

# Bachelorarbeit

Titel der Arbeit // Title of Thesis

**Qlue-ls, a powerful SPARQL language server**

---

Akademischer Abschlussgrad: Grad, Fachrichtung (Abkürzung) // Degree

**Bachelor of Science (B.Sc.)**

---

Autorenname, Geburtsort // Name, Place of Birth

**Ioannis Nezis, Breisach am Rhein**

---

Studiengang // Course of Study

**Informatik**

---

Fachbereich // Department

**Technische Fakultät**

---

Erstprüferin/Erstprüfer // First Examiner

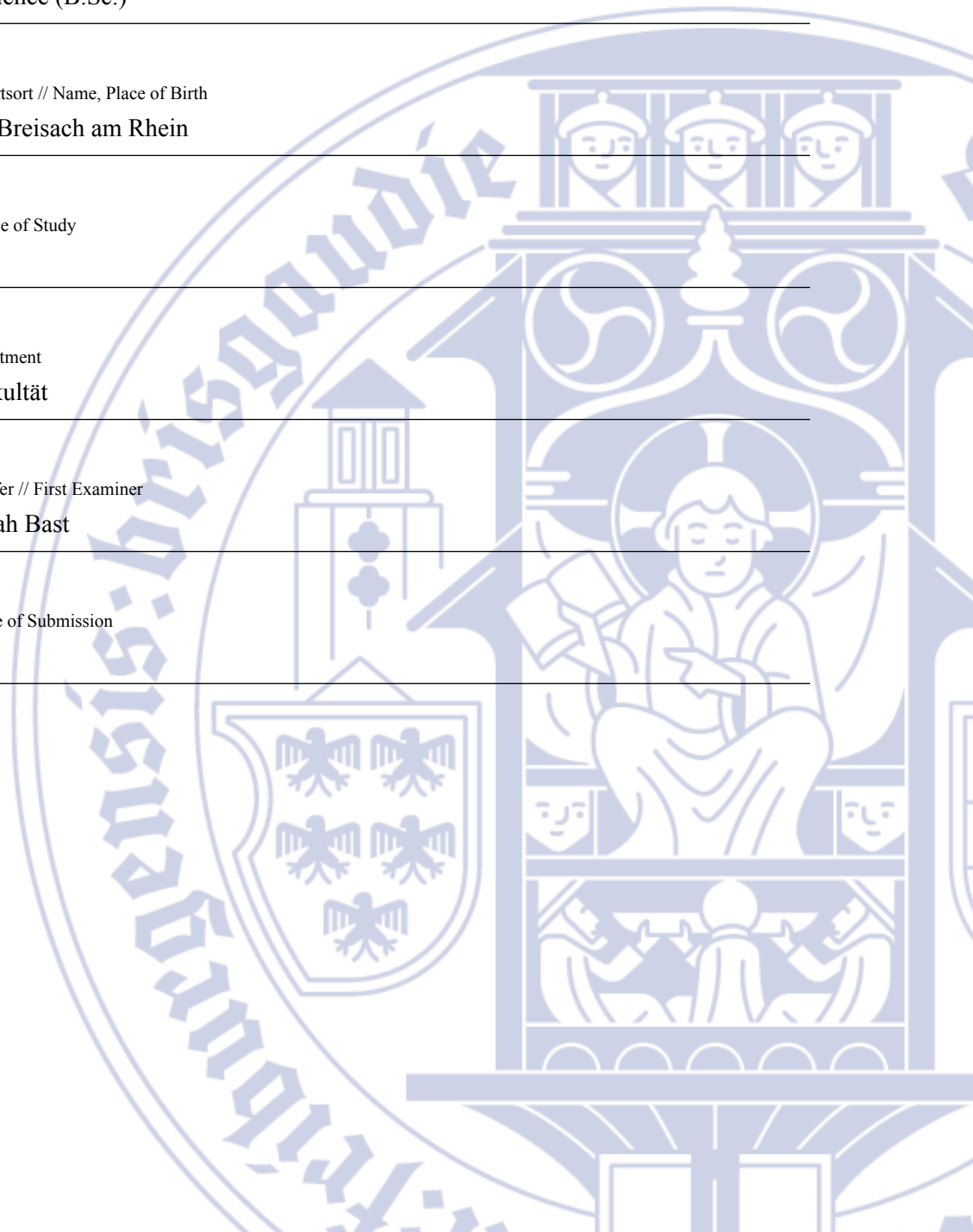
**Prof. Dr. Hannah Bast**

---

Abgabedatum // Date of Submission

**26.09.2025**

---



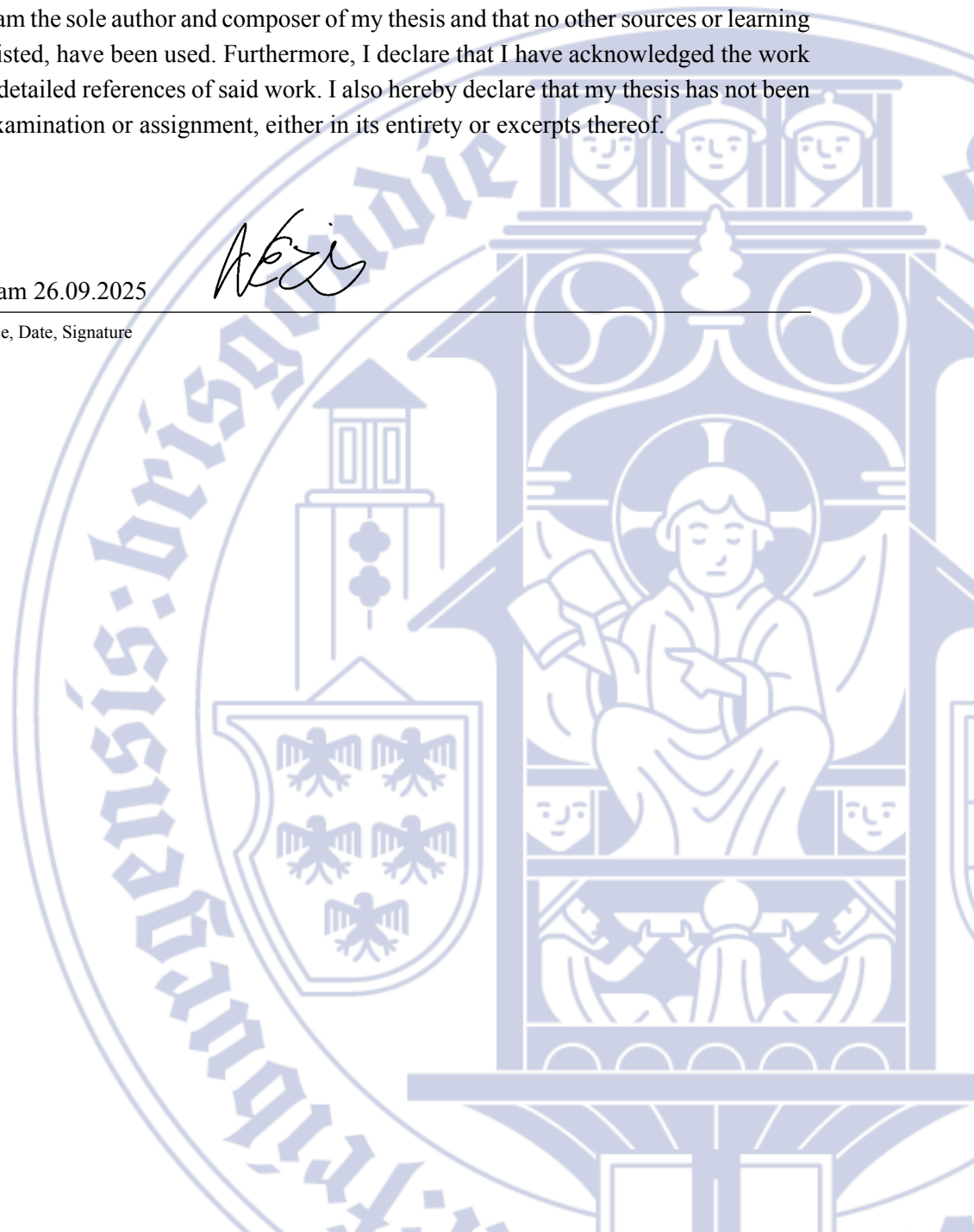
## DECLARATION

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

Freiburg im Breisgau, am 26.09.2025

Ort, Datum, Unterschrift // Place, Date, Signature

*Heiz*



## Abstract

The Resource Description Framework (RDF) formalizes a way to describe Information in a graph-like structure. *SPARQL* is the standard query language for interacting with RDF data, but writing queries can be challenging, especially without auto completion support. In this work I present Qlue-ls, a *SPARQL* language server. Powered by a parser that is able to parse incomplete queries, it's able to provide support when writing *SPARQL*. By using the language server protocol *Qlue-ls* can be used by any editor that provides language server support.

## Contents

1	Introduction	1
1.1	Problem definition .....	1
1.2	Contributions .....	3
2	Background	3
2.1	Resource Description Framework .....	3
2.2	SPARQL .....	4
2.3	Language Server Protocol .....	6
3	Related work	8
3.1	Completion .....	8
3.2	Formatting .....	11
3.3	Diagnostics .....	11
3.4	Hover .....	11
4	Architecture	12
5	Parser	14
5.1	Grammar .....	15
5.2	Extended Backus-Naur Form .....	18
5.3	SPARQL grammar .....	19
5.4	Lexical Analysis .....	19
5.5	Parsing Algorithm .....	20
6	Capabilities	29
6.1	Completion .....	29
6.2	Formatting .....	37
6.3	Diagnostics .....	39
6.4	Code Actions .....	44
6.5	Hover .....	47
7	Theoretical Analysis	49
7.1	Parsing Algorithm .....	49
8	Empirical Analysis	50
8.1	Completion .....	51
8.2	Formatting .....	54
8.3	Diagnostics .....	55
8.4	Code Actions .....	55

---

8.5 Hover .....	55
9 Acknowledgements	55
10 Appendix	56
10.1 Completion categories .....	56

# 1 Introduction

Modern developing environments have transformed software development by offering intelligent assistance while writing code. Features like automatic code formatting, error detection or completion suggestions allow developers to iterate faster and work more efficiently. These features are often provided by language servers – programs that offer language specific capabilities.

For many popular programming languages (like python, C++, Rust, Java, JavaScript, ...), language support is very extensive. Domain specific languages (DSL) often lack modern tooling. This makes them harder to use and harder to learn. One such language is *SPARQL*.

*SPARQL*[1] is a query language standardized by the [W3C](#). It is designed to retrieve and manipulate data stored in Resource Description Framework (RDF)[2] format. It plays a key role in the semantic web.

Despite it's importance in this field, *SPARQL* lacks proper tooling. Specialized editors that provide some support, mainly completion, exist. But there is currently no mature language support available for *SPARQL*.

This thesis presents **Qlue-Is**, a language server for *SPARQL* that brings modern language support.

## 1.1 Problem definition

This thesis addresses the problem of providing *language support* for *SPARQL*.

The term ‘*language support*’ is vague. It refers to a range of features that assist with development in a specific formal language. I will refer to these features as **capabilities**.

To make things concrete, I define a fixed set of capabilities that count as ‘*language support*’ in this thesis.

### 1.1.1 Completion

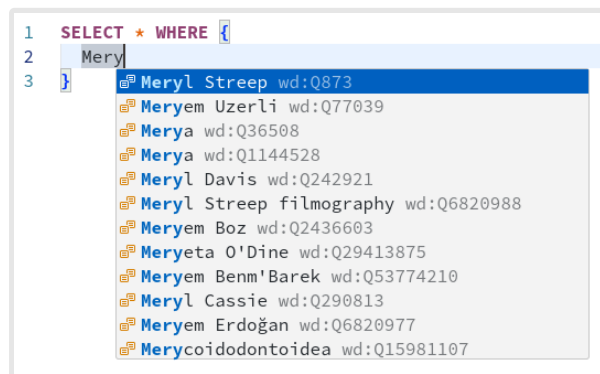


Figure 1: Completion suggestions after typing “Mery”, in the Wikidata dataset.



Figure 2: Editor state after accepting the first suggestion. The suggestion had the label “Meryl Streep” but the wd:Q873 was inserted. wd:Q873 is the resource identifier for the actress “Meryl Streep in the wikidata dataset.”

Given a cursor position within a *SPARQL*, the *completion* capability returns a set of suggestions. Suggestions are Variables, IRIs, Literals, or snippets of *SPARQL* that could validly follow at the cursor position.

### 1.1.2 Formatting

```

1 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2 PREFIX wdt: <http://www.wikidata.org/prop/direct/>
3 PREFIX pq: <http://www.wikidata.org/prop/qualifier/>
4 PREFIX ps: <http://www.wikidata.org/prop/statement/>
5 PREFIX p: <http://www.wikidata.org/prop/>
6 PREFIX wd: <http://www.wikidata.org/entity/>
7 SELECT ?film_id ?film ?award WHERE {
8   wd:Q873 p:P166 ?p166 .
9   ?p166 ps:P166 ?award_id ; pq:P1686 ?film_id .
10  ?award_id wdt:P31 wd:Q19020 ;
11  | rdfs:label ?award . FILTER (LANG(?award) = "en")
12  ?film_id rdfs:label ?film . FILTER (LANG(?film) = "en")
13 }

```

Figure 3: Unformatted SPARQL query.

```

1 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2 PREFIX wdt: <http://www.wikidata.org/prop/direct/>
3 PREFIX pq: <http://www.wikidata.org/prop/qualifier/>
4 PREFIX ps: <http://www.wikidata.org/prop/statement/>
5 PREFIX p: <http://www.wikidata.org/prop/>
6 PREFIX wd: <http://www.wikidata.org/entity/>
7 SELECT ?film_id ?film ?award WHERE {
8   wd:Q873 p:P166 ?p166 .
9   ?p166 ps:P166 ?award_id ;
10   | pq:P1686 ?film_id .
11   ?award_id wdt:P31 wd:Q19020 ;
12   | rdfs:label ?award . FILTER (LANG(?award) = "en")
13   ?film_id rdfs:label ?film . FILTER (LANG(?film) = "en")
14 }

```

Figure 4: Formatted SPARQL query.

