula purely based on the types of the atomic formula's terms. In C++, propositions are viewed as functions that take their terms as arguments and return a boolean value. This means that if a proposition has n arguments of type T_1, T_2, \dots, T_n then the type of the proposition is `std::predicate<T1, T2, . . ., Tn>` [9]. This is mirrored in Java in the `Predicate` class, where $n = 1$, and `BiPredicate` class, where $n = 2$ [6].

This type system classifies atomic formulas syntactically according to the type of terms that they describe. However, Logical English as a language seeks to minimise the syntactic rules it imposes on the user. I believe that grouping atomic formulas by their syntax is unintuitive; a Logical English document may contain atomic formulas that agree in the order and type of their arguments, but differ in their semantic meaning. Consider, for instance, the templates in Listing [?]

```
*a person* works at *a company* for *a number* years.
*a person* sues *a company* for *a number* dollars.
*a person* leaves *a company* for *a number* days.
```

Listing 4.1: Logical English templates that have differing semantic meaning, but under a syntactic type system would be grouped under the same type.

Each template in Listing [?] has the same three types in the same order. This means that, under a syntax-based type system, atomic formulas that have the structure of either of the three templates would be assigned the same type¹. A syntax-based type system will therefore not notice any errors if such atomic formulas are interchanged. However, users of Logical English may feel that these templates ought to be categorised in a way that differ from each other. Therefore, unlike the above object-oriented languages, I introduce a type system for Logical English where atomic formulas are assigned types according to their semantic meaning, as determined by the user.

¹In the notation of C++, this type would be `std::predicate<Person, Company, Number>`

Chapter 5: Project Requirements

In this section I describe the requirements that my project was to meet. These were decided on through discussions between myself and my supervisors. The requirements were based purely on the user functionality that the editor provides.

The editor must consist of three components to assist Logical English: a Syntax Highlighter, a Language Server and a Language Client. The syntax highlighter and language server must be cross-editor and cross-platform. This requirement meant that the editor can be used with many of the most popular programming editors on various operating systems with minimal configuration. Out of the three components, it is the language server that is the most complex.

5.1 The Language Client

The user uses the language client to interact with the features provided by the language server and syntax highlighter. Therefore, the language client must convey changes in the document to the syntax highlighter and language server. The language client must also incorporate the responses from the language server and syntax highlighter.

5.2 The Syntax Highlighter

The syntax highlighter must identify and label grammatical features in a Logical English given document. This must be done in a way that the language client can then use the labels to highlight the document. The syntax highlighter must identify both micro-features such as keywords and variable names, and macro-features such as section headers.

5.3 The Language Server

The language server must provide three functionalities to help the user write Logical English documents: code completion, error diagnostics, and suggested error fixes.

These requirements were not concrete at the outset. After researching and explaining what is reasonably possible, these three requirements were agreed on through discussion between myself and my supervisors during the early to middle stages of creating the editor.

My supervisors and I suggested that if there was time, after the above three features had been implemented, I could explore implementing a type-checking system in the editor.

5.3.1 Code Completion

When a user is in the process of writing an atomic formula, if the atomic formula could match a template, an option must appear for the editor to complete the remainder of the atomic formula according to the form of the matching template.

5.3.2 Error Diagnostics

The user must be informed if they make the following types of errors:

1. a atomic formula has been written that does not match any template
2. a clause has been written where the precedence of the connectives has not been made clear by appropriate indentation

5.3.3 Suggested Error Fixes

This feature was not a strict requirement, but was desirable. When the user writes a atomic formula that does not match a template, making error (1) above, an option should appear for the editor to write a new template that matches the atomic formula. This feature is not required since it was not clear at the outset when it is possible to algorithmically generate such a template, nor how difficult such an algorithm would be to implement.

5.4 The Type System and Type Checker

5.4.1 Requirements of the Type System

The Logical English development team were considering introducing a type system. This type system would re-interpret the argument names of an atomic formula's template as the types of the values in the atomic formula. This type system would be used to check for inconsistent uses of values: errors where

- an atomic formula contains a value x , where x is assigned the type A

- another atomic formula contains the same value x , where x is assigned the type B
- A and B are incompatible types.

I was tasked with specifying a suitable type system and implementing this type system in the editor. If finished, the editor would assign values their according types and notify the user when a type error was made. However, this feature was a suggestion, only to be explored if there was sufficient time once the other features had already been implemented.

