

TempTing

A Template Development Environment for Abstract Wikipedia

Matthew Craig
University of Cape Town
Cape Town, South Africa
CRGMAT002@myuct.ac.za

ABSTRACT

A vision for a truly multilingual Wikipedia has been proposed, architected, and specified. The vision, titled Abstract Wikipedia, aims to generate Wikipedia articles from base, abstract representations of content. Despite progress towards this goal, there exist practical barriers preventing users from contributing. This report details the development of a software tool that broadens the accessibility and functionality of Abstract Wikipedia. This tool provides a web-based development environment for template creation. These templates are necessary for realising abstract content in natural language. A custom parser was developed for the template syntax. The parser provides context-aware diagnostics, syntax highlighting, and autocompletion capabilities to the development environment. This paper covers the tool’s purpose, design, implementation, and successes. It was demonstrated that the system could adequately handle valid and invalid user submissions. The system was designed to allow for its components can be repurposed in future efforts within the Abstract Wikipedia project.

CCS CONCEPTS

• **Software and its engineering** → **Parsers**; *Designing software*; • **Computing methodologies** → Natural language generation.

KEYWORDS

Abstract Wikipedia, templates, parsing, templatic NLG

1 INTRODUCTION

1.1 Context

Abstract Wikipedia is a much broader project, of which this project forms only a small part. Abstract Wikipedia envisions a system of open collaboration between people of diverse, multilingual backgrounds [16]. Its goal is to leverage *Wikidata* [15] and *Wikifunctions* [16] to facilitate the creation of language-agnostic representations of content. Natural Language Generation (NLG) techniques will be utilised to produce articles from these abstract representations in human languages.

Abstract Wikipedia aims to deterministically generate Wikipedia articles from the knowledge stored in Wikidata, an established knowledge graph [15]. The content is to be composed in language-independent representations called *constructors* [2]. A constructor specifies content to be extracted from Wikidata. Each of these constructors is to have, associated with it, a set of language-specific *templates* [6]. A template acts as a function that transforms selected content into natural language. Constructors pass selected Wikidata content to a template as arguments. A template details the language-specific arrangement of this content. The templates arrange this

content with use of dependency-relation labelling, lexical function calls, and sub-template invocations [6].

Constructors and templates serve as representations within the natural language generation (NLG) pipeline [2]. Templates, after receiving content from a constructor, can be realised as a natural language article. The project discussed in this report focuses on improving the process of writing templates for use in the realisation process.

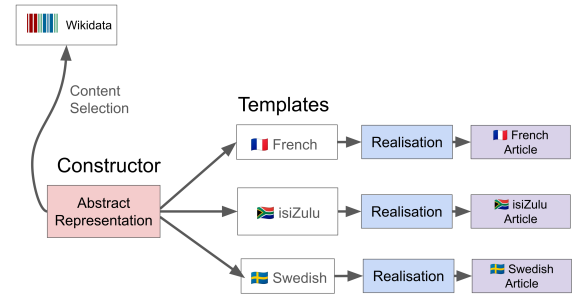


Figure 1: Simplified Abstract Wikipedia NLG Pipeline

1.2 Problem Statement

This project seeks to address some Abstract Wikipedia’s current shortcomings, particularly those pertaining to templates. These can be loosely categorised into two categories: *Functionality* and *Accessibility*.

1.2.1 Problem: Functionality. The template functionality required by Abstract Wikipedia is largely absent. Partial implementations - discussed in section 2.2 - have been attempted, but progress towards a production-ready system has not yet begun. This project focuses on the issue of absent template validation and parsing. Templates are unable to be realised as natural language without a system that can handle these tasks.

1.2.2 Problem: Accessibility. Significant technical barriers are preventing the general public from contributing to and benefiting from Abstract Wikipedia. Even if templates were to be fully functional in selecting and realising content, there persist significant hurdles in interacting with them. There are no existing tools that improve the ease of use, accessibility, or comprehensibility of the project.

The template creation process is complex and unintuitive without guidance. This deters a majority of potential users. It is infeasible to expect users to devote time to understanding the syntax and functionality. As it stands, the only means by which users can acquire

knowledge of templates is through reading the specification [6]. These accessibility problems with template creation are a priority of this project.

1.3 Project Overview and Aims

The project proposed here aims to solve many of the problems that currently affect Abstract Wikipedia’s templates. To solve these problems, a web tool has been developed. This tool, titled *TempTing* is a featureful development environment specialising in template management and creation.

TempTing was developed in parallel with two other projects. The first of which is a system for optimally extracting relevant Wikidata content for use in constructors. The second is a tool similar to TempTing but for managing the creation of constructors. Ideally, all three components will eventually be integrated. This project focuses entirely on templates, as the other two projects have covered the constructor and Wikidata aspects.

TempTing is a user-facing tool that aims to streamline template creation. The tool allows users of various languages to create templates necessary for realising constructors in natural language. Realisation (NLG) falls outside of the scope of this project. The components of this project have, however, been designed modularly such that they can be easily integrated into future realisation efforts. This will be discussed further in section 4.5.

The project aimed to produce a robust user interface and suite of tools that improve the template creation experience. These tools are seamlessly integrated into the web app and include:

- A custom parser
- A syntax validator
- A linter with in-line diagnostics
- Syntax highlighter
- Autocompletion
- A template browser

This web app intends to reduce friction and barriers that currently disincentivize engagement with Abstract Wikipedia.

2 BACKGROUND AND RELATED WORK

2.1 Abstract Wikipedia

The project will be developed with influence from the work of Denny Vrandečić. Vrandečić justified [14] and proposed [16] the Abstract Wikipedia project. As discussed in section 1.1, Abstract Wikipedia is a project that aims to develop a process by which Wikipedia articles can be generated from base, abstract representations of content [14]. The ability to deterministically generate Wikipedia articles is particularly beneficial for low-resourced languages. isiXhosa Wikipedia, for example, has 2 092 articles and only a single admin [11]. This contrasts with English’s 6 873 087 articles and 854 admins [11]. Low-resourced languages often lack in article quantity, quality, and recency. A single language agnostic representation would allow for the generation of articles in all supported languages, ensuring that they are immediately substantive and up-to-date.