The *formatting* capability transforms a SPARQL query into a standard form. Only whitespace is added or removed, the structure and meaning of the query stays the same. The standard form is defined by a configuration and the formatting algorithm.

### 1.1.3 Diagnostics

```

1 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2 PREFIX wdt: <http://www.wikidata.org/prop/direct/>
3 PREFIX pq: <http://www.wikidata.org/prop/qualifier/>
4 PREFIX ps: <http://www.wikidata.org/prop/statement/>
5 PREFIX p: <http://www.wikidata.org/prop/>
6 SELECT ?film_id ?film ?award WHERE {
7   wd:Q873 p:P166 ?p166 .
8   ?p166 ps:P166 ?award_id ;
9   | pq:P1686 ?film_id .
10  ?award_id wdt:P31 wd:Q19020 ;
11  | rdfs:label ?award . FILTER (LANG(?award) = "en")
12  ?film_id rdfs:label ?film . FILTER (LANG(?film) = "en")
13 }

```

Diagnostic message: 'wd' is used here, but was never declared  
 qlue-ls(undeclared-prefix)  
 Academy Awards  
[View Problem \(Alt+F8\)](#) [Quick Fix... \(Ctrl+.\)](#)

Figure 5: In this SPARQL query the PREFIX wd is used. But since this PREFIX was never declared in the PROLOGUE this will cause an error. The diagnostic message is showing this information to the user and the error is underlined in the editor.

Given a SPARQL query, the *diagnostics* capability reports issues in the query, each tied to a specific range in the document. Each issue has one of four severity levels: Error, Warning, Info, or Hint.

### 1.1.4 Code Actions

```

1 SELECT * WHERE {
2   ?s ?p ?o
3 }

```

More Actions...

- Add to result
- Add Lang-Filter
- Add Filter

Figure 6: SPARQL query with action menu

```

1 SELECT * WHERE {
2   ?s ?p ?o FILTER (LANG(?o) = "en")
3 }

```

Figure 7: SPARQL query with language filter

The *code action* capability suggests edits to improve or modify the query. These edits can be simple or complex. Code actions typically automate common tasks — for example, adding a language filter.

### 1.1.5 Hover



Figure 8: Hover information for **FILTER**

Given a *SPARQL* query and a cursor position, the *hover* capability shows textual information about the element under the cursor.

## 1.2 Contributions

I consider the following as my main contributions:

- I implemented a *SPARQL* parser. When the input is incomplete or incorrect, the parser still returns a partial parse tree for the longest possible prefix that is valid.
- I implemented the 5 capabilities defined above.
- I implemented a fully functioning *SPARQL* language server called **Qlue-ls**.
- I published Qlue-ls publicly available, open source project on [GitHub](#).

The remainder of this thesis is structured as follows.

Chapter 2 provides background on RDF, *SPARQL*, the language server protocol. Chapter 3 lists existing notable *SPARQL* editors and *SPARQL* tooling in general. Chapter 4 describes the structure of the implementation. Chapter 5 describes the parser and how it works, and the implemented capabilities in detail. Chapter 6 describes the individual capabilities and how they are implemented. Chapter 7 evaluates the parsers theoretical time complexity. Finally Chapter 8 evaluates the systems empirical performance.

## 2 Background

This section provides the background knowledge required to understand the concepts in this thesis.

### 2.1 Resource Description Framework

The Resource Description Framework (RDF)<sup>1</sup> is a data model, standardized by the W3C.

RDF data is a set of triples.

A triple consists of (1) a subject, (2) a predicate and (3) an object.

In Listing 1 there are 6 triples separated by dots.<sup>2</sup>

<sup>1</sup>To be exact RDF 1.1.

<sup>2</sup>This format is called *n-triples*.



```

1 <freiburg> <type> <city> .
2 <freiburg> <name> "Freiburg" .
3 <freiburg> <hasPopulation> "237244" .
4 <münster> <type> <city> .
5 <münster> <name> "Münster" .
6 <münster> <hasPopulation> "307071" .

```

Listing 1: Six RDF triples, assigning two cities their name and population size.

Each triple statement forms the edge of a directed graph. The subject is the start node, the predicate is a annotated edge, and the object is the end node.

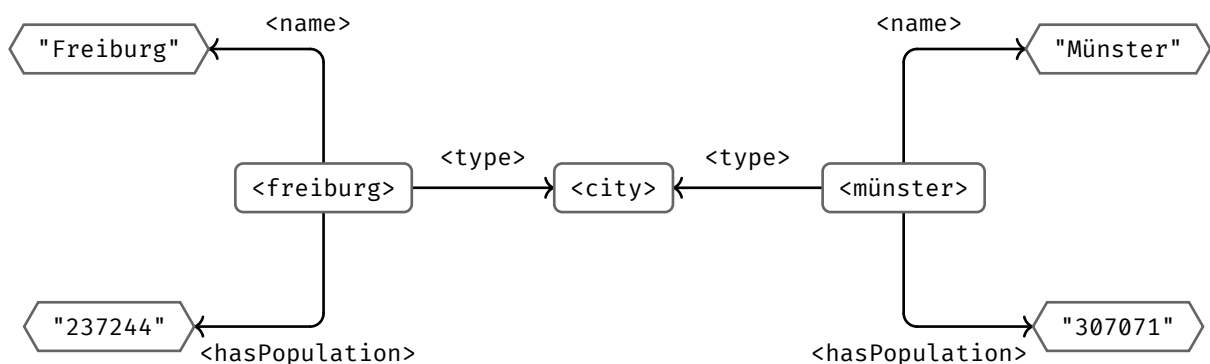


Figure 9: Visualization of Listing 1 as a graph.

Resources are represented with rectangles, literals with hexagons

The subject of an RDF statement is either an Internationalized Resource Identifier (IRI) or a blank node (I will ignore those) and denotes a resource. A blank node represents an anonymous resource without a global identifier. For the purposes of this thesis, IRIs are just strings without whitespace enclosed by angle brackets. The predicate of a RDF statement also has to be an IRI. The object is an IRI, a blank node, or a string literal.

Data in RDF format is often called *RDF graph* or *knowledge graph*.

## 2.2 SPARQL

*SPARQL* is the standard query language for RDF data.

It allows users to retrieve and manipulate data stored in RDF knowledge graphs.

The two actions – retrieving and manipulating data – are separated into two disjoint sublanguages:

- *SPARQL* 1.1 Query Language: to retrieve data
- *SPARQL* 1.1 Update: to manipulate data – e.g. insert, delete, change

In this thesis I will focus on the *SPARQL* query language.

*SPARQL* queries use pattern matching as a central mechanism. A pattern describes a structure of a subgraph. Such a pattern is called *group graph pattern*. Listing 2 shows a simple query with a group graph pattern.

```

1 SELECT * WHERE {
2   ?city <type> <city> .
3   ?city <name> ?name
4 }

```

Group  
Graph  
Pattern

(1)

Listing 2: SPARQL query with a simple group graph pattern

Here the group graph pattern (1) has 2 triples. `?city` and `?name` are variables. They function as wildcards and match any node in the graph.



Figure 10: Visualization of the graph pattern from Listing 2.  
Variables are represented with octagons.

This graph pattern matches 2 subgraphs in the knowledge graph.  
The first solution is shown in Figure 11.

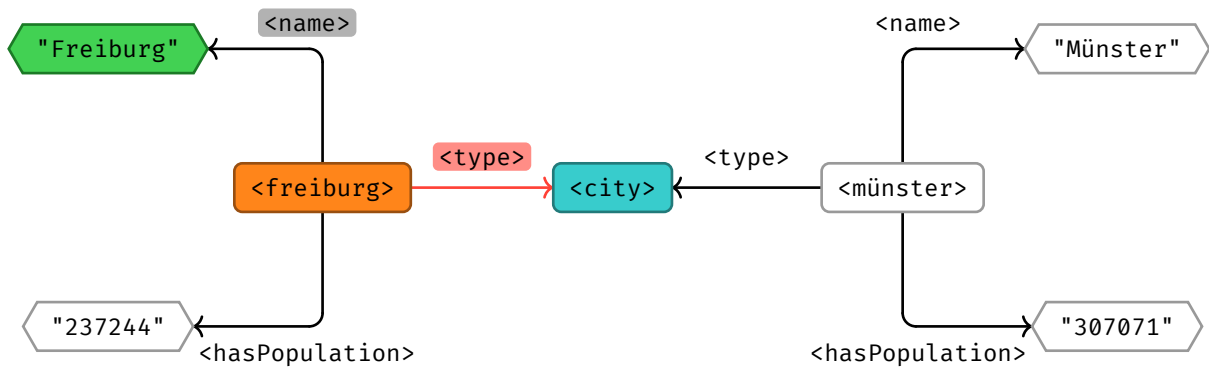


Figure 11: RDF graph with a highlighted subgraph that matches the group graph pattern from Listing 2.

Note that in this solution `?city` matches `<freiburg>` and `?name` matches `"Freiburg"`.

The second solution is shown in Figure 12.

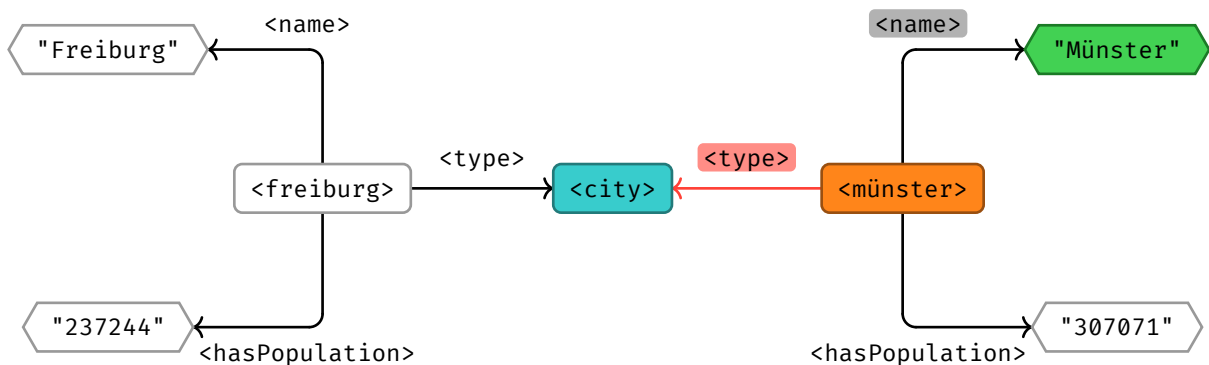


Figure 12: RDF graph with a highlighted subgraph that matches the group graph pattern from Listing 2.

Note that in this solution **?city** matches **<münster>** and **?name** matches **"Münster"**.

In each match, every variable matches a literal or IRI. In other words every match provides a *binding* for every variable. The result is a table with a column for each variable. For every match a row is added to the result, listing the bindings.

The result for the example would be shown in Table 1.

?city	?name
<freiburg>	"Freiburg"
<münster>	"Münster"

Table 1: *SPARQL* result table.

This is the basic query mechanism.

*SPARQL* supports a wide range of features:

- Selecting a specific variable
- Binding variables to given values
- Filtering for conditions
- Grouping and aggregating
- Ordering
- Limiting
- some conversions via build-in functions
- ...

*SPARQL* is described in detail, with examples, in its [specification](#).

## 2.3 Language Server Protocol

The Language Server Protocol (LSP) [3] is a protocol used between editors and language servers that provide language specific tooling.

The idea is that an editor should only implement language agnostic features and everything language specific should be provided by the language server. The editor and the language server form a client – server architecture.

The protocol is JSON-RPC based. That means the messages are JSON strings that follow a specific structure. In JSON-RPC there exist 3 types of messages:

- Requests, containing a method name, parameters and an id
- Responses, containing results or errors and the corresponding request id
- Notifications, containing a method name, parameters but no id

During the initialization stage, the client and server exchange their capabilities.

When the language server does not implement a capability – for example formatting – the client knows that and will not send formatting requests.

Every change in the editor is sent to the language server so it can maintain a synchronized version of the input.

Each capability has a unique method name – for example formatting: “textDocument/formatting”.

The client can trigger a capability by sending a request with the respective method and parameters.

The request for formatting requires the following parameters:

- the documents identifier (a URI)
- formatting options

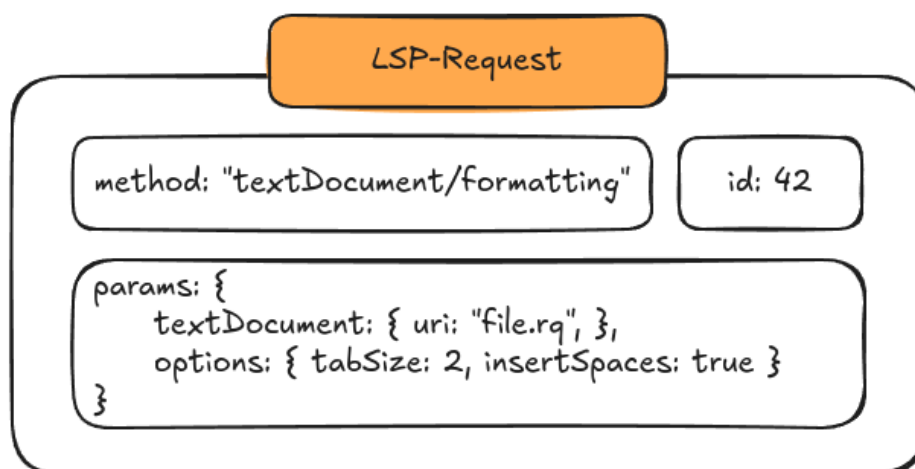


Figure 13: Visualization of a formatting LSP-request.

The server processes these messages and responds with a result or error.

In the case of formatting, the result is an array of text edits. A text edit is represented by a range in the document, and a string. The range is supposed to be replaced with that string.

