Imperial College London

Department of Computing

An editor for the Logical English programming language

Supervisor:
Dr. Fariba Sadri

Second Marker:
Prof. Robert Kowalski

Author:
Nikolai Merritt

Advisers:
Prof. Robert Kowalski
Dr. Jacinto Dávila
Mr. Galileo Sartor

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

September 2022

Acknowledgments

I would like to thank my advisers, Professor Robert Kowalski, Dr. Jacinto Dávila and Galileo Sartor, for their consistent advice and help throughout this project. I would also like to thank my friends and family for their support through this three months of difficult work.

Contents

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 discussed. 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 [28]. Although Logical English has an online editor hosted on the SWISH platform [16], 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 [40], a new editor was needed that was custom-built for writing Logical English.

2. Logical English

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. [28].

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.
    disha khan 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 that determine when an atomic formula can be logically derived. Examples include:

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

disha khan eats at a cafe if
    disha khan feels hungry
    and the cafe sells sandwiches.

disha khan feels hungry.
```

Listing 2.2: Examples of clauses in Logical English.

Clauses begin with a 'conclusion', which is the atomic formula that is logically implied by the rest of the clause. If there are no other atomic formulas, then the conclusion is logically implied in all cases. Otherwise, the conclusion is followed by the keyword `if`, then a number of atomic formulas that logically imply the conclusion. These 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 conclusion `disha khan eats at cafe jaune` would be logically implied through the second clause.

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 Higher-order Atomic Formulas

A higher-order atomic formula contains another atomic formula as a term. A higher-order atomic formulas can be written with the keyword `that` preceding 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 higher-order atomic formula `there is a requirement that bob pays money to LE bank`.

2.2 Program Structure

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.

disha khan 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 the section title `the templates are:`, in which the templates are defined.

Each Logical English document is also implicitly given several general-purpose templates: templates such as `*a thing* = *a thing*`, or `*a date* is immediately before *a date*`. A full list of these templates can be found in the Logical English handbook [8].

2.2.2 Clauses

The Knowledge Base

The program's clauses are given in the knowledge base. The knowledge base starts with the section title `the knowledge base <name> includes:`, where `<name>` is a name that can optionally be given.

Connectives in Clauses

The atomic formulas in a clause are separated by logical connectives. The precedence of these connectives is determined by indentation; connectives that have higher precedence are indented further. For example, what one might write in English as `(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.
```

Categories of terms

Logical English supports three categories of terms:

1. **Constants**. These terms represent a single value and cannot be broken down into any constituent terms. Examples include names such as `mary jane` or `LE Bank`, as well as numbers.

2. **Complex terms**. These are terms that are built from other terms. Examples include lists, such as `[one, two, three]`, that are built from any other terms, as well as dates, such as `01-01-1970`, which are built from numbers. An atomic formula can also be a complex term, since atomic formulas can include other terms, and can feature as arguments of higher-order atomic formulas.

3. **Atomic Formulas**. Atomic formulas are terms, since they can feature as arguments of higher-order atomic formulas. Similar to complex terms, atomic formulas can contain other terms. However, unlike complex terms, atomic formulas have a logical meaning: clauses are made up of atomic terms.

4. **Variables**. Variables are terms that range over other terms. The syntax of variables is described in Section 2.1.2.

2.2.3 Scenarios

Various scenarios can optionally be given. Scenarios contain clauses that are just conclusions, and can be used when running a query. Scenarios must have a name, and begin with the section title `scenario <name> is:`.

2.2.4 Queries

The final section of a Logical English program contains 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:
which person travels to which resort.
```

Listing 2.6: Another short Logical English program.

gives the answer

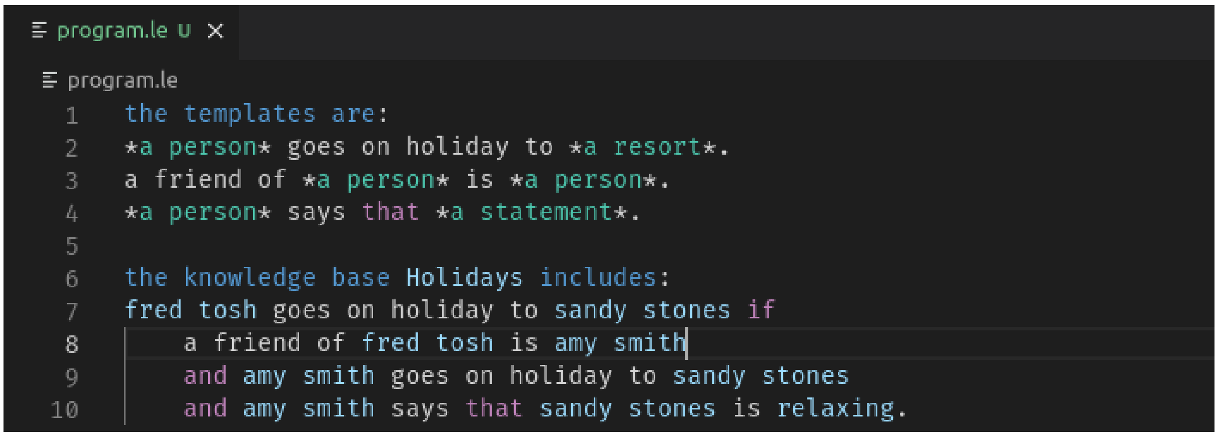
```
fred bloggs travels to the blue lagoon.
```

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

3. Editor Features

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

3.1 Highlighting



```
1   the templates are:
2   *a person* goes on holiday to *a resort*.
3   a friend of *a person* is *a person*.
4   *a person* says that *a statement*.
5
6   the knowledge base Holidays includes:
7   fred tosh goes on holiday to sandy stones if
8       a friend of fred tosh is amy smith
9       and amy smith goes on holiday to sandy stones
10      and amy smith says that sandy stones is relaxing.
```

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

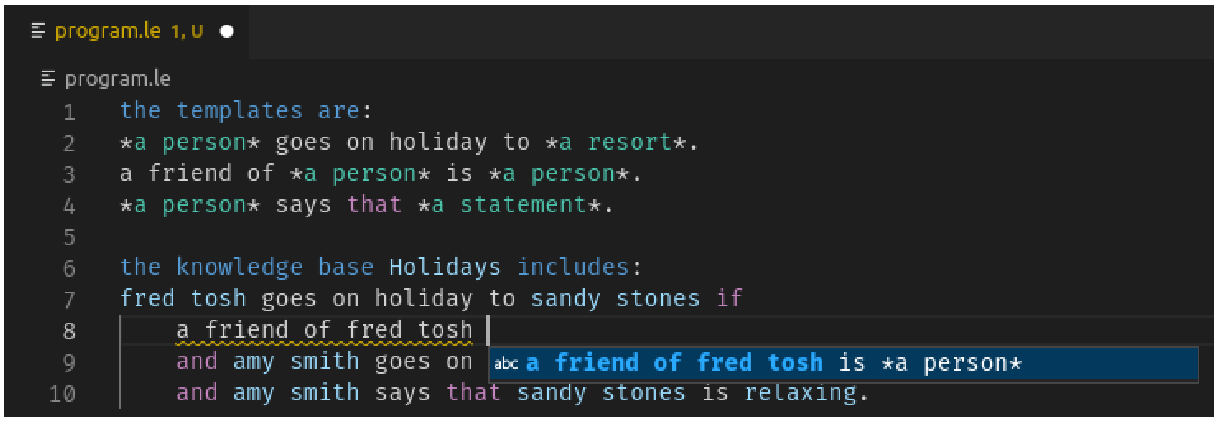
Figure 3.1 shows the editor highlighting grammatical features of Logical English. The editor highlights

- titles of sections
- template variable names
- logical connectives between atomic formulas, and
- terms in atomic formulas.

As shown on line 12 of Figure 3.1, the editor highlights terms in atomic formulas recursively. If a higher-order atomic formula contains another atomic formula, then its terms are also highlighted.

Figure 3.1, as well as subsequent screenshots, are taken using the built-in 'Default Dark+' colour theme of Visual Studio Code . 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



```
the templates are:
*a person* goes on holiday to *a resort*.
a friend of *a person* is *a person*.
*a person* says that *a statement*.

