UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS Honours Project Final Paper 2024

Title: Implementing a Multilingual Backend for the CoSMo Language

Author: Jordy Kafwe

Project Abbreviation:

Supervisor(s): Maria Keet

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	20
Theoretical Analysis	*0*	*25*	0
Experiment Design and Execution	*0*	*20*	0
System Development and Implementation	*0*	*20*	20
Results, Findings and Conclusions	*10*	*20*	10
Aim Formulation and Background Work	*10*	*15*	10
Quality of Paper Writing and Presentation	*10*		10
Quality of Deliverables	*10*		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	*0*	*10*	
Total marks		**80**	

Implementing a Multilingual Backend for the CoSMo Language

Jordy Kafwe

University of Cape Town

Cape Town, South Africa

KFWJOR001@myuct.ac.za

Abstract

The development of a multilingual backend for the CoSMo language addresses the need for abstract representation in applications such as database view specification and natural language generation. This paper presents the creation of a backend system for CoSMo, focusing on parsing, query mapping, and data retrieval from Wikidata. The system is designed to support Abstract Wikipedia's multilingual goals by using CoSMo constructors, which are modular, language-independent representations of content. By implementing a query generation module that converts CoSMo syntax into SPARQL or SQL queries, the backend demonstrates platform independence and suggests related items to enhance content selection. The system's architecture, built as an HTTP API, ensures interoperability with other applications within the Abstract Wikipedia project. Key results include the successful parsing of CoSMo constructors and the generation of accurate queries. This work highlights the potential applications of CoSMo in creating a truly multilingual Wikipedia, overcoming language disparities in content availability.

1 Introduction

There currently exists large disparities in content availability and quality across Wikipedia's language editions [10]. For instance, while the English Wikipedia has over six million articles, many other language editions have significantly fewer articles, and the comprehensiveness of these articles can vary greatly [10]. This uneven distribution means that speakers of less-represented languages often have limited access to the breadth of knowledge available in more dominant languages. In addition, smaller language communities face the daunting task of creating and maintaining a comprehensive encyclopedia with limited resources and contributors.

Abstract Wikipedia [8] is a project which seeks to create a truly multilingual Wikipedia by generating articles from structured data stored in Wikidata and functions from Wikifunctions[2, 10]. This approach is designed to overcome the challenges posed by the vast number of languages that Wikipedia needs to cover to fulfill its vision of providing free access to the sum of all human knowledge.

To achieve its goal, Abstract Wikipedia will use Wikidata. Wikidata is an open, collaborative knowledge base [9]. Wikidata houses a large collection of labelled entities and the relationships they have with other entities. This data is stored as an RDF store accessible through a public SPARQL endpoint. Its goal is to provide a diverse collection of machine-readable knowledge that anyone may contribute to and benefit from [9]. 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.

CoSMo, a novel Content Selection Modelling language, which has recently been formalised, will help Abstract Wikipedia to this end [1]. CoSMo will help in selecting data housed in Wikidata as base, abstract representations of content. CoSMo is platform-independent, meaning it functions effectively regardless of the system in which the data is stored, whether it be RDF, XML, SQL, or others. In addition, CoSMo is natural language-independent, which supports Abstract Wikipedia's multilingual goals. The abstracted representations of content are given the term *constructors* [10]. 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. CoSMo is, thus, the modelling language for constructors.

There is, presently, no software implementation of CoSMo constructors. This paper gives particular concern to the current absence of content selection and Wikidata integration [10]. Without the functionality of content extraction from Wikidata, the constructors are unable to fulfill their intended purpose within Abstract Wikipedia.

In this paper, we show that we were able to build an initial backend system that realises the functionality of CoSMo. The system is available as an HTTP API that offers its functionality through various endpoints. Firstly, it parses the constructors. This involves accurately following CoSMo syntax to ensure that the content selection is correctly interpreted and validated. Secondly, it implements a query generation module that converts the CoSMo syntax into the appropriate SPARQL or SQL query. Although Wikidata currently uses SPARQL, having SQL query generation was implemented to demonstrate that CoSMo is in fact platform-independent. This query generation module also implements query expansion to suggest items related to the entities of interest.

To emphasise, the main objective of this project was to create a backend system that implements the functionality of CoSMo. This paper will demonstrate how we accomplished this objective and explain the reasoning behind our conclusions. We will begin by reviewing related works and providing background information. Next, we will outline the design process of the application and its implementation. Subsequently, we will describe the evaluation of each component of the project and present the results of these evaluations.

