Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Project report: rough draft

Supervisor:
Dr. Fariba Sadri

Second Marker:
Prof. Robert Kowalski

Author:
Nikolai Merritt

Advisors:
Prof. Robert Kowalski
Dr. Jacinto Quintero
Mr. Galileo Sartor

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

September 2022

Contents

1 Introduction **4**
- 1.1 Abstract . . . 4
- 1.2 Motivation . . . 4
  - 1.2.1 Why a new editor? . . . 4
  - 1.2.2 Why a Language Server? . . . 4
  - 1.2.3 Why a separate Syntax Highlighter? . . . 5

2 Literature Review **6**

3 Project Requirements **7**
- 3.1 Logical English . . . 7
  - 3.1.1 An overview . . . 7
  - 3.1.2 The structure of a Logical English program . . . 8
- 3.2 Project Requirements . . . 11
  - 3.2.1 Syntax Highlighter . . . 11
  - 3.2.2 Language Server . . . 11
- 3.3 Project Implementation Plan . . . 11
  - 3.3.1 Project Timeline . . . 12

4 Design **13**
- 4.1 Design Methodoligy . . . 13
- 4.2 Data Representation . . . 13
  - 4.2.1 Description of Problem . . . 13
  - 4.2.2 Element Representation . . . 14
  - 4.2.3 Literal Representation . . . 14
  - 4.2.4 Section Representation . . . 15

5 Implementation **16**
- 5.1 Semantic Highlighting . . . 16
  - 5.1.1 How a Language Server highlights . . . 16
  - 5.1.2 Template Parsing . . . 16
  - 5.1.3 Extracting the terms of a literal . . . 16
  - 5.1.4 Matching a template to a literal . . . 17
  - 5.1.5 Finding the best match . . . 17
- 5.2 Completion . . . 17
  - 5.2.1 How a language server completes code . . . 17

5.2.2 Completing the remainder of a literal 17
5.3 Error diagnosis 18
5.3.1 How a language server diagnoses errors 18
5.3.2 Diagnosing literals that have no matching template 18
5.3.3 Diagnosing clauses that have misaligned connectives 18
5.3.4 Diagnosing type mismatches 19
5.4 Quick Fixes 20
5.4.1 How a language server provides quick fixes 20
5.4.2 Fixing a lack of template for literals 20

6 Evaluation **22**

7 Conclusions and Further Work **23**

Introduction

1.1 Abstract

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

1.2 Motivation

1.2.1 Why a new editor?

his where was first oduced?

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

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

a source e about s vs epad

1.2.2 Why a Language Server?

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

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

This prompted us to then consider Monaco.

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

.

In the end, we found that what we really needed was a language server. This language server would be editor-agnostic, meaning that one language server would be able to connect to any front-end that supports the Language Server Protocol. This includes online editors such as Codemirror 6 [12] and Monaco [13], along with desktop editors such as Visual Studio Code [5], Visual Studio [3] and IntelliJ [2] some of the most popular code editors [10]. I also found that the language server I produced was able to communicate error messages with Codemirror 5. Logical English is still in an early stage of development, and producing a language server would give us the flexibility needed to branch out to all kinds of coding environments.

1.2.3 Why a separate Syntax Highlighter?

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

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

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

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

Literature Review

Language Servers have proven to be a powerful tool in creating cross-editor support for a wide variety of programming languages. As noted by Rask et al [11], the Language Server Protocol, which language servers use to communicate with code editors, "changed the field of IDEs". This is because a language server can easily communicate with any IDE that supports the protocol, thus allowing IDEs to easily support a new language. Further, In surveying the effectiveness of language servers when building a language server for OCaml, Bour et al [7] note that "adding support for a new editor to a language server requires no language-specific logic". This allows people who are not yet familiar with a given language to link a language server to their IDE of choice and begin programming.