Upon initial inspection, such a system may feel unnecessary. Why not simply use machine translation to make English content available in all other languages? The problem is that doing so risks a

form of cultural imperialism. English, and by extension the culture of the Anglosphere, would be asserted as the core source of truth. Abstract content has the distinct benefit of supporting contributions from anyone, regardless of language or culture [14].

2.1.1 Wikidata. Wikidata is an open, collaborative knowledge base [15]. Wikidata houses a large collection of labelled entities and the relationships they have with other entities. Its goal is to provide a diverse collection of machine-readable knowledge that anyone may contribute to and benefit from [15]. The content is, however, largely inaccessible to the broader public. This is due to the technical barrier prohibiting interaction with the content. Abstract Wikipedia intends to solve this problem by producing human-readable natural language from the knowledge housed in Wikidata. [16].

Each concept stored in Wikidata is associated with a unique identifier prefixed with Q. For example, Douglas Adams is identified with Q42 [2]. These concepts are related to each other by property identifiers (prefixed with P) such as `has occupation (P106)`. Constructors, the abstract representations (section 2.1.2), specify content through use of these Q and P identifier.

2.1.2 Constructors. The abstracted representations of content are given the term *constructors* [16]. Constructors are declarative statements of content to be selected from Wikidata. The declarations can be conceptualised as expressive arrangements of language-independent Wikidata identifiers. They are modular representations of content that can be composed to form an article [2]. A modelling language for constructors, *CoSMo*, has been formally specified [2].

2.1.3 Templates. Constructors are inherently multilingual [2] and thus require an intermediary language-specific representation before they can be realised as natural language. These representations, called templates, are specific to a constructor and language. Each template describes how constructor-specified content is to be arranged in a particular language [6]. Natural language, such as Wikipedia articles, is to be generated from the realisation of these templates. The specifics of the template structure and syntax is discussed in section 2.3. Earlier work, such as a Vrandečić’s proposals [14] [16], use the term “renderer” to refer to a concept roughly equivalent to templates. These renderers were only given a pseudo-syntax and existed as a placeholder concept.

2.1.4 Wikifunctions. Wikifunctions, another project by the Wikimedia Foundation, has recently launched [10]. Wikifunctions provides a platform that aims to democratise access and contribution to functions [16]. Currently, only functions written in the Javascript and Python programming languages are supported. It is intended that the templates, lexical functions, constructors, and NLG functionality of Abstract Wikipedia are to be hosted on Wikifunctions [6]. This prospect aims to broaden Abstract Wikipedia’s potential for open collaboration.

2.2 Existing Implementations

2.2.1 Ninai/Udiron. Morshed (2023) [12] outlines the specification and development of an NLG system for Abstract Wikipedia titled Ninai/Udiron. Ninai/Udiron consists of two core components:

- (1) *Ninai*: Searches Wikidata for items and lexemes for concepts relating to a constructor. It processes the results into syntax trees.
- (2) *Udiron*: Manipulates the syntax trees and converts them into natural language.

Ninai/Udiron serves as a demonstration that NLG within Abstract Wikipedia is viable. The system is, however, highly coupled to the Python programming language. Constructors, rather than using the CoSMo syntax [2], are defined as nested Python functions. Templates, as discussed in sections 2.1.3 and 2.3 do not exist in Ninai/Udiron. Instead, reminiscent of Vrandečić’s proposals [16], Python functions called *renderers* are used to realise constructors. This reliance on Python-implemented representations is less accessible than the combination of CoSMo [2] and templates.

2.2.2 Scribuntu Implementation. A prototype implementation of the template system has been created [5]. It is integrated within Scribuntu, Wikipedia’s embedded Lua-based scripting environment. This prototype demonstrates that the proposed template system is adequate for realising natural language from Wikidata content. The implementation was not, however, designed to be a robust final system. The parser takes concessions such as omitting support for multiple function arguments or nested function calls [5]. Additionally, the system is not equipped to handle invalid templates. No diagnostic information is provided to guide users through template creation.

2.3 Template Syntax

A syntax for templates, which this project adheres to, has recently been proposed ([6]). In this syntax, templates are composed of slots (enclosed in braces: { }) interspersed with free text. Slots can represent interpolations of arguments, lexical function calls, or sub-templates [6]. Slots are to be given dependency labels which identify their grammatical role and relation to other syntactic elements. The specific role (dependency label) identifiers used here are based on those provided by Universal Dependencies [18]. Free text is simply a string of text in the chosen language; typically used when dependency labelling does not affect its realised output. A full specification of the grammar can be found in the appendix (table 6).

2.3.1 Template Example 1. For example, take a template for the English language titled *item-in-container* (fig. 2). The appendix includes a figure (fig. 8) demonstrating the template within the context of the TempTing editor. The template takes an *item* and a *container* as arguments from a constructor. When given the arguments *item* = "ant" (Q115705859) and *container* = "box" (Q188075), the template would be realised as “There is an ant in the box”.

It’s worth noting a few aspects of this template’s presentation in fig. 2. Each slot (enclosed in braces { }) is presented on its own line. This does not affect the output but is useful for demonstration. Free text and strings and highlighted normal-weighted black. The arguments that the template will receive from a constructor are highlighted in green. Roles, part of a template’s dependency label, are highlighted in purple and suffixed with “:”. These roles state

```
{expl:"There"}
{root:"is"}
{det<nsbj:Lexeme("a")}
{nsbj:item}
in the
{obj:container}
```

Figure 2: Template: item-in-container
Arguments: item, container

the dependency relation a slot has; here, “is” is the root of the template. Some roles are succeeded by a “<” symbol which indicates their dependency relation with another syntactic element. For example, the Lexeme("a") slot has the dependency label *det<nsbj*. This indicates that the slot has the role *det* (determiner) and depends on the *nsbj* (nominal subject). In this case, the *nsbj* is an invocation of the *container* argument. For slots where “<” is omitted, it is inferred to be dependent on the root. In the case of role-identifier conflicts, the role must be followed by an index. Indexes are prefixed with “_” (eg. *expl_2*).

Function invocations can either be lexical functions or sub-template invocations. In fig. 2, function invocations are highlighted with bold black. In this example, the lexical function *Lexeme* is invoked with “a” (the indefinite article) as an argument. The template syntax specification indicates that the *Lexeme* function would actually take the *Lexeme* identifier L2767 (English indefinite article), however, “a” has been used for demonstrative clarity. This *Lexeme* function is used so that the lexeme (lexical unit) can be realised as either the “a” or “an” lemmas (dictionary forms). The choice between “a” and “an” resolved by the *det<nsbj* dependency label. The indefinite article (*det*) is dependent on the *item* argument (*nsbj*).