2 Background and Related Work

The CoSMo specification paper established that current content selection tools often require considerable technical expertise, hindering accessibility [1]. Existing tools, such as TexToData, offer some multilingual capabilities, but they rely on translation methods that are not inherently multilingual [1]. In addition, while graphical query builders like VSB and RDF Explorer facilitate query construction, they lack modularity and multilingual support[1]. Furthermore, they do not accept conceptual data models, which are necessary for a constructor implementation [1]. These limitations, therefore, made it necessary to develop a novel content selection tool for use in Abstract Wikipedia.

The development of CoSMo was driven by the recognition that existing modelling languages do not adequately support the unique requirements of multilingual content selection and representation [1]. The language is specifically tailored to function within the Abstract Wikipedia project, which aims to create a truly multilingual Wikipedia by generating articles from data stored in Wikidata [10]. This project necessitates a modeling language that can integrate with the diverse linguistic and functional needs of the Wikipedia ecosystem.

To achieve this, CoSMo incorporates several features that distinguish it from traditional modeling languages. It embeds multilinguality directly into the language, allowing users to declare content in their preferred natural language [1]. This is achieved through a modular approach where modules, referred to as 'constructors', can be defined at both the class and instance levels, enabling flexible and dynamic content selection. In addition, CoSMo supports the inclusion of both static content and functions, allowing for the computation of derived content, such as calculating a person's age from their date of birth [1].

The rigorous design process of CoSMo involved a comprehensive analysis of requirements, stakeholder consultations, and iterative refinements. This process ensured that the language not only meets the technical needs of content selection but also aligns with the collaborative and open nature of the Wikimedia community. CoSMo's specification includes a formal syntax and semantics, graphical notation, and a preliminary evaluation that demonstrates its effectiveness in real-world applications [1].

Ninai/Udiron is a notable attempt at implementing a natural language generation (NLG) system for Abstract Wikipedia [3]. However, Ninai currently only makes use of constructors on a surface level, which limits its flexibility in content selection and generation [3].

The NLG pipeline for Abstract Wikipedia involves several steps, including content selection, discourse planning, and linguistic realisation [7, 10]. CoSMo addresses the content selection stage by providing a language that allows users to specify what information should be verbalised from structured data. This is a critical step in the pipeline, as it determines the content that will be included in the generated text.

In summary, while existing tools and systems provide some functionality for content selection and NLG, they do not fully meet the requirements of the Abstract Wikipedia project. CoSMo fills this gap by offering a multilingual, modular, and user-friendly language for content selection, enabling the generation of articles in multiple languages from structured data.

3 Design

The design of the backend for the CoSMo language was informed by a comprehensive requirements analysis conducted by Keet et al. during the development of CoSMo [1]. This analysis identified several critical requirements that CoSMo must fulfill in order to support the Abstract Wikipedia project [1]. We designed the backend to fulfill as a many of these requirements as possible.

3.1 Requirements Analysis

Using the CoSMo paper [1] as a base, the requirements for this initial backend were then further refined in face-to-face meetings with the project supervisor. Through this process the following requirements were identified:

(1) The backend must support multilinguality, enabling the retrieval and display of labels in the user's preferred language [1]. In addition, the backend must be platform-independent, capable of functioning regardless of the underlying database technology.

(2) The backend should allow for the reuse and combination of constructors. [1]. It must also enable the specialisation and generalisation of constructors, including creating subconstructors and refining or generalising existing ones.

(3) Roles are integral to the CoSMo language [1], and the backend must support roles as first-class citizens. This includes the ability to name and manage roles within constructors [1]. To allow detailed and precise content selection, the backend should also support declarations at both the instance and type levels, allowing users to retrieve specific instances or types from Wikidata.

(4) The backend should support the specification of value constraints and mandatory participation for elements within constructors which ensures that the selected content meets specific criteria and requirements [1]. Facilitating join or merge operations between elements is necessary for combining similar or related items.

(5) The backend should be capable of being used by other applications within the Abstract Wikipedia project. This interoperability ensures that the CoSMo language can be effectively integrated into the broader ecosystem of Wikimedia tools and applications.

(6) The backend should also allow the discovery of related Wikidata data that may be of interest to the user based on their initial CoSMo query.