Chapter 6: Design

6.1 System Overview

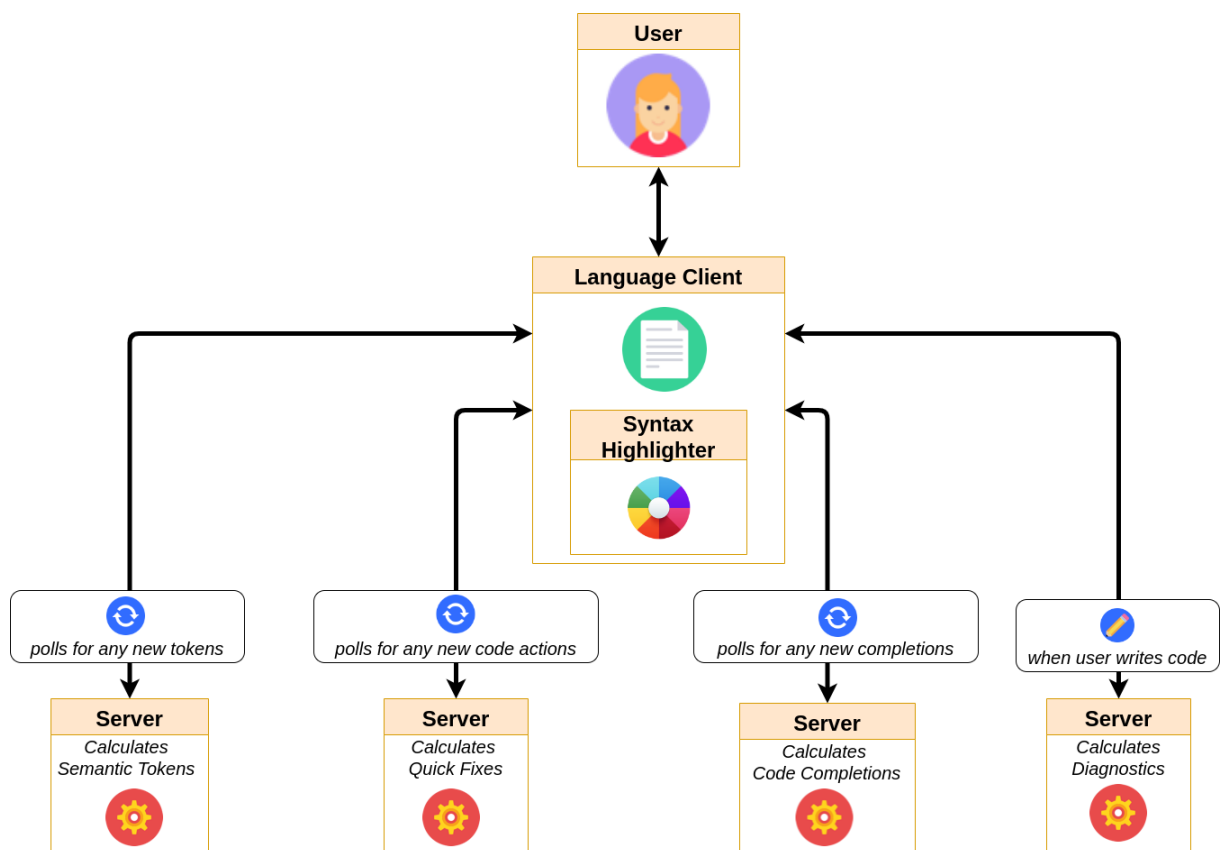


Figure 6.1: A diagram showing the overall structure of the Logical English editor.

Increase the font size.

The overall structure of the editor is described in Figure [?]. The user interacts with the editor through the language client. The language client takes on the rules of the syntax highlighter to highlight basic syntactic features of the user's document. The language client highlights semantic features of the document, along with presenting any code

completions or quick fixes, by routinely polling the language server for this data. The language server presents error diagnostics by requesting diagnostic data from the server whenever the user makes a change to the document.

6.2 Requirements Approach

6.2.1 Why a Language Server?

When deciding what type of language extension to create, we surveyed a variety of options.

Logical English is currently edited on the SWISH platform. This runs on Codemirror, a JavaScript framework for creating web-based code editors. This meant that writing the language server entirely in Codemirror was the initial 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, developing the extension in Codemirror 5 was also a risk. This was because migration to Codemirror 6 would involve a number of breaking changes [2], for example, changes that would have entirely restructured interacting with Logical English documents.

This prompted us to then consider Monaco.

Why were we considering Monaco? How would Monaco have worked with SWISH?

.

After researching other options, I found that an editor-agnostic language server would leave us open to using a variety of editors. Using the Language Server Protocol, a single language server would be able to connect to any language client that supports the protocol. This includes language clients written in online editors such as Codemirror 6 [31] and Monaco [32], along with some of the most popular desktop code editors [27], such as Visual Studio Code [14], Visual Studio [11] and IntelliJ [4].

I was hoping that Codemirror 5 would support the Language Server Protocol, so that I could immediately publish the editor to the SWISH platform. Unfortunately, the Codemirror 5 development team are not willing to integrate with the protocol [1].

Looking elsewhere, I found an unmaintained, alpha release of a Codemirror 5 plugin that claimed to be able to communicate with language servers using the Language

Server Protocol [7]. To test this, I created a simple language server, starting with the sample language server created by Microsoft [8] that had error diagnostic and code completion features, and extended the server with code action features. The sample server came with a language client, which displayed the three kinds of features as expected. However, after connected the language server to the Codemirror 5 plugin according to the plugin's instructions, the Codemirror 5 language client was only able to display the error diagnosis feature. Although I cannot be sure of why the Codemirror 5 client failed to display all three of the features, after inspecting the TCP packets with Ubuntu Linux's built-in `netcat` command, the cause appeared that the Codemirror 5 plugin was using a different encoding for its requests than the encoding that the language server used.

This lead my supervisors and I to have to decide between two options:

1. a Codemirror 5 extension, that would be usable immediately but may soon become outdated
2. a language server that would need to be used through a new editor, but could be used through a variety of editors

I advocated for the latter option and my supervisors agreed. Logical English is still in an early stage of development and adoption, so producing a language server would give the flexibility needed to branch out to all kinds of coding environments.

6.2.2 Why a Visual Studio Code client?

Although a language server could be connected to many different language clients, to keep the scope of the project manageable I focused on developing a language client for a single IDE. I had two requirements when searching for the right IDE to develop for: I was looking to maximise both the popularity of the IDE, and the IDE's ease of use.

Visual Studio Code is a highly popular IDE. Stack Overflow, one of the most popular websites for the programming community, conducts global surveys every year monitoring the programming community's trends. In 2021, out of over 80,000 responses, Visual Studio Code had "a significant lead as the IDE of choice across all developers", with over 70% of responders using the IDE [27]. This made Visual Studio Code a clear choice in terms of popularity.

However, I also considered two runner-up IDEs in the survey, Visual Studio and IntelliJ, with 29% and 33% of responders using the IDEs. As mentioned in the previous section, all three IDEs support language servers. However, unlike Visual Studio Code, Visual Studio and IntelliJ are complex IDEs with a larger application size, longer install time, and larger user interface. A large part of the target audience of Logical English are logicians and lawyers, neither group involving itself with enterprise programming.

This lead me to decide that the simpler the IDE, the better; having to use an enterprise-programming IDE would be inconvenient and off-putting.

6.2.3 Why a separate Syntax Highlighter?

Although language servers can mark code for highlighting, it is common to delegate the majority of the highlighting to the language client. This is done for efficiency reasons. It is often the case that some highlighting features can be described by applying simple search patterns to the document: rules such as ‘`if` is a keyword’ or ‘text of the form `*a _*` is a template argument’. These rules are computationally simple, so they do not need to be computed by the language server. In fact, doing so would incur delay in waiting for the server to receive the request to highlight the document, and respond to the client with the highlighting data. For this reason, the client applies these search patterns itself. This is done by specifying these search patterns in a document, referred to as the ‘syntax highlighter’ or ‘language grammar’, which the language client reads at it launches.