the knowledge base Holidays includes:
fred tosh goes on holiday to sandy stones if
    a friend of fred tosh
    and amy smith goes on   a friend of fred tosh is *a person*
    and amy smith says that sandy stones is relaxing.
```

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

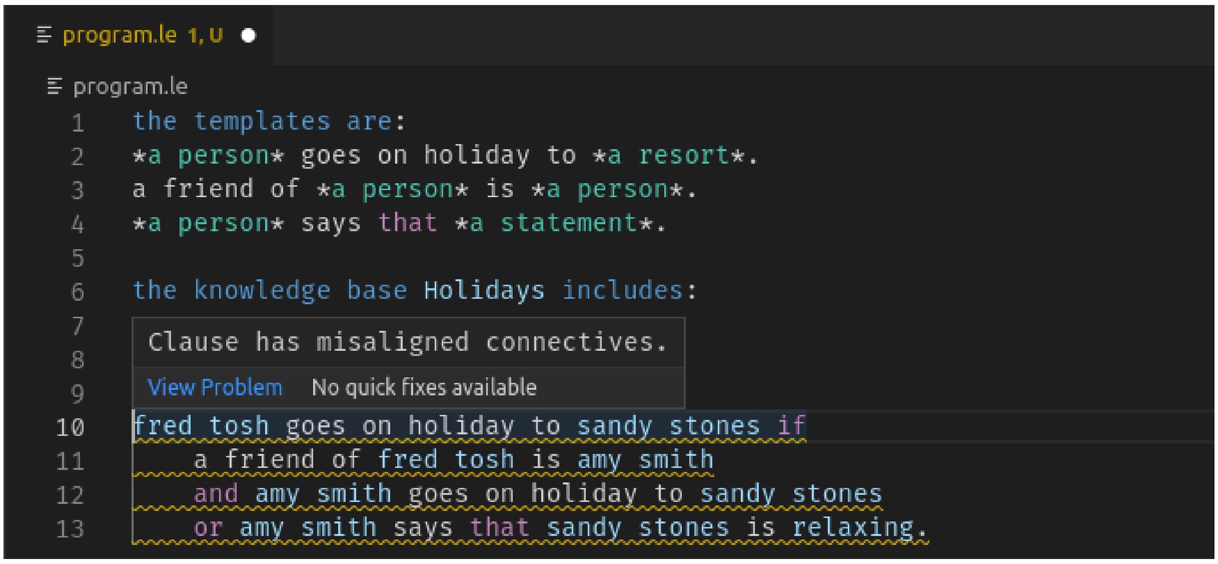
Figure 3.2 shows the editor suggesting the remainder of an incomplete atomic formula. This happens as the user is typing the atomic formula once the line the user is typing begins to conform to a template. When selecting the suggested formula, by clicking on the suggestion or pressing Tab, the suggested atomic formula is inserted with the remaining template arguments (such as 'a person' in Figure 3.2) 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

3.4 Clauses with misaligned connectives

As shown in Figure 3.3, the editor marks a clause with a warning when the clause has connectives whose order of precedence is not stated through indentation [1]. On hovering over the clause, the editor produces the error message 'Clause has misaligned connectives.'

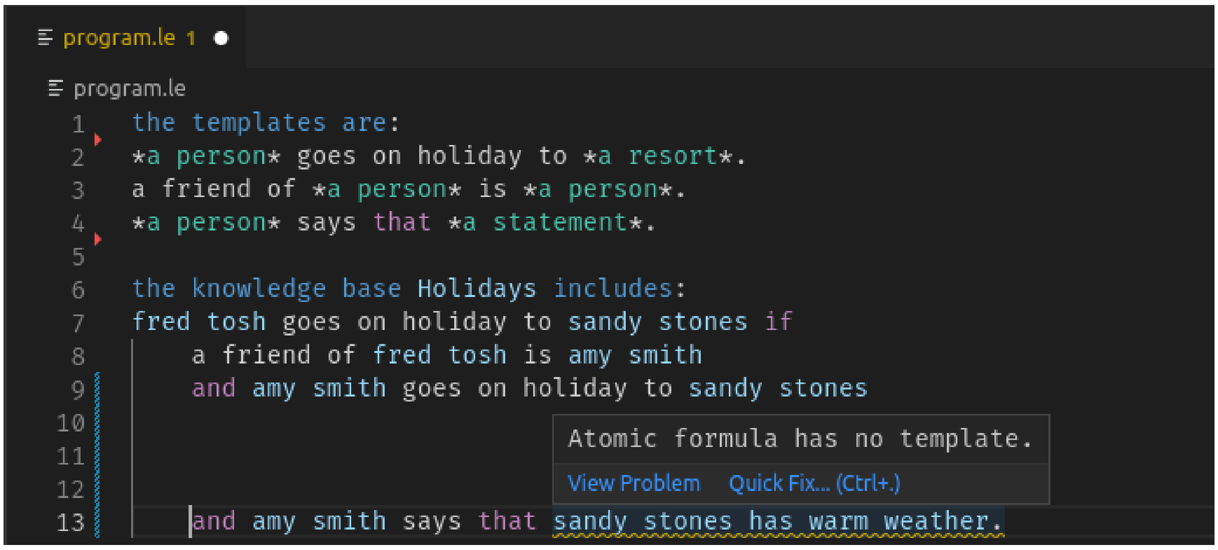
[1]This is discussed in section 2.2.2



```
program.le 1, U
program.le
the templates are:
*a person* goes on holiday to *a resort*.
a friend of *a person* is *a person*.
*a person* says that *a statement*.

the knowledge base Holidays includes:
    Clause has misaligned connectives.
    View Problem    No quick fixes available
fred tosh goes on holiday to sandy stones if
    a friend of fred tosh is amy smith
    and amy smith goes on holiday to sandy stones
    or amy smith says that sandy stones is relaxing.
```

Figure 3.3: The editor identifying a clause with misaligned connectives.

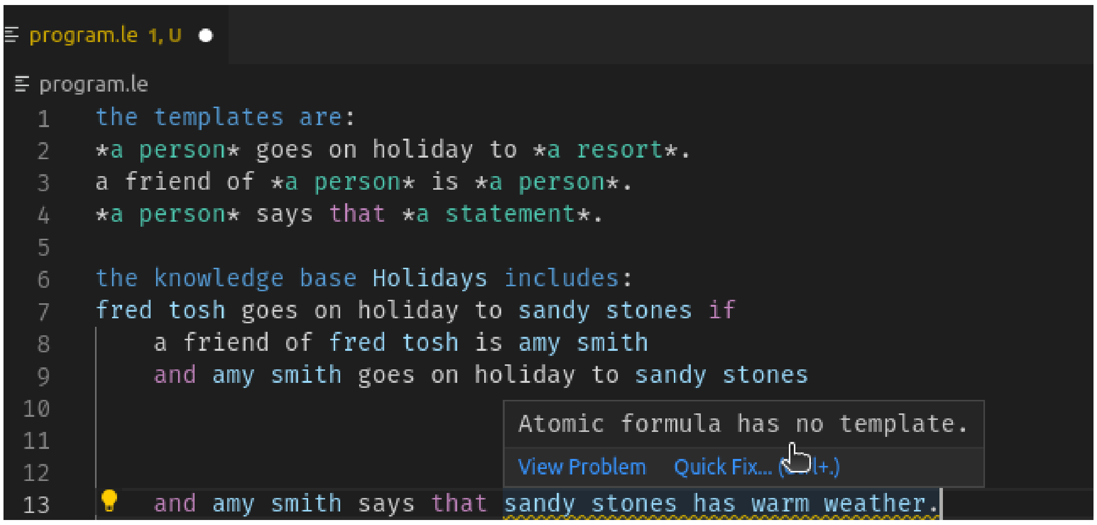
3.4.1 Atomic formulas without templates



```
program.le 1
program.le
the templates are:
*a person* goes on holiday to *a resort*.
a friend of *a person* is *a person*.
*a person* says that *a statement*.

the knowledge base Holidays includes:
fred tosh goes on holiday to sandy stones if
    a friend of fred tosh is amy smith
    and amy smith goes on holiday to sandy stones
                           Atomic formula has no template.
                           View Problem   Quick Fix... (Ctrl+.)



    and amy smith says that sandy stones has warm weather.
```

Figure 3.4: The editor identifying an atomic formula that does not match any of the templates.

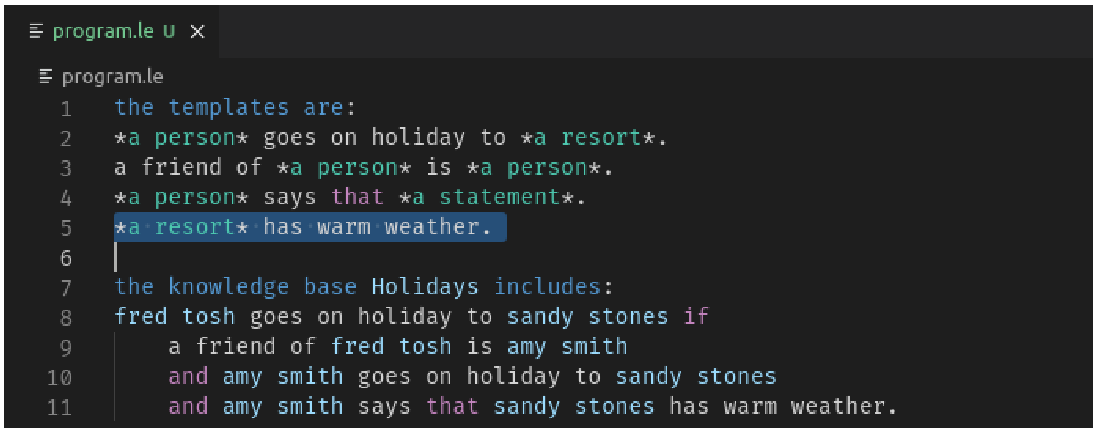
As shown in Figure 3.4, if the document contains an atomic formula that does not conform to any template, the editor marks the atomic formula with an underline representing a warning. On hovering over the atomic formula, the editor produces an explanatory error message.



```
the templates are:
*a person* goes on holiday to *a resort*.
a friend of *a person* is *a person*.
*a person* says that *a statement*.

the knowledge base Holidays includes:
fred tosh goes on holiday to sandy stones if
    a friend of fred tosh is amy smith
    and amy smith goes on holiday to sandy stones



    and amy smith says that sandy stones has warm weather.
```

```
Atomic formula has no template.
```

((a)) The document before the template is auto-generated.