The requirements were then transformed into software objectives with certain tasks prioritised based on the project goals. Using an iterative waterfall method, we worked on achieving these objectives. The iterative waterfall method allowed a structured approach to completing tasks whilst allowing iteration within a phase, if needed.

Therefore we determined that we needed to develop a system that fulfills the following objectives:

(1) Processes Constructors.

(2) Produces queries.

(3) Interfaces with Wikidata.

(4) Provides access though an API Endpoint

3.2 System Components

The CoSMo backend system is structured using a layered architecture, a design approach that organises the system into distinct layers, each with a specific responsibility. This architecture is chosen for its ability to separate concerns, enhance modularity, and improve maintainability and scalability. Each layer focuses on a specific function. Layers interact sequentially and they communicate with adjacent layers.

The rationale for adopting a layered architecture is multifaceted. Separation of concerns allows focused development. Debugging becomes easier. Modularity is a key advantage as each layer can be developed, tested, and maintained independently. This modularity facilitates updates and integration with other systems, as changes in one layer do not necessarily impact others. The architecture supports scalability by enabling individual layers to be optimised or expanded as needed without affecting the entire system. This flexibility is crucial for accommodating future enhancements without disrupting existing functionality. Furthermore, the architecture ensures seamless interaction with external applications, particularly within the Abstract Wikipedia project, thereby enhancing interoperability.

Overall, the backend system is designed as an HTTP API to facilitate integration with other applications in the Abstract Wikipedia project. By receiving CoSMo statements and returning responses in JSON format, the system meets the requirement for interoperability with external applications. This design choice abstracts the internal workings of the backend from users, who only need to know the appropriate endpoint for sending their statements and the expected response format; users do not need to understand the underlying modules that comprise the backend. The layers of the CoSMo backend system, in the order they interact with each other, are: the HTTP API Layer, the Parser Layer, the Semantic Analysis Layer, the Mapping Layer, the Execution Layer, the Result Processing component, and finally back to the HTTP API Layer to return the response.

3.2.1 Parser Layer

This layer is responsible for interpreting CoSMo syntax. It uses ANTLR (Another Tool for Language Recognition) to generate a parser from a BNF (Backus-Naur Form) notation [4, 5]. ANTLR provides a robust framework for building language parsers [**?**], which are essential for converting CoSMo's high-level constructs into a form that can be processed by the backend system.

3.2.2 Semantic Analysis Layer

After the parser layer has parsed the constructors and determined they are syntactically correct, this layer will conduct a few basic semantic checks. First, that variables are not referenced before they have been declared. Secondly, that variables are not any CoSMo keywords.

3.2.3 Mapping Layer

Once the CoSMo syntax is confirmed to be semantically and syntatically correct, the mapping layer translates the parsed constructs into SPARQL or SQL queries. This involves mapping CoSMo's declarative elements to equivalent query constructs. The engine ensures that the multilingual and modular aspects of CoSMo are preserved in the query output.

3.2.4 Execution Layer

This layer handles the execution of the generated queries against the target database. It interfaces with RDF triple stores for SPARQL queries or relational databases for SQL queries, ensuring that the data retrieved is consistent with the specifications outlined in the CoSMo constructors.

3.2.5 Result Processing Layer

After query execution, the results are processed to fit the requirements of the natural language generation pipeline. This includes formatting the data and integrating it with other components of the system to produce coherent and contextually appropriate outputs.

3.2.6 Relational Database Design

The relational database design for storing Wikidata data is determined by the *wd2sql* tool we are using. This schema includes several key tables: a *meta* table that contains the English label and description for each entity, with columns for the entity ID, label, and description; and separate tables for different types of claim values, such as *string*, *entity*, *coordinates*, *quantity*, and *time*, where each table corresponds to a property's value type and includes columns for the subject entity ID and the property ID. In addition, *none* and *unknown* tables store claims with "no value" and "unknown value," respectively, identified by their entity and property IDs. Therefore, all property (P) and Wikidata entity (Q) values are stored in the *entity* table. Althpug

The *wd2sql* tool encodes Wikidata IDs, which consist of a type prefix ($Q/P/L$) followed by an integer, into a single integer format. That is, entity IDs are represented simply by the integer part of their ID, so Q42 becomes 42. Property IDs are adjusted by adding an offset of 1 billion to differentiate them. For example, P271 becomes 1000000271.