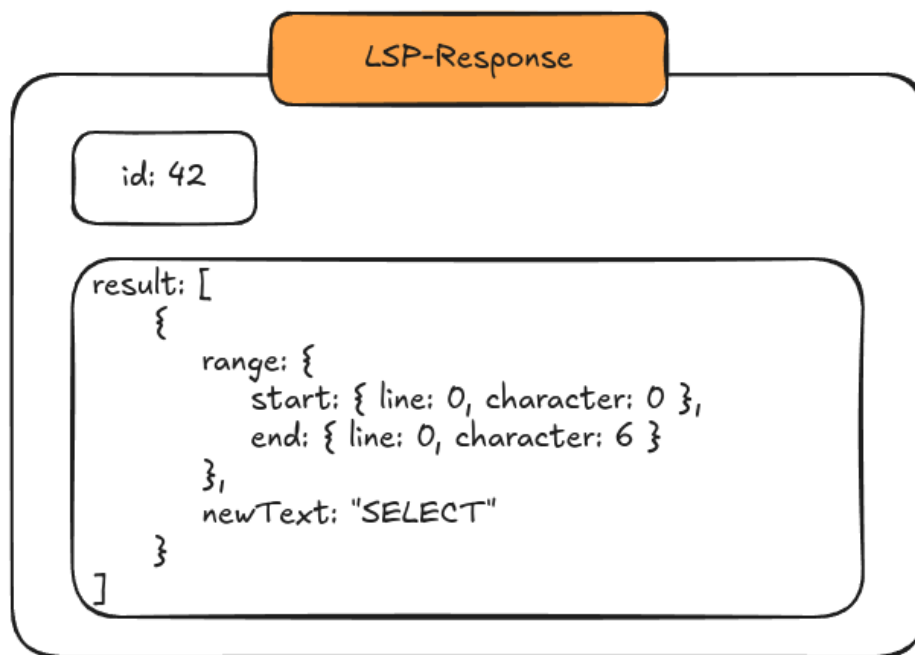


Figure 14: Visualization of a LSP-response to a formatting request.

This is how almost all capabilities are implemented. They are registered during initialization, initiated by a request to the language server and solved by a response.

## 3 Related work

I defined *language support* as the collection of the five capabilities: completion, formatting, diagnostics, code actions and hover. I will discuss the related work for each of these capabilities.

### 3.1 Completion

#### 3.1.1 Data-driven completion

Hannah Bast, Johannes Kalmbach, Theresa Klumpp, Florian Kramer and Niklas Schnelle [4] proposed and demonstrated the completion strategy: “*SPARQL Autocompletion via SPARQL*”.

In this strategy, the suggestions are retrieved from the knowledge-graph itself via *SPARQL* queries. This has the advantage that the completion suggestions are very accurate. The disadvantage is that this puts a lot of load on the triple store, since for each completion request at least one *SPARQL* query is sent. The *SPARQL* query used to compute the completions (called *autocompletion query*) can also become slow, or throw an error. Here is a example:

Given the knowledge-graph from before:

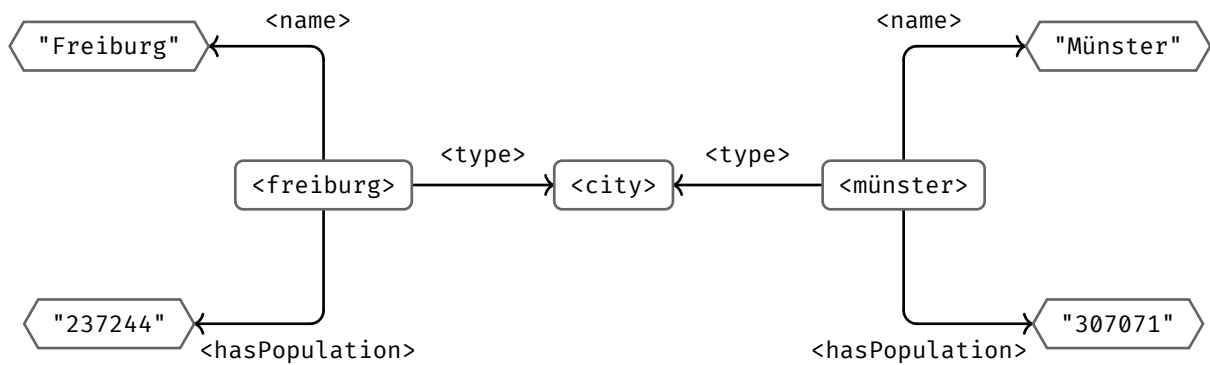


Figure 15: Example knowledge graph, containing name and population of two german cities.

And this query:

```

1 SELECT * WHERE {
2   ?city <type> <city> .
3   ?city <name> 
4 }
```

(The red box is not part of the input, but the cursor position.)

A good *autocompletion query* would be

```

1 SELECT ?suggestion WHERE {
2   { ?city <type> <city> }
3   ?city <name> ?suggestion
4 }
```

The marked section is the computed *completion context*, i.e. everything that constrains the result. The result of this query would be:

?suggestion
“Freiburg”
“Münster”

What’s omitted in this example is a relevancy based ranking of the suggestions.

This autocompletion strategy is used in the “*QLever-UI*”, a web-based *SPARQL* editor for the “*QLever*” RDF triple store. *QLever-UI*, *QLever* and *Qhue-Is* (this thesis) are projects of the Algorithms and Data structures group at the University Freiburg lead by Hannah Bast.

The source of *QLever-UI* is publicly available on GitHub: <https://github.com/ad-freiburg/qllever-ui>. And a demo is available on this webpage: <https://qllever.cs.uni-freiburg.de/>.

The implementation of “*SPARQL* autocompletion via *SPARQL*” in the *QLever-UI* works, but has weaknesses. The biggest problem is that it does not use a parse tree. It uses regular expressions to locate cursor in the query grammar.

*SPARQL* has nested structure such as balanced parentheses and cannot be captured by regular expressions. This limitation is formalized in the Chomsky hierarchy of languages [5]. Therefore this approach is inherently limited.

In this thesis I will improve this approach. My goal is that *QLever-UI* will use *Glue-Is* for its completion capability.

### 3.1.2 Schema-driven completion

Vincent Emonet, implemented an autocompletion strategy that could be summarized as “Schema-driven completion”.

In contrast to “*SPARQL* Autocompletion via *SPARQL*” it does not use the data in the knowledge-graph, but the structure of the data, sometimes called *schema*.

RDF itself does not enforce any *schema* or rules on the data.

For example: “If an entity has the type *city*, it has to have a *<name>* and *<hasPopulation>* property”. However, there exist systems to describe and enforce such schemas. One way to describe the schema with RDF is with a *VOID description* [6].

Vincent Emonet’s implementation of “Schema-driven completion” uses the *VOID description* of the knowledge-graph to provide completion for classes and properties.

The advantage of this approach is that completions are computed very fast. Since the schema can be fetched once and then no further queries are required.

The disadvantage is that the accuracy of the results is limited to the schema. Also a *VOID description* is required and the *syntactic location* is limited to predicate and object.

This completion strategy is implemented in a *SPARQL* editor of the Swiss Institute of Bioinformatics.

The source of this editor is publicly available here: <https://github.com/sib-swiss/sparql-editor>.

And a demo is available on this webpage: <https://sib-swiss.github.io/sparql-editor/>.

### 3.1.3 Prefix completion

Vercruysse, Arthur and Rojas Melendez, Julian Andres and Colpaert, Pieter presented a language server called “Semantic Web Language Server” in 2025 [7].

This language server has a completion capability.

When typing a prefix at the predicate location of a triple it suggests valid completions. When the predicate is “a” (a special keyword) it also supports prefix completion for the object location.

It also suggests every keyword at any location.

The source of this language server is publicly available here: <https://github.com/SemanticWebLanguageServer/swls>.

And a demo is available on this webpage: <https://semanticweblanguageserver.github.io/swls/>.

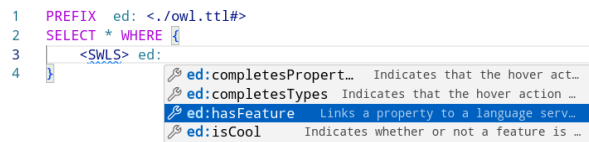


Figure 16: Prefix completion at predicate location.

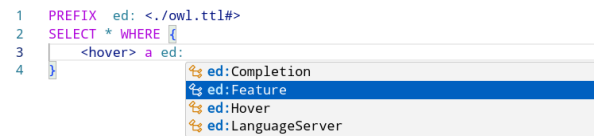


Figure 17: Prefix completion for a class.

## 3.2 Formatting

### 3.2.1 Sparqling sparql-formatter

Hirokazu Chiba, implemented a *SPARQL* formatter tool called *sparql-formatter* [8].

It works quite well but it does have some issues.

I’ve written a detailed comparison and analysis in my [blog post about Qlue-ls](#)[9].

The source of *sparql-formatter* is publicly available here: <https://github.com/sparqling/sparql-formatter/>. And a demo is available on this webpage: <https://sparql-formatter.dbcls.jp/>.

### 3.2.2 YASGUI

*YASGUI* is a *SPARQL* client. It was presented in the paper “YASGUI: Not Just Another SPARQL Client” by Laurens Rietveld1 and Rinke Hoekstra in the year 2013 [10].

It’s is a web based *SPARQL* client that provides a full user experience.

From selecting the *SPARQL* endpoint, writing the query to finally fetching and displaying the results.

At it’s time, according to [10], *YASGUI* was one the first *SPARQL* editors to provide modern features like formatting, autocompletion or syntax highlighting to the table.

The source of *YASGUI* is publicly available here: <https://github.com/TriplyDB/Yasgui>.

And a demo is available on this webpage: <https://yasgui.tripty.cc/>.

The key combination to trigger formatting is Ctrl + Shift + f.

## 3.3 Diagnostics

### 3.3.1 Undefined prefix

The *Semantic Web Language Server* [7] also supports one diagnostic called “Undefined prefix”.

This diagnostic does exactly the same as my implementation of the “undeclared prefix” diagnostic, described in Section 6.3.2. When a prefix is used but was never declared, this diagnostics is shown to the user via Error-diagnostic.

## 3.4 Hover

### 3.4.1 Data-driven hover

The *QLever-UI* also provides the hover capability.

When hovering IRIs in *QLever-UI* it tries to show the label of this resource.



```
1 PREFIX wd: <http://www.wikidata.org/entity/>
2 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3 SELECT WHERE {
4   wd:Q2833 rdfs:label "Freiburg im Breisgau"@en .
5 }
```

Figure 18: query in the *QLever-UI*, hovering *wd:Q2833*

Just like the *completion* capability of *QLever-UI*, this is implemented via *SPARQL* queries. This makes the shown content configurable. With the right query the *QLever-UI* could show anything from the knowledge graph, like comments or information about the class of the entity.

## 4 Architecture

I want to briefly discuss the architecture of *Qlue-ls* to provide an overview before I discuss each component in greater detail.

*Qlue-ls* is a language server, that means it's a program that sends and receives LSP messages.

The “connection” module is responsible to send and receive these messages.

The “message dispatch” module is responsible to recognize the methods of incoming messages and dispatch it to a message handler.

Each capability has a “message handler” module that is responsible to handle the messages. The “message handler” modules use the “connection” module to send a LSP response to the client.

The “parser” module is responsible for building parse trees, it's available from every “message handler” module.

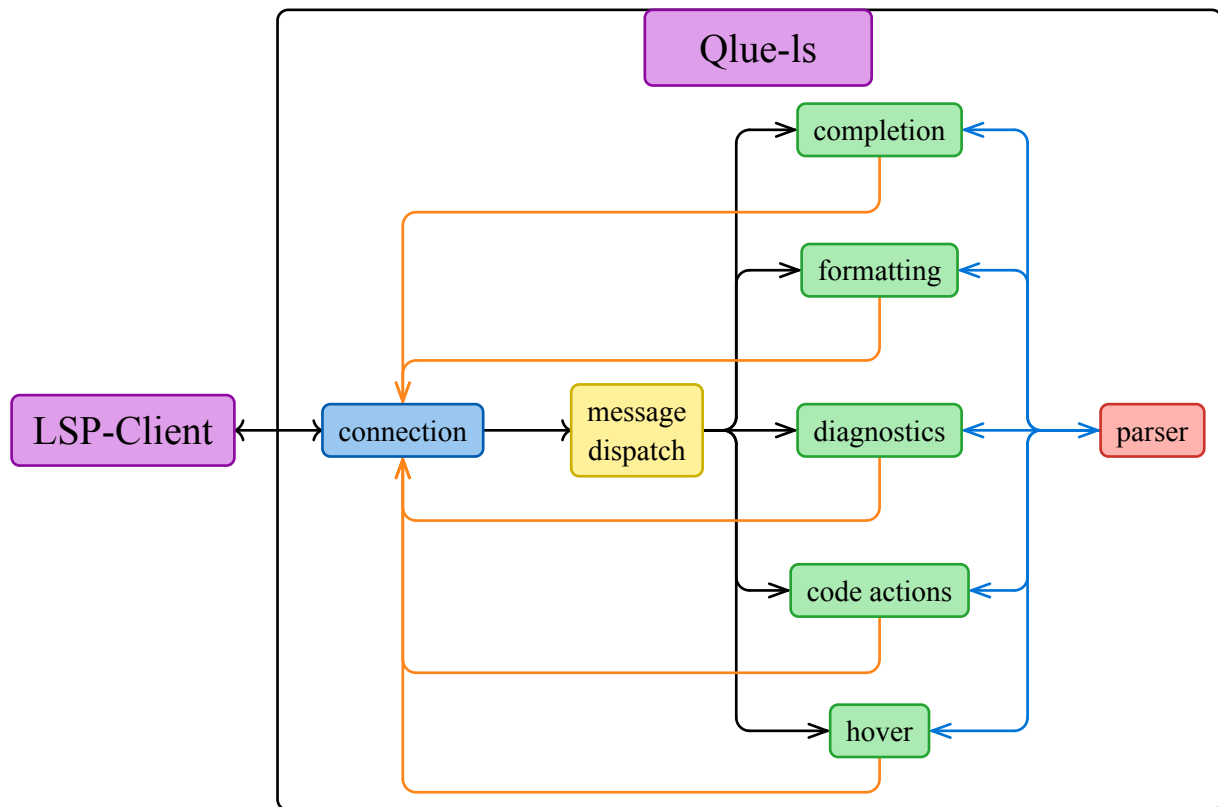


Figure 19: Architecture of the *Qlue-ls* language server.