However, building a language server does not come without difficulties. Bour et al [7], and the Visual Studio Code Language Server Extension documentation [5], describe two main challenges that Language Servers face, that of "incrementality and partiality":

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

In building their language server for OCaml, Bour et al solved these two issues by building their own parser, generated using an enhanced version of Menhir. This was needed because OCaml has a complex, recursive grammar, which made parsing incomplete portions of code a highly complex task. Logical English, however, has a very simple, non-recursive grammar, and our language server only concerns itself with parsing certain aspects of the language. Thus I expect it to be feasible for our language server to parse Logical English documents itself.

There is also existing literature on boilerplate generation from existing code. Wang et al [14] created a powerful compilation agent that auto-generates Java boilerplate code from more succint, annotated Java. The boilerplate code is generated at the Abstract Syntax Tree (AST) level: the code generator starts with the AST representing the annotated code and, using the Lombok compilation agent, produces an AST corresponding to non-annotated, boilerplate Java. Since neither templates nor heads of rules are recursive, I will likely find that a less complex representation can be favoured over AST.

Project Requirements

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

3.1 Logical English

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

3.1.1 An overview

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

```
1    fred bloggs eats at cafe bleu.
2
3    emily smith eats at a cafe.
4
5    cafe bleu sells sandwitches.
```

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

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

```
1    fred bloggs eats at cafe bleu if
2        fred bloggs feels hungry.
3
4    emily smith eats at a cafe if
5        emily smith feels hungry
6        and the cafe sells sandwitches.
7
8    emily smith feels hungry.
```

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

true.

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

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

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

3.1.2 The structure of a Logical English program

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

```
1     the templates are:
2     a person travels to a place.
3     a place has an amenity.
4
5     the knowledge base Travelling includes:
6     fred bloggs travels to a holiday resort if
7         the holiday resort has swimming pools.
8
9     emily smith travels to a museum if
10        the museum has statues
11        and the museum has ancient coins.
12
13    scenario A is:
14    the blue lagoon has swimming pools.
15    the national history museum has statues.
16
17    query one is:
18    which person travels to which place.
```

Listing 3.1: A short Logical English program.

Templates

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

Knowledge base

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

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

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

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

```
1    A
2        and B
3    or C.
```

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

```
1    A
2    and B
3        or C.
```

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

```
1    A
2    and B
3    or C.
```

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

Scenarios

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

Queries

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

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

3.2 Project Requirements

The project will consist of developing two tools for Logical English: a Syntax Highlighter and a Language Server. These two tools will be cross-editor, meaning that they can be used with many of the most popular programming editors with minimal configuration.

3.2.1 Syntax Highlighter

The Syntax Highlighter will identify both micro-features of Logical English such as keywords and variable names, and macro-features such as section headers. It will identify these features using TextMate grammar. This way, the features identified by the grammar can be recognised and styled by the default themes of many popular code editors.

3.2.2 Language Server

The Language Server will allow the user to generate new templates from rules. If a set of rules do not match any existing templates, the Language Server will communicate this to the editor. It will allow the user to, at the click of a button, generate a template that matches the rules.

If there is time, I will give the language server the feature to alert the user of certain type mismatch errors. The user will be notified of errors where a rule is supplied in the knowledge base with a type that conficts with the corresponding type in the rule's template. To determine whether the one type conficts with the other, the Language Server would consider type inheritance as supported by Logical English.

The language server will communicate with potential language clients using the Langauge Server Protocol. This way, many popular code editors will be able to easily communicate with the language server.

3.3 Project Implementation Plan

The syntax highlighter will be implemented using TextMate grammar, since this has the widest range of editor support. The Language Server will be implemented in TypeScript using the `vscode-languageserver` NPM package. This package has clear, thorough documentation which describes multiple example language servers. In testing, both the language server and syntax highlighter will be tested on a Visual Studio Code language client. This choice is made due to Visual Studio Code's powerful debugging features for language plugins.

3.3.1 Project Timeline

The timeline for developing and testing these two tools is below. This plan has us completing both the template generation and type error detection features of the language server. However, if any large problems arise, I will prioritise solving these over working on type error detection.