4 Implementation

4.1 Development Environment and Tools

The backend was developed using Python, with FastAPI [6]. serving as the framework for building the HTTP API. Python was chosen due to its simplicity, large ecosystem and currently being one of three languages supported by Wikifunctions [**?**]. FastAPI was selected due to its high performance and ease of use when building HTTP APIs. The parser layer was developed using ANTLR4 [5], which generated a Python parser and lexer for CoSMo. This ensured robust syntax interpretation for both shorthand and long form CoSMo language constructs. Docker was used to containerise

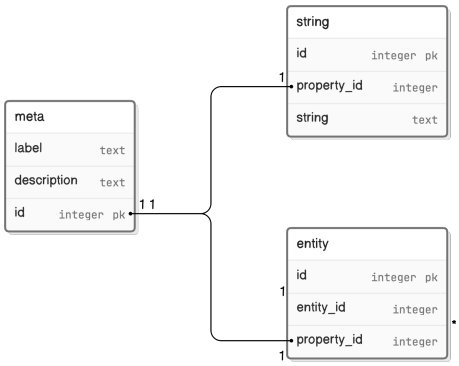


Figure 1: Entity Relationship Diagram of SQLite instance

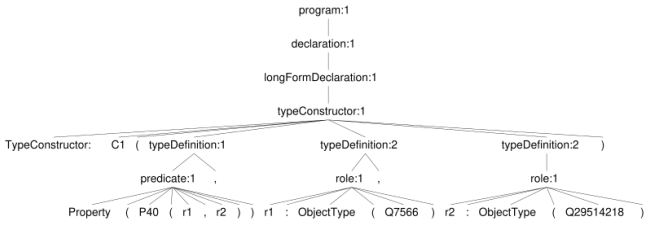
the application, providing a consistent environment across development and testing. For data storage and query execution, the publicly available Wikidata SPARQL endpoint was used for interfacing with the RDF triple store. A local SQLite instance, created by the wd2sql tool, was used as the relational database.

4.2 CoSMo Parser

ANTLR v4 [5] was chosen for its ability to generate native Python parser and lexer and that it separates the grammar from the actual logic, allowing the grammar to be reused in more contexts [4]. In our case, this means that we can use the exact same grammar for both SPARQL and SQL. A Python parser is beneficial because the Python FastAPI HTTP server can directly use the parser. The CoSMo parser to interpret language statements was designed according to the Backus-Naur Form (BNF) written by Keet et al [1]. The BNF was converted into the ANTLR eBNF grammar language. This eBNF is what ANTLR used to generate the Python parser. A few lexical rules, shown in Listing 2, were also added to the initial BNF from [1] to allow the syntax (or parser) rules to work on actual statements. The parser was implemented to work with both shorthand and long form notation of the CoSMo language. In order to meet the multilingual goal, the parser uses the pivot with CSMxxx identifiers [1]. This allows the compiler to work correctly, even as more natural languages are supported by CoSMo. CSMxxx identifiers are used to map each CoSMo language construct that would appear in the natural language of the constructor.

```
PItem: 'P' DIGIT+;
QItem : 'Q' DIGIT+;
ZItem : 'Z' DIGIT+;
VARIABLE : [a-zA-Z][a-zA-Z0-9]*;
DIGIT : [0-9];
COMMA : ',';
NUMBER : [0-9]+ ('.' [0-9]+)?;
STRING : '"' .*? '"';
WS : [ \t\r\n]+ -> skip;
```

Figure 2: CoSMo Lexer rules




```
TypeConstructor:C1(
Property(P40(r1,r2)),
r1:ObjectType(Q7566),
r2:ObjectType(Q29514218)
```

Figure 3: Example CoSMo Parse Tree

4.3 Mapping Layer

The mapping layer uses the parse tree (as shown in figure 3) created by ANTLR parser after successfully interpreting a statement. The ANTLR Visitor pattern is then used to walk the tree, progressively generating the appropriate SQL or SPARQL query. The visitor pattern is unguided tree traversal as ANTLR performs a depth first search, ultimately visiting every node in the tree. To do this a class that inherits from ANTLR's ParseTreeVisitor is generated by the library. In this class, there is a method associated with every CoSMo language type defined in the eBNF. We completed these methods to adhere to our mapping algorithm. Each construct and variable in the CoSMo statement is directly available in code according to the labelx it was given in the ANTLR eBNF grammar.