For example, let *container* = "box" (Q188075):

- When *item* = "ant" (Q115705859) → “There is an ant in the box.”
- When *item* = "person" (Q215627) → “There is a person in the box.”

2.3.2 Template Example 2. The previously discussed example can be modified to facilitate numbered items. The new template, *items-in-container*, would take in the arguments *count*, *item*, and *container*. Through dependency labelling and the use of the *Cardinal* function, the chosen lemmas are dependant the plurality of *count*.

Let *item* = "person" (Q215627) and *container* = "box" (Q188075):

- When *count* = 1 → “There is one person in the box.”
- When *count* = 2 → “There are two people in the box.”

To achieve the aforementioned pluralised template, a sub-template invocation will be used. In fig. ??, the template invokes the sub-template *items* 4. The *items* template (4) takes in an *item* and a *count* and transforms them into a numbered singular or plural form. *items*(2, "person") → “two people”. The *nummod* dependency label identifies *Cardinal*(2) as a numeric modifier.

While equivalent realisation could be achieved with a single template, this example demonstrates the utility of template nesting.

```
{expl:"There"}
{root:Lexeme("is")}
{nsbj:items(count, item)}
in the
{obj:container}
```

Figure 3: Template: items-in-container
Arguments: count, item, container

```
{nummod:Cardinal(count)}
{root:item}
```

Figure 4: Template: items
Arguments: count, item

Templates are composable, modular, and reusable. This property intends to accelerate template development.

3 REQUIREMENTS AND DESIGN

3.1 Requirements

Requirements were determined and prioritised through analysis the current state of Abstract Wikipedia and existing implementations. This revealed that two primary components would be required:

- A web application that aids in the editing and management of templates.
- A custom-built parser for the template syntax.

3.1.1 Web App Requirements. Wikimedia projects, such as Wikifunctions [10], are typically typically distributed via the web. To align with adjacent projects, it was considered a requirement that TempTing be implemented as a web app.

Vrandečić’s Proposal states a need for editing tools within Abstract Wikipedia [16]. These tools must make it easy to “create, refine, and change” [16] content. The proposal gives a hypothetical example of a tool which gives contributors immediate feedback [16]. This feedback that would help them learn to “express themselves in the constrained language the system understands” [16]. Based on these statements, it was concluded that TempTing should provide an integrated template-editing environment with features that enable immediate feedback. The following editing features were chosen to achieve this:

- Detailed diagnostic and linting information. Contributors receive real-time feedback on template validity.
- Syntax highlighting of the elements of template syntax.
- Autocompletion for dependency labels and argument invocations.

Screenshots of each features of these within the TempTing editor can be found in the appendix (fig. 9, 11, ??). Additionally, to aid in the refinement and modification of existing templates, a template management system was deemed a requirement.

While the template syntax could accommodate any dependency-labelling system, Universal Dependencies prioritised. Both Ninai/Ud-iron [12] and the template-syntax specification [6] make use of Universal Dependencies. Thus, TempTing’s editor was required to support Universal Dependencies.

3.1.2 Parser Requirements. Requirements for the parser centered around the template language specification [6]. In particular, adherence with the provided grammar (fig. 6) was considered essential. There are some discrepancies between the provided grammar and provided examples. Notably, the suggested functions accept atypical invocations such as lexeme identifiers, Wikidata Q-items, and integers [6]. These invocations do not exist in the grammar. Additionally, the Hebrew examples make use of the | character for conditional elision [6]. This is despite its absence in the grammar. It was decided that adherence to the grammar was preferable, and thus TempTing’s parser did not require support for these edge cases. Comparable functionality can, however, still be achieved with the more minimal syntax. Lexeme and Q-item identifiers could be wrapped as strings. Conditionals could be achieved through function nesting, rather than with dedicated syntax.

The parser’s requirements were influenced by the shortcomings of the Scribuntu prototype [5]. TempTing’s parser was required to offer full support of the syntax specified in the grammar. This includes multiple function arguments and function nesting. The parser was required to emit context-specific error messages when encountering invalid syntax. To improve editing feedback, the parser would need to be fault-tolerant. This means that each slot must be parsed independently, resulting in the production of a list of error messages. This contrasts with a parser that would fail upon encountering its first error.

It is idealised that components of Abstract Wikipedia, such as the parser, would eventually run on Wikifunctions. This means that the implementation’s programming language must be supported by Wikifunctions. Additionally, in order for the parser to be integrated with the web app, it must be able to run in a browser.

3.2 System Overview

The web app has been designed with a separate frontend and backend. The frontend, accessible via a browser, communicates with the backend via its REST API. The parser, detailed in section 3.3, is designed to be integrated in the frontend. The parser communicates with the editor so that it may provide the required features (section 3.1.1). A high-level overview of this system can be seen in figure (fig. 5).

3.3 Parser Design

3.3.1 Parser Prototype. During the parser design process, an exploratory prototype was developed. This prototype was rapidly implemented with unified lexing and parsing stages. It was created to determine an adequate design that would accommodate the required parsing features (section 3.1.2). This aided in designing a modular system that could support contextual error messages, fault tolerance, and nested function calls.

3.3.2 Parser Pipeline. The parser was designed as a pipeline (fig. ??). A template is transformed, by the lexer stage, into a stream of tokens. The tokens are processed by the parser stage to produce either an abstract syntax tree or list of errors. The exploratory prototype revealed that an additional error handling stage would be useful. The error handling stage produces formatted error messages from the information in error list. Thus far, the term “parser” has referred to the entire parsing pipeline. The “parser stage” is considered to

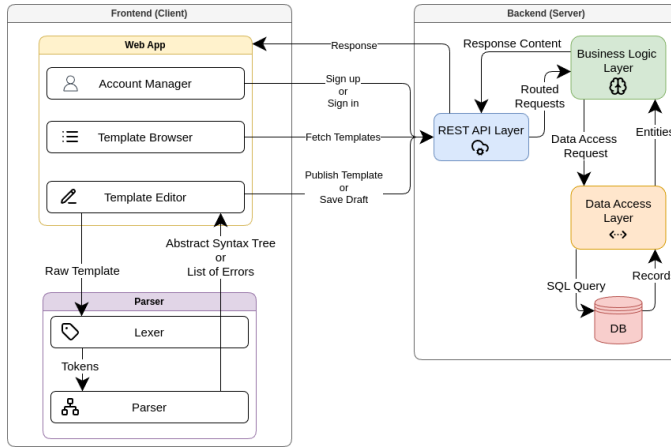


Figure 5: System Overview

be a sub-component of the parsing pipeline. The parsing pipeline was designed with inspiration from Ball’s interpreter textbook [3].