## 5 Parser

Every capability implementation uses the parser to “understand” the syntactic structure of the input. The parser transforms a simple string into a structured representation called **parse tree**. In this section I will describe what a parse tree is and how I compute it for *SPARQL*.

In the next sections I describe how the actual capabilities are implemented.

Assume you have a *SPARQL* query input as a vector of bytes. Suppose you want to extract the variables listed in the SELECT clause. This information is not directly accessible from the raw bytes. One might attempt to locate the substring “SELECT” and extract variable names using regular expressions. However, this approach is error-prone, inefficient, and difficult to maintain.

Given the string: “SELECT ?s WHERE { ?s ?p ?o }”

I want to compute the tree in Figure 20.

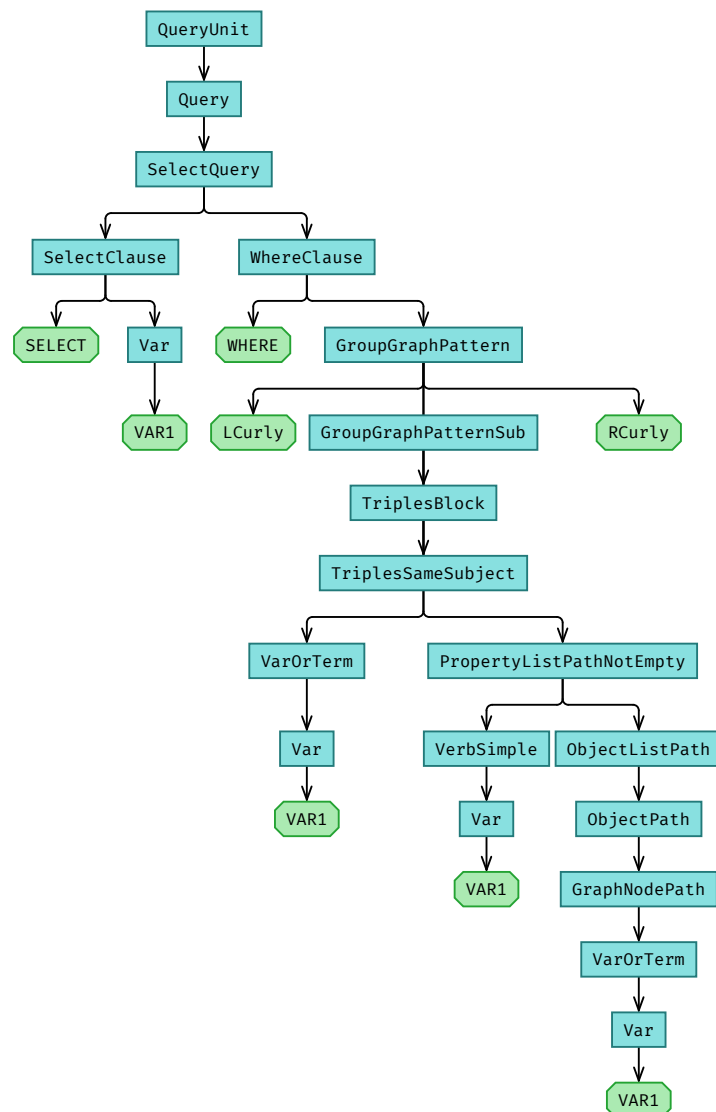


Figure 20: Parse tree for the input “SELECT ?s WHERE { ?s ?p ?o}”.

Given this tree, finding the variables listed in the SELECT clause is easy.

The program that builds this tree is called **parser**.

An important detail to keep in mind is that the given *SPARQL* query does not have to be complete. Especially for the completion capability, the input is often invalid.

## 5.1 Grammar

To build the parser, I use a **context free grammar** as a theoretical basis.

For simplicity I will write “grammar” instead of “context free grammar” in the rest of this thesis.

A grammar can be used to describe the syntax of formal languages, such as *SPARQL*.

Essentially a grammar is a set of rules for rewriting strings, along with a “start symbol” from where the rewriting starts.

The core idea is simple. For example a grammar would be:

```
1 There are 2 rules:
2 - Replace 'A' with '4B'.
3 - Replace 'B' with '2'
4 Start with 'A' and apply the rules above.
```

We start with ‘A’ apply the first rule and get ‘4B’ then we apply the second rule ‘42’.

This is the core principle of a formal grammar.

Formally a grammar  $G$  consist of four components:

- a finite set  $\mathcal{N}$  of *nonterminal* symbols
- a finite set  $\Sigma$  of **tokens**, known as *terminal* symbols, that is disjoint from  $\mathcal{N}$
- a finite set  $P$  of productions, where each production consists of a nonterminal, called *left hand side (lhs)*, and a sequence of tokens and/or nonterminals, called *right hand side (rhs)*.

A rule states that the *lhs* can be replaced with the *rhs*.

Formally written, a production has this form:  $\underbrace{\mathcal{N}}_{\text{lhs}} \rightarrow \underbrace{(\Sigma \cup \mathcal{N})^*}_{\text{rhs}}$

where  $*$  is the Kleene star operator, which denotes zero or more repetitions.

- a nonterminal  $S \in \mathcal{N}$  that is the *start symbol*.

The grammar formally is defined as the tuple  $(\mathcal{N}, \Sigma, P, S)$ .

These four components form a rewriting system:

Start with the *start symbol*  $S$ , repeatedly apply rules from  $P$  to replace nonterminals from  $\mathcal{N}$  until only tokens from  $\Sigma$  are left.

- the strings that get produced are called **words**
- the set of all words is called **language**
- each replacement step is called **derivation-step**
- the sequence of **derivation-steps** is called **derivation**

### 5.1.1 Grammar example

Given the Grammar  $G = (\mathcal{N}, \Sigma, P, S)$  with:

$$\mathcal{N} = \{A, B\}$$

$$\Sigma = \{\text{id}, (, ), +\}$$

$$P = \{A \rightarrow B, B \rightarrow (B + B), B \rightarrow \text{id}\}$$

$$S = A$$

Now we start with  $A$  and apply rules from  $P$ .

$$A \Rightarrow B \Rightarrow (B + B) \Rightarrow (\text{id} + B) \Rightarrow (\text{id} + (B + B)) \Rightarrow (\text{id} + (\text{id} + B)) \Rightarrow (\text{id} + (\text{id} + \text{id}))$$

We have shown how the string “(id + (id + id))” is generated from the start symbol.

To indicate zero or more steps we can write  $\Rightarrow^*$ , as in  $A \Rightarrow^* (\text{id} + (\text{id} + \text{id}))$

### 5.1.2 Parse tree

Let  $G = (\mathcal{N}, \Sigma, P, S)$  be a Grammar

and  $S = w_0 \Rightarrow w_1 \Rightarrow \dots \Rightarrow w_n$  be a derivation, with  $w_i \in (\mathcal{N} \cup \Sigma)^*$ .

Then we get the parse tree as follows:

- The parse tree is initialized with a single root node  $S$
- The tree is constructed stepwise. After step  $i$ , the leaves – read from left to right – are always  $w_i$
- If in derivation-step  $w_i \Rightarrow w_{i+1}$  the rule  $V \rightarrow u$  is applied, then  $V$  gets  $|u|$  children that get annotated – from left to right – with the symbols of  $u$ .

For the example derivation above:

$$A \Rightarrow B \Rightarrow (B + B) \Rightarrow (\text{id} + B) \Rightarrow (\text{id} + (B + B)) \Rightarrow (\text{id} + (\text{id} + B)) \Rightarrow (\text{id} + (\text{id} + \text{id}))$$

We would get this parse tree:

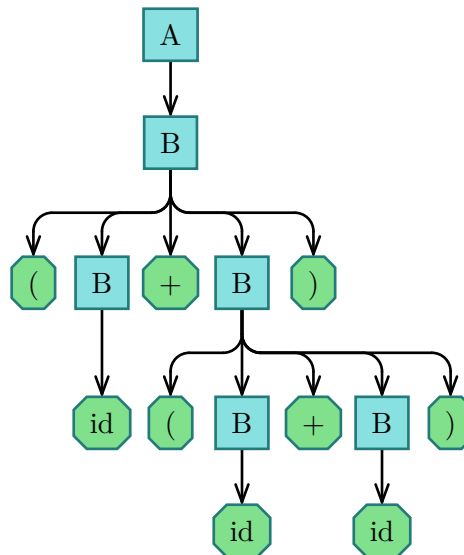


Figure 21: A parse tree for the word “(id + (id + id))”.

Nonterminals are blue and rectangles, tokens are green and octagons.

Note that there are multiple derivations that can yield the same parse tree.

The parse tree above is also the parse tree of this derivation:

$$A \Rightarrow B \Rightarrow (B + B) \Rightarrow (\text{id} + B) \Rightarrow (\text{id} + (B + B)) \Rightarrow (\text{id} + (B + \text{id})) \Rightarrow (\text{id} + (\text{id} + \text{id}))$$

### 5.1.3 Leftmost derivation

A **leftmost derivation** is a derivation where always the leftmost variable gets replaced.

This is a leftmost derivation:

$$A \Rightarrow B \Rightarrow (B + B) \Rightarrow (\text{id} + B) \Rightarrow (\text{id} + (B + B)) \Rightarrow (\text{id} + (\text{id} + B)) \Rightarrow (\text{id} + (\text{id} + \text{id}))$$

This is not:

$$A \Rightarrow B \Rightarrow (B + B) \Rightarrow (\text{id} + B) \Rightarrow (\text{id} + (B + B)) \Rightarrow (\text{id} + (B + \text{id})) \Rightarrow (\text{id} + (\text{id} + \text{id}))$$

### 5.1.4 LL(k) property

Formally this is the definition of LL(k):

Let  $G$  be a grammar and  $k \geq 1 \in \mathbb{N}$ .

$G$  is  $LL(k)$ , if and only if for any two leftmost derivations:

1.  $S \Rightarrow \dots \Rightarrow wA\alpha \Rightarrow w\beta\alpha \Rightarrow \dots \Rightarrow wu$
2.  $S \Rightarrow \dots \Rightarrow wA\alpha \Rightarrow w\gamma\alpha \Rightarrow \dots \Rightarrow wv$

The first  $k$  tokens of  $u$  and  $v$  are the same, then  $\beta = \gamma$ .

In this definition:

- $S$  is the start symbol
- $A$  is a nonterminal
- $w$  is the already derived input
- $u$  and  $v$  are the remaining inputs
- $\alpha$ ,  $\beta$  and  $\gamma$  are sequences of grammar symbols

Let's unpack this definition as it will become very important.

Given a word  $w' = wu$ .

Suppose we found a partial leftmost derivation  $S \xRightarrow{*} wA\alpha$ .

If the grammar is LL(k) then, only by looking at the first  $k$  symbols of  $u$  we know that there is exactly one rule to expand  $A$  that is allowed to apply next.

This is relevant, because the *SPARQL* grammar is LL(1).

### 5.1.5 FIRST set

For the construction of the parser, FIRST sets are important.

FIRST assigns each nonterminal a set of tokens. This set contains exactly the first tokens of all words that **could** be derived from the nonterminal.

For example:

- If  $A$  can only be extended to the words “abc” or “def” then  $\text{FIRST}(A) = \{'a', 'd'\}$
- If  $A$  can only be extended to the empty word  $\varepsilon$  and “x” then  $\text{FIRST}(A) = \{'x', \varepsilon\}$

The letter  $\varepsilon$  denotes the empty word, not a grammar symbol. If the FIRST set of a nonterminal contains  $\varepsilon$ , that means it can be extended to “nothing”.

To formally define FIRST, I will first define  $\text{FIRST}_{\text{rhs}}$ .

$\text{FIRST}_{\text{rhs}}$  does the same as FIRST just for right hand sides of production rules instead of nonterminals.

Let  $G = (\mathcal{N}, \Sigma, P, S)$  be a grammar, then  $\text{FIRST}_{\text{rhs}}$  is defined as follows.

$$\text{FIRST}_{\text{rhs}} : (\mathcal{N} \cup \Sigma)^* \rightarrow \mathcal{P}(\Sigma \cup \{\varepsilon\}) :$$

$$\text{rhs} \mapsto \begin{cases} \{\varepsilon\} & \text{if rhs} = \varepsilon \\ \{t\} & \text{if rhs} = t \text{ rhs}' \\ \text{FIRST}(A) & \text{if rhs} = A \text{ rhs}' \wedge \varepsilon \notin \text{FIRST}(A) \\ \text{FIRST}(A) \cup \text{FIRST}_{\text{rhs}}(\text{rhs}') & \text{if rhs} = A \text{ rhs}' \wedge \varepsilon \in \text{FIRST}(A) \end{cases}$$

In this definition  $t$  is a token  $A$  is nonterminal and  $\text{rhs}'$  is another right hand side of a production rule (possibly empty).

Now FIRST is defined as

$$\text{FIRST} : \mathcal{N} \rightarrow \mathcal{P}(\Sigma \cup \{\varepsilon\}) : V \mapsto \bigcup_{V \rightarrow \text{rhs} \in P} \text{FIRST}_{\text{rhs}}(\text{rhs})$$

With these definitions FIRST and  $\text{FIRST}_{\text{rhs}}$  can be recursively computed.

For example, given this grammar:

```
1 A → B
2 B → (B + B)
3 B → id
```

The FIRST sets would be:

$$\text{FIRST}(B) = \text{FIRST}_{\text{rhs}}((B + B)) \cup \text{FIRST}_{\text{rhs}}(\text{id}) = \text{FIRST}_{\text{rhs}}((B + B)) \cup \{\text{id}\} = \{ (, \text{id} \}$$

$$\text{FIRST}(A) = \text{FIRST}_{\text{rhs}}(B) = \text{FIRST}(B) = \{ (, \text{id} \}$$

## 5.2 Extended Backus-Naur Form

*SPARQL* is provided in the **Extended Backus–Naur Form** (EBNF).