4.4 Execution Layer

SQLModel is used as the object relational mapper (ORM) for SQLite. The ORM, in combination with FastAPI's 'Depends()' method, allows us to perform dependency injection with the database. This results in cleaner, more easily testable code. The SPARQL queries are made using the SPARQLWrapper library, which provides a simple interface to query the Wikidata public SPARQL endpoint. This approach abstracts the details of making HTTP requests and parsing responses, resulting in more streamlined and maintainable code.

4.5 HTTP API Layer

Two main endpoints were implemented. /sparql/query using the public Wikidata SPARQL endpoint and /sql/query using the local SQLite instance. The Both these endpoints are HTTP POST endpoints and accept the the CoSMo syntax in the payload which is then decoded and passed to the parser.

4.6 Algorithms

4.6.1 Mapping

The algorithm for mapping CoSMo constructs to SPARQL queries efficiently parses CoSMo statements while tracking the usage of type constructors by instance constructors and ensuring that each triple is associated with its corresponding constructor. The process begins with initialising a visitor class, 'CoSMoToSPARQLVisitor',

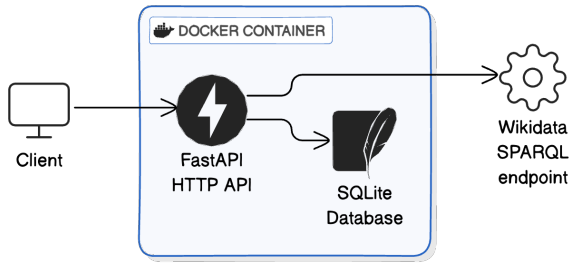


Figure 4: High level system diagram

which includes data structures: a list for 'Triple' objects, a dictionary for CoSMo variable mappings, a dictionary for type constraints, and a set for tracking type constructor usage.

As the visitor traverses the parse tree, it processes CoSMo constructs like type constructors, instance constructors, and properties. The focus is on tracking information, not immediate SPARQL mapping. When visiting type constructors, the visitor records expected object types in the 'subject_type_constraints' dictionary. The dictionary has a constructor variable name as a key and then the value itself is another dictionary. In the sub-dictionary, the key is a role or variable identifier, and the value is the expected object type. This ensures subjects in the SPARQL query match specified types. The visitor also tracks type constructor usage. For instance constructors, it checks role conformity and updates the set of referenced type constructors.

The algorithm maintains a set, 'used_type_constructors', to track referenced type constructors. Each time an instance constructor references a type constructor, its identifier is added to this set. RDF triples are constructed by identifying the subject, predicate, and object from the CoSMo statement. These are stored as instances of the 'Triple' class, which includes fields for the subject, predicate, object, and constructor association. This class encapsulates necessary information for SPARQL queries.

The 'getSPARQLQuery' method assembles the SPARQL query after processing all constructs. It iterates over stored triples, constructing the 'SELECT' and 'WHERE' clauses. Type constraints are added using 'wdt:P31', based on the 'subject_type_constraints' dictionary. This ensures results match specified types. After processing, the algorithm checks for unused type constructors. If a type constructor is not used to create an instance-level constructor, the algorithm generates a SPARQL query to fetch all objects that match that type constructor. Conversely, if a type constructor is used to create an instance-level constructor, the query is tailored to fetch only those objects that match the specific instance constructor. This distinction ensures that the query results are accurate and relevant, reflecting the intended scope of the CoSMo constructs. The algorithm finalises the SPARQL query, ensuring all patterns and constraints are included. The pseudo code for this algorithm is depicted in Algorithm 1.

The above was simplified to only explain the more general case – to convery the big idea. In the actual implementation, we also keep track of Joins, IsMandatory and SubConstructor. Join and SubConstructor are both dictionaries where the key is one part of the relationship and the value is the other. IsMandatory is simply a set of all the mandatory variables.

```
SELECT ?entityId _subject_ ?label
WHERE {
    // triples generated going here
    // followed by type checking of subjects here

  BIND(STRAFTER
  (STR(?r1), "http://www.wikidata.org/entity/")
  AS ?entityId)

  SERVICE wikibase:label {
   bd:serviceParam wikibase:language "[AUTO_LANGUAGE],
    _language_" .
    _subject_ rdfs:label ?label .
  }
}
```