6th June - 10th June	Write a TextMate grammar for Logical English.
13th June - 17th June	Using the Visual Studio Code documentation [6], create a proof-of-concept language server with dummy error highlighting, warning highlighting, and code generation.
20th June - 25th June	In the language server, convert Logical English templates to a suitable TypeScript representation. Using this representation, determine whether a Logical English rule conforms to a template.
27th June - 8th July	Create a template from first two, then arbitrarily many, rules.
11th July - 23rd July	Create a TypeScript representation for Logical English types, to be used by the language server. Use this type representation in to augment the template representation with types of the template's variables. Consider types when determining whether a rule conforms to a template.

Talk about types and type hierarchy. Talk about what is required.

Talk about how lax the requirements were that were given, and how I suggested most of the features, or they were agreed on through dialogue.

Design

4.1 Design Methodoligy

Research more about the methodology I used of low-state, functional-based class, and alternate methodologies

4.2 Data Representation

4.2.1 Description of Problem

When editing a document, the initial task of the editor is to extract the document's literals and templates. It was clear early on that this is the fundamental problem: the better the representation the editor has for the literals and templates, the easier all subsequent work on them will be. It was important to design representations that:

- was lightweight: literals and templates would be reloaded every time the document received an update
- focused on the semantics, making it easy to perform queries and operations based on the meaning
- did not stray too far from the Logical English syntax, as converting to Logical English should be simple
- were not too distinct from each other: templates and literals are often used together, such as querying whether a literal matches a template

Based on the last requirement, I focused on first desgining a representation for templates, and then used those ideas to design a similar rerpresentation for literals.

Initial Template Design

Initially, the most obvious and simple design for a template was as a list of tokens, called 'elements' [1], each element being a string. These elements would either be a type, such as `*a person*`, or text that lies between the types, such as `goes shopping`. This list of elements was easily acheived by splitting a Logical English template by a

[1] This is done to distinguish these tokens from the tokens used in highlighting the document

regular expression that identified substrings of the form `*a _*` or `*an _*`.

Although the design was lightweight and easy to implement, the more I used this design the clearer it became that it did not do enough work for me. Lots of duplicate work had to be re-done whenever this representation was used: specifically, identifying which elements are types and which are not, and filtering the list of elements to obtain all the types, or all the surrounding text.

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

4.2.2 Element Representation

Thus the obvious level of abstraction was to abstract these two different kinds of substrings into two different types.
The class `Type` was created to represent the type of a template argument. The class `Surrounding` was created to represent surrounding text that lies between types. As per my design philosophy, these are both lightweight types that simply store immuntable data.

4.2.3 Literal Representation

Once I had solved the `Template` design problem, I applied the same principles to creating a design for the literals. I found throughout creating the editor that literals did not need as elaborate a design as templates: in fact, they did not even need their own class. In broad terms, a template acts on literals – mainly through extracting terms, or checking whether a given literal matches its form. So, while templates have rich functionality, literals are passive objects that are acted on. Thus there is no need for a literal class with methods or mutating state.

Like a template, a Logical English literal consists of text that is either a typed term, or is surrounding text. Like with templates, the natural representation is simply a list of `LiteralElement` elements, where a `LiteralElement` is either a `Surrounding` or a `Term`.

Term

In logical english, a term is a value with an associated type. The natural data structure to represent this is therefore:

```
1     class Term:
2         name: string
3         type: Type
```

with each of these properties being immutable. It is important that the `type` property is a reference to the corresponding type, not a copy. This is crucial to see if two uses of the same Logical English term have conflicting types.

4.2.4 Section Representation

Along with representing Logical English data, it was also important to be able to refer to where the data is in the document. This is crucial in highlighting, providing diagnostic error underlines and identifying the current literal that the user is typing. Attaching position data to these representations themselves was a bad idea for two reasons: by the design principle of concerns , the representations are "abstract" in that they represent what a logical english construct is, not where it happens to lie in a document. Further, it is important to know where bodies of raw text (i.e. `string`s) are: these, of course, do not have position data.

Name the de sign principl

ContentRange¡T¿

This is a generic type, allowing the type to store any kind of data in its field. For a given type `T` (a `string`, a `Template` or any other kind of content), a `ContentRange<T>` has a `content: T` field, and a `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. Note that `T` could be any type whatsoever, including types that do not make sense.