3.3.3 Pipeline Stages. The design for the pipeline stages is visualised in fig. 6.

The lexer stage exists to convert a plain-text template into labelled tokens. These labels include information about the token’s line and column number. This is necessary so that the produced errors may include positional details.

The parser stage receives tokens from the lexer. Using slot-delimiter tokens, a list of slots is produced. Each slot is parsed independently so that the required fault-tolerance can be achieved. The parser, depending on the context in which an error is encountered, produces a constructed error-entity. These errors are later be unpacked, by the error handler, as formatted error messages. Free text is also be parsed to identify punctuation marks.

The errors handled by the parsing pipeline are limited to syntax errors. The linter integrated into TempTing will use parser output, provided arguments, and chosen language to produce post-parsing errors.

3.4 UI Design

Effective user interface design was deemed a priority for the project. An inviting user interface can often be the determining factor in ensuring that a user adequately engages with a system. The burden of learning the template syntax poses a significant user-retention hurdle that a good UI could help alleviate.

3.4.1 UI Views. The UI was designed to include three distinct views. The template creator, template browser, and account manager. Fig. ?? presents their context within the broader system. The template creator includes the template editing panel and inputs for specifying template arguments, title, description, and language. The template browser displays existing templates and allows users to edit them.

3.4.2 Responsive Design. The UI is intended to be used on a desktop, however, it has been designed to support mobile. The entire application utilises responsive design to adapt to differing screen sizes. Some components, such as the template creator, are given

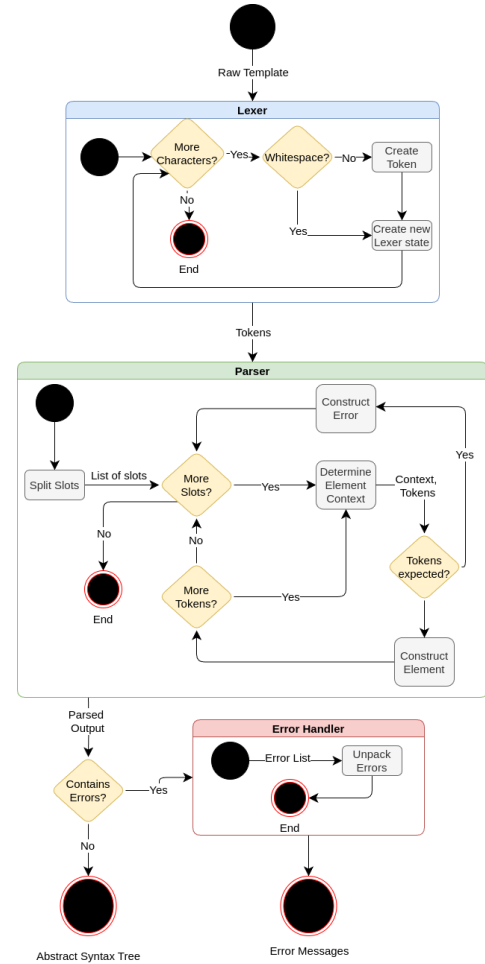


Figure 6: Parsing Pipeline

unique layouts that cater towards smaller screens. Mobile compatibility was targeted to broaden the potential user base.

3.5 Backend Design

The backend intends to be relatively simple, existing primarily to manage user authentication, template storage, and template retrieval. Template and user information is stored in a relational database that is managed by the backend.

The backend, visualised in fig. ??, uses a simple three-tiered architecture. The *REST API layer* acts as the presentation layer. This layer receives requests from the frontend and routes them to processing functions in the *business logic layer*. The *business logic layer* processes the provided information and requests data from the *data access layer*. The *data access layer* constructs the necessary SQL queries and communicates with the database.

3.6 Database Design

Data management does not make up a significant portion of this project’s capabilities or focus. The database design, thus, prioritises simplicity and stability. Only data regarding users and templates are

stored. The database follows a relational schema that is portrayed in fig. 7.

The users table information about user accounts. Passwords are hashed before they are stored, protecting the user in the event of a leak. Upon login, a session key is generated, hashed, and stored in the sessionKeys table. The client is sent a copy of the session key to store locally. The client must send a valid session key when attempting to perform actions such as publishing a template.

The published table uniquely identifies each published template. Multiple language implementations of a template, stored within the versions table, can associate with a single published entry. The actual content and metadata of a template implementation is stored within the templates table. The content (template table) is decoupled from the versions table so that it may be referenced from the drafts table.

The database does not store the available roles for each language. These roles are delivered directly to the client. This is articulated in section 4.4.

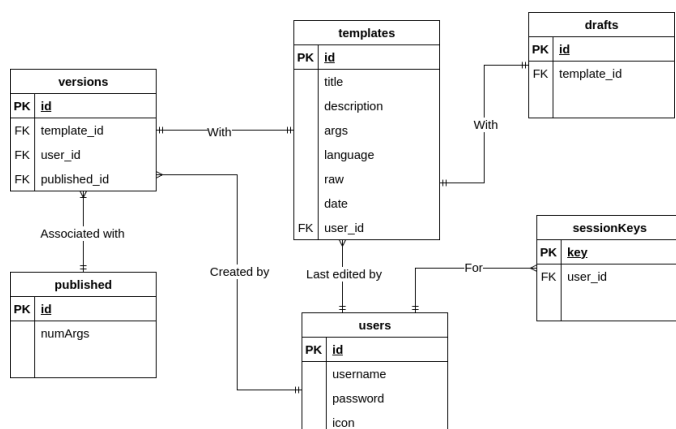


Figure 7: Crow's Foot ER Diagram

4 IMPLEMENTATION

4.1 Parser Implementation

4.1.1 Programming Language Choice. To provide the desired text-editing functionality, the parser needs to run on the client. Ideally, the parser would also be able to run on Wikifunctions. It is for these reasons that the parser must execute as Javascript code. The parser was not, however, written in vanilla Javascript. The parser was written in Gleam, a modern, statically-typed, functional language that can transpile to Javascript [13]. This transpilation, works similarly to Typescript. Gleam uses a Hindley–Milner type system [13], which is particularly well-suited to parser development. Gleam's control flow is handled exclusively through exhaustive pattern matching. In the context of parser development, this type-system and control flow forces one to handle all possible execution paths. It was concluded that Gleam would be an ideal fit for developing the parser as it offers assurance that all edge cases are covered.