```
the templates are:
*a person* goes on holiday to *a resort*.
a friend of *a person* is *a person*.
*a person* says that *a statement*.
*a resort* has warm weather.

the knowledge base Holidays includes:
fred tosh goes on holiday to sandy stones if
    a friend of fred tosh is amy smith
    and amy smith goes on holiday to sandy stones
    and amy smith says that sandy stones has warm weather.
```

((b)) The document after the template is auto-generated.

Figure 3.5: 2 Figures side by side

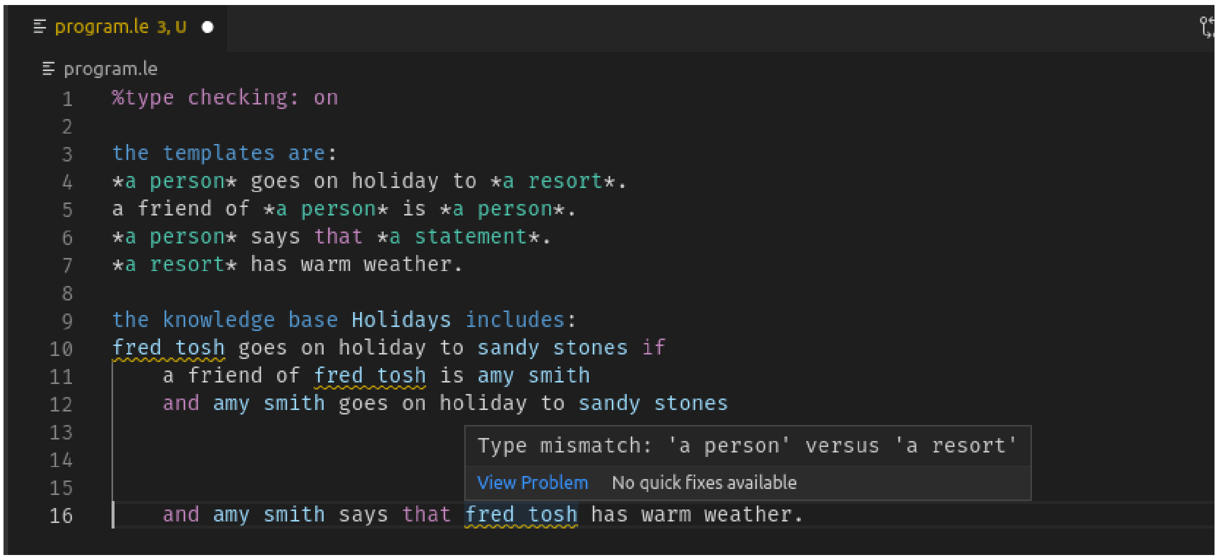
3.5 Code Actions

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, hovering over any one of the atomic formulas produces an error message with the option 'Quick Fix'. When selected, the editor auto-generates a single template that conforms to each atomic formula that was marked with this error.

If the marked atomic formulas contain terms which also feature in atomic formulas that conform to templates, then the types of those terms will feature in the auto-generated template. This is shown in Figure 3.5: the term `sandy stones` features in prior atomic formulas formulas and has type `a resort`. This allows the auto-generated template to feature the type `a resort`.

3.6 Type Checking

3.6.1 Type Mismatch Errors



program.le 3, U

program.le

```
1   %type checking: on
2
3   the templates are:
4   *a person* goes on holiday to *a resort*.
5   a friend of *a person* is *a person*.
6   *a person* says that *a statement*.
7   *a resort* has warm weather.
8
9   the knowledge base Holidays includes:
10  fred tosh goes on holiday to sandy stones if
11      a friend of fred tosh is amy smith
12      and amy smith goes on holiday to sandy stones
13
14
15
16      and amy smith says that fred tosh has warm weather.
```

```
Type mismatch: 'a person' versus 'a resort'
```

Figure 3.6: The editor identifying a type mismatch error.

The comment `%type checking: on` (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, where 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 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.

This is demonstrated in Figure 3.6. The term `fred tosh` is assigned the type `a person` in the atomic formulas on line 10 and line 11, but is assigned the type `a resort` in the atomic formula on line 16. Since these two types are incompatible, an error message is produced.

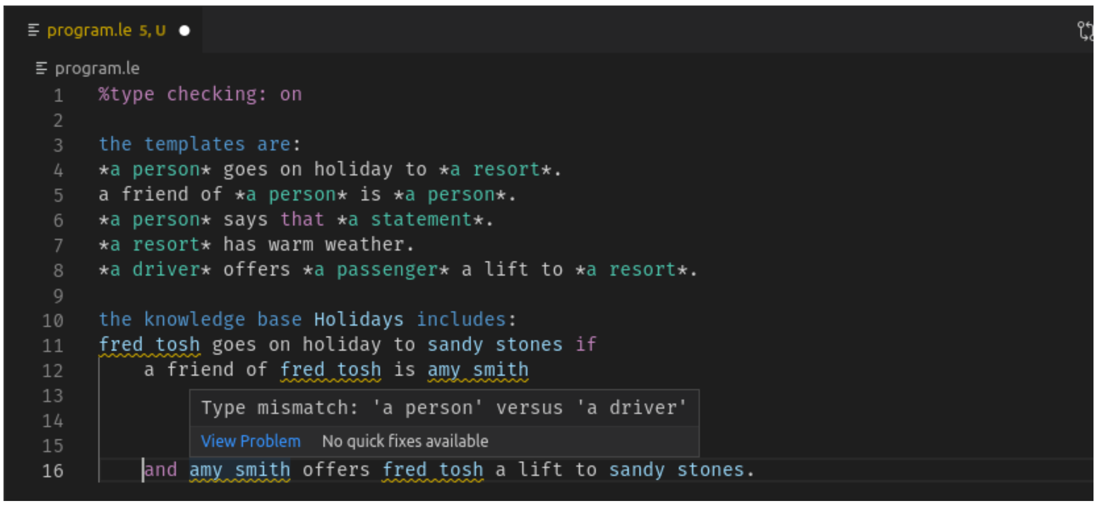
3.6.2 The Type Hierarchy

The title `the type hierarchy is:` marks the optional section where a type hierarchy can be written. To write that the type B is a subtype of the type A, the type name of B is written below the type name of A, indented further than A by a single tab.

The editor uses this type hierarchy in checking for type errors. Figures 3.7 and 3.8 show this in action with the introduction of the template

```
    *a driver* offers *a passenger* a lift to *a resort*.
```

As shown in Figure 3.7, when writing an atomic formula that matches the above template, supplying terms that are of type `a person` as the first two arguments causes a type mismatch error. The error is resolved in Figure 3.8 by introducing a type hierarchy that states that `a driver` and `a passenger` are two subtypes of the type `a person`.

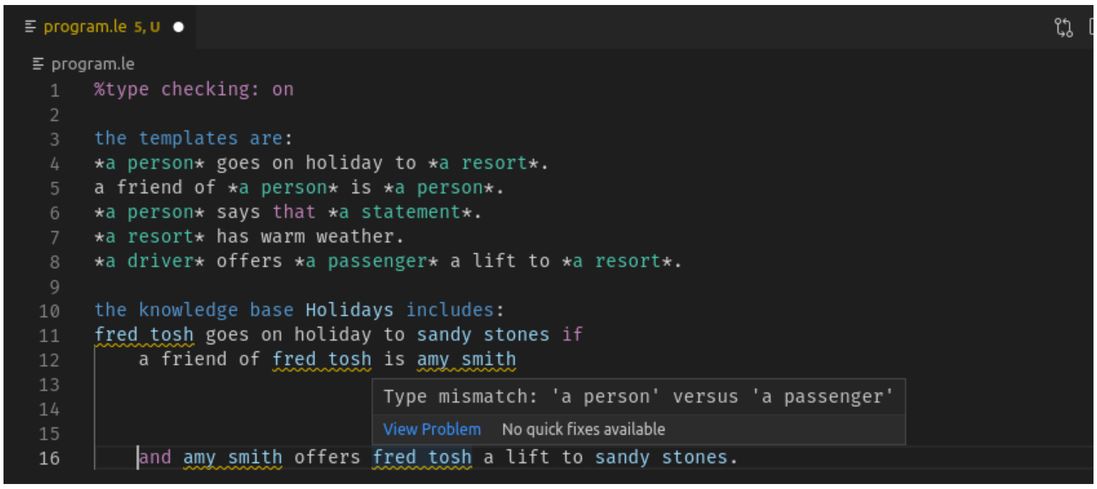


program.le 5, U ●

program.le

```
%type checking: on

the templates are:
*a person* goes on holiday to *a resort*.
a friend of *a person* is *a person*.
*a person* says that *a statement*.
*a resort* has warm weather.
*a driver* offers *a passenger* a lift to *a resort*.