Implementation

5.1 Semantic Highlighting

5.1.1 How a Language Server highlights

Research this in more detail

5.1.2 Template Parsing

First, the lines containing templates are found. These are found by starting after the header `the templates are:`, continuing until either another header or the end of the document is reached. A corresponding template is constructed from each non-empty line by the template class: sub-strings wrapped in asterisks are taken to contain types, and the relevant `Type` object is put in place, taken from the type tree.

5.1.3 Extracting the terms of a literal

The fundamental problem here is, given a template `T` and a literal `L` that is assumed to match, how to extract the elements of the literal? In short, this is done by leveraging the assumptions that the template and the literal share the same surroundings. Under this assumption, comparing the template's surroundings against the literal yields the literal's terms. The algorithm for this is given below.

Write up algorithm here

This algorithm was constructed using the idea of incremental problem solving in my Algorithsm course. There is a slight subtlety here: what happens if a surrounding also appears as a term. For example, a scenario where a merchant packages and sends items could be described with the template `*a merchant* ships *an item*`. If we have a merchant who packages and sends ships, then the corresponding literal would be `the merchant ships ships`.

Describe how this nuance is handled

Talk about how this algorithm works with the edge case of incomplete literals also

5.1.4 Matching a template to a literal

Once all the template representations have been created, the templates can be used to highlight the terms in each literal. Starting with a given literal, the templates are filtered by whether or not they match the literal. For each template, once the literal's elements have been extracted, the elements are checked to see whether they are isomorphic to the template's elements. That is, whether the surroundings match up between them.

5.1.5 Finding the best match

Old design: takes first template that matches. Problematic, especially with 'default templates' e.g. "*a thing* is *a thing*"
New design with ranking *partially written* templates according to how well they match

Initially, it was assumed that a literal can only match one template. 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. With the above example, if a user then supplies a template `*a beneficiary* is included in *a will*`, then any literal that matches the latter template – for example, `jane is a beneficiary of the will` – will also match the former template. However, using the former template to extract terms will be incorrect, such as highlighting `a beneficiary of the will` as a single term.

This motivated a 'match score' between a literal and a template. The higher the score, the better the template matches the literal, and the more certain we can be that it should be used to extract the terms.

Write up scoring algorithm

Con: this is not normalised – how are comparisons then meaningful?

5.2 Completion

5.2.1 How a language server completes code

Research this in more detail

5.2.2 Completing the remainder of a literal

To offer completion for a literal, first its corresponding templates are found. This is not as straightforward as searching for which templates match the literal, because the literal will be incomplete, and so will not match any template. Instead, the templates are ranked by their match score against the literal, and the top three are taken.

There is already some nuance here. Some matches will be "obviously" irrelevant, such as (the dreaded) `*a thing*` `is` `*a thing*` against `the person is a beneficiary of` . These cannot be ruled out algorithmically, however, since they do match. They will, however, be out-ranked by templates with better-matching surroundings, and will appear lower in the list. Finding best three templates according to match score. Filling in templates with terms that the literal has so far.

Justify why at most three literals are suggested. Research user design.

5.3 Error diagnosis

5.3.1 How a language server diagnoses errors

Research this in more detail

5.3.2 Diagnosing literals that have no matching template

Given the above discussion it is fairly straightforward to find literals that do not match any template. The nuance is in using this as the deciding factor actually being too eager. Specifically, incomplete literals (e.g. literals that are being typed) will, in general, not match a template. This means that every literal that is being typed will be diagnosed as incorrect until it is finished.

Talk about how this is impossible to fix: on diagnosis time, all that is given is the text document, not the cursor position.

5.3.3 Diagnosing clauses that have misaligned connectives

This is another problem that sounds simple in theory, once the above framework was developed, but turned out to have nuances in practice. Each clause is split into its lines, and the lines containing the keywords `and` and `or`, which have equal precedence, have their indentation compared. If two literals with different connectives have the same whitespace, then the precedence of the connectives is ambiguous.