4.1.2 Parser Justification. A significant portion of this project's development focused on the development of a custom parser for the

template syntax. This was done despite the availability of parser generators. A custom parser was developed because it provides significantly more control over what contextual information is returned in the event of a syntax error. For example, “_” is used in slots to prefix a role index. New users may attempt to use “_” in a snake_cased function name. Given the following slot:

```
{
  myRole:example_function()
}
```

- TempTing's parser produces the following syntax error:

```
Syntax Error:

Function name: example_function

'_' is not a valid character for Function name.

Invalid Function name format:
Format must be in the form:
One-word identifier.
It can be the name of another template,
which will invoke it as a sub-template

TIP: ` ` is prefixed before role indexes.
Separation of multi-word identifiers can be done with:
camelCase (capitaliseEachWord)
or kebab-case (put-a-hyphen-between-each-word).
```

- A parser generator, such as *Nearley* [1], would produce the following syntax error:

```
Line 2, column 13: Expected ![{_<:(), \n\r\t"},
end of input, or whitespace but "_" found.
```

The error returned by the parser generator is unable to provide adequate feedback to the user. TempTing's parser gives instructions to resolve the issue.

4.1.3 Parser Algorithm. The lexer and parser were written in a pure functional style. Reflecting on section 3.3.2, the lexer transforms an input string into a list of Tokens. The parser transforms the list of Tokens (produced by the lexer) into a Template. The role of the lexer is to produce a token list that ignores whitespace and to assign positioning (line and column number) information to each token. The parser produces a tree (AST) from these tokens and returns a contextual error message when an unexpected token is encountered. The parser is implemented as a recursive descent parser [3] but with heavy reliance on Gleam's type system.

The Lexer type is a product type containing a list of previously lexed tokens, the current position, and a context function. The Lexer's context function takes in a lexer and a character and produces a new lexer. The lex function folds the list of input characters with a Lexer as an accumulator. For each input character, the current Lexer's context function is called to produce the subsequent Lexer. A context function pattern matches based on the input character and returns a new Lexer which may have an added Token and new context function. For example, in the label context, “_” would add an IndexPrefix token to the token list and change the

context function to index. The core lexing function is as follows (note that `|>` is the pipe operator in gleam):

```
fn lex_with_lexer(input: tok.Chars, lexer: Lexer) {
  input
  |> fold(lexer, fn(l, char) {
    l |> l.context(char) |> update.increment
  })
}
```

The `Template` type models a template as a tree of sum and product types. The parse function takes the output list of tokens outputted by the `lex` function and transforms them into a `Template`. A `Template` is treated as a list of `Results` so that it may be fault-tolerant. Each slot is parsed independently and can produce its own errors. This is useful for linting purposes as it allows for multiple slots in a template to separately display diagnostic information. A syntax error in one slot will not prevent another from being parsed. Each slot is parsed recursively to create a `Slot` constructor. Recursive descent parsing is necessary because function calls are nestable. The parsing operations recursively call contextual parsing functions (eg. `parse_slot` or `parse_invocation`) until an element is returned. Gleam's monadic bind operations such as `try` and `guard` are used. These operations ensure that the parser short-circuits in the event of a syntax error and returns the contextual error type. The error types are themselves constructed as a tree and are unwrapped to produce their context-aware descriptions. This is the error handler stage of the pipeline discussed in section 3.3.2.

The subsequent block is the `Template` type definition in `template.gleam`. In Gleam, the `type` keyword, when followed by `=` denotes a type alias. When followed by a `{ }` block, the `type` keyword denotes a set of type constructors. The types are implicitly sum types with each type constructor being a product type. Here, an `Element` is constructed with either `Slot(dependency_label, invocation)` or `FreeText(text)`.

A Gleam instance of this `Template` type is produced as parser output. A final step is employed to convert this `Template` instance into JSON, so that it may integrate with other systems more seamlessly.

```
pub type Template =
  List(Result(Element, err.Syntax))

pub type Element {
  Slot(dependency_label: DependencyLabel,
        invocation: Invocation)
  FreeText(text: FreeText)
}

pub type FreeText {
  LexemeOrString(lexeme_or_string: String)
  Punctuation(punctuation: String)
}

pub type DependencyLabel {
  DependencyLabel(label: Label, source: SourceLabel)
  Root
}

pub type SourceLabel {
  SourceLabel(label: Label)
  SourceRoot
}

pub type Label {
  Label(role: String, index: Int)
}

pub type Invocation {
  Function(name: String, args: Array(Invocation))
  Interpolation(interpolation: String)
  StringInvocation(string: String)
}
```

4.2 Frontend Implementation

4.2.1 Frontend Language and Libraries. The frontend was implemented as a Typescript *React* single-page application. React, like other Javascript frameworks, enables structured and efficient declarations of web UIs [4]. React shines due to its popularity and ecosystem. *Shadcn* [9], a React component library was the particular standout which solidified React as the framework of choice. Ideally, the frontend would've also been built in Gleam but it does not yet support any libraries comparable to *Shadcn*.

Shadcn is a collection of flexible pre-built React components [9]. Unlike other component libraries, *Shadcn* components are inserted directly into the project's `src/` directory and can be modified at will. *Shadcn* components are styled with *TailwindCSS*, a CSS utility class framework [17]. *Tailwind* improves the consistency and development efficiency of component styling. Through *Shadcn* and *Tailwind*, *TempTing* was able to greatly improve the look and feel of its UI.

The template editor was implemented with the use of the *CodeMirror* library. *CodeMirror* is an extensible in-browser text-editing library [7]. *CodeMirror* facilitated the development of the linting, syntax highlighting, and autocompletion functionality. The integration of *CodeMirror* into *TempTing* is discussed further in section 4.3.