the knowledge base Holidays includes:
fred tosh goes on holiday to sandy stones if
    a friend of fred tosh is amy smith
        Type mismatch: 'a person' versus 'a driver'
        View Problem    No quick fixes available
    and amy smith offers fred tosh a lift to sandy stones.
```

((a)) The editor marking the type mismatch error with the term `amy smith`.



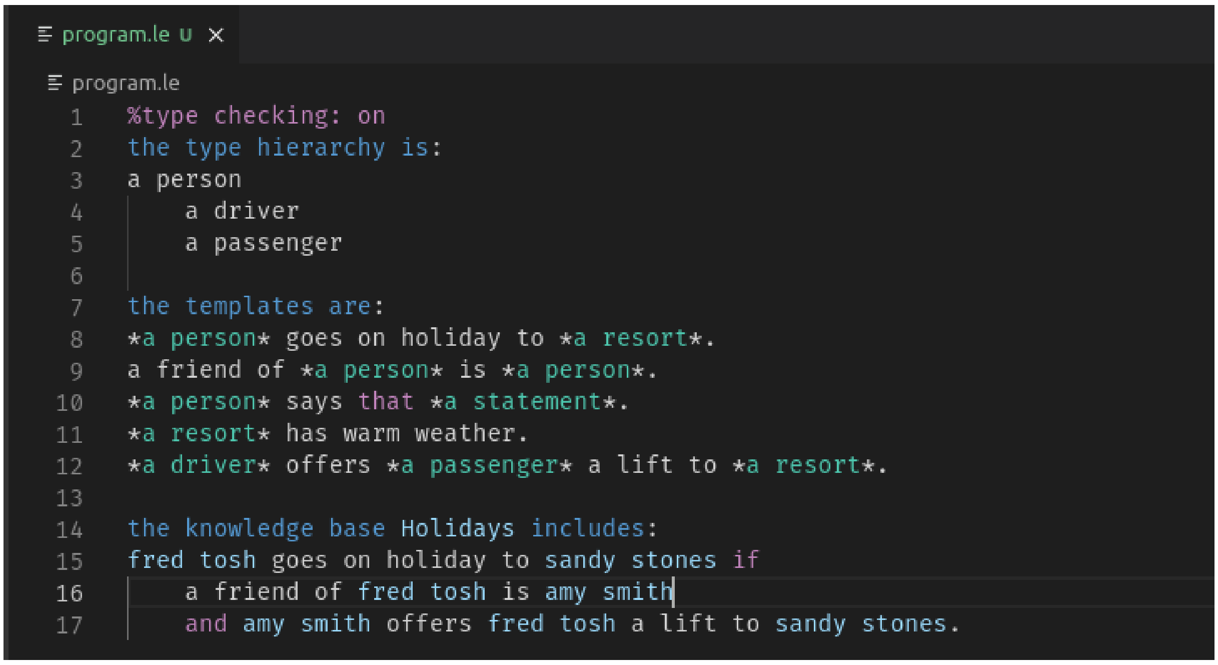
program.le 5, U ●

program.le

```
%type checking: on

the templates are:
*a person* goes on holiday to *a resort*.
a friend of *a person* is *a person*.
*a person* says that *a statement*.
*a resort* has warm weather.
*a driver* offers *a passenger* a lift to *a resort*.

the knowledge base Holidays includes:
fred tosh goes on holiday to sandy stones if
    a friend of fred tosh is amy smith
                         Type mismatch: 'a person' versus 'a passenger'
                         View Problem    No quick fixes available
    and amy smith offers fred tosh a lift to sandy stones.
```

((b)) The editor marking the type mismatch error with the term `fred tosh`.

Figure 3.7: The editor marking two type mismatch errors. The errors are caused by the types `a driver` and `a passenger` being incompatible with the type `a person`.



```
%type checking: on
the type hierarchy is:
a person
    a driver
    a passenger

the templates are:
*a person* goes on holiday to *a resort*.
a friend of *a person* is *a person*.
*a person* says that *a statement*.
*a resort* has warm weather.
*a driver* offers *a passenger* a lift to *a resort*.

the knowledge base Holidays includes:
fred tosh goes on holiday to sandy stones if
    a friend of fred tosh is amy smith
    and amy smith offers fred tosh a lift to sandy stones.
```

Figure 3.8: A type hierarchy is used to resolve the type mismatch errors shown in Figure 3.7.

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 [35], 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 [24] 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 [24], and the Visual Studio Code Language Server Extension documentation [17], describe two main challenges that Language Servers face, that of "incrementality" and "partiality":

- Due to efficency constraints, the IDE may only being able to send 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 [11]. 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 higher-order 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 [39] 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. Inspired by this, I will also use a tree-like structure to represent the recursive nature of atomic formulas.

4.2.2 Code Completion

Bruch et al [19] create the 'Best Matching Neighbours' algorithm for suggesting method calls in object-oriented code. The Best Matching Neighbours algorithm suggests code completions of library methods by learning from patterns found in large, pre-existing codebases. This does not have an analogue in Logical English: there are no code libraries that many documents may draw from; each Logical English document defines all the templates and atomic formulas that it uses. This means that, in suggesting the completion of an atomic formula, the Logical English editor must use a simpler, rule-based approach rather than a learning algorithm.

In their study, Bruch et al hypothesise that useful code completion [19, p. 214]:

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

Bruh 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, expressing the program's fundamental structure [20, 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 a proposition based on the types of the proposition's arguments. 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 `T1`, `T2`, ..., `Tn` the type of the proposition is `std::predicate<T1, T2, . . ., Tn>` [12]. This is mirrored in Java in the `Predicate` class, where $n = 1$, and in the `BiPredicate` class, where $n = 2$ [7].

This type system assigns types to atomic formulas syntactically, according to the type of terms that the atomic formulas describe. However, Logical English as a language seeks to minimise the syntactic rules it imposes on the user. I believe that assigning types to atomic formulas based on 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 4.1 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 type [1]. A syntax-based type system will therefore not notice 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.

Type systems are also heavily studied in the field of Type Theory. Type Theory is a rich and diverse field of study, with there being a wide range of both logical backgrounds from which a type theory can be constructed and types of expressions which a type theory permits [25, p. 12]. However, many type theories have certain characteristic features in common. A central process in type theory is creating terms from other terms in a process known as β-reduction, where variables are substituted for terms in λ-abstractions[25, p. 3]. These λ-abstractions are themselves terms, meaning they can be substituted into further λ-abstractions, and so on. Due to the importance of λ-abstractions, the types of λ-abstractions, given restrictions on the types of their terms, is a key area of focus in many type theories [33, p. 22]. In general, the relations between terms and their types are expressed as logical judgements and are given by rules of inference that may differ between theories.

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

There are slight parallels between Logical English and such type theories. Logical English allows for the creation of new terms by substituting existing terms into constructs. This could be viewed as an analogue to β-reduction, with each construct corresponding to a λ-abstraction. However, in the current version of Logical English, there are only two such constructs – that of lists, and that of dates – contrasted with the infinite amount of λ-abstractions allowed for in Simply-Typed Lambda Calculus and other type theories. Further, it is impossible to refer to the λ-abstraction itself in Logical English, as it is impossible to refer to a list or a date in Logical English without referring to concrete terms. Until Logical English implements arbitrary function symbols, with each function symbol itself being a valid term, Type Theory will only have a weak relevance to Logical English.

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 writers: 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 connected to many of the most popular programming editors on various operating systems. 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 communicate the responses from the language server and syntax highlighter to the code editor.

5.2 The Syntax Highlighter

The syntax highlighter must identify and label grammatical features in Logical English documents. This must be done in a way that the editor can highlight the document according to the labels. 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.

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 was not required since it was not clear at the outset in what cases 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. The type system would be used to check for inconsistent uses of values: errors where

- an atomic formula contains a value c, where c is assigned the type A
- another atomic formula contains the same value c, where c 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.

6. Design

6.1 System 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.

6.1.1 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.

6.1.2 Syntax Highlighter

Contrary to what the name "syntax highlighter" may suggest, the syntax highlighter is merely a static document that contains the grammar of Logical English. The language client parses the syntax highlighter when the client launches, using the grammatical rules in the syntax highlighter to highlight aspects of Logical English. The syntax highlighter is written according to the TextMate grammar specification, which allows it to be used by language clients for IDEs other than Visual Studio Code.

6.1.3 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.

6.1.4 How the components interact

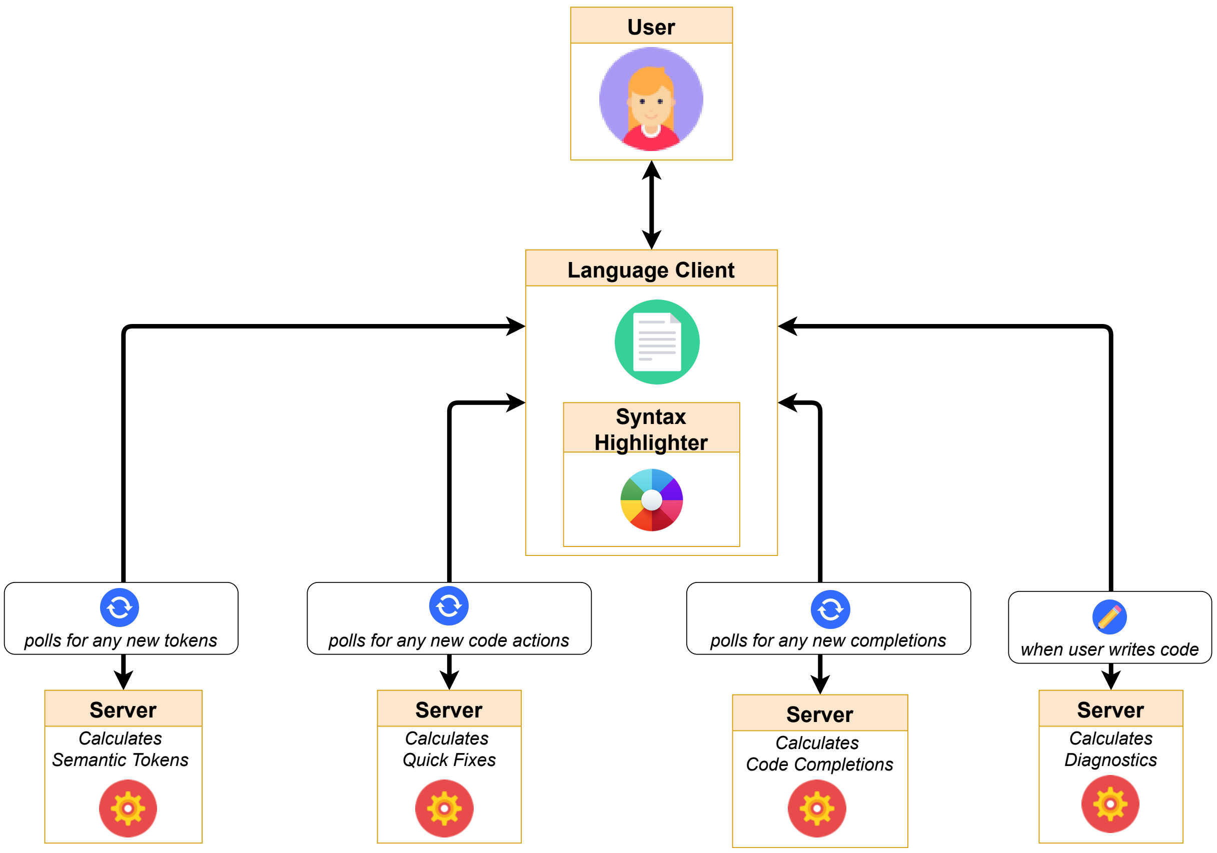


Figure 6.1: A diagram showing the overall structure of the Logical English editor.

The overall structure of the editor is described in Figure 6.1. 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.

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 [36] and Monaco [38], along with some of the most popular desktop code editors [31], such as Visual Studio Code [17], Visual Studio [14] and IntelliJ [5].

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 [9]. To test this, I created a simple language server, starting with the sample language server created by Microsoft [10] 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 [31]. 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. As noted by Zayour and Hajjdiab in their survey on IDE productivity [40], "the complexity of the [coding] environment creates additional burdens", mostly in the ability to fix errors. 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 [29]. This is standard amongst language clients, with Codemirror 6, Visual Studio and IntelliJ also supporting syntax highlighting using TextMate grammar [3], [15], [6].

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# [14], and the `vscode-languageserver` library, written for TypeScript [17]. 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 destructuring 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` [30]. 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.

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

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

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

The overall design methodology was guided by the principles outlined in "Java to Kotlin: A Refactoring Guidebook" by Duncan McGregor and Nat Pryce [23]. 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. [23] 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[22]. 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 [21],

> 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.

7. The Type System

7.1 Overview

The type system for Logical English must model Logical English's domain of discourse. The type system must assign types to terms of atomic formulas, in a way that can identify inconsistent uses of terms.

7.2 The Type Hierarchy

In Section [**?**] I determined that a new type system is needed which is fundamentally user-driven. The user would group terms assigning types that are entirely of their own choosing, having no connection to 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. 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, it was found that this relation obeys the three axioms of a partial order [37]:

- **Reflexivity:** Every type A must be a subtype of itself, since terms of type A can be used whenever terms of type A are expected.
- **Anti-symmetricity:** 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[27, 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 top type of the type hierarchy

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

Other types in the type hierarchy

The default templates introduce their own types: as well as `a thing`, the default templates feature the types `a number`, `a list`, `a date`. This means that the latter three types must be subtypes of `a thing`.

Similarly, it may be useful to allow higher-order atomic formulas to apply to any other atomic formula: for instance, modal atomic formulas of the form `the judge concurs that x`. This would require a default type for atomic formulas; a possible name for this top type could be `a proposition`.

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 two 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 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 of characters. 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

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

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

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 argument names: 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 the design 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, filtering the list of elements to obtain the list of argument names, or to obtain 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, the design was improved.

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 argument [2]. This `Type` class contained the template argument's name, along with additional structure used to implement the type hierarchy, discussed in Section 8.5.3. The class `Surrounding` was created to represent the surrounding text that lies between types. As per the design philosophy, these are both lightweight classes 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 construct a `Template` class to be a wrapper class around this list. The `Template` class provided a read-only view to the list of elements, exposing methods that allowed the elements to be queried. These methods included obtaining the template's types or surrounding text – what was previously done manually – along

[2]Before 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.

with more advanced queries that will be discussed later.

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

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

    class Surrounding:
        text: string
        ...

    class Template:
        elements: (Type | Surrounding)[]
        getTypes(): Type[]
        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, the same design principles were applied to create a representation for 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: using a type `Term` to represent terms of an atomic formula, the `AtomicFormula` class would hold an `elements` list that consists of either `Surrounding` or `Term` objects. An overview for the `AtomicFormula` class is given in Listing 8.2

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

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

Unfortunately, designing the type `Term` was not straightforward. As discussed in Section 2.2.2, there are four categories of terms:

1. constants,

2. complex terms,

3. variables, and

4. atomic formulas.

Due to time constraints, parsing complex terms as constituting smaller terms was left for future work: instead, complex terms are parsed as constants. This then motivated two new classes: `Constant` to represent constant terms, and `Variable` to represent variable terms. From this, `Term` objects were allowed to either be `Constant`, `Variable` or `AtomicFormula` objects.

Initially it seemed that the `AtomicFormula` class was enough to capture the content of user-written clauses. However, when I began working on parsing atomic formulas it became clear that many Logical English documents would require another class. Consider the Logical English extract in Listing 8.3.