Figure 5: Base query used to fetch data specificed in constructors

Algorithm 1
CoSMo to SPARQL Mapping Algorithm (Partial)

initialize *triples* ← list of Triple objects
initialize *subject_type_constraints* ← dictionary of CoSMo variable mappings
initialize *used_type_constructors* ← set of type constructors
initialize *sparql_queries* ← empty dictionary
function getSPARQLQueries
  for all *triple* **in** *triples* **do**
    subject ← getSubject(*triple*)
    constructor ← getConstructor(*triple*)
    if *constructor* ∉ *sparql_queries* **then**
      sparql_queries[*constructor*] ← "SELECT * WHERE {\n"
    end if
    sparql_queries[*constructor*] ← *sparql_queries*[*constructor*]+ " " + str(*triple*) + "\n"
  end for
  for all *subject*, *expected_type* **in** *subject_type_constraints.items*() **do**
    constructor ← getConstructorFromType(*expected_type*)
    sparql_queries[*constructor*] ← *sparql_queries*[*constructor*]+ " ?" + *subject* + " p:P31 ?" + *expected_type* + " .\n"
  end for
  return *sparql_queries*
end function

4.6.2 Semantic Analysis

The semantic analysis algorithm for this language is designed to ensure that all variables are declared before they are used and that each variable is declared uniquely. The process begins by initialising a set, *DeclaredVariables*, which will store the names of all declared variables. The use of a set is advantageous due to its constant time, or O(1), lookup performance, and its inherent property of disallowing duplicate entries. This ensures efficient management of variable declarations and prevents the same variable from being declared multiple times for different purposes.

As the algorithm traverses the parse tree generated by ANTLR, it identifies nodes that represent variable declarations and usages, thanks to the explicit labeling in the grammar. When a variable declaration node is encountered, the algorithm extracts the variable name and checks if it is already in the *declared_variables* set. If not, the variable is added to the set, marking it as declared. This step ensures that each variable is declared only once. Conversely, when a variable usage node is encountered, the algorithm checks if the variable is present in the *declared_variables*. If the variable is not found, a semantic error is reported, indicating that the variable has been used without prior declaration. The pseudo-code for this algorithm is depicted in Algorithm 2.

Algorithm 2
Semantic Analysis for Variable Declaration and Usage

```
initialize declaredVariables ← {}                    ▷ An empty set
function declareVariable(varName)
    if not varName ∈ declaredVariables then
        add varName to declaredVariables
    else
        print "Semantic Error: Variable '" + varName + "' already
declared"
    end if
end function
function isVariableDeclared(varName)
    return varName ∈ declaredVariables
end function
function traverseParseTree(node)
    if node is a variable declaration node then
        varName ← getVariableName(node)
        declareVariable(varName)
    end if
    if node is a variable usage node then
        varName ← getVariableName(node)
        if not isVariableDeclared(varName) then
            print "Semantic Error: Undeclared variable '" +
varName + "' at line " + getLineNumber(node)
        end if
    end if
    for all childNode in children(node) do
        traverseParseTree(childNode)
    end for
end function
```

This approach efficiently ensures that the program adheres to the language's semantic rules regarding variable declarations and uniqueness, leveraging the properties of the set data structure for optimal performance.

4.6.3 Query expansion

The query expansion algorithm fetches all nodes directly connected to the entity of interest. The SPARQL query used for this is shown in figure 6. The same process is followed for the SQL database, but without ordering by sitelinks as those are not available. We then order the results by the number of sitelinks because a higher number of sitelinks suggests that a piece of information is widely used across Wikimedia projects. This widespread usage implies that the information is likely highly relevant to the entity in question. By prioritising data with more sitelinks, the algorithm helps users discover on the most significant and commonly referenced attributes of the entity, providing a more relevant overview of its characteristics and associations.

```
SELECT ?property ?propertyLabel
       ?value ?valueLabel ?sitelinks
WHERE
{
  # Replace Qxxx with the Wikidata ID of interest
  wd:Qxxx ?property ?value .

  # Get the property label
  ?prop wikibase:directClaim ?property ;
        rdfs:label ?propertyLabel .
  FILTER(LANG(?propertyLabel) = "_language_")

  # Get the number of sitelinks for the value
  OPTIONAL { ?value wikibase:sitelinks ?sitelinks }

  # Service to add labels and descriptions
  SERVICE wikibase:label {
    bd:serviceParam wikibase:language "_language_,
    [AUTO_LANGUAGE]".
    ?value rdfs:label ?valueLabel .
    ?prop rdfs:label ?propertyLabel .
  }
}
ORDER BY DESC(?sitelinks)
```

Figure 6: SPARQL Query used for Query Expansion

5 Testing

The testing phase of this project encompassed a comprehensive approach to validate the backend system, language parsing, and API components. A combination of unit testing, integration testing, functional testing methodologies was used to ensure the robustness and reliability of the system. Unit tests were developed to verify the correct functionality of individual components in isolation. These tests focused on validating specific functions, methods, and classes within each module.