A language client written for Visual Studio Code requires a syntax highlighter written in a JSON document according to the TextMate grammar [25]. This is standard amongst language clients, with Codemirror 6, Visual Studio and IntelliJ also supporting syntax highlighting using TextMate grammar [3], [12], [5].

6.3 Development Framework

6.3.1 Language Server

The following features were needed from the framework used to create the language server.

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 latest 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, the framework had to be built around a strongly-typed programming language. This was both in order to avoid mistakes that could be detected before run-time, and to 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 frameworks: the `Microsoft.VisualStudio.LanguageServer` library, written for C# [11], and the `vscode-languageserver` library, written for TypeScript [14]. Since these language server libraries were both created by Microsoft ¹, they are both 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 it is easier to work with JSON in TypeScript rather than in C#. Useful features include de-structuring JSON, giving JSON objects unique types based on their fields, and treating JSON objects as implementing interfaces that have the same fields. This decision being made, the resulting framework was TypeScript run on `Node.JS` using the library `vscode-languageserver`.

6.3.2 The Syntax Highlighter

Since the syntax highlighter for a Visual Studio Code client is written in a single JSON document, the technology stack required was minimal. Rather than writing the JSON document directly, the TextMate grammar was written in a YAML document, from which the JSON document was then automatically generated using the command `yq` [26]. 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. According to the TextMate grammar, the search patterns used to specify which parts of the documents to highlight are written as regular expressions, which have a simpler syntax in YAML.

6.3.3 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.

¹This is because Microsoft also created the Language Server Protocol.

6.4 Design Methodology

The Language Client and Syntax Highlighter needed very little design. Hence the design discussed in this section will concern the language server.

6.4.1 Initial Design

My initial design methodology was to use global variables to store the document's current type hierarchy and current collection of templates. These two variables would be recalculated whenever the document changes its content. They would be read, via a read-only view, whenever they were needed to deliver features to the user.

On first consideration this seemed to be the most practical design. Re-calculating these two objects whenever the document received an update should ensure that these objects would stay up-to-date with the document. Because most of the language server's features depended on these two objects, it could be easy to accidentally modify their contents. If this happened, it would be quite hard to find the source of the error. Accessing the objects through a read-only view, rather than directly, should prevent such errors from occurring.

However, once I had set up a minimal language server, it was clear that this design approach would lead to concurrency issues. The language server interacts with the client asynchronously, whenever any one of the following events happen:

- the document changes its content
- the client requests code completion
- the client requests code actions (such as a quick fix)
- the client requests semantic highlighting information

For example, if the user updates their templates, the client will send the server a notification that the document changed, and may, shortly afterwards, send a request for semantic highlighting. The server would receive the notification that the document changed, and would start rebuilding the list of templates. However, the server may receive the request for semantic highlighting before the list of templates has finished updating. This would lead to the semantic highlighting being incorrect.

This was an issue that was explored in the Operating Systems course. One solution that was taught was to apply locks to the list of templates and the type tree. This would ensure that processes that wish to access the type tree or the list of templates only do so once the objects have finished being modified.

However, there is a special case of this concurrency problem that the use of locks would

not solve. The client communicates with the language server through standard console input/output. This means that all communication happens through a single channel. Therefore, if the document update notification is sent first, but on a separate thread to the semantic highlighting request, then it is possible for the semantic highlighting request to be written to console input first. This means that the server would receive the semantic highlighting request before it updates the template list – irrespectively of whether the template list has locks or not.

While the above special case might be rare in practice, the possibility of this issue hinted to me that there might be an overall better design than using global variables for the template list and type tree. The only way to avoid such an issue is for the templates and type tree to be updated when processing each request. This way, the templates and type trees could now be constant, local values to each request. Although this would result in duplicate work in the case that the document does not change between requests, this is the only way to avoid the concurrency issues. This lead me to research design based around a lack of mutable state.

6.4.2 Current Design Methodology

Avoiding Mutable State

The overall design methodology was mainly guided by the principles outlined in “Java to Kotlin: A Refactoring Guidebook” by Duncan McGregor and Nat Pryce [20]. Although taking examples from refactoring Java code into Kotlin, the book outlined design principles based on the ideas of minimising mutable state in classes and using ‘pure’, stateless functions.

McGregor and Pryce give two relevant scenarios in which avoiding mutable state in classes and functions is beneficial. Firstly, objects and functions without mutable state do not change as their contents is iterated over, or otherwise accessed across multiple instructions. [20] This is exactly the solution that I was looking for in avoiding the concurrency problems. Further, pure functions produce the same output if they are called multiple times[19]. This is ideal as with each type of request now generating its own local type tree and list of templates, if the document has not changed between two requests, then the same templates and type tree should be produced. In other words, given the same document as input, the factory methods that generate the templates and type tree should produce the same output.

Another important reason for favouring minimal state programming is for the ease of understanding the code. Limiting the use of global, mutable state in turn limits the scope for error. As put by McGregor and Pryce, when looking for the source of an error [18],

If there is a possibility that a function could mutate shared state, we have

to examine the source of the function and, recursively, every function that it calls, to understand what our system does. Every piece of global mutable state makes every function suspect.

This principle extends even to beyond debugging errors. By the same principle, the greater the dependency on global, mutable state, the harder it is to reason about the code's behaviour. This is true both from a formal standpoint, and from a point of practical understanding. Limiting the scope of mutable state to local to a function, a `for` loop or an `if` statement, limits the effect the mutable state has, and therefore limits the complexity of the code.

This principle is especially important because of the time scope of this project; the Logical English team will maintain and extend the editor once my work is complete. Therefore it is a priority that the editor's source code must be easy to understand, debug and improve on for this external team who are not familiar with the source code.

Chapter 7: The Type System

Add sources.

7.1 Requirements

The type system for Logical English had to model Logical English's domain of discourse. Logical English has two categories of terms:

1. atomic terms, such as names of real-world objects
2. atomic formulas, which feature as terms in expressions of modality.

Since atomic formulas may contain terms, these terms will also need to be typed.

Correct the terminology.

The type system has two purposes: to effectively assign types to the two categories of terms, and to use the assigned types to identify inconsistent placements of terms.