```
the templates are:
*a person* plays tennis on *a day*.

the knowledge base Tennis includes:
abdul plays tennis on a day if
    the day is a bank holiday.
```

Listing 8.3: An extract of a Logical English document in which the condition of a clause does not have any matching template. Because of this, the `AtomicFormula` class can properly represent the argument atomic formula.

In Listing 8.3, the entry `the day is a bank holiday` appears to be an atomic formula, since it is a condition in a clause. However, since the entry does not conform to any template, terms cannot be parsed from it. This means that the `AtomicFormula` class cannot represent the entry.

This prompted a new class, `TemplatelessFormula`, to represent entries that appeared in clauses but did not conform to a template. Writing such entries is an error in Logical English, so this class was crucial in diagnosing errors.

Since it was always a possibility that a condition in a clause could be a `TemplatelessFormula` when an atomic formula was expected, 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 classes themselves was not an option for two reasons: by the principle of Separation of Concerns [26, p. 183] the classes are "abstract" representations of Logical English construct: 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-langaugeserver` uses [3].

8.2 Parsing the Document

8.2.1 Parsing templates

Each template is found below the header `the templates are:`. The types of the template are found by finding substrings of the form `*a __*` or `*an __*`: all remaining substrings are surrounding text.

8.2.2 Parsing clauses

The clauses are found below headers of the form `the knowledge base __ includes:`. This text is split into clauses by using the fact that each clause must end with a full stop. Each clause, along with its range in the document, is passed to the error diagnosis functionality as a `ContentRange<string>`.

The conclusion and conditions of each clause were found through splitting the clause by Logical English's connectives. Of these strings, those that had matching templates were parsed into atomic formulas using the techniques in Section 8.2.3. Each atomic formula was packaged with its range in the document and passed to the semantic highlighting section as `ContentRange<AtomicFormula>` objects. If a conclusion or condition did not match any template, it was passed to the error diagnosis and quick-fix sections as a `ContentRange<TemplatelessFormula>` object.

[3] A 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 is not enough commonality between valid values of `T`, such as `string`, `Template` or `AtomicFormula`, to constrain `T` effectively.

8.2.3 Extracting the document's atomic formulas

Atomic formulas are parsed into their `AtomicFormula` representation according to their matching template. This is a complex task and is broken down into multiple sub-problems.

The core sub-problem is to extract an atomic formula with respect to a given template. Assuming that the atomic formula matches the template, the template's form is used to extract the atomic formula's elements.

This algorithm is used in multiple places. Extracting the full representation of a higher-order atomic formula is done by recursively applying the algorithm to the higher-order atomic formula's terms. Surprisingly, the algorithm is also used in what is logically the prior task: determining how well an atomic formula, or a partially-written atomic formula, matches a template.

Extracting an atomic formula's elements with respect to a template

The 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 is condensed into Algorithm 1.

Algorithm 1 iterates over each of the template's surrounding elements. For each surrounding element, the corresponding surrounding text is searched for in the atomic formula. If there is a type element immediately before the surrounding element, then a term is extracted from the text before the atomic formula's surrounding text. If the term begins with `a`, `an` or `the`, then the term is a variable, otherwise, it is a constant. The term and the surrounding text are appended to a list of formula elements. The atomic formula that has been visited is discarded.

If the template element ends with a type, then this type will not be reached by looking for types that precede surrounding elements. This case is handled after the loop, with any remaining atomic formula text interpreted as a final term.

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). Through the procedure `FindSurrounding`, 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

Algorithm 1 An algorithm to extract the elements of an atomic formula according to a template.

```
procedure ExtractTerms($template\,elements$ , $formula$)
    $formula\,elements \leftarrow$ empty list
    for $i \leftarrow 1$ to $template\,elements.length$ do
        if $template\,elements[i]$ is a $Surrounding$ then
            $surrounding\,text \leftarrow$ FindSurrounding($elements[i].text$, $formula$)
            if $surrounding\,text$ was not found then
                end loop

            if $i > 1$ then
                $term\,name \leftarrow formula$ before $surrounding\,text$
                $t \leftarrow$ TermFromName($term\,name$ , $template\,elements[i-1]$)
                append $t$ into $formula\,elements$