We used the constructors given through the CoSMo [1] paper to test the parser and overall system. Th

After development was completed, we conducted manual integration tests to assess the interaction between different components of the system. These tests ensured that the parser, other backend logic, and API worked cohesively. Functional testing validated that the system met its specified requirements and produced expected outputs for given inputs.

The testing process used several tools and frameworks to streamline and automate the validation procedures. Pytest, a popular Python testing framework, was used for writing and executing unit tests. HTTPie was used for manual integration testing by allowing the creation and automation of API requests.

The combination of these testing methodologies, tools, and frameworks resulted in a thorough validation of the system. It allowed for the identification and resolution of issues at various levels of the application stack. This comprehensive testing approach significantly contributed to the overall quality and reliability of the final product.

6 Results and Discussion

6.1 Results

All promised functionality was achieved with some caveats. Wikifunctions could not be integrated with as it is not yet functional [9]. Although we demonstrated that CoSMo is platform-independent by also integrating with a SQL database. The SQL database was working on a much simpler schema with no qualifiers and also only in English.

The backend's layered architecture enhances modularity, facilitating easy maintenance and future expansion. This was shown by the SQL addition made to the originally SPARQL only system.

As can be seen in figure 7, the parser is able to parse CoSMo constructors with 0 ambiguities in milisecond speeds.

6.2 Discussion

The development of CoSMo presented several challenges. One significant challenge was ensuring that the constructors accurately reflected the data representation in Wikidata. For instance, when designing a constructor to fetch locations where capybaras are endemic, it was discovered that the initial assumptions about data structure did not match Wikidata's representation. Locations were categorised as instances of "country" rather than a general "region" or "location" QID. This discrepancy highlighted the importance of understanding and adapting to the actual data structures in Wikidata to ensure the effectiveness of CoSMo.

Another challenge was the absence of certain data entries in Wikidata, which are crucial for constructing accurate content selection models. A specific example was the attempt to create a constructor for Edith Eger's parentage, only to find that her parents were not listed in Wikidata. Such data gaps pose significant obstacles, as they prevent the creation of complete and accurate constructors. Although, this could be fixed by using IsMandatory() notation [1] which requires the data to exist for the constructor to fetch data, data not being present may hinder non-expert users.

Rule	Invocations	Time (ms)	Total k	Max k	Ambiguities	DFA cache miss
program:0	5	0.0085	5	1	0	5
declaration:1	5	0.027161	5	1	0	4
longFormDeclaration:2	5	0.005072	5	1	0	4
shortFormDeclaration:3	0	0.0	0	0	0	0
typeConstructor:4	7	0.005932	7	1	0	3
typeConstructor:5	2	0.011178	2	1	0	2
instanceConstructor:6	4	0.004683	4	1	0	3
instanceConstructor:7	1	0.006707	1	1	0	1
instanceOf:8	1	0.007248	1	1	0	1
subConstructorOf:9	1	0.00655	1	1	0	1
typeDefinition:10	2	0.00783	2	1	0	1
typeDefinition:11	4	0.005197	4	1	0	2
typeDefinition:12	1	0.004682	1	1	0	1
typeDefinition:13	0	0.0	0	0	0	0
typeDefinition:14	0	0.0	0	0	0	0
typeDefinition:15	0	0.0	0	0	0	0
typeDefinition:16	7	0.005232	7	1	0	3
instDefinition:17	1	0.005313	1	1	0	1
instDefinition:18	2	0.004539	2	1	0	1
instDefinition:19	1	0.003997	1	1	0	1
instDefinition:20	0	0.0	0	0	0	0
instDefinition:21	0	0.0	0	0	0	0
instDefinition:22	4	0.004506	4	1	0	3
role:23	6	0.001521	6	1	0	1
role:24	6	0.015869	6	1	0	2
function:25	1	0.002004	1	1	0	1
join:26	0	0.0	0	0	0	0
shortTypeConstructor:27	0	0.0	0	0	0	0
shortInstanceConstructor:28	0	0.0	0	0	0	0
shortTypeDefinition:29	0	0.0	0	0	0	0
shortInstDefinition:30	0	0.0	0	0	0	0
shortRole:31	0	0.0	0	0	0	0
shortRole:32	0	0.0	0	0	0	0
shortFunction:33	0	0.0	0	0	0	0
shortJoin:34	0	0.0	0	0	0	0
argumentList:35	1	0.002245	1	1	0	1