7.2 The User-based Type Hierarchy

In Section [?] I determined that a new type system is needed which is fundamentally user-driven. The user would group atomic terms and atomic formulas in the same way: by assigning types that are entirely of their own choosing, having no connection to either either syntax or logical semantics. Here I give a specification of this new type system, based on existing features of Logical English and on the needs of the user.

7.2.1 The need for a hierarchy

The type system relates types to each other through a hierarchy, where types lower in the hierarchy are subtypes of types directly above them.

Motivation for a type hierarchy comes from the structure of English sentences. Sentences in English often contain terms whose implicit types change in specificity.

Find a study of this.

For instance, consider the sentence “I love my pet: I adore the way she wags her tail.” This simple English sentence has the term ‘pet’ have three different levels of specificity of its type, from ‘a pet’ to ‘a female pet’ to ‘a female pet with a tail’ (presumably a dog). As more detail is specified in a sentence, the type of its terms narrows in refinement.

Supporting scenarios such as this in Logical English was then a key feature of the type system. This meant that a ‘subtype’ relation \leq between types was needed, where $A \leq B$ specifies that terms of type A can be used where terms of type B are expected.

When studying what kind of a type hierarchy would emerge from this relation, I found that this relation obeys the three axioms of a partial order:

- **Reflexivity:** Every type A must be a subtype of itself, since, of course, terms of type A can be used whenever terms of type A are expected.
- **Anti-symmetry:** If $A \leq B$ and $B \leq A$ then the type A and B can be used interchangeably. This means that, in terms of checking for incompatible usages of terms, A and B are equal.
- **Transitivity:** If terms of type A can be used whenever terms of type B are expected, and terms of type B can be used whenever terms of type C are expected, then it is safe to use terms of type A whenever terms of type C are expected.

7.2.2 The structure of the type hierarchy

The structure induced by the subtype relation

Since \leq is a partial order, the type hierarchy induced by \leq forms a Transitive Directed Acyclic Graph[23, p. 49]. This imposes the following requirements on the type hierarchy:

1. Edges in the hierarchy are directed, with edges pointing from types to their subtypes
2. There are no loops in the type hierarchy: it must be impossible to start at a type and, by only visiting other subtypes, arrive back at the same type
3. If $A \leq C$, then the connection from A to C can be omitted if there is another type B with $A \leq B$ and $B \leq C$

The type hierarchy is disconnected

In Logical English, atomic terms and atomic formulas are two disjoint categories of terms: there are no terms that are both atomic terms and atomic formulas, making it

an error to supply an atomic term where an atomic formula is expected, and vica versa. This means that the type hierarchy is made up of two disconnected Transitive Directed Acyclic Graphs: one containing the types of atomic terms, and the other containing the types of atomic formulas.

Top types of the type hierarchy

Some atomic formulas apply to any atomic terms, no matter their type: for instance, propositions of the form x is x . This requires a ‘top type’ for these terms, of which every other type is a subtype. A suitable name for the top type could be `a thing`, taken from everyday English.

Similarly, there may be modal atomic formulas that apply to any other atomic formula: for instance, modal atomic formulas of the form `the judge concurs that` x . This would also require a ‘top type’ for atomic formulas; a possible name for this top type could be `a proposition`.

Draw a diagram of this hierarchy once `unknown` has been clarified.

Chapter 8: Implementation

8.1 Data Representation

8.1.1 Description of Problem

When providing diagnostics, semantic highlighting, code completion or quick fixes, the initial task of the language server is to extract the document's templates and atomic formulas. It was clear early on that these are the fundamental problems: the better the representation the editor has for the templates and atomic formulas, the easier all subsequent work on them becomes. It was important to design representations that:

- were lightweight: templates and atomic formulas would be reloaded every time the document received an update
- focused on grammar: it should be easy to query atomic formulas or templates based on their grammatical structure
- did not stray too far from the Logical English syntax: converting between Logical English and the editor's representations should be kept simple
- were not too distinct from each other: templates and atomic formulas often feature together in queries and have related syntax

Based on the last requirement, I first designed a representation for templates, then applied the ideas to design a similar representation for atomic formulas.

Initial Template Design

Initially, the most obvious and simple design for a template was as a list of tokens, called 'elements' ¹, each element being a string. These elements would either refer to a template's argument name, or text that surrounded the arguments (and therefore constituted part of the predicate name). For instance, the Logical English template

`*a person* shops at *a shop* to buy *an item*.`

¹The name 'elements' is used to distinguish from the tokens used in highlighting the document

is represented as the list of strings

```
["*a person*", "shops at", "*a shop*", "to buy", "*an item*"]
```

It was quite efficient to generate this list of elements. The list was achieved through splitting a Logical English template by a regular expression that identified substrings of the form `*a __*` or `*an __*`.

Although the design was lightweight and easy to implement, the more the design was used the clearer it became that it did not capture enough of Logical English's grammar. Lots of duplicate work had to be re-done whenever this representation was used: specifically, identifying which elements are template arguments, along with filtering the list of elements to obtain the list of argument names, or the list of strings that constituted the predicate name.

It became clear that, effectively, the list consisted of two different types of items: template arguments, on the one hand, and surrounding text on the other. Based on the awkwardness of use and the fact that I had plans to give template arguments a richer type structure, it was clear that the design needed improving.

8.1.2 Element Representation

The next level of abstraction was to abstract the two different kinds of elements into two different types. The class `Type` was created to represent a template argument². This `Type` class contained the template argument's name and was given additional structure when the type hierarchy was implemented. The class `Surrounding` was created to represent surrounding text that lies between types. As per my design philosophy, these are both lightweight types that store immutable values.

8.1.3 Template Representation

A template's elements were now a list consisting of either `Type` or `Surrounding` objects. The next logical step was to have a `Template` class to be a wrapper class around this list. The `Template` class provided a read-only view to the list of elements. It also exposed methods that queried the elements. These methods included obtaining the template's types or surrounding text – what was previously done manually – along with more advanced queries that will be discussed later.

An overview of the above design is given in pseudocode in Listing 8.1

²Before I implemented the type-checking system, this class was called 'TemplateArgument'. However, it will be easier to now only describe the final iteration of the editor.

```
class Type:
    name: constant string
    ...

class Surrounding:
    text: constant string
    ...

class Template:
    elements: constant (Type | Surrounding)[]
    public getTypes(): Type[]
    public getSurroundings(): Surrounding[]
    ...
```

Listing 8.1: An overview of the `Type`, `Surrounding` and `Template` classes.