            $s \leftarrow$ new $Surrounding(s.text \leftarrow surrounding\,text)$
            append $s$ into $formula\,elements$
            $formula \leftarrow formula$ after $surrounding\,text$

    if $formula$ is not empty and $template\,elements.last$ is a $Type$ then
        $t \leftarrow$ TermFromName($formula$, $template\,elements.last$)
        append $t$ into $formula\,elements$
    return $formula\,elements$

procedure FindSurrounding($surrounding\,name$ , $formula$)
    if $surrounding\,name$ is in $formula$ then
        return $surrounding\,name$
    else
        for $i \leftarrow 1$ to $formula.length$ do
            $end\,of\,formula \leftarrow formula$ after index $i$
            if $surroundingname$ ends with $end\,of\,formula$ then
                return $end\,of\,formula$
    return fail

procedure TermFromName($term\,name$ , $type$)
    if $term\,name$ starts with ‘a’ or ‘an’ or ‘the’ then
        $v \leftarrow$ new $Variable(v.name \leftarrow term\,name , v.type \leftarrow type)$
        return $v$
    else
        $c \leftarrow$ new $Constant(c.name \leftarrow term\,name , c.type \leftarrow type)$
        return $c$
```

that allows the diagnostic, code completion and semantic highlighting features to occur as the user is writing the document.

To extract all terms that feature in a higher-order atomic formula, Algorithm 1 is used recursively. First, Algorithm 1 is applied to the higher-order atomic formula, yielding each term as a `Variable` or `Constant` object. If there are any constants whose text matches a template (in the sense discussed in 8.2.4), then the constant is treated as an atomic formula. Algorithm 1 is used to extract the atomic formula's elements using the matching template.

Algorithm 1 handles the edge case where a surrounding also appears as a term. A simple example of this problem is given by the the template `*a merchant* ships *an item*`. The atomic formula `the merchant ships ships`, describing a merchant that packages and transports ships, is an atomic formula that matches the template. By treating the atomic formula as a queue, removing text that has been visited, this edge case is handled by the algorithm.

8.2.4 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.

8.2.5 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 Beneficiary includes:
chuan is a beneficiary of disha'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 `chuan is a beneficiary of disha'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 `chuan is a beneficiary of disha's will` will lead to `a beneficiary of disha'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 Semantic Highlighting

The semantic highlighting feature highlights the terms of each atomic formula in the document. This is a straightforward application of parsing the document. Each atomic formula is parsed from the document, along with its location, as a `ContentRange<AtomicFormula>` object. This is done by finding the atomic formula's matching template with the highest match score, which is used to extract the atomic formula's elements. The associated range data of the `ContentRange<AtomicFormula>` is used to find the location of each term. Each term's location is then highlighted in the document. The semantic highlighting process is recursive: if a higher-order atomic formula contains an atomic formula as an argument, then the atomic formula's terms are also highlighted. Using the location of the higher-order atomic formula to find the location of the terms of its atomic formula argument was a technically difficult process, but involved no higher-level design and does not warrant discussion.

8.4 Completion

8.4.1 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 templates match the atomic formula, in the sense of Section 8.2.4, because the atomic formula will be incomplete, and so may not contain all 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 erroneous templates cannot be ruled out algorithmically. They will, however, be out-ranked by templates with longer matching surroundings according to the match score algorithm. 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.

8.5 Error diagnosis

8.5.1 Diagnosing template-less atomic formulas

The editor diagnoses errors where an atomic formula does not have a matching template. The approach is similar to the approach used in highlighting the terms of the document.

Each atomic formula that does not match any template is read, along with its range in the document, as a `ContentRange<TemplatelessAtomicFormula>`. The document is then marked with an error message that spans the range of the templateless atomic formula.

At first glance there appeared to be no problem with this simple solution. However, as I was testing this feature I found that the criterion used was too broad. Specifically, incomplete atomic formulas, atomic formulas that are being typed, will, in general, not match a template. This means that, in most cases, until the user finishes typing an atomic formula, the atomic formula will be marked with an error.

Unfortunately, this is impossible to fix in the current version of the Language Server Protocol. When the client requests diagnosis information, the only information that is supplied to the language server is the current state of the document. To determine which atomic formula is being typed by the user, the user's cursor position would also be needed. This means that the error diagnosis feature cannot identify which atomic formula is being typed to make it exempt from errors.

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, in the hope that this position stays accurate when calculating error diagnoses. However, I expect that this approach is highly ineffective in practice.

8.5.2 Diagnosing clauses that have misaligned connectives

The editor also diagnoses errors where a clause has misaligned connectives. If a clause contains two lines that have the same indentation, but one begins with `or` while the other begins with `and`, then the precedence of the connectives is ambiguous.

The approach is similar to the method used in atomic formulas that match no template. Each clause is read from the document, along with its location, as a `ContentRange<string>`. The clause is then split into an array of its lines. If two lines begin with the same level of indentation, but different connectives, then the clause's range is marked in the document with the error message "clause has misaligned connectives".

The nuance here is with the term 'same indentation'. There are two common ways to indent lines: tabs, and spaces, which are approximately equally common amongst programmers [13]. However, there is no standard 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 atomic formulas using tabs, and the other using spaces, then it is up to interpretation as to whether the indentation is unambiguous.

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 [34]. Since this is a tangential issue to finding misaligned connectives, I left implementing such a feature for future work.

8.5.3 Diagnosing type mismatches

Feature Overview

The type checking system was one of the most complex features to implement. 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 wish to be notified of type-checking errors.

The approach I chose in implementing type checking was to have the feature disabled by default, 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. Types were assigned to each term of an atomic formula according to the template's argument name. If two terms were found that had the same name but different types, a type mismatch error message was generated.

This design was a lot more inconvenient to use than I expected, since many more error messages were generated than 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 would differ from 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 in Listing 8.5 would not need to be stated, since it is included by default.) The type-checking system would generate a type mismatch error for Listing 8.5 since `[]` is both of type `an A` and `a B`. 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 rewritten as

```
*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 I was careful not to rename argument names such as `a date` that were used the Logical English engine.

Diagnosing with a type hierarchy

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 hierarchical tree structure in text form: the one that balanced both ease of readability and ease of writing was the indented list [18], 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 [18].

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. According to the type hierarchy specification in Chapter 7.1, 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 [4].

8.6 Quick Fixes

8.6.1 Generating a new template to match 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 further.

The first candidate template is generated according to the Least General Generalisation algorithm. This is based on algorithm written by Gordon Plotkin [32, p .155] that gives, from two or more atomic formulas [4], a single generalisation which has variables wherever the two atomic formulas have differing terms. The result is a generalisation in the sense that both atomic formulas 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 Algorithm 2. This is an adaptation of Gordon Plotkin's original algorithm. The algorithm iterates over each whitespace-separated word in an arbitrarily-chosen (here, first) formula. If the current word features in every formula, then the current word must be part of a surrounding. The word is appended to a buffer that collects the current surrounding. Since a surrounding has been reached, a

[4]Gordon Plotkin's algorithm uses the term 'word', which is a string of symbols that contains no variables. Thus atomic formulas can be viewed as 'words'. Templates, which conform to an atomic formula but contain variables, can be viewed as their generalisation.

Algorithm 2 An algorithm to obtain the least general template that matches a list of formula strings.

```
procedure LEASTGENERALGENERALISATION($formulas$)
    $first\,formula\,words \leftarrow formulas.first\,string$ split by whitespace
    $template\,elements \leftarrow$ []
    $passed\,term \leftarrow$ false
    $buffer \leftarrow$ empty string buffer
    $visited\,term \leftarrow$ false

    for $word$ in $first\,formula\,words$ do
        if $f$ includes $word$ for every $f$ in $formulas$ then
            append $word$ into $buffer$
            if $visited\,term$ then
                append new Type into $template\,elements$
                $visited\,term \leftarrow$ false

            for $f$ in $formulas$ do
                remove first occurrence of $word$ from $f$

        else
            $visited\,term \leftarrow$ true
            if $buffer$ is not empty then
                $s \leftarrow$ new Surrounding($s.text \leftarrow buffer$)
                append $s$ into $template\,elements$
                clear $buffer$

    $T \leftarrow$ new Template($T.elements \leftarrow template\,elements$)
    for $f$ in $formulas$ do
        if $f$ does not match $T$ then
            return failed
    return $T$
```

term may have been passed by. If so, a type with an arbitrary type name is added to the template elements. Finally, the current word's first occurrence is removed from every formula to prevent double-counting.

If the current word does not feature in every formula, then the current word is part of a term. This means that we have reached the end of the surrounding stored in the buffer. The buffer's contents is appended to the template elements as a `Surrounding` object, then cleared.

Once each word has been iterated over, a new template is created from the template elements. The formulas passed into the algorithm may not have conformed to a common template, in which case not every formula will match the template generated. If this is so, the algorithm fails. Otherwise, the template is returned.

Since Algorithm 2 takes care to prevent double-counting, the algorithm does not fail in the edge case where a term or surrounding occur multiple times in each formula. Algorithm 2 was therefore a useful starting point in generating a common template. This is discussed further in the next section.

Template Refinement from the re-use of terms

It was often the case that the Algorithm 2 alone was not enough to generate an accurate template. This was the case when the atomic formulas did not vary in each of their arguments. In trying to fix this problem, I noticed that Algorithm 2 was not exploiting the clause in which the atomic formulas were written. The surrounding clause of an atomic formula may contain crucial information: if a term of a template-less atomic formula appears in another atomic formula that conforms to a template, then the term and its type are known. Identified in this way, the term can be generalised into an argument.

To incorporate this feature, the `Template` class was given a method that finds occurrences of a given term in its surroundings, replacing those occurrences with a new variable. By starting with the template obtained by Algorithm 2 and successively applying the method to each known term of the surrounding clause, a much more accurate template was obtained.

This procedure had two additional improvements over simply using Algorithm 2. A single atomic formula could now be generalised into a template – a feature that is impossible with least general generalisation. Further, whenever variables were added from existing terms, the term's corresponding type could be used, giving a more accurate variable name.

9. Evaluation

As described in Chapter 8, the editor has been carefully designed to implement both the required and optional features. Two methods were used to determine how well the features are implemented: user feedback and automated testing.

9.1 User Feedback

9.1.1 Collecting user feedback

User feedback was collected through a survey form. The survey form, like the editor, was designed with people new to Logical English in mind. The form guided the responder through writing a preset Logical English program, constructed so that the responder would encounter the editor's features sequentially. Once the responder met each feature, they were asked to rate the feature by its intuitiveness and usefulness. At the end of the form, the responder was invited to suggest issues or points of improvement for the editor.