4.2.2 Web App Features. The editor, discussed in section 4.3, is embedded within a web app that aims to complement the editing experience. The template creation page offers several enhancements. Users may give their templates a title, description, language, and arguments. The supported languages are, currently, based on those

with Universal Dependency treebanks [18]. The editor frame exists within a dynamic split-screen view. The user can adjust the portion dedicated to the editor or the utilities panel. The utilities panel displays collapsible menus for diagnostics, and parser output.

The editor panel is overlaid with zoom and maximise buttons. The zoom buttons allow the user to change the font size in the editor, irrespective of their browser zoom-level. The maximise button moves the editor into an isolated floating panel so that editing may be done free from the distractions of a complete interface.

Once users have created an account and logged in, they can save and publish their templates. Templates can be saved as drafts, prior to publication, if they wish to revisit it later. Publishing a template makes it accessible to all other users for viewing, editing, are composition as a sub-template. Existing templates can be explored, filtered, and searched through the template browser. Users may create new language variations of existing templates from the template browser. The associations between template language variations indicate a common constructor. These associations have not yet been linked with a constructor implementation. It is enforced that associated templates have the same number of arguments to ensure future constructor compatibility. The name of a template and its arguments, however, can vary by language implementation.

4.3 Editor Implementation

4.3.1 Text Editor Features. The application’s core features are provided through a text-editing interface. The user is guided through the template creation process through the additional IDE-like features that the editor provides. Screenshots of the features are found in the appendix (fig. ??, ??, ??).

One prominent template-creation aid is the included linter. The linter is capable of detecting syntax errors and invalid identifier usage. The user is provided diagnostic information that describes their error as well as hints that suggest fixes. Problematic areas of the user’s template are given a red, wavy underline. This borrows from the spell-checker idiom that users are likely exposed to through word-processing software. The diagnostic descriptions are displayed to the user upon mouse hover. Additionally, the user can browse a list of errors in the resizable utility panel discussed in section 4.2.2. The linter provides feedback to the user that directs them towards writing valid templates.

The linter calls the parser once the user pauses typing for 200ms. In the event of parser failure, a detailed syntax error is produced based on the failure’s context. The AST produced through parser success is further validated by the linter. It is ensured that invocations match the arguments provided to the template. Additionally, roles are compared with the language-specific dependency-relations provided by *Universal Dependencies* [18]. Specifically, the revised v2 Universal Dependency relations were used. The parser is not dependent on this set, however, and thus the linter could accommodate alternate roles.

TempTing provides contextual autocompletion functionality in the editor. Within the editor window, it can complete argument interpolations and roles.

The template writing process is significantly streamlined due to the provided syntax highlighting. Each parsed symbol is displayed with a unique colour, depending on its context. This can be seen

in fig. 8. The primary benefit of syntax highlighting is improved template readability. Additionally, it gives immediate confirmation to a user that a template adheres to their intent.

4.3.2 Text Editor Library. CodeMirror, the text editor library, is highly configurable and offers support for custom language implementations. CodeMirror provides modular frontends for linting, syntax highlighting, and autocompletion.

Initially, TempTing achieved syntax highlighting with the custom parser. The parsed symbols were wrapped with `` tags and given classes dependent on their token type. Each CSS class was then given a colour. This approach was abandoned, however, as it did not integrate natively with CodeMirror. While CodeMirror does support custom parsers for syntax highlighting, it prioritises its own bespoke parser-generator. CodeMirror’s *Lezer* parser system is built specifically for syntax highlighting CFGs such as the template syntax [7]. The primary benefit of a custom parser was the context-aware diagnostics and thus, it was considered acceptable to use CodeMirror’s *Lezer* system for syntax highlighting. A grammar, found in the appendix, for the template syntax was written in *Lezer*’s notation. Admittedly, this approach is inelegant. The editor runs two independent parsers to achieve its required features. This inelegance was considered to be a necessary tradeoff improved integration with CodeMirror. This improved integration allows for the syntax highlighting to be exported as a CodeMirror plugin, which feature projects may utilise.

CodeMirror’s linting capabilities fit nicely with the custom parser’s design. CodeMirror’s linter is simply a function that is called, with a debounce, on editor change. Each time the editor changes, it’s contents is passed to the `parse_template` function (section 4.1). If any errors are returned, their positions ranges (start and end positions) are returned to the linter with the designated diagnostic messages.

4.4 Backend Implementation

Gleam, the language that the parser was implemented in, was also used for the backend. Gleam is capable of transpiling to Erlang as well as well Javascript [13]. Erlang transpilation allows the backend to run on the BEAM VM. The choice of backend language was, however, largely inconsequential. The backend is a simple REST API which receives HTTP requests, performs necessary CRUD operations, and returns an HTTP response. Gleam was chosen to open the possibility of parsing templates on the server.

SQLite was the chosen relational database. *SQLite* has many shortcomings; these include a minimal datatype set and relatively poor scaling [8]. Despite this, however, *SQLite* was chosen due to the database being contained entirely within an in-project binary. This simplifies development and deployment when compared with running a separate DB service. It is unlikely that TempTing will ever need to scale beyond *SQLite*’s capabilities. If necessary, the database can easily be migrated to a more featureful database in future.

4.5 Modular Implementation

Each component of the system - backend, frontend, and parser - exists within its own decoupled git repository. The parser is imported into the frontend as a `node_module`. If the parser were to

be needed on the backend, it could be imported via Gleam’s package manager. The parser includes an API wrapper that simplifies Javascript interoperability.

The template editing features developed in this project could be packaged as a CodeMirror extension. This would allow the TempTing features to be embedded in other web apps, such as Wikifunctions. Wikifunctions currently uses an older library, Ace, for its editing functionality. If Wikifunctions were to integrate the template syntax, it would either have to migrate to CodeMirror completely or make an exception for the template syntax.

Due to being a web app, TempTing is highly portable. Deployment and scaling would not be major hurdles. The parser, in its ability to target both the Javascript and Erlang runtimes, can be utilised within multiple ecosystems.

4.6 Dependency Labels Implementation

Universal Dependencies provides language-specific treebanks which are free to download and use [18]. Dependency Labels for the supported language were extracted from these treebanks. Most languages make use of a small subset of dependency labels. Thus, this extraction was performed to provide more accurate dependency label autocompletion and diagnostics.