8.1.4 Atomic Formula Representation

Once I had solved the `Template` design problem, I applied the same principles to creating a design for the atomic formulas. Templates consist of either surrounding text or type names; atomic formulas consist of either surrounding text or terms. Thus the analogue design for atomic formulas was clear. Create a type `Term` to represent terms of an atomic formula. Then create an `AtomicFormula` class with an `elements` list that consists of either `Surrounding` or `Term` objects. The description for the `AtomicFormula` class is given in Listing 8.2

```
class AtomicFormula:
    elements: constant (Surrounding | Term)[]
    type: constant Type
    public getTerms(): Term[]
    public getSurroundings(): Surrounding[]
    ...
```

Listing 8.2: An overview of the `AtomicFormula` class.

Unfortunately, designing the type `Term` was not straightforward. As discussed in Section 7.1, a term of an atomic formula can either be data, or (if the atomic formula is higher-order) another atomic formula. The `Data` class holds the Logical English term it represents as a simple string. However, since both kinds of terms are typed, both the `Data` class and the `AtomicFormula` class also hold a reference to their corresponding `Type` object.

Initially it seemed like the `Data` and `AtomicFormula` classes were enough to capture the possibilities of the values of Logical English terms. However, when I began working on parsing atomic formulas it became clear that there was a third option. Consider the Logical English extract in Listing 8.3.

```
the templates are:  
*a person* believes that *a proposition*.  
  
the knowledge base Higher-Order Atomic Formula includes:  
fred believes that the moon is made of cheese.
```

Listing 8.3: An extract of a Logical English document containing a higher-order atomic formula, in which the argument atomic formula does not match a template. Neither the `Data` class nor the `AtomicFormula` class can properly represent the argument atomic formula.

In Listing 8.3, the two arguments to the atomic formula `fred believes that the moon is made of cheese` are `fred` and `the moon is made of cheese`. The latter term `the moon is made of cheese` appears to be another atomic formula, since it is prefaced by `that`. This means that it is incorrect to represent the term with the `Data` class. However, the term does not have any corresponding templates; Logical English cannot extract its terms or its surrounding text. This means that the `AtomicFormula` class cannot represent the term either.

This prompted a third class, `TemplatelessFormula`, to represent atomic formulas that did not conform to a template and could not have their elements extracted. Since it was always a possibility that when an atomic formula – or any term at all – was expected, the result could be a `TemplatelessFormula`, this introduced many edge cases into the codebase. This was a symptom of the problem of partiality discussed in Section 4.1.

8.1.5 Section Representation

Along with representing Logical English data, it was also important to be able to refer to where the data lies in the document. This is crucial in highlighting features of the document, providing diagnostic error underlines, and identifying the current atomic formula that the user is typing.

The immediate approach would have been to add a `range` field to each of the above classes that specifies where the data begins and ends in the document. However, attaching range data to the representations themselves was not an option for two reasons: by the principle of Separation of Concerns [22, p. 183] the representations are “abstract”: they represent what a Logical English construct is, not where it happens to lie in a document.

`ContentRange<T>`

The alternative solution was to have a class that wraps data, supplying an additional range field. For a given type `T` (a string, a `Template` or any other kind of content), a `ContentRange<T>` has a `content` field of type `T`, and an immutable `range` field. The `range`

field stores the beginning and the end of the content, in the (line number, character number) form that `vscode-languageserver` uses ³.

8.2 Semantic Highlighting

8.2.1 An Overview

The semantic highlighting feature highlights the terms of each atomic formula in the document. To identify the terms, the templates are first read from the document and represented as `Template` objects. To each atomic formula, the closest-matching `Template` object is assigned. Using these `Template` objects, the terms of atomic formulas terms are recursively identified and highlighted.

The central problem here is extracting the elements of an atomic formula according to the template that the atomic formula corresponds to. Following the principle of Problem Decomposition, the problem of judging how well an atomic formula matches a template is broken down using the above problem:

1. extract the atomic formula elements under the (temporary) assumption that the atomic formula fully matches the template
2. compare the resulting elements against the template's elements to score how well the atomic formula matches the template

8.2.2 Extracting the elements of an atomic formula

This is a fundamental problem that is used by many of the language server's features, such as type checking a literal's terms and ranking literal completions, along with semantic highlighting. This lead me to spend a long time trying different approaches to this problem.

Talk about these other algorithms and their limitations.

The final algorithm leverages the assumption that the template matches the atomic formula. By this assumption, the template and the atomic formula share the same surroundings. Thus comparing the template's surroundings against the atomic formula yields the atomic formula's terms.

The resulting algorithm written in TypeScript is 55 lines long. However, with a liberal use of pseudocode, it is condensed in Listing 1. First, the trivial case is dealt with

³A downside to this approach is that `T` could be any type whatsoever, including types that do not make sense (such as the `void` type, or the type of a function). However, there are not enough commonality between valid values of `T`, such as `string`, `Template` or `AtomicFormula`, to constrain `T` effectively.

Algorithm 1 An algorithm to extract the elements of a literal according to a template

```

1: procedure EXTRACTTERMS(template elements , formula)
2:   formula elements  $\leftarrow$  []
3:   if template elements.length = 1 then
4:     s  $\leftarrow$  new Surrounding(s.text  $\leftarrow$  formula)
5:     append s into formula elements
6:     return formula elements
7:   end if
8:
9:   for [type, surrounding] sequence in template elements do
10:    if surrounding.text is substring of formula then
11:      s  $\leftarrow$  surrounding
12:    else if surrounding.text starts with an end substring of formula
then
13:      text  $\leftarrow$  the end substring of formula
14:      s  $\leftarrow$  new Surrounding(s.text  $\leftarrow$  text)
15:    else
16:      end loop
17:    end if
18:
19:    data  $\leftarrow$  formula substring before s.text
20:    if data is not empty then
21:      d  $\leftarrow$  new Data(d.value  $\leftarrow$  data, d.type  $\leftarrow$  type)
22:      append d into formula elements
23:    end if
24:
25:    append s into formula elements
26:    formula  $\leftarrow$  formula substring after s
27:  end for
28:
29:  el  $\leftarrow$  elements.last element
30:  if formula is not empty and el is a Type then
31:    d  $\leftarrow$  new Data(d.value  $\leftarrow$  formula, d.type  $\leftarrow$  el)
32:    append d into formula elements
33:  end if
34:  return formula elements

```

where the template list has no types. In this case, the atomic formula has no terms, so the resulting list of elements consists of the entire atomic formula as a `Surrounding` object.