9.1.2 Feedback Scores

The survey form was filled out by seven people, none of whom had used Logical English before. The responders varied in programming ability: three had no experience in programming, one had limited experience, and the other three were confident student or professional programmers. Both groups benefited the survey. Those who were experienced in programming could judge the editor's features against editors they had used for other programming languages, while those with limited programming experience would judge the intuitiveness of the editor from a fresh perspective, with little prior experience to guide them to how to use certain features.

Figure 9.1 shows the scores obtained. Figure 9.1 shows that each feature was overall rated as highly intuitive and useful, with each feature receiving an average score of no less than 8 out of 10. However, there is a large difference in the scores given to the highlighting and code completion features, with the scores for highlighting ranging from 5 to 10, and the scores for code completion ranging from 6 to 10. This is sur-

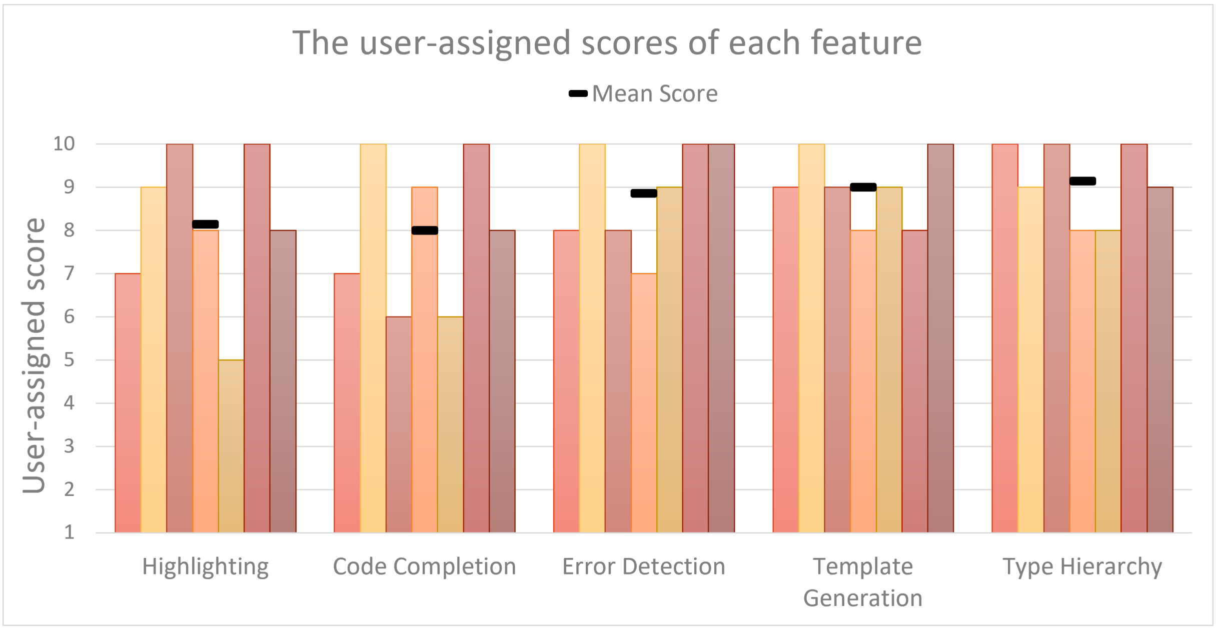


Figure 9.1: The scores assigned to each feature from a survey with 7 participants. The features are scored by their intuitiveness and usefulness from 1 to 10, with a higher score meaning that a feature is more intuitive and useful.

prising; I expected the template generation and type hierarchy features, which were more experimental, to receive the lower scores, if any. This result is due to some users encountering minor errors with the features, as explained in their comments about the editor.

9.1.3 Feedback Comments

Although the editor was tested heavily before being submitted for review, the form responders encountered some minor issues when testing the editor. Three issues stood out in particular: problems with highlighting unsaved documents, auto-completing section headers and irrelevant code completions being suggested.

Overall comments

The comments that the editor received were overall highly positive. Users were surprised at the intuitiveness of the template auto-generation, especially at the the use of template refinement to give the template accurate variable names. Some users, especially those with little programming experience, found the need for a type hierarchy difficult to understand at first. This was despite the form's example of the type hierarchy being necessary to fix a type mismatch error that should not occur. However, once the type hierarchy was understood, those users found it quite intuitive to write, especially when they chose to experiment with their own examples. The overall popularity of these two features is evidenced also by their scores, with the two features both scoring an average of approximately 9 out of 10, and no score being lower than 8 out of 10.

The scores given to highlighting and code completion contrast with the scores of the template generation and type hierarchy. This is because some users reported two issues with the features: unsaved documents receiving no highlighting, and erroneous auto-completion for section headers.

Highlighting unsaved documents

Visual Studio Code allows the user to write in a new document before it is saved. The Logical English editor is usually activated when a new file is saved, or an existing file is opened, with the `.le` file extension. However, the editor can also be manually activated by the user on unsaved documents. Users reported that, if activated manually, the editor provided no features other than simple syntax highlighting. [1].

On investigating the issue, I found that the issue is caused by the language server failing to launch when the editor is activated manually on unsaved files. This issue is reproducible on both Windows and Ubuntu Linux platforms. The language client documentation does not mention this issue and, as found through preliminary testing, the TypeScript language extension does not suffer from this issue either. Since this issue can be avoided entirely by saving the document, investigating this issue further is left for future work.

Auto-Completing Section Headers

To make writing the document easier for new users, I added the ability to auto-complete the headers for the type hierarchy, template and knowledge base section. However, selecting the suggestion `the type hierarchy is:` lead to the section header `the templates are:` being written. This was a simple mistake on my part which was quickly fixed.

9.2 Automated Testing

Once the editor had been adjusted in light of the user feedback, the functionality and correctness of the editor were assessed with automated tests. With automated testing, the thoroughness and certainty of the tests can be verified by other testers. If the editor is developed further, the tests can be carried out during development to ensure that no existing features are impaired.

[1]Juging by how the other features scored highly, I assume that the users who encountered this issue then tried saving the document, which caused the rest of the features to activate.

9.2.1 Functionality Tests

End-to-end tests are used to test the editor's functionality. The highlighting, diagnostics, code completion and quick-fix functionalities are tested through a total of 21 end-to-end test cases.

These tests tested the editor's functionality from the user's perspective. In each test case, the editor opens a Logical English program. The editor's behaviour when providing a feature is then queried and checked against the desired behaviour.

The editor is tested in every feature listed in Section 5 against its required behaviour. These features are tested in areas of standard use: each test case examines the editor's behaviour when editing a realistic Logical English document. These test cases cannot claim to be exhaustive, since Logical English's grammar is not specified to a level of detail that determines edge cases. However the end-to-end tests are broad in scope, testing both when features should and should not occur.

9.2.2 Correctness Tests

Unit tests are used to test the correctness of the language server. The behaviour of the two most complex classes, `Template` and `TypeTree`, is tested with 25 unit tests. These tests are supplemental to the end-to-end tests. Since the behaviour of Logical English has not been specified in edge cases, the unit tests are used to determine whether the `Template` and `TypeTree` classes can handle edge cases in their input in a meaningful way. If so, this would allow the editor to be easily updated to incorporate a more specific grammar.

9.2.3 Test Results

The editor passes each of the 21 end-to-end tests. The language server passes each of the 25 unit tests. Instructions on how to confirm this result can be found in Section 11.3.

10. Conclusion

10.1 Challenges and Reflection

Although the end result was successful, each step in developing the editor came with significant challenges. As is common when undertaking any significant task, these were challenges that were not anticipated. I expected work on the project to be easier initially, as I was developing simpler features, and slowly become harder as the features became more complex. In reality, the project's difficulty was the opposite of my expectations: the initial two phases were significantly harder than all subsequent work, with the work getting easier as I became more familiar with developing the language server.

The initial research phase was surprisingly difficult. Logical Programming was a completely unfamiliar field of Computer Science prior to this project, with my only exposure being tangential references to Logical Programming in university courses on Logic. Although Logical English is an extremely simple programming language (compared to, for instance, TypeScript, which was used to develop the editor), the lack of documentation on Logical English, coupled with my unfamiliarity with its Logical or Programming context, made it a difficult task to gain the thorough understanding needed to develop an editor for the language. I would like to thank my supervisors for their support through this, who were very eager and helpful in helping me learn the language that they had created.

Surprisingly, the most difficult period of the project was in creating an language server with simple examples of each feature. This was due to the lack of documentation. Although the Language Server Protocol itself is heavily described by Microsoft, there is no documentation on how to create a language server that implements more than the most basic features of this protocol. Specifically, the `vscode-languageserver` library on which the language server is built gives little guidance on anything other than supplying simple diagnostic or auto-completion messages, only through hints in comments found in the library. In the end, the most helpful instructions were found in the source code of other language servers, which have been credited wherever my own source code built on them. These language servers were hard to find; many language servers connected to Visual Studio Code use a language server built in another language, often Java or

C++, that do not use Microsoft's language server libraries. Of the language servers that do, many use the now deprecated `vscode` library as opposed to `vscode-languageserver`.

Once basic examples of each of the features had been built, developing the language server became more straightforward. Although plenty of errors were made along the way, finding their underlying causes became significantly easier. It is a lot easier to fix errors in a language server that interacts with the editor, giving a meaningful error message when something goes wrong, rather than a language server that does not run or fails to connect. This principle applied throughout working on the language server's features. Although implementing the features at a higher level came with their challenges, the most frustrating moments were in trying to fix the many regular expressions that the language server depends on, which give no meaningful response when they do not work as intended.