EBNF is a notation to declare grammars. Unfortunately there exist many different variations.

For this thesis the notation is not really important, what is important is the constructs EBNF brings to the table. EBNF is syntactic sugar for formal grammars. Every grammar that can be described in EBNF could be described in a formal grammar and vice versa.

EBNF provides the following constructs to the right hand side of rules:

Let  $E_1, E_2$  be EBNF right hand side terms then so are:

Concept	Notation	Semantic
concatenation	$E_1 E_2$	$E_1$ followed by $E_2$
alternation	$E_1 \mid E_2$	Either $E_1$ or $E_2$
repetition	$E_1^*$	Repeat $E_1$ <b>zero</b> or more times

Concept	Notation	Semantic
non zero repetition	$E_1^+$	Repeat $E_1$ <b>one</b> or more times
option	$E_1?$	$E_1$ or the empty word $\varepsilon$
group	$(E_1)$	Group term for precedence

Note that *concatenation* takes precedence over alternation.

The table above creates a recursive definition for a EBNF term. The base cases for this recursion is: Every grammar symbol is a EBNF term.

Every EBNF term could be rewritten using formal grammar production rules.

The proof is left as an exercise for the reader.

### 5.2.1 FIRST sets for EBNF terms

Now that I introduced EBNF, the definition for  $\text{FIRST}_{\text{rhs}}$  needs to be extended.

$$\text{FIRST}_{\text{rhs}}(\text{rhs}) := \begin{cases} \dots & \dots \\ \text{FIRST}_{\text{rhs}}(E_1) & \text{if } \text{rhs} = E_1 E_2 \wedge \varepsilon \notin \text{FIRST}''(E_1) \\ \text{FIRST}_{\text{rhs}}(E_1) \cup \text{FIRST}''(E_2) & \text{if } \text{rhs} = E_1 E_2 \wedge \varepsilon \in \text{FIRST}''(E_1) \\ \text{FIRST}_{\text{rhs}}(E_1) \cup \text{FIRST}''(E_2) & \text{if } \text{rhs} = E_1 \mid E_2 \\ \text{FIRST}_{\text{rhs}}(E) \cup \{\varepsilon\} & \text{if } \text{rhs} = E^* \\ \text{FIRST}_{\text{rhs}}(E) & \text{if } \text{rhs} = E^+ \\ \text{FIRST}_{\text{rhs}}(E) \cup \{\varepsilon\} & \text{if } \text{rhs} = E? \\ \text{FIRST}_{\text{rhs}}(E) & \text{if } \text{rhs} = (E) \end{cases}$$

## 5.3 SPARQL grammar

*SPARQL* was standardized by the World Wide Web Consortium.

It is defined in the specification [SPARQL 1.1 Query Language](#) [1].

The *SPARQL* Grammar is a context free grammar.

It consists of 138 nonterminals, a production rule each and 161 terminals.

There are two entry points into the grammar: *QueryUnit* and *UpdateUnit*.

Formally, only one entry to a grammar is allowed. It would be more correct to say there are 2 grammars defined, one with *starting symbol* ‘*QueryUnit*’ and one with *starting symbol* ‘*UpdateUnit*’.

In Section 2.2 I mentioned that there are two disjoint sublanguages. ‘*QueryUnit*’ is the *starting symbol* for the *SPARQL* query language and ‘*UpdateUnit*’ is the *starting symbol* for the *SPARQL* update language.

Both grammars use the same set of production rules and both are LL(1).

## 5.4 Lexical Analysis

A parser gets a sequence of tokens as input and transforms them into a parse tree.

A few examples of tokens of the *SPARQL* grammar are: *SELECT*, *WHERE*, *{*, *\**, *VAR1*, *INTEGER*

However the input is sequence of chars. A sequence of chars that make up a single token is called *lexeme*.

For example “?var123” is a *lexeme* of the token *VAR1*.



The role of the *lexical analyzer* is to read the input chars and produce a sequence of tokens. This transformation is also called *lexing*.

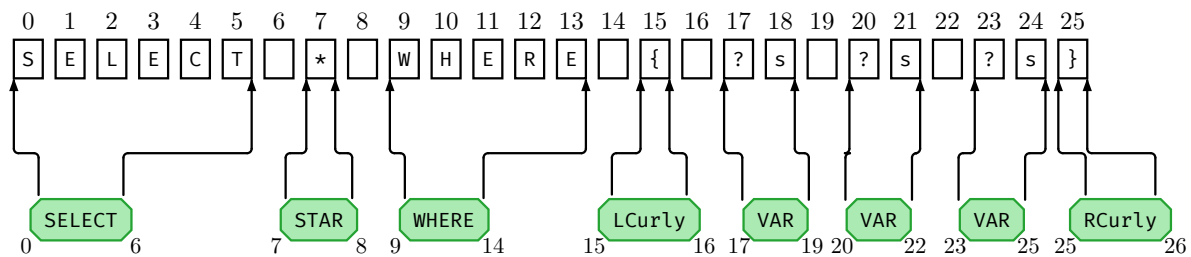


Figure 22: The string **SELECT \* WHERE { ?s ?p ?o }** transformed into *SPARQL* tokens.

For my application it's important to remember the lexeme of each token.

Figure 22 shows how "SELECT \* WHERE { ?s ?p ?o }" would get lexed. In this thesis I always represent tokens with octagons. The numbers and arrows show which slice of the input is the lexeme of each token. Also the token "{" is written "RCurly".

In this thesis, I do that with all "symbol-tokens" like '+' → 'Plus', '|' → 'Bar' and so on.

In the example whitespace is being skipped. In reality this is not the case. The parser I built is **lossless**. That means that the input can always be reconstructed from the parse tree. To achieve that, whitespace is not skipped but tokenized as well. When a parse-tree has this property we call it a *lossless concrete syntax tree*.

In the *SPARQL* specification each terminal is defined by regular expressions.

To convert the input sequence into a byte sequence, we repeatedly match the remaining input against these regular expressions, consuming the input token by token.

To create an efficient lexer is more complex than that, but since this is not the topic of this work, we will keep it simple.

## 5.5 Parsing Algorithm

In this section I describe the parser I built to parse *SPARQL*.

The input is a sequence of tokens  $w_1, w_2, \dots, w_n$  and the output a parse tree.

I build a **predictive recursive decent parser**.

*Recursive decent parsing* is a parsing method where each nonterminal is assigned one procedure.

These procedures call each other recursively to process the input.

The input sequence will be scanned from left to right.

*Predictive parsing* means that there is no backtracking – no token of the input is scanned twice.

This is possible due to the LL(1) property of the grammar. Whenever a procedure is required to make a decision, the next token (called **lookahead**) is used to unambiguously determine the correct procedure.

A predictive parser can be implemented by mutually recursive procedures. These procedures can be computed by automata.

For each nonterminal an automaton is constructed. Each automaton has a finite number of states. One state is a start state and one state is a final state. These states can be connected by directed edges.

Edges are annotated with a token, nonterminal or a special symbol  $\epsilon$ . I draw edges, annotated with a nonterminal, with a dashed line.

Here are 3 example automaton:

Automaton A:

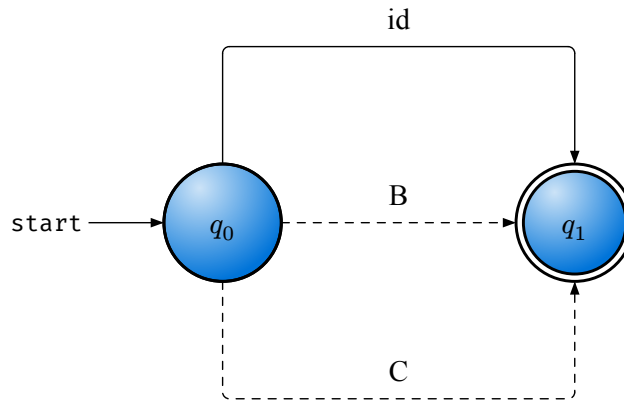


Figure 23: Automaton for nonterminal A with rule  $A \rightarrow B \mid C \mid \text{id}$ .

Automaton B:

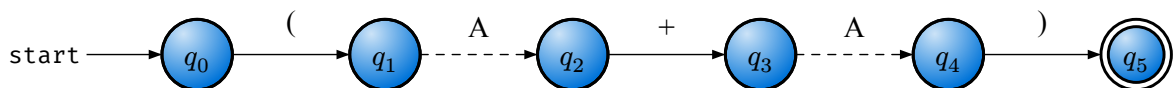


Figure 25: Automaton for nonterminal B with rule  $B \rightarrow ( A + A )$ .

Automaton C:

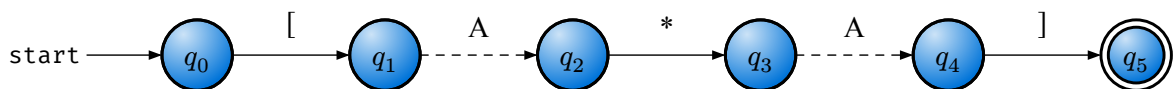


Figure 27: Automaton for nonterminal C with rule  $C \rightarrow [ A * A ]$ .

The parser executes these automata the following way:

The input is a sequence of tokens:  $w_1, \dots, w_n$ .

The parser “reads” the tokens one by one from left to right.

The next token to read is called *lookahead*. Initially the parser has not read any tokens and the *lookahead* is  $w_1$ .

The parser starts in the start state of the start symbol. Assume that after some steps it is in state  $s$ . If there is an edge from  $s$  to  $t$  annotated with token  $a$  and the *lookahead* matches  $a$ , the parser reads the next token into the *lookahead* and moves to state  $t$ .

If the edge is annotated with a nonterminal  $A$  and the *lookahead* is in  $\text{FIRST}(A)$ , the parser moves to the start state for  $A$  without reading from the input sequence. If the parser reaches the final state for  $A$ , the parser moves to state  $t$ .

Only if there is no edge as just described, but an edge annotated with  $\epsilon$ , the parser moves to  $t$  without reading from the input sequence.

To construct the automaton I first declare a help procedure.

The procedure is called *connect* and takes 3 arguments, the first two are states and the last is an EBNF rhs.

If  $X = a$  where  $a$  is a token:

Connect  $s$  and  $t$  with an edge annotated with  $a$ .

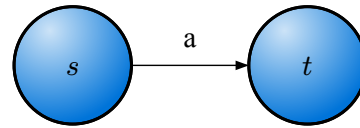


Figure 29: Sub-automaton created by calling *connect*, where the rhs is a token.

If  $X = A$  where  $A$  is a nonterminal:

Connect  $s$  and  $t$  with an edge annotated with  $A$ .

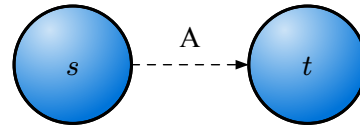


Figure 30: Sub-automaton created by calling *connect*, where the rhs is a nonterminal.

If  $X = X_1 \mid X_2 \mid \dots \mid X_n$  (alternation):

For each  $i \in [1, \dots, n]$  apply *connect* to  $s, t$  and  $X_i$ .

I draw a dotted edge if two states will be connected by applying *connect*.

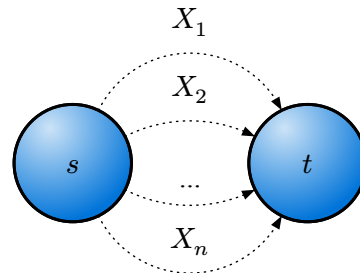


Figure 31: Sub-automaton created by calling *connect*, where the rhs is an alternation.

If  $X = X_1 X_2 \dots X_n$  (sequence):

Create  $n - 1$  states  $q_2, \dots, q_n$ . Apply *connect* to  $s, q_2$  and  $X_1$ . Apply *connect* to  $q_n, t$  and  $X_n$ .

For each  $i \in [2, \dots, n - 1]$  apply *connect* to  $q_i, q_{i+1}$  and  $X_i$ .



Figure 32: Sub-automaton created by calling *connect*, where the rhs is a sequence.

If  $X = X'?$  (optional):

Connect  $s$  and  $t$  with an edge annotated with  $\varepsilon$ .

Apply *connect* to  $s, t$  and  $X'$ .

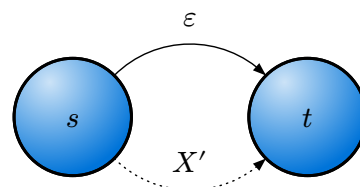


Figure 33: Sub-automaton created by calling *connect*, where the rhs is an optional construct.

If  $X = X'^*$ :

Connect  $s$  and  $t$  with an edge annotated with  $\varepsilon$ .

Apply *connect* to  $s$ ,  $s$  and  $X'$ .

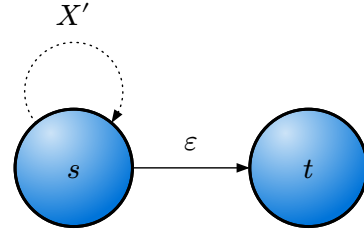


Figure 34: Sub-automaton created by calling *connect*, where the rhs is a repetition construct.

If  $X = X'^+$ :

Connect  $t$  and  $s$  with an edge annotated with  $\varepsilon$ .

Apply *connect* to  $s$ ,  $t$  and  $X'$ .

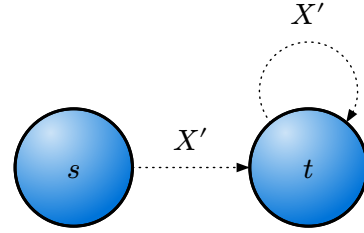


Figure 35: Sub-automaton created by calling *connect*, where the rhs is a non-zero repetition construct.

With this help procedure we can construct an automaton for every nonterminal.

For every production  $A \rightarrow X$  in the *SPARQL* grammar

1. create start state  $s$  and finite state  $t$ .
2. apply *connect* to  $s$ ,  $t$  and  $X$ .