In the general case, the algorithm iterates over each type and subsequent surrounding in the template element list. If the surrounding is contained in the atomic formula, then the text before the surrounding is interpreted as data. The data and surrounding are appended to the list of formula's elements. The atomic formula is treated as a queue: the data and surrounding is removed from the atomic formula. This loop continues until a surrounding does not match the atomic formula.

In iterating over sequences of the form `[type, surrounding]`, if the template list ends with a type then it will not be visited in the loop. In this case, whatever is left of the atomic formula string is interpreted as data corresponding to the last type.

Algorithm 1 can also extract the elements of 'incomplete atomic formulas': substrings that an atomic formula begins with. These incomplete atomic formulas occur when the user is writing an atomic formula (in the usual way, by appending text to the end). With the `else if` clause on line 12, incomplete atomic formulas that end with the beginning substring of a surrounding can also be parsed. This means that elements can be extracted as the user is writing an atomic formula. This is a highly useful property that allows the diagnostic, code completion and semantic highlighting features to occur as the user is writing the document.

The reason why such a long algorithm is needed is to deal with the edge case of what happens if a surrounding also appears as a term. For example, a scenario where a merchant packages and sends items could be described with the template `*a merchant* ships *an item*`. If we have a merchant who packages and sends ships, then the corresponding literal would be `the merchant ships ships`. By treating the atomic formula as a queue, removing text that has been visited, this edge case is handled by algorithm.

8.2.3 Matching a template to an atomic formula

Determining whether a template matches an atomic formula is a simple application of Algorithm 1. Once the atomic formula's elements have been extracted, it suffices to check whether the surroundings of the atomic formula match the surroundings of the template.

8.2.4 Finding the template that best matches an atomic formula

Initially, it was assumed that only one template can match an atomic formula. This was convenient, as I could simply use the first (assumed only) template that matches the atomic formula to extract its terms.

However, as the editor was being developed, I soon saw how this was often false. This was most clearly visible when “default” templates were implemented – general templates, such as **a thing* is *a thing** that were implicitly present in every Logical English document. Consider the following Logical English document:

```
the templates are:
*a thing* is *a thing*.
*a person* is a beneficiary of *a will*.

the knowledge base Counter-Example includes:
jane is a beneficiary of her father's will.
```

Listing 8.4: A Logical English document containing two templates that both match an atomic formula.

In Listing 8.4, the first template to match the literal *jane is a beneficiary of her father's will* is the template **a thing* is *a thing**. However, this is not the template that *should* match. Using this template to extract the terms of the atomic formula *jane is a beneficiary of her father's will* will lead to a beneficiary of her father's will being treated as a single term.

This motivated a ‘match score’ between a template and an atomic formula: the higher the score, the closer the match. Since the surroundings are used to determine whether a template matches an atomic formula, I reduced the problem to judging the match between the surroundings. In this way, the match score between a template and an atomic formula was determined as the total length of the surroundings extracted in Algorithm 1. Under this match score, the highest scoring template is used to extract the literal's terms.

8.3 Completion

8.3.1 Completing the remainder of an incomplete atomic formula

When the user is writing an atomic formula, various options for the remainder of the atomic formula are suggested. These suggestions are calculated using the templates that the incomplete atomic formula could correspond to. This is not as straightforward as searching for which template match the atomic formula, in the sense of 8.2.3, because the atomic formula will be incomplete, and so may not contain each of the template's surroundings. Instead, the templates are ranked by their match score against the atomic formula.

There is already some nuance here. Templates that are ranked highly may be irrelevant: for instance, the template **a thing* is *a thing** shares the surrounding *is* with the incomplete atomic formula *a person is a beneficiary of*. These templates cannot be

ruled out algorithmically. They will, however, be out-ranked by templates with longer matching surroundings. If the top three templates are taken every time, then the results from these erroneous matches may either appear lower in the list, or may not appear at all.

Each of the three best-matching templates are then used to suggest the rest of the atomic formula. To generate the rest of the atomic formula, the terms that the incomplete atomic formula contains are substituted into each template. Any remaining template arguments are presented to the user as placeholders. When the user selects a template, these placeholders can be instantly navigated to by pressing Tab, allowing the user to quickly fill in the placeholders. This is done through Visual Studio Code's 'code snippet' feature, whereby text wrapped in `{ }` is treated as a placeholder that can be navigated to.

What other language clients support this? Does this affect the universality of the language server?

8.4 Error diagnosis

8.4.1 How a language server diagnoses errors

Research this in more detail

8.4.2 Diagnosing template-less atomic formulas

The editor diagnoses errors where an atomic formula does not have a matching template. Having already solved the problem of determining whether a template matches a literal, this at first glance appeared simple. However, as I was testing this feature I found that this criterion was too broad. Specifically, incomplete literals (e.g. literals that are being typed) will, in general, not match a template. This means that every literal that is being typed will be diagnosed as incorrect until it is finished.

Unfortunately, this is impossible to fix in the current version of the Language Server Protocol. When the client requests diagnosis information, all that is supplied is the current state of the document. To determine which literal is being typed by the user, the user's cursor position would also be needed. Since the client supplies the cursor position when requesting auto-completions, one possible workaround could be to store the cursor position when auto-completions are requested and hope that this position stays accurate when it is time to provide diagnostic information. However, I expect that this approach is highly ineffective in practice.

8.4.3 Diagnosing clauses that have misaligned connectives

The editor also diagnoses errors where a clause has misaligned connectives. Logical English requires that each literal begins on its own line. The lines starting with the keywords `and` and `or`, which have equal precedence, have their indentation compared. If two literals have the same whitespace, but one begins with `or` while the other begins with `and`, then the precedence of the connectives is ambiguous. By keeping track of where the clause occurs in the document, the clause is then marked with the error message “clause has misaligned connectives”.