Each treebank includes a `stats.xml` file in its route directory. The `stats.xml` file includes a list of dependency relations that were present in the language’s corpus. A simple Python script was written to extract the list of unique relations for each language. In the event that a language had multiple treebanks, the results were merged. These lists are used as the basis for role validation and autocomplete within the editor. Problematically, a significant number of these role labels include “:” to denote subtyping. This conflicts with template syntax specification; “:” is the separator between dependency labels and invocations within a slot. To avoid this collision, all “:” in subtyped roles were replaced with a “-”, an otherwise unused symbol in the syntax.

The dependency labels that were extracted are not stored in the database. Instead, the dependency labels are stored directly in hashmap literals within Javascript delivered to the client. While this does increase the frontend payload slightly, it is marginal when compared to the entire bundled application. This approach was followed to avoid additional network calls while the user edits a template. Linting latency was prioritised over an incremental difference in page-load times. It is imperative that the user receives immediate feedback when they produce an invalid or valid template. Relying on network requests and database communication risks displaying stale data to the user.

5 DISCUSSION AND RESULTS

5.1 Evaluation

To ensure TempTing met the requirements, an evaluation was performed. Separate evaluations were run to determine whether the system could adequately handle valid and invalid templates.

To evaluate valid templates, 1000 template syntax trees were generated. From these generated ASTs, raw templates were produced. These valid templates were then re-parsed and the outputs were compared with the original ASTs. This strategy ensured coverage over many possible template combinations.

To evaluate invalid templates, a minimally triggering template for each possible error case was written. These templates were written by hand to ensure expected errors were produced.

5.2 Comparison with Project Aims

TempTing aimed to solve the problems that currently affect Abstract Wikipedia’s templates: *Functionality* and *Accessibility*.

5.2.1 Functionality. The TempTing project resulted in the successful development of a custom parser. This parser validates templates and transforms them into a machine-readable AST. The output from this parser can be utilised within the realisation process of the NLG pipeline. In section 2.3.1, a template example was introduced. The functionality of TempTing’s parser supersedes that of the Scribuntu prototype [5]. TempTing’s parser supports multiple function arguments, nested function calls, and context-aware error messages.

TempTing successfully covers the prioritised functionality problems it aimed to solve. However, this does not cover all functionality required by Abstract Wikipedia. In particular, TempTing did not cover realisation. Future efforts, as discussed in section 5.4, will have to grapple with this.

5.2.2 Accessibility. The aforementioned parser confronts functionality problems directly. Additionally, TempTing leverages this parser as an indirect boon for accessibility. The parser, through integration with the web-based editor, is utilised for the provision of the text-editing features. Validation, linting, autocompletion, and syntax-highlighting give feedback to users in real-time. A caveat, explained in section 4.3.2, is that the syntax-highlighter was implemented with a separate parser.

This immediate feedback enables users to learn to “express themselves in the constrained language the system understands”, as Vrandečić’s Proposal necessitates [16].

5.3 Problems Encountered

There only exists one source from which information about the template syntax could be acquired [6]. This template specification includes some inconsistencies mentioned in section 3.1.2. The example lexical functions are shown to accept arguments that are not supported by the grammar specification. Due to this discrepancy, concessions had to be made regarding the parser’s implementation.

The introduction of a secondary parser for syntax highlighting was not ideal (section 4.3.2). There is a risk of misalignment between the two parsers. Such misalignment could lead to a situation whereby the syntax highlighting communicates differently from error messages, potentially confusing users. If a reader wishes to replicate this project, it is recommended that a compatibility layer be written that bridges the parser with CodeMirror’s syntax highlighter.

The choice to implement the parser in Gleam did improve its robustness. The type-system and pattern matching were of significant value. However, if the parser were to be hosted on Wikifunctions, the transpiled Javascript would need to be used. This could hurt future maintainability as the transpiled output is less readable than hand-written code.

5.4 Future Work

5.4.1 TempTing Developments. TempTing currently only offers support for the languages that have Universal Dependency treebanks. Some language groups, such as the Niger-Congo B languages, have limited inclusion. Future work may seek to broaden the sources used for language integration.

The user interface, despite the goal of language-agnostic contribution, is entirely in English. A future system would require the UI to synchronise with the supported template languages.

The syntax highlighting and linting integration could be packaged as a CodeMirror plugin. This would enable seamless embedding in other projects.

5.4.2 Integration with Abstract Wikipedia. Future work should prioritise deeper integration with the rest of the Abstract Wikipedia project. As originally envisioned, the parser, templates, and lexical functions could be integrated with Wikifunctions.

The two concurrent projects, mentioned in section 1.3, could be unified with TempTing. This would help coordinate constructors and templates.

While clear progress has been made regarding template functionality, there is still work to be done building the complete NLG pipeline. A realisation algorithm must be developed to utilise the current parser implementation.

6 CONCLUSION

This paper presented the process of designing and implementing TempTing, a template-editing software tool. TempTing was developed to address functionality and accessibility problems that affected Abstract Wikipedia’s templates. A parser for the template syntax was built and integrated into a web-based template development environment. The produced web app facilitated the management and editing of templates with real-time feedback.

TempTing design and requirements were influenced by previous contributions to Abstract Wikipedia, ensuring its relevance and applicability. The parser component shows promise for future integration within the Natural Language Generation pipeline, potentially serving as a precursor to the realisation phase. The implemented editing features streamline template creation and improve the accessibility of the project. These features include syntax highlighting, diagnostics, and autocompletion.

While TempTing successfully addresses the requirements set for the project, there are opportunities for future work. This should focus on broader language support and deeper integration with the rest of Abstract Wikipedia, including constructors, Wikifunctions, and the realisation.

TempTing represents a step forward in Abstract Wikipedia’s template capabilities. By improving the accessibility and functionality of template editing, the tool has potential to extend engagement Abstract Wikipedia and contribute to its vision of creating a truly multilingual Wikipedia.