Here are 2 examples from the *SPARQL* grammar:

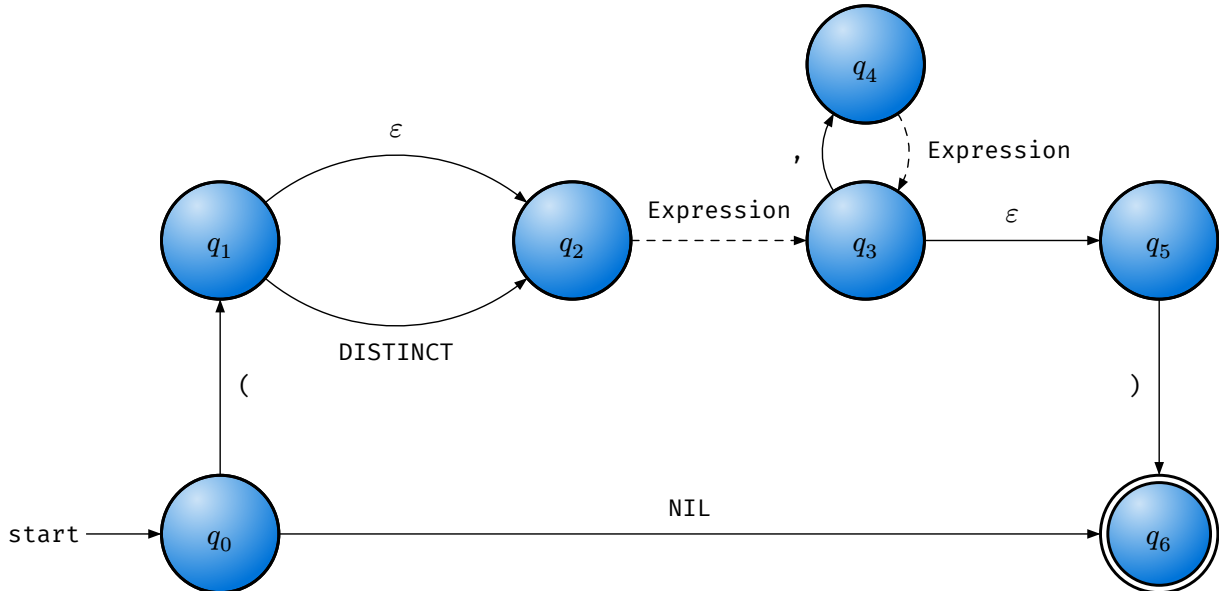


Figure 36: Automaton for the production:

$\text{ArgList} ::= \text{NIL} \mid '(' \text{'DISTINCT'? Expression} ( ',' \text{Expression} )^* ')'$

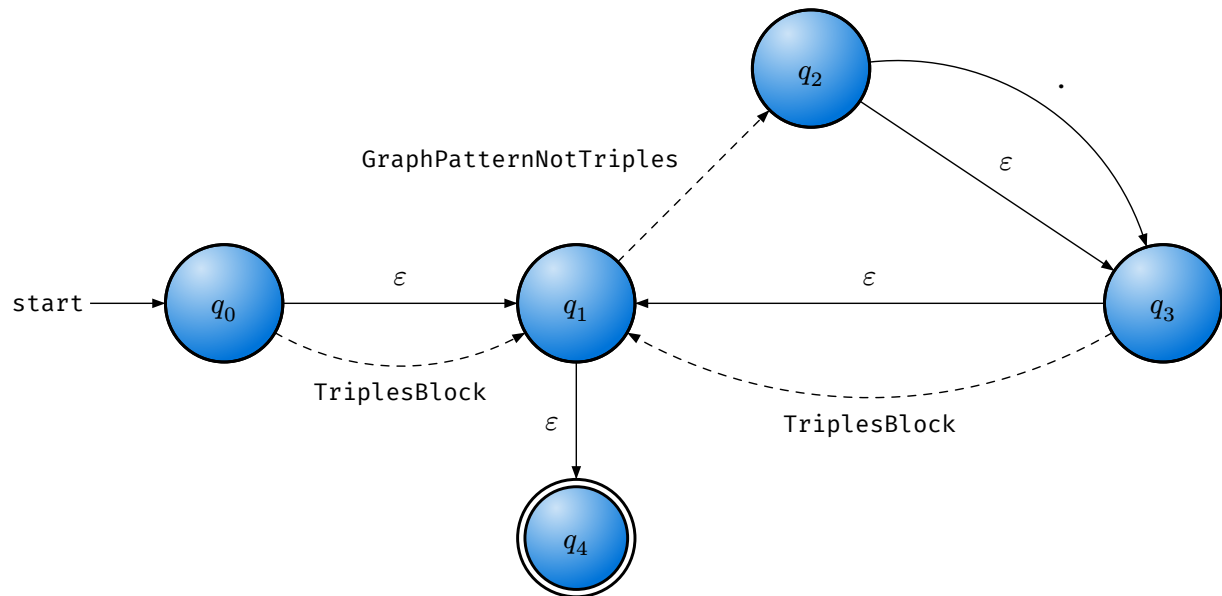


Figure 37: Automaton for the production:

`GroupGraphPatternSub = TriplesBlock? ( GraphPatternNotTriples '.'? TriplesBlock? )*`

If the parser reaches the final state of the parsing procedure for `QueryUnit` or `UpdateUnit` the input was a valid word of the *SPARQL* language.

To construct a parsing tree from these procedures:

1. Create a root node and mark it as *active*.
2. When an edge annotated with a token is used, add a leaf node marked with that token to the *active* node.
3. When an edge, annotated with an  $\epsilon$  is used, nothing happens.
4. When a procedure starts, a child node is added to the *active* node, marked with the corresponding nonterminal of the procedure. The new node is marked as the new *active* node. When the procedure is finished, the parent node is again marked as the *active* node.

### 5.5.1 Properties

This parsing algorithm creates a **LL** parser.

That means it is a **top-down** parser reads the input from **Left-to-right** performing **Leftmost** derivation.

#### 5.5.1.1 Top-down

Top-down refers to a parsing strategy where we begin with the grammars' start symbol and expand it step by step to match the input. As we do this, the parse tree grows from the top (the start symbol) downwards toward the leaves, which correspond to the input tokens.

My parsing algorithm does this as it starts with the grammars' start symbols by calling the procedure for `QueryUnit` or `UpdateUnit`. The constructed parse tree grows by adding nodes to the *active* node. So the constructed tree grows from the top (the root node) down to the leaves.

This is perfect for my use-case, because at any point during parsing, I have a partial tree rooted at the start symbol — making it easy to represent and inspect intermediate structure.

#### 5.5.1.2 Left-to-Right

The input to our parser is a sequence of tokens—for example:



Figure 38: Token sequence for the string “SELECT \* WHERE {“

Left-to-right means that the parser processes these tokens in the same order they appear in the input.

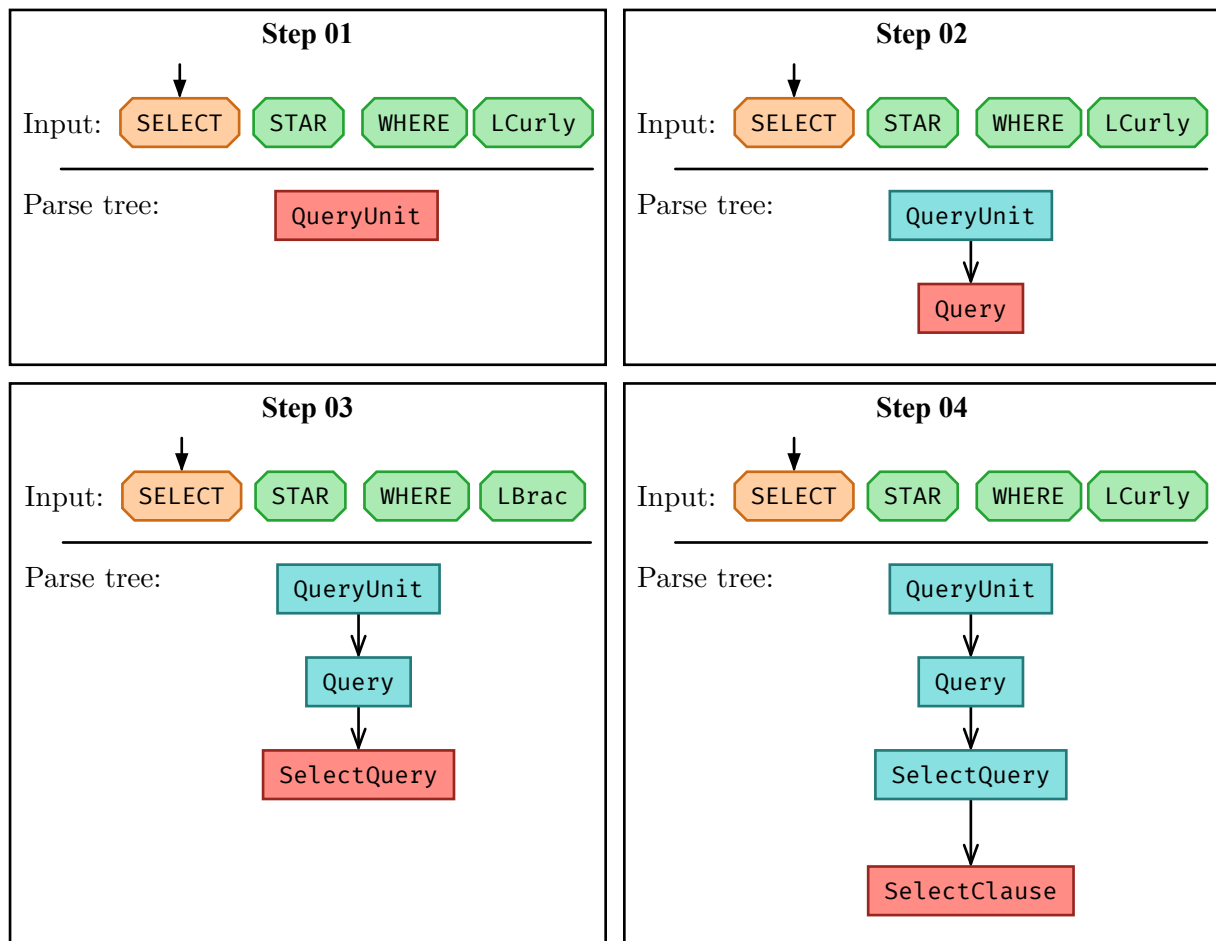
My parsing algorithm does this because it starts with the first token and then reads the tokens one by one from left to right.

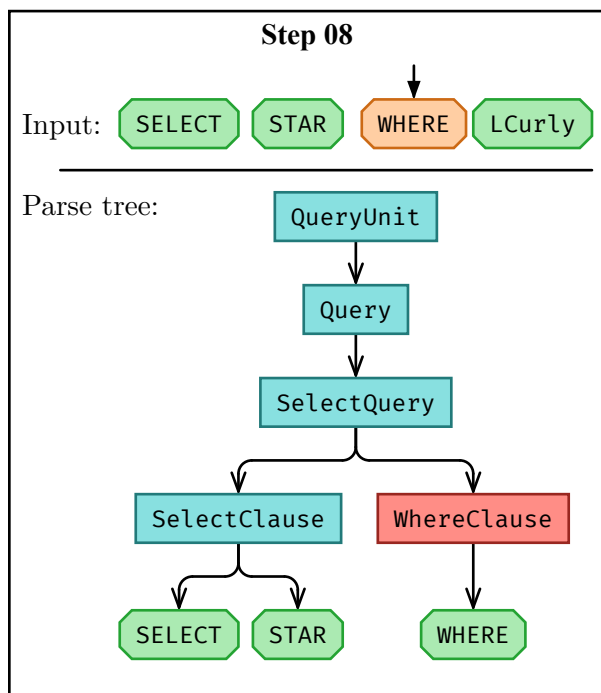
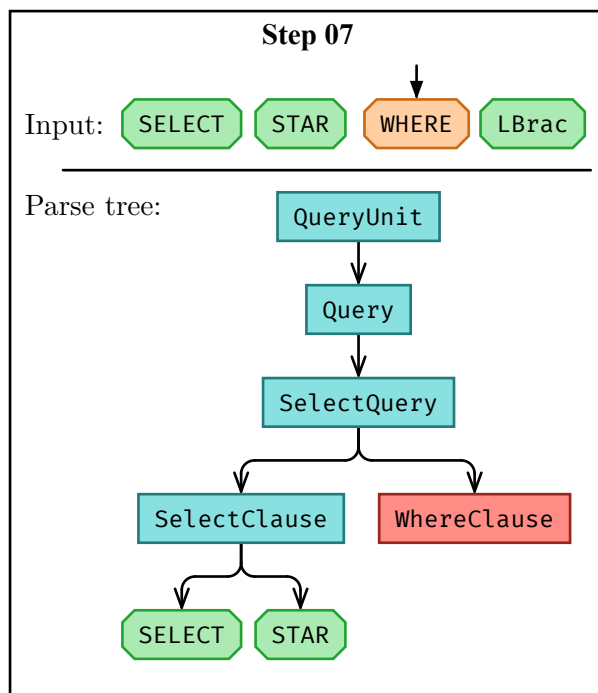
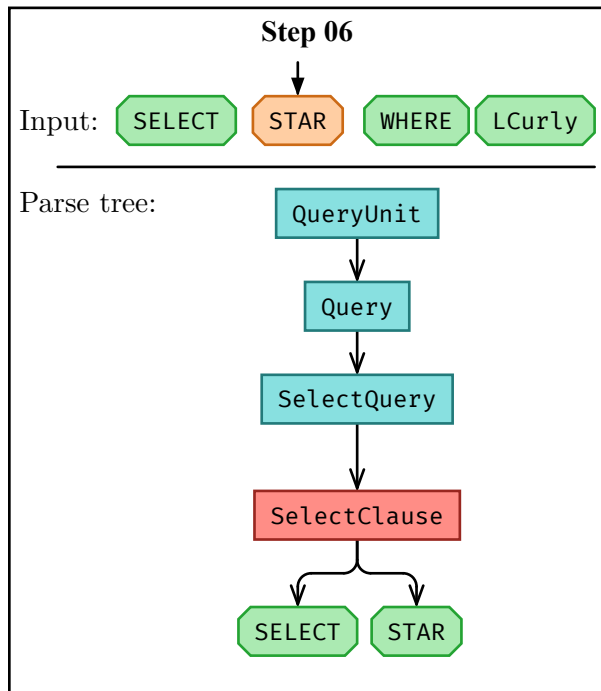
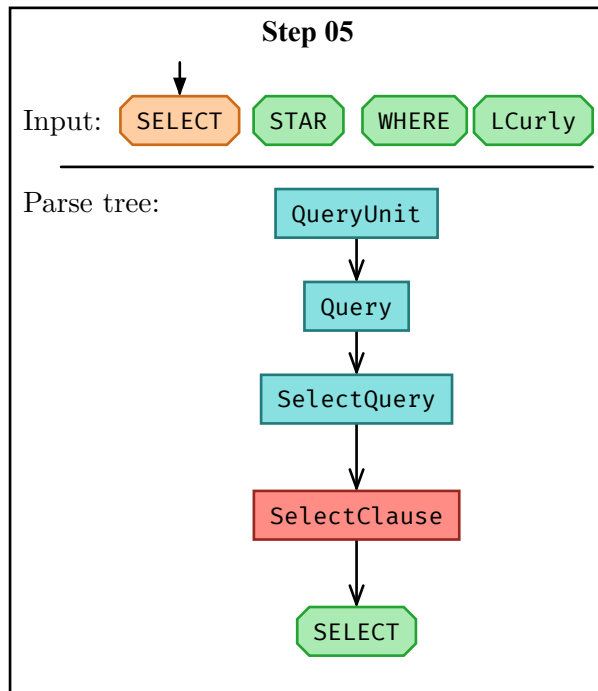
This is again a perfect fit for my use-case. If the input unexpectedly ends or if there is an error token, my algorithm will parse everything up to this point.