The nuance here is with the term ‘same whitespace’. There are two common ways to indent lines: tabs and spaces, roughly equally common amongst programmers [10]. There is no standard, single size of a tab in terms of spaces. For instance, IBM documents the popular ‘Courier’ font to allow a tab size ranging from 0.7 points to 20 points. Thus, if one person indents literals using tabs, and the other using spaces, then it is up to interpretation as to whether the indentation is correct or not.

The popular programming language Python 3, which has a similar dependence on consistent indentation, attempts to infer a reasonable amount of spaces that a tab must represent based on the document. If it cannot do so, it raises an error [29]. Since this is a tangential issue to finding misaligned connectives, I left implementing such a feature for future work.

8.4.4 Diagnosing type mismatches

Feature Overview

The most complex feature to implement was the type checking system. Since this feature is experimental, it was important not to clash with any existing features or hinder the experience of a user who did not want their types to be checked.

The approach I chose in implementing type checking was to have the feature disabled, only being enabled with an explicit `%type checking: on` comment. The type hierarchy was also designed to be as minimal as possible in its presentation, requiring little wording and being easy to read.

Initial design: flat type hierarchy

When experimenting with implementing this feature, I first implemented type checking before introducing a type hierarchy. In each clause, the literals were extracted, and, using the document’s templates, the terms were extracted from each literal. These terms were assigned a type, but there was no notion of sub-type or super-type. If two terms were found that had the same value but different types, a type mismatch error message was generated.

This design was a lot more inconvenient to use than I expected: many more error messages were generated than I anticipated. This was mainly because of the default templates. Since the default templates are very broad, they use short, generic placeholder type names, such as *an A*, *a B*, *a C*, *Or a thing*.

This caused problems for two reasons. Firstly, if a clause featured an atomic formula that conformed to a built-in template, then the type names clashed with the more specific type names used in other atomic formulas. This problem could only be resolved through a type hierarchy by ensuring that the types used in the default templates would be super-types of all other types.

However, there was a second issue, in which the type names often clashed amongst themselves. Take, for example, a Logical English program in which the empty list is reversed.

```
the templates are:
% a default template
*an A* is the reverse of *a B*.

the knowledge base Type-Clashing includes:
[] is the reverse of []. %type mismatch error
```

Listing 8.5: A Logical English program using a default template in which the types are inconsistent.

(In reality, the template above would not need to be stated, since it is included by default.) In Listing 8.5, a type mismatch error is generated. This is because `[]` is both of type *an A* and *a B*.

Cite the default templates, or at least, talk about them in the LE specification.

This required the template argument names to be renamed in light of the fact that the names were now used as types. With the template in Listing 8.5, for instance, being replaced with

```
*a list* is the reverse of *a list*.
```

there would be no type clashing error. Renaming the types would not cause any errors with any older Logical English code, since the argument names of templates were not used by the Logical English engine.

Are the template argument names used at all by the engine?

Diagnosing with a type hierarchy

The type hierarchy was implemented as a node-based tree structure. Each node, represented by the class `Type`, stored the name of the type that it represented, along with a list of references to each immediate subtype. Due to time constraints, the type hierarchy was given one top type, named `a thing`, as opposed to the two separate top types prescribed in 7.2.

One point of consideration was the format in which the user would be required to write the type hierarchy. There are various ways of representing a type hierarchy in text form: the one that balanced both ease of readability and ease of writing was the indented list [15], where each parent is followed by an indented list of its children. (For a further description of how to format such a list, see the User Guide.) It is a standard algorithmic problem to convert an indented list to a tree, to which the solution is well-known.

Once the type hierarchy was implemented, I could build on the type identification system by using the type hierarchy to check whether two types were compatible. Two types are compatible if they are equal, or if one is a child of the other in the type hierarchy. This reduced checking for type compatibility to a standard tree search problem, to which the solution is again well-known.

8.5 Quick Fixes

8.5.1 How a language server provides quick fixes

Research this in more detail

8.5.2 Generating a new template that matches given atomic formulas

Template Generation using Least General Generalisation

When a document contains atomic formulas that have no matching template, the editor seeks to generate a new template that matches them all. This is done in multiple iterations, with each iteration generalising the template.

The first candidate template is generated according to the Least General Generalisation algorithm. This is an algorithm written by Gordon Plotkin [28, p.155] that gives, from two or more ‘words’ (corresponding to ‘atomic formulas’), a single generalisation which has variables wherever the two words have differing terms. This is a generalisation in the sense that both ‘words’ can be obtained by substituting the necessary terms in place of the generalisation’s variables. The generalisation is ‘least general’ in the sense

that it has no more variables than necessary to be a generalisation.

The algorithm followed by the editor is an adaptation of Plotkin's original algorithm. Each literal is split into a list of space-separated words. It is assumed that there is a common template which matches all the literals.

Initially, the common surroundings are identified as the intersection of all the space-separated words. On testing, this criteria failed to account for when the same surrounding occurs more than once in each atomic formula. Instead, the surroundings are calculated by stepping through the words of an arbitrarily chosen atomic formula. Each word is checked to see whether it is contained in all other atomic formulas: if so, then it is added to the list of surroundings, and its first occurrence is removed from the other lists.

Having found all of the surroundings, the variables are found by taking the first atomic formula and removing the surroundings. If this template matches all the other atomic formulas, then it is returned. Otherwise, the atomic formulas are assumed to have been incompatible.

Template Refinement from the re-use of terms

It was often the case that the least general generalisation was not enough to generate an accurate template. Either there were too few atomic formulas, or the atomic formulas did not vary in every term. In trying to fix this problem, I noticed that I was not exploiting the context(i.e. the clause) in which the literal was written. If a literal borrows a term from another literal, then we know about that term, and can generalise it into a variable.

This was done by giving the `Template` class a method to generate a more general template by generalising a given term into a variable. Applying this method successively to all the surrounding terms that feature in each template-less literal gave much more accurate templates. This also allowed a single literal to be generalised into a template – a feature that is impossible with least general generalisation. This also told us the type to place inside the template, since the type of each term is known.