Figure 7: ANTLR Profiler with Figure 8 as input

In addition, the compatibility of CoSMo's formal syntax with different parser generators posed a technical challenge. Initially, the CoSMo BNF specification did not work with the lightweight Lark parser generator[4], which is often preferred for its simplicity, due it raising to many ambiguities. However, it was successfully implemented using ANTLR [5], a more robust parser generator capable of handling complex grammar specifications. While ANTLR provided the necessary support for CoSMo's complex BNF, it also required a greater investment in learning due to its complexity compared to Lark. This means that making changes to the existing code may require more time and effort if other maintainers were needed to oversee such a system, as ANTLR is generally more complicated to use.

The current system was completely developed and tested on a local machine. Although, it met all the requirements, this is not the same environment that it would encounter if it is was to be integrated into the Abstract Wikipedia project. This current system also did not integrate with Wikifunctions as it was not yet.

Overall, this work lays a solid foundation for achieving Abstract Wikipedia's multilingual goals, contributing to the vision of providing free access to the sum of all human knowledge across languages.

7 Conclusion and Future Work

In conclusion, the development of a multilingual backend system for the CoSMo language marks a significant advancement in tackling the challenges of content selection and representation within the Abstract Wikipedia project. The system effectively implements CoSMo's core functionalities, such as parsing, query generation, and data retrieval, demonstrating its platform independence and multilingual capabilities. Although integration with Wikifunctions is pending due to its current unavailability, the backend's ability to interface with both SPARQL and SQL databases highlights its versatility. The project underscores the necessity of understanding Wikidata's data structures for accurate constructor creation and addresses the challenges posed by data gaps. The backend's layered architecture enhanced modularity, facilitating easy maintenance and future expansion. Future efforts should focus on integrating with Wikifunctions, addressing Wikidata's data gaps, and deploying the system in a production environment to ensure successful integration into the Abstract Wikipedia project.

```
TypeConstructor:C1(
Property(P40(r1,r2)),
r1:ObjectType(Q7566),
r2:ObjectType(Q29514218))

InstanceOf(C2, C1)

InstanceConstructor:C2(
Property(P40(r1,r2)),
r1:ObjectType(Q7566),
r2:ObjectType(Q29514218),
ObjectType(Q29514218)={Q62070381})

SubConstructorOf(C3, C1)

TypeConstructor:C3(
Property(P40(r1,r2)),
r1:ObjectType(Q7566),
r2:ObjectType(Q29514218),
Function(Z12345(Q29514218)))
```

Figure 8: Profiled constructors used in figure 7

References

[1] 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

[2] L. Martinelli. 2023. *Wikifunctions FAQ.* https://www.wikifunctions.org/wiki/Wikifunctions:FAQ

[3] M. 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

[4] Francisco Ortin, Jose Quiroga, Oscar Rodriguez-Prieto, and Miguel Garcia. 2022. An empirical evaluation of Lex/Yacc and ANTLR parser generation tools. *Plos one* 17, 3 (2022), e0264326.

[5] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf, 17–20.

[6] Sebastián Ramírez. [n. d.]. *FastAPI.* https://github.com/fastapi/fastapi

[7] EHUD REITER and ROBERT DALE. 1997. Building applied natural language generation systems. *Natural Language Engineering* 3, 1 (1997), 57–87. https://doi.org/10.1017/S1351324997001502

[8] D. Vrandecic. 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

[9] D. Vrandečić and M. Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (sep 2014), 78–85. https://doi.org/10.1145/2629489

[10] D. Vrandečić. 2020. Architecture for a multilingual Wikipedia. (2020). https://doi.org/10.48550/arXiv.2004.04733 arXiv:2004.04733 [cs.CY]