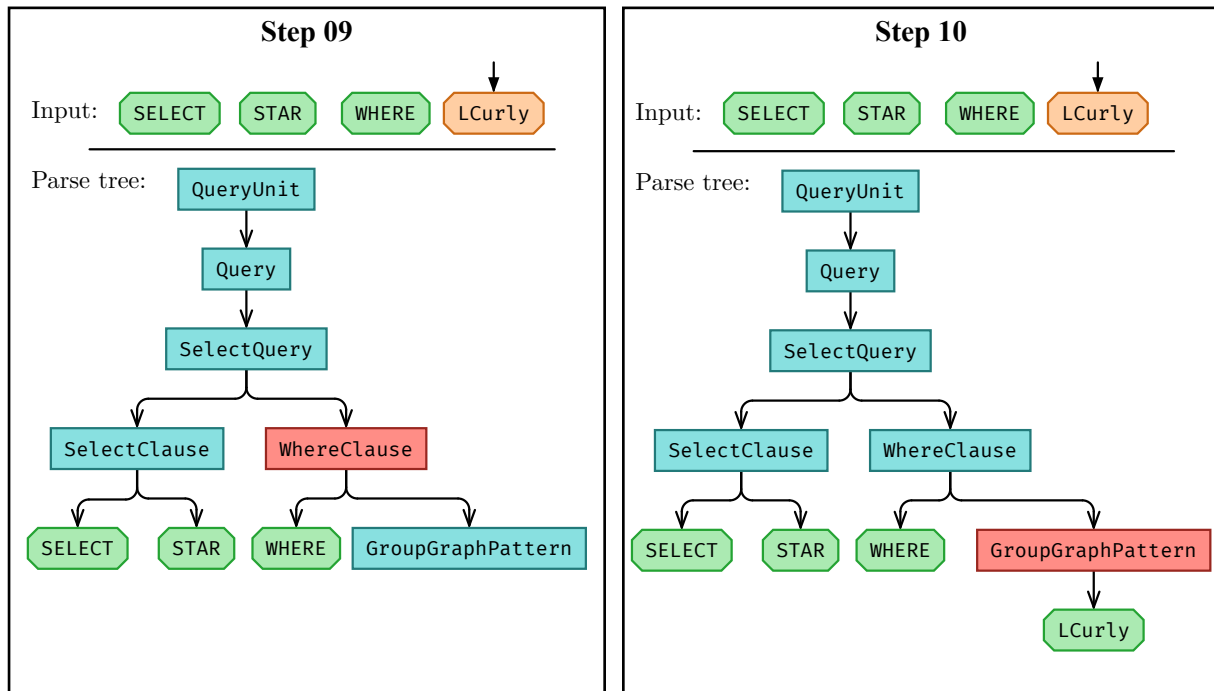
Here is for example how the parse tree gets built step by step.

The **orange** octagon represents the *lookahead* token.

The **red** box represents the currently active node in the parse tree.







Note how the input is incomplete, but I still get a valid partial parse tree.

### 5.5.1.3 Leftmost Derivation

I defined in Section 5.1.3 what a leftmost derivation is.

The constructed automaton performs leftmost derivations because of how sequences are handled. When there is a sequence of nonterminals the automaton handles them from left to right, therefore performing leftmost derivations.

## 5.5.2 Operation Identification

The *SPARQL* grammar is composed of 2 formal grammars:

- *SPARQL* 1.1 Query Language, with *start symbol* QueryUnit
- *SPARQL* 1.1 Update, with *start symbol* UpdateUnit

My algorithm can parse both. To parse a *SPARQL* query input, call `parse_QueryUnit` and to parse a *SPARQL* update input, call `parse_UpdateUnit`.

To parse any *SPARQL* input, I need to decide if the given input is a *SPARQL* query or *SPARQL* update input.

Here are the first two rules of *SPARQL* query:

```

QueryUnit ::= Query
Query ::= Prologue
        ( SelectQuery | ConstructQuery | DescribeQuery | AskQuery )
        ValuesClause
  
```

And here are the two rules of *SPARQL* update:

```

UpdateUnit ::= Update
Update ::= Prologue ( Update1 ( ';' Update )? )?
  
```



Both languages start with Prologue. The nonterminal Prologue can produce the empty word  $\varepsilon$ . After Prologue *SPARQL* query continues with ( SelectQuery | ConstructQuery | DescribeQuery | AskQuery ). This rule can't produce  $\varepsilon$  and has the first set:

$$\text{first\_set\_query} = \{\text{'SELECT'}, \text{'CONSTRUCT'}, \text{'ASK'}, \text{'DESCRIBE'}\}$$

*SPARQL* update follows Prologue with Update1, which has the first set:

$$\begin{aligned} \text{first\_set\_update} = \{ & \text{'LOAD'}, \text{'CLEAR'}, \text{'DROP'}, \text{'CREATE'}, \text{'ADD'}, \\ & \text{'MOVE'}, \text{'COPY'}, \text{'INSERT'}, \text{'INSERT\_DATA'}, \\ & \text{'DELETE'}, \text{'DELETE\_DATA'}, \text{'DELETE\_WHERE'}, \varepsilon \} \end{aligned}$$

To decide what operation the input is, I iterate over the input tokens until I find a token from either of those sets.

If it is in set *first\_set\_query*, the input is in *SPARQL* query, if it is in set *first\_set\_update* it is in *SPARQL* update. If no token is in either of those sets, it can't be decided.

## 6 Capabilities

In Section 5 I have shown in depth how the heart of Qlue-Is – the parser – works. In this section I will show how I use the parser to provide language support for *SPARQL*.

### 6.1 Completion

Given a cursor position in the input, the completion capability returns a list of suggestions. These suggestions are inserted at the cursor position, or replace a range in front of the cursor.

Completion requests can be sent from anywhere in the input string  $s$ .

```
1 SELECT * {
2   |
3 }
```

Listing 3: *SPARQL* query with cursor at the start of a *GroupGraphPattern*

```
1 SELECT * {
2   }
3   |
```

Listing 4: *SPARQL* with the cursor after the *SelectQuery*

For example in Listing 3 the cursor is where a triple would start.

Here valid suggestions would be: variables, IRIs or constructs like *FILTER* or *OPTIONAL*. But in Listing 4 the cursor is right after the *SelectQuery*. Here none of the suggestions above would be legal but instead only *SolutionModifier* like *ORDER BY* or *GROUP BY* and a *ValuesClause* would work.

The difference between those positions are the possibilities of how the prefix  $t$  could be continued.

In the next section I will provide a formalization for this idea.

The outline of the completion algorithm looks like this:

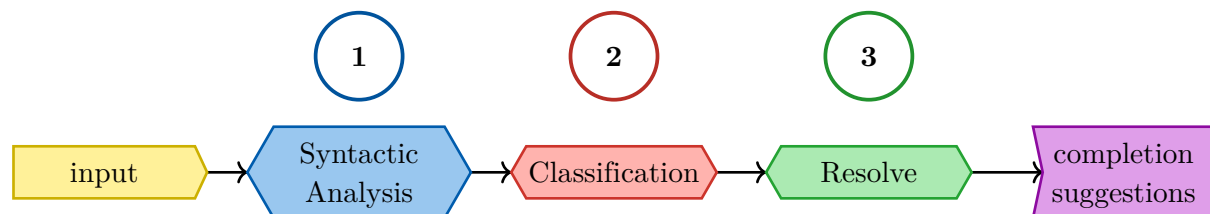


Figure 39: Diagram of the completion algorithm

**Input:** The input is a sequence of UTF-8 bytes  $s$  and a natural number  $c \in [0..|s|]$ .

**Step 1:**  $s$  is tokenized and parsed into a parse tree.

**Step 2:** The input is classified, each input gets a **syntactic location** assigned.

**Step 3:** Based on the *syntactic location*, the completion suggestions are computed.

**Step 4:** The output is a list of suggestions.

In the next 2 sections I focus on step 2 and 3.

#### 6.1.1 Syntactic Location Classification

The idea here is to put all inputs that can be resolved the same way into one bucket.

Inputs can be resolved the same way if they can be continued the same way.

Let  $s$  be the input string and  $c \in [0, |s|]$  be the cursor position.

Let  $W = (w_0, w_1, \dots, w_n)$  be the sequence of tokens produced by lexing  $s$ .

For each  $w \in W$ :

- $\text{range}(w) \in \{[a, b) \mid a, b \in [0..|s|] \wedge a \leq b\}$  is a tuple s.t.  
 $s[\text{range}(w).\text{first}() : \text{range}(w).\text{last}()]$  is the string slice of the lexeme of  $w$  in  $s$ . Ranges are inclusive-exclusive, meaning the range  $(0, 0)$  does not include  $s[0]$  but  $(0, 1)$  does.
- $\text{type}(w) \in \Sigma \cup \{\text{Error}\}$ , where  $\Sigma$  is the set of tokens of the *SPARQL* grammar.

Tokens are constructed s.t. their ranges are consecutive and non-overlapping.

Since our input can be incomplete or contain errors, I introduce a special token: Error.

Let  $tree$  be the parse tree of  $s$ , and  $\text{parent}(v)$  be the parent node of a node  $v \in tree$ .

The cursor position  $c$  lies at one of the following positions relative to the token sequence:

1. Before the range of the first token  $w_0$
2. Within the range of some token  $w_i$
3. Between the ranges of two consecutive tokens  $w_i, w_{i+1}$
4. After the range of the last token  $w_n$

Define the **trigger token**  $w_{\text{trigger}}$  as:

$$w_{\text{trigger}} = \begin{cases} w_0 & \text{if } c \leq \text{range}(w_0).\text{first}() \\ w_i & \text{if } \text{range}(w_i).\text{first}() < c < \text{range}(w_i).\text{last}() \\ w_i & \text{if } \text{range}(w_i).\text{last}() \leq c \leq \text{range}(w_{i+1}).\text{first}() \\ w_n & \text{if } c \geq \text{range}(w_n).\text{last}() \end{cases}$$

The **anchor token** is defined as:

$$w_{\text{anchor}} = w_j \text{ where } \max\{j \mid j < \text{index}(w_{\text{trigger}}), \text{type}(\text{parent}(w_j)) \neq \text{Error}\}$$

In natural language: The *anchor token* is the nearest preceding token of  $w_{\text{trigger}}$  whose parent node in  $tree$  is not an *Error* node.

### Example:

Given the input:

```
1 SELECT * WHERE {
2   FREI
3 }
```

Listing 5: Incomplete *SPARQL* query. The cursor – marked in red – is right after the word “Frei”.

The cursor at position 23, as shown in Figure 40.

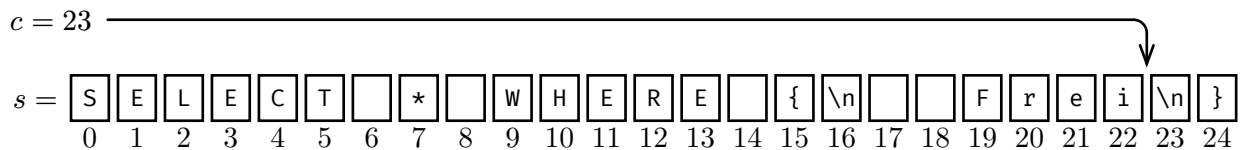


Figure 40: The input from Listing 5 visualized as a sequence of chars.

“Frei” is not a valid *SPARQL* lexeme and will get recognized as *Error*. The cursor position  $c$  is between the ranges of the tokens  $w_4$  and  $w_5$ . Therefore, the *trigger token*  $w_{\text{trigger}}$  is  $w_4$ , marked red in Figure 41.

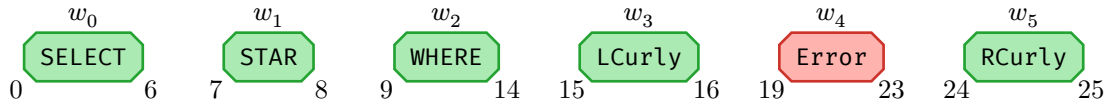


Figure 41: The input from Listing 5 visualized as a sequence of chars.

The *anchor token* is the  $w_3$  marked yellow in Figure 42.