Chapter 9: Evaluation

9.1 Testing

9.2 Feedback

Chapter 10: Conclusions and Further Work

10.1 Conclusion

10.2 Challenge and Reflection

10.3 Further Work

We haven't finished yet.

Chapter 11: User Guide

11.1 Installing the extension for Visual Studio Code

The easiest way to install the extension is from Visual Studio Code. This is done by navigating to the Extensions window and searching for `NikolaiMerritt.logical-english-vscode`. The extension is hosted at <https://marketplace.visualstudio.com/items?itemName=NikolaiMerritt.logical-english-vscode> and can be installed from there too. Once installed, Visual Studio Code will keep the extension up to date automatically.

11.2 Running the language server manually

To run the language server manually, you will need

- node (<https://nodejs.org/en/download/>) version 16.4.5 or newer
- node package manager (npm) version 8.12.2 or newer

Having downloaded the repository at <https://github.com/nikolaimerritt/logical-english-vscode>, navigate to the `vscode-package` directory. Run `npm install`. This installs all the node packages for the syntax highlighter, language server, and visual studio code client.

To run the language server, from the `vscode-package/server` folder, run `npm link`. (You may have to run this command with super user or administrator privileges). This creates the language server as a node package called `le-server`. Test that it launches without errors by running `le-server --stdio`. This should produce no output.

As per the Language Server Protocol, the language server, `le-server`, listens to input from standard console input (`stdio`).

11.3 Reading the source code

The source code for the entire extension, which includes the language client, syntax highlighter and language server, can be found at <https://github.com/nikolaimerritt/logical-english-vscode> under the MIT license.

Bibliography

- [1] Correspondence with the codemirror 5 development team on language server protocol integration, . URL <https://github.com/codemirror/codemirror5/issues/4326>. pages 26
- [2] Codemirror: Migration guide from codemirror 5 to codemirror 6, . URL <https://codemirror.net/docs/migration/>. pages 26
- [3] Textmate grammars support for codemirror, . URL <https://github.com/zikaari/codemirror-textmate>. pages 28
- [4] Lsp4intellij - language server protocol support for the jetbrains plugins, . URL <https://github.com/ballerina-platform/lsp4intellij>. pages 26
- [5] (textmate highlighting in intellij) textmate, . URL <https://www.jetbrains.com/help/idea/textmate.html>. pages 28
- [6] Package java.util.function (java standard reference). URL <https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>. pages 21
- [7] lsp-editor-adapter (alpha), . URL <https://github.com/wylieconlon/lsp-editor-adapter>. pages 27
- [8] Lsp example, . URL <https://github.com/microsoft/vscode-extension-samples/tree/main/lsp-sample>. pages 27
- [9] std::predicate (c++ standard reference). URL <https://en.cppreference.com/w/cpp/concepts/predicate>. pages 21
- [10] Stack overflow developer survey 2017. URL <https://insights.stackoverflow.com/survey/2017>. pages 45
- [11] Add a language server protocol extension, . URL <https://docs.microsoft.com/en-us/visualstudio/extensibility/adding-an-lsp-extension?view=vs-2022>. pages 26, 29

- [12] (textmate highlighting in visual studio) add visual studio editor support for other languages, . URL <https://docs.microsoft.com/en-us/visualstudio/ide/adding-visual-studio-editor-support-for-other-languages?view=vs-2022>. pages 28
- [13] Logical english on the swish online editor, 2022. URL <https://logicalenglish.logicalcontracts.com/>. pages 5
- [14] Language server extension guide, 2022. URL <https://code.visualstudio.com/api/language-extensions/language-server-extension-guide>. pages 19, 26, 29
- [15] Anonymous. Tree datastructures (converting an indented list to a node-based tree). URL http://rosettacode.org/wiki/Tree_datastructures#Python. pages 47
- [16] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605580012. doi: 10.1145/1595696.1595728. URL <https://doi.org/10.1145/1595696.1595728>. pages 20
- [17] R. Davies. *Practical Refinement-Type Checking*. 2005. URL <https://www.cs.cmu.edu/~rwh/students/davies.pdf>. pages 20
- [18] N. P. Duncan McGregor. *Java to Kotlin: A Refactoring Guidebook*. O'Reilly Media, Inc., . pages 31
- [19] N. P. Duncan McGregor. *Java to Kotlin: A Refactoring Guidebook*. O'Reilly Media, Inc., . pages 31
- [20] N. P. Duncan McGregor. *Java to Kotlin: A Refactoring Guidebook*. O'Reilly Media, Inc., . pages 31
- [21] B. et al. Merlin: A language server for ocaml (experience report), 2018. URL <https://dl.acm.org/doi/pdf/10.1145/3236798>. pages 19
- [22] J. Ingeno. *Software Architect's Handbook*. O'Reilly Media, Inc. pages 39
- [23] D. Jungnickel. *Graphs, Networks and Algorithms, Algorithms and Computation in Mathematics*. 2012. URL <https://books.google.com/books?id=PrXxFHmchwcC>. pages 34
- [24] R. Kowalski. Logical english, 2020. URL <https://www.doc.ic.ac.uk/~rak/papers/LPOP.pdf>. pages 5, 7
- [25] macromates.com. Textmate grammars specification, 2021. URL https://macromates.com/manual/en/language_grammars. pages 28

- [26] mikefarah. yq. URL <https://github.com/mikefarah/yq>. pages 29
- [27] S. Overflow. 2021 developer survey, 2021. URL <https://insights.stackoverflow.com/survey/2021/most-popular-technologies-new-collab-tools>. pages 26, 27
- [28] G. D. Plotkin. *A note on inductive generalization*. 1970. URL https://homepages.inf.ed.ac.uk/gdp/publications/MI5_note_ind_gen.pdf. pages 47
- [29] Python. Indentation in python. URL https://docs.python.org/3/reference/lexical_analysis.html#indentation. pages 45
- [30] e. a. Rask. The specification language server protocol: A proposal for standardised lsp extensions, 2021. URL <https://arxiv.org/abs/2108.02961>. pages 19
- [31] M. Ridwan. Using language servers with codemirror 6. URL <https://hjr265.me/blog/codemirror-lsp/>. pages 26
- [32] TypeFox. Monaco language client and vscode websocket json rpc. URL <https://github.com/TypeFox/monaco-languageclient>. pages 26
- [33] 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 20