The nuance is with the term 'same indentation'. There are two equally common ways to indent lines: tabs and spaces. There is no standard way to display a tab in terms of spaces: any range from two to eight spaces is common. Thus, if one person indents literals using tabs, and the other using spaces, then it is up to interpretation as to whether the indent is correct or not.

Cite sources on this. Talk about how this is a problem in Python, and that there is no solution. Talk about how it should therefore be an error to mix spaces and tabs, but I do not want to check for this.

Issue: spaces versus tabs

5.3.4 Diagnosing type mismatches

Feature Overview

Although not part of the editor's requirements, the Logical English development team wanted a way to introduce experimental support for typed terms. The argument labels in templates would give the type of their argument. This feature would be used for finding type errors in multiple uses of the same term across a clause. Since this feature is experimental, it was not supposed to clash with any existing features or hinder the experience of a user who did not want this.

The way I chose to implement this feature was to have type checking turned off by default, only being turned on with an explicit `type checking: on` comment. The type hierarchy was also designed to be as minimal as possible.

Move this discussion to Requirements section.

Initial design: flat type hierarchy

When experimenting with implementing this feature, I first implemented type checking before implemnting a type hierarchy. In each clause, the literals were extracted, and, using the document's templates, the terms were extracted from each literal. These terms were typed, but there was no notion of sub-type or super-type. 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, with many more surprising error messages being generated. This was mainly because of the default templates, and for two reasons. Firsly, since the default templates are very broad, they use placeholder type names, such as `A`, `B`, `C`, `thing`, et cetera. Ths caused problems for two reasons. Firstly, the type names clashed with the more specific type names found in real-world Logical English examples, if they were used together. Secondly, the type names often clashed amoungst themselves: take, for example, the template `*an A* appended to *a B* gives *a C*`, and the literal `[] appended to the list gives the list`. Here `the list` is both of type `B` and `C`.

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

The solution to the first problem could only be resolved through a type hierarchy, making sure that the types used in the default templates would be super-types of whatever is used in more specific templates. However, the solution to the second problem was immediately resolvable: rename the types. All general type names such as `A`, `B`, `C` et cetera were renamed to `thing`.

Diagnosing with a type hierarchy

First, the type tree is read from the document under the header `the type hierarchy is:`. Parsing a text version of a tree is a well-known algorithmic problem.

Cite the solution.

Now instead of simply checking whether two type names are equal, the two types are checked to see if one is a subtype of the other. Since types are rerpesented in a tree, this is a simple tree search problem to see whether either type node has a child node with the same name as the other type.

Cite the solution.

5.4 Quick Fixes

5.4.1 How a language server provides quick fixes

Research this in more detail

5.4.2 Fixing a lack of template for literals

Template Generation using Least General Geneneralisation

The first candidate template is generated according to the principle of least general generalisation. This is a principle from Logic-Based Learning which states how to generate a predicate that matches a set of given instantiations of the predicate, that is no more general than is necessary to match all of them.

Research and describe this more.

The way this is done in the editor is as follows. Each literal is split into a list of space-separated words. It is assumed that there is a common template which matches all the literals. In the initial design, the predicate words were identified as the intersection of all the words. This criteria is slightly refined, and will be discussed later.

Having found all of the predicate words, the terms of the first literal are identified as the words that are not predicate words. Knowing both the predicate words and the terms of the first literal, I then generate a template that matches this. If this same template matches all the other literals, then it is returned.

Write up the algorithm.

Template Refinement from the re-use of terms

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

This was done by giving the `Template` class a method to generate a more general

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

Evaluation

We haven't done any evaluation yet.

Conclusions and Further Work

We haven't finished yet.

Bibliography

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

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

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

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

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

[6] Programmatic language features, 2022. URL https://code.visualstudio.com/api/language-extensions/programmatic-language-features. pages 12

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

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

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

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

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

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

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

[14] Y. Wang, H. Zhang, B. C. d. S. Oliveira, and M. Servetto. Classless java. *SIGPLAN Not.*, 52(3), oct 2016. ISSN 0362-1340. doi: 10.1145/3093335.2993238. URL `https://doi.org/10.1145/3093335.2993238`. pages 6