A major mistake that I carried forward throughout most of the project was in underestimating the importance of frequent, thorough testing. Throughout most of my time working on the editor, tests were rushed and cursory. The only testing I would perform was to open one or two Logical English documents in the editor and check the features manually with a few brief examples. Since testing of this sort is both inconsistent and inconclusive, various errors and edge cases went unnoticed until the evaluation phase of development, caught by unit and end-to-end testing. Time and effort would have been saved if automated tests were regularly performed throughout the editor's development.

10.2 Further Work

10.2.1 Parsing complex terms

Due to time constraints, the editor parses complex terms, such as lists and dates, as atomic terms with no nested structure. This means that a clear avenue for further work is to build on the recursive parsing of atomic formulas to parse the recursive structure of other terms. To achieve this, the tree structure that the `AtomicFormula` class uses to contain other atomic formulas as terms may be adapted to a general Abstract Syntax Tree. An Abstract Syntax Tree will be useful in not only combine the parsing of atomic formulas with that of lists, but also in allowing Logical English to be extended to include user-defined functions with possibly nested applications.

10.2.2 Extending the type system

The type system was designed to be powerful enough to be useful in its current form whilst being minimal enough to be easily extended. Future work would be to incorporate the type system in its current version into the Logical English engine, so that type mismatch errors are found during, or perhaps before, runtime.

The type hierarchy could also be extended to better fit Logical English's domain of discourse. Atomic formulas are, in some ways, distinct from other compound terms in Logical English. For instance, equality is described by templates such as

```
    *a thing* is *a thing*
```

which, since all terms have their type as subtypes of the type `a thing`, applies to atomic formulas as well as other terms. However, users may expect equality for atomic formulas to differ from equality for other terms. Two constants are considered equal by comparing the constants names, however, users may consider atomic formulas with differing names as 'equal' if they are logically equivalent. This could motivate disconnecting the type hierarchy of atomic formulas from the type hierarchy of other terms, so that errors are caught when atomic formulas are used when any other term is expected.

10.2.3 Improving template auto-generation

One area of development for template generation is to support generalising terms that are assigned different types. Currently, in creating a template from an atomic formula, the process of template refinement (described in Section 8.6.1) replaces terms that reappear in other atomic formulas with their types. There is an issue to this straightforward approach, evidenced by Listing 10.1

```
 1 the type hierarchy is:
 2 a person
 3     a candidate
 4     a boss
 5
 6 the templates are:
 7 *a candidtate* accepts the job offered by *a boss*.
 8
 9 the knowledge base Jobs includes:
10 fred bloggs accepts the job offered by a boss if
11     fred bloggs likes football
12     and the boss likes football.
```

Listing 10.1: An extract of a Logical English program in which an atomic formula without a template appears twice, with arguments of different types.

From Listing 10.1, the template refinement process would create the template `*an employee* likes football` when generalising the atomic formula on line 11, and the template `*a boss* likes football` when generalising the atomic formula on line 12. However, neither template matches both the atomic formulas. To arrive at the correct template, `*a person* likes football`, the template refinement process should check if, when replacing a term with its type, using a type higher in the type hierarchy would result in a template that matches more atomic formulas. More specifically, if a term to be generalised has type A, then the type B is a candidate type to generalise the term when

1. A is equal to B, or A is a subtype of B
2. the template where B is used maximises the amount of matching atomic formulas that do not match any other template
3. B is a lowest type in the type hierarchy for which (1.) and (2.) hold

10.3 Concluding Remarks

The language client, syntax highlighter and language server were successfully built and connected to a Visual Studio Code editor. The three components correctly implemented the required features of syntax highlighting, code completion and error detection, along with the optional type checking feature. For the latter feature, a type hierarchy was proposed for Logical English, which the editor successfully implements. When tested by users new to Logical English each feature is rated highly, with average scores no lower than 8 out of 10.

The editor I have developed is, in many ways, an exploration. This is the first purpose-built code editor for Logical English and, as such, various features are experimental. It was unclear how feasible or accurate template auto-generation would be, and the type system is a new and tentative contribution to the language. However, the success of the implementation, both with respect to its requirements and as shown in the user survey, motivate and encourage further growth. This leaves the Logical English editor as complete with respect to its requirements, but with many avenues open for further development.

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 Testing the Editor

The editor is supplied with unit and end-to-end tests. The easiest way to execute these tests is through Visual Studio Code. Having opened the `vscode-package` directory in

Visual Studio Code, open the Debug menu from the Activity Bar. From here, select the ‘E2E + Unit Test’ option, then press Run. The same result can also be achieved by executing the end-to-end and unit tests through the command line, with the required commands listed in the `launch.json` file.

11.4 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 31

[2] Codemirror: Migration guide from codemirror 5 to codemirror 6, . URL https://codemirror.net/docs/migration/. pages 31

[3] Textmate grammars support for codemirror, . URL https://github.com/zikaari/codemirror-textmate. pages 33

[4] Dfs traversal of a tree using recursion. URL https://www.geeksforgeeks.org/dfs-traversal-of-a-tree-using-recursion/. pages 52

[5] Lsp4intellij - language server protocol support for the jetbrains plugins, . URL https://github.com/ballerina-platform/lsp4intellij. pages 31

[6] (textmate highlighting in intellij) textmate, . URL https://www.jetbrains.com/help/idea/textmate.html. pages 33

[7] Package java.util.function (java standard reference). URL https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html. pages 24

[8] Logical english handbook. URL https://github.com/LogicalContracts/LogicalEnglish/blob/main/le_handbook.pdf. pages 11

[9] lsp-editor-adapter (alpha), . URL https://github.com/wylieconlon/lsp-editor-adapter. pages 31

[10] Lsp example, . URL https://github.com/microsoft/vscode-extension-samples/tree/main/lsp-sample. pages 31

[11] Menhir home page. URL http://gallium.inria.fr/~fpottier/menhir/. pages 22

[12] std::predicate (c++ standard reference). URL https://en.cppreference.com/w/cpp/concepts/predicate. pages 24

[13] Stack overflow developer survey 2017. URL `https://insights.stackoverflow.com/survey/2017`. pages 50

[14] Add a language server protocol extension, . URL `https://docs.microsoft.com/en-us/visualstudio/extensibility/adding-an-lsp-extension?view=vs-2022`. pages 31, 33

[15] (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 33

[16] Logical english on the swish online editor, 2022. URL `https://logicalenglish.logicalcontracts.com/`. pages 6

[17] Language server extension guide, 2022. URL `https://code.visualstudio.com/api/language-extensions/language-server-extension-guide`. pages 22, 31, 33

[18] Anonymous. Tree datastructures (converting an indented list to a node-based tree). URL `http://rosettacode.org/wiki/Tree_datastructures#Python`. pages 52

[19] 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 23

[20] R. Davies. *Practical Refinement-Type Checking*. 2005. URL `https://www.cs.cmu.edu/~rwh/students/davies.pdf`. pages 23

[21] N. P. Duncan McGregor. *Java to Kotlin: A Refactoring Guidebook*. O'Reilly Media, Inc., . pages 36

[22] N. P. Duncan McGregor. *Java to Kotlin: A Refactoring Guidebook*. O'Reilly Media, Inc., . pages 36

[23] N. P. Duncan McGregor. *Java to Kotlin: A Refactoring Guidebook*. O'Reilly Media, Inc., . pages 36

[24] B. et al. Merlin: A language server for ocaml (experience report), 2018. URL `https://dl.acm.org/doi/pdf/10.1145/3236798`. pages 22

[25] N. Guallart. An overview of type theories, 2014. URL `https://arxiv.org/abs/1411.1029`. pages 24

[26] J. Ingeno. *Software Architect's Handbook*. O'Reilly Media, Inc. pages 44

[27] D. Jungnickel. *Graphs, Networks and Algorithms, Algorithms and Computation in Mathematics*. 2012. URL `https://books.google.com/books?id=PrXxFHmchwcC`. pages 39

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

[29] macromates.com. Textmate grammars specification, 2021. URL `https://macromates.com/manual/en/language_grammars`. pages 33

[30] mikefarah. yq. URL `https://github.com/mikefarah/yq`. pages 34

[31] S. Overflow. 2021 developer survey, 2021. URL `https://insights.stackoverflow.com/survey/2021/most-popular-technologies-new-collab-tools`. pages 31, 32

[32] G. D. Plotkin. *A note on inductive generalization*. 1970. URL `https://homepages.inf.ed.ac.uk/gdp/publications/MI5_note_ind_gen.pdf`. pages 52

[33] T. U. F. Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013. pages 24

[34] Python. Indentation in python. URL `https://docs.python.org/3/reference/lexical_analysis.html#indentation`. pages 50

[35] e. a. Rask. The specification language server protocol: A proposal for standardised lsp extensions, 2021. URL `https://arxiv.org/abs/2108.02961`. pages 22

[36] M. Ridwan. Using language servers with codemirror 6. URL `https://hjr265.me/blog/codemirror-lsp/`. pages 31

[37] C. Simovici, Dan A. Djeraba. ”partially ordered sets”. mathematical tools for data mining: Set theory, partial orders, combinatorics, 2008. URL `https://books.google.co.uk/books?id=6i-F3ZNcub4C`. pages 38

[38] TypeFox. Monaco language client and vscode websocket json rpc. URL `https://github.com/TypeFox/monaco-languageclient`. pages 31

[39] 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 23

[40] I. Zayour and H. Hajjdiab. How much integrated development environments (ides) improve productivity? *J. Softw.*, 8(10):2425–2431, 2013. pages 7, 32