REFERENCES

- [1] 2024. *Nearly*. <https://github.com/kach/nearly>
- [2] K. Arrieta, P.R. Fillottrani, and C.M. Keet. 2024. CoSMo: A multilingual modular language for Content Selection Modelling. *ACM/SIGAPP Symposium on Applied Computing (SAC ’24)* 39 (2024). <https://doi.org/10.1145/3605098.3635889>
- [3] T. Ball. 2020. *Writing an Interpreter in Go*. Germany. <https://interpreterbook.com/>
- [4] Facebook. 2024. *React*. <https://github.com/facebook/react>
- [5] A. Gutman. 2022. *Abstract Wikipedia/Template Language for Wikifunctions/Scribunto-based implementation*. https://meta.wikimedia.org/wiki/Abstract_Wikipedia/Template_Language_for_Wikifunctions/Scribunto-based_implementation
- [6] A. Gutman and C.M. Keet. 2024. *Abstract Wikipedia/Template Language for Wikifunctions*. https://meta.wikimedia.org/wiki/Abstract_Wikipedia/Template_Language_for_Wikifunctions#Example_templates
- [7] M. Haverbeke. 2024. *CodeMirror5 Github*. <https://github.com/codemirror/codemirror5>
- [8] R.D. Hipp. 2024. *SQLite*. sqlite.org
- [9] H. Johnston. 2024. *Shadcn-Svelte Github*. <https://github.com/huntabyte/shadcn-svelte>
- [10] L. Martinelli. 2023. *Wikifunctions FAQ*. <https://www.wikifunctions.org/wiki/Wikifunctions:FAQ>
- [11] Meno25. 2024. *List of Wikipedias*. https://meta.wikimedia.org/wiki/List_of_Wikipedias
- [12] Mahir Morshed. 2023. Using Wikidata Lexemes and Items to Generate Text from Abstract Representations. *Semantic Web* (2023). <https://www.semantic-web-journal.net/content/using-wikidata-lexemes-and-items-generate-text-abstract-representations-0>
- [13] Louis Pilfold. 2024. *Gleam Language*. <https://github.com/gleam-lang/gleam>
- [14] D. Vrandečić. 2020. Collaborating on the Sum of All Knowledge Across Languages. (10 2020). <https://doi.org/10.7551/mitpress/12366.003.0016> arXiv:https://direct.mit.edu/book/chapter-pdf/2247832/9780262360593_c001200.pdf
- [15] D. Vrandečić and M. Krötzsch. 2014. Wikidata: a free collaborative knowledge-base. *Commun. ACM* 57, 10 (sep 2014), 78–85. <https://doi.org/10.1145/2629489>
- [16] D. Vrandečić. 2020. Architecture for a multilingual Wikipedia. (2020). <https://doi.org/10.48550/arXiv.2004.04733> arXiv:2004.04733 [cs.CY]
- [17] A. Wathan. 2024. *Tailwind Github*. <https://github.com/tailwindlabs/tailwindcss>
- [18] D. Zeman. 2024. *Universal Dependencies Github*. <https://github.com/UniversalDependencies>

Table 1: Template Grammar [6]

Template	→ Element
Element	→ {Slot} Text Element Element
Text	→ lexeme punctuation string
Slot	→ DependencyLabel : Invocation Invocation
DependencyLabel	→ Label < SourceLabel Label root
Label	→ Role _ Index Role
Role	→ ...
Index	→ 1 2 ...
SourceLabel	→ Label
Invocation	→ FunctionInvocation Interpolation String
FunctionInvocation	→ F(ArgumentList) F()
ArgumentList	→ invocation invocation
F	→ functionName templateName
Interpolation	→ interpolation
String	→ "string"

TempTing

Templates

Help

Account

Template Title

template-in-container

Template Description

Optional

Template Language

English

Arguments

item

container

+

1 {expl:"There"}

2 {root:"is"}

3 {det<subj:Lexeme("a")}

4 {subj:item}

5 in the

6 {obj:container}

190%

No errors.

View Parser Output.

New template.

Save Draft

Publish Template

Figure 8: Template: item-in-container


```

@top template { Element* }

Element {"{" Slot "}" | text }

text {LexemeOrString | Punctuation }
Slot {DependencyLabel ":" invocation | invocation}
DependencyLabel {Label "<" Label | Label }
Label {Role "_" Index | Role}
Role {identifier}



```



Templates ▾

Help ▾

Account ▾

Template Title

Template Description

Template Language

Arguments

1 {invalid template}

Syntax Error:

Invocation: invalid_template

'.' is not a valid character for Invocation.

Invalid Invocation format:

Format must be in the form:

Function (eg. sub-template)

or

Interpolation of argument

or

String

TIP: '.' is prefixed before role indexes.

Separation of multi-word identifiers can be done with

camelCase (capitaliseEachWord)

or kebab-case (put-a-hyphen-between-each-word).

100%

Title Errors

'SPACE' is not allowed in template title.

Language Errors

Please select a language. Templates must be assigned a specific language.

Syntax Errors

New template.

Figure 9: TempTing Diagnostics

```

Index {index}
invocation {Function | Interpolation | String}
Function {Fname("Args") | Fname("")}
Fname {identifier}
Args{list<invocation>}
Interpolation{identifier}
LexemeOrString{lexOrStr}
Punctuation{punctuation}

@tokens {
  String { '"' char* '"' }
  char {![ " ]}
  whitespace { $[ \n\r\t ] }
  identifier {(?![_<:() \n\r\t"])+}
  index {@digit+}
  punctuation{
    $[.?!:;—()[\]'\/,|~@#%&*^_<>«»·¿¡,„`-]
  }
  lexOrStr{
    ![.?!:;—()[\]'\/,|~@#%&*^_<>«»·¿¡,„`{ \n\r\t-]+
  }
}

@skip { whitespace }

list<item> { item ("," item)* }

```

TempTing

Templates ▾ Help ▾ Account ▾

🔒

⚙️

Template Title

template-name

Template Description

Optional

Template Language

German

Arguments

example-for-completion ✕

example-2 ✕

example-3 ✕

+

1 {sub-template(exam)}

example-2

example-3

example-for-completion

↗️

⋮

🔍

160%

🔍

⚠️

Argument Errors

▾

New template.

Save Draft

Publish Template

Figure 10: Argument Autocompletion

{ }

TempTing

Templates ▾Help ▾Account ▾

Template Title

Template Description

Template Language

Arguments

example-for-completion X

example-2 X

example-3 X

+

1 {sub-template(exam)}

x example-2

x example-3

x example-for-completion

↻

160%

🔍

⚠️ Argument Errors ▾

New template.

Save Draft

Publish Template

Figure 11: Role Autocompletion