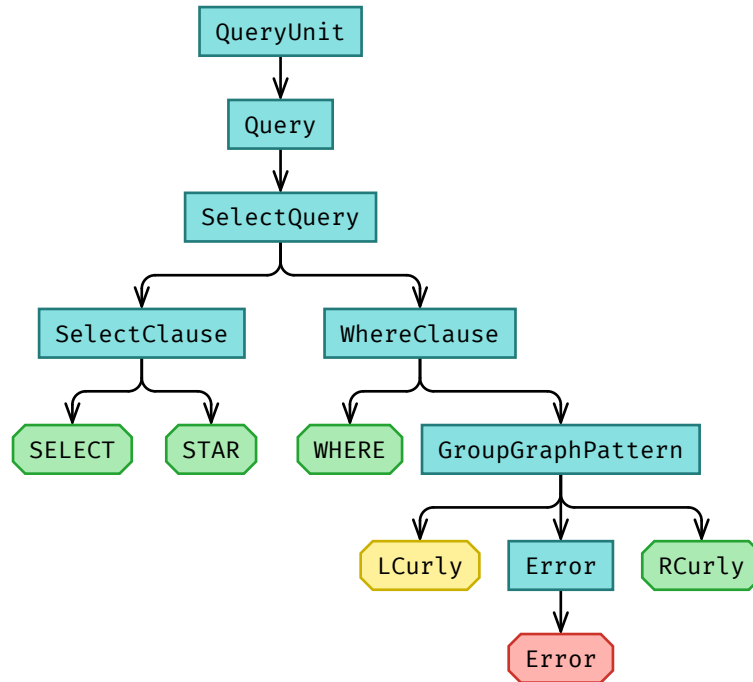


Figure 42: A parse tree for the input from Listing 5.

The *trigger token* marked red, and the *anchor token* marked yellow.

The token  $w_{\text{anchor}}$  is the starting point for the completion. The substring from the end of  $w_{\text{anchor}}$  up to  $c$  is the text that will get replaced by the suggestions.

**Definition (completion prefix & search term):**

$$\begin{aligned} \text{completion prefix} &:= s[0 : \text{range}(w_{\text{anchor}}).\text{last}()] \\ \text{search term} &:= s[\text{range}(w_{\text{anchor}}).\text{last}() : c] \end{aligned}$$

### Example

For the input Listing 5:

- the *completion prefix* is: “SELECT \* WHERE {“
- the *search term* is: “\n Frei”

### Definition (continuations set)

Let  $p$  be the *completion prefix* for  $s$  and  $c$ .

Let  $W' = (w_1, \dots, w_{\text{anchor}})$  be the sequence of tokens produced by lexing  $p$ .

By definition  $w_{\text{anchor}}$  is a non-*Error* token and child of a non-*Error* node.

For simplicity I assume that  $p$  is a valid prefix of a *SPARQL* query.

That implies that there exists an error-free partial parse tree for  $p$ , where the rightmost leaf node is  $w_{\text{anchor}}$ .

Let  $\mathcal{S}$  be every possible *SPARQL* query.

Let  $\mathcal{S}_p = \{pr \mid pr \in \mathcal{S}\}$  be every *SPARQL* query starting with  $p$ .

Let  $\mathcal{T}_p = \{\text{tree}(x) \mid x \in \mathcal{S}_p\}$  be the parse trees of queries in  $\mathcal{S}_p$ .

In every tree  $t \in \mathcal{T}_p$ , the leftmost leaves are exactly  $W'$ .

Let  $\text{next}(t, w)$  denote the node that immediately follows  $w$  in an in-order traversal of the parse tree  $t$ .

Then the **continuations set** is:

$$\{\text{type}(\text{next}(t, w_{\text{anchor}})) \mid t \in \mathcal{T}_p\}$$

To summarize:

- The *completion prefix* is the fixed part of the input that will remain unchanged.
- The *search term* is what the user typed after the *completion prefix*.
- The *continuations set* is every possible grammatical symbol that could continue the parse tree of the *completion prefix*

### Example:

For the input Listing 5 the *continuations set* is  $\{\text{GroupGraphPatternSub}, \text{SubSelect}, \text{RCurly}\}$ .

Figure 43 shows the partial parse tree for  $p = \text{"SELECT * WHERE \{"}$ .

Figure 44 shows a possible continuation of this tree where the next node is *GroupGraphPatternSub*.

Figure 45 shows a possible continuation of this tree where the next node is *SubSelect*.

Figure 46 shows a possible continuation of this tree where the next node is *RCurly*.

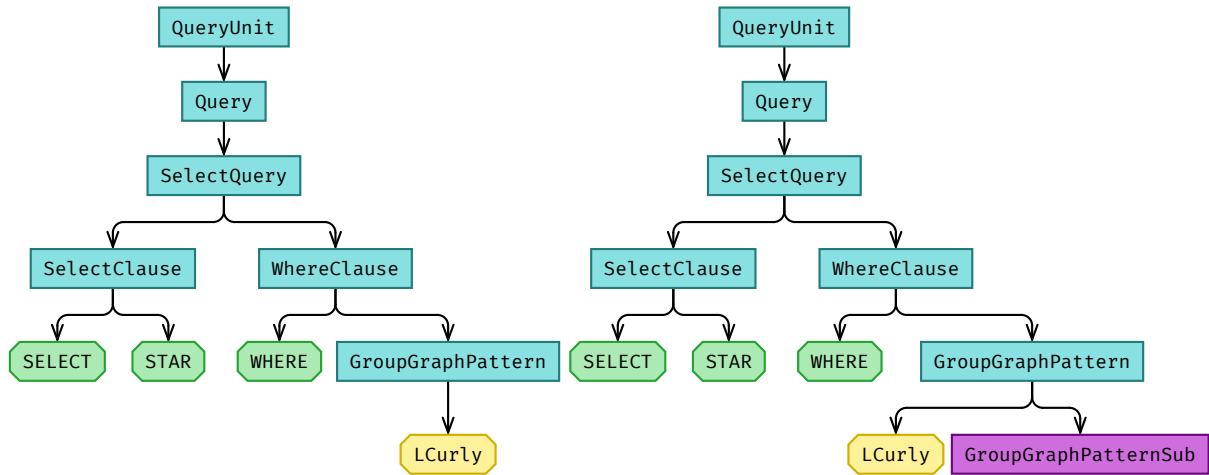


Figure 43: Partial parse tree for *completion prefix*  $p = \text{"SELECT * WHERE \{"}$       Figure 44: Continuation of the tree in Figure 43.

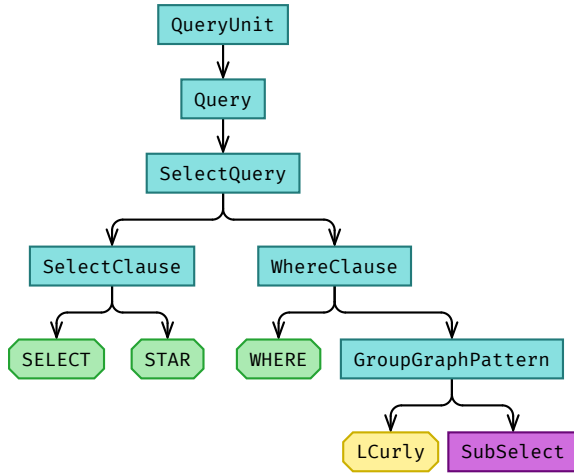


Figure 45: Continuation of the tree in Figure 43.

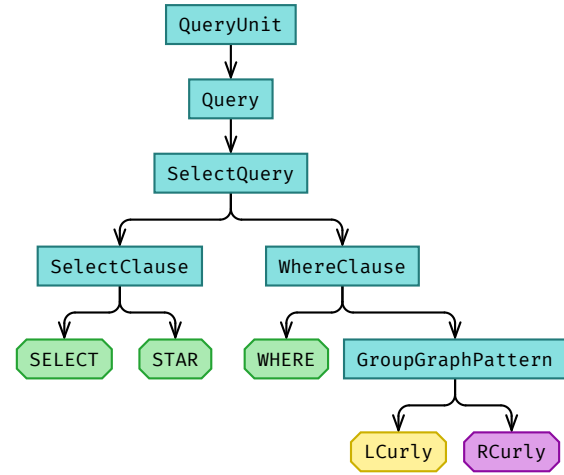


Figure 46: Continuation of the tree in Figure 43.

Given input  $s$  and  $c$  I define the *syntactic location* via the Table 2.

The *syntactic location* in the first column is assigned if the *continuations set* of  $p$  contains **any** of the symbols in the second column.

If the *continuations set* of  $p$  contains **none** of the symbols in the second column, the *syntactic location* is unknown.

Syntactic location	Continuations	example
Start	Prologue	1 <span style="border: 1px solid red; padding: 0 2px;"> </span>
Subject	GroupGraphPatternSub TriplesBlock GraphPatternNotTriples DataBlockValue GraphNodePath	1 SELECT * { 2 <span style="border: 1px solid red; padding: 0 2px;"> </span> 3 }
Predicate	PropertyListPathNotEmpty PropertyListPath Path VerbPath VerbSimple PathEltOrInverse PathSequence PathElt PathNegatedPropertySet PathOneInPropertySet PathAlternative	1 SELECT * { 2   ?a ?b ?c; 3 <span style="border: 1px solid red; padding: 0 2px;"> </span> 4 }
Object	ObjectListPath ObjectPath ObjectList Object	1 SELECT * { 2   ?a ?b ?c; 3       ?d <span style="border: 1px solid red; padding: 0 2px;"> </span> 4 }
SolutionModifier	SolutionModifier HavingClause OrderClause LimitOffsetClauses LimitClause OffsetClause	1 SELECT * { 2   ?a ?b ?c; 3 } 4 <span style="border: 1px solid red; padding: 0 2px;"> </span>
GroupCondition	GroupCondition	1 SELECT * { 2   ?a ?b ?c; 3 } 4 GROUP BY <span style="border: 1px solid red; padding: 0 2px;"> </span>
OrderCondition	OrderCondition	1 SELECT * { 2   ?a ?b ?c; 3 } 4 ORDER BY <span style="border: 1px solid red; padding: 0 2px;"> </span>
FilterConstraint	Constraint	1 SELECT * { 2   FILTER ( <span style="border: 1px solid red; padding: 0 2px;"> </span> ) 3 }

## 6.1.2 Compute Completions

For each *syntactic location* a unique resolve strategy is used to provide completions.

### 6.1.2.1 Start Completions

If the *syntactic location* is Start, query templates for all 4 query types are suggested.

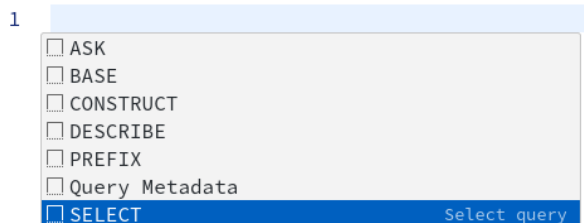


Figure 47: Suggestions for the *syntactic location* Start.

```
1 SELECT * WHERE {
2   ?s ?p ?o
3 }
```

Figure 48: After accepting the SELECT suggestion.

### 6.1.2.2 Subject Completions

For subject completions the *search term* is used to determine if the user is typing a variable or not. If the first non whitespace char of the *search term* is ? or \$, the user is typing a variable.

Otherwise the user is typing an IRI or a GraphPatternNotTriples. A GraphPatternNotTriples is for example a FILTER or OPTIONAL expression.

If the user is typing a variable, all variables in scope are collected and suggested.

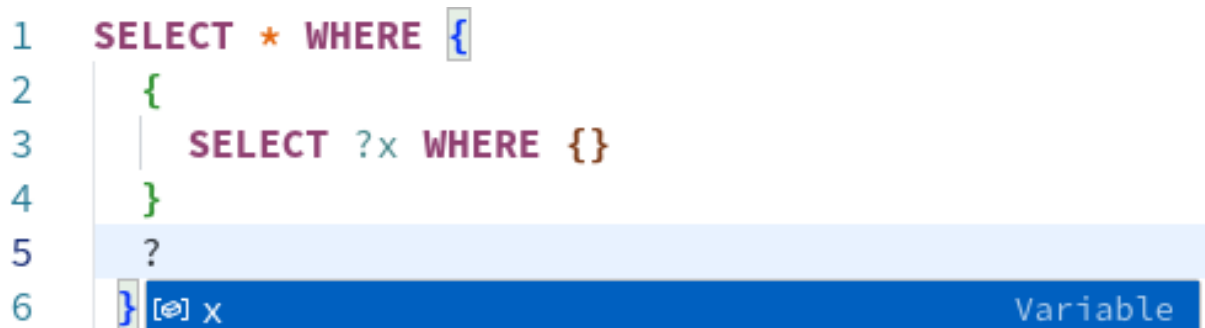


Figure 49: Variable completion

If the user is not typing a variable, GraphPatternNotTriples and IRI completions are mixed. Just like the Start completions, the completions for GraphPatternNotTriples are static.

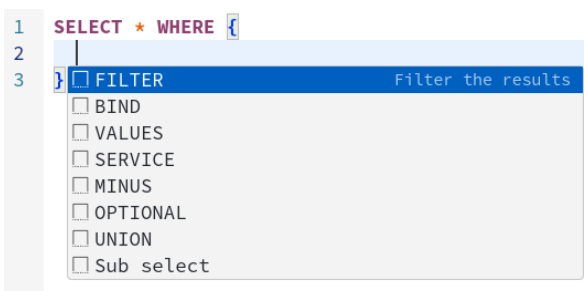


Figure 50: Static suggestions for the *syntactic location* Subject

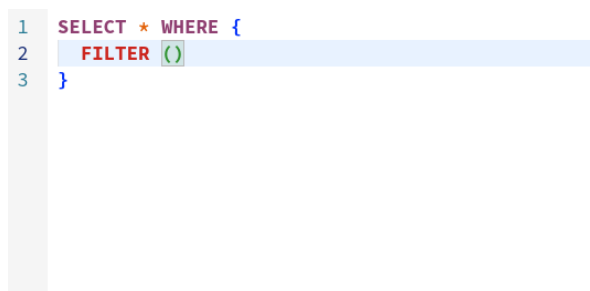


Figure 51: After accepting the FILTER suggestion



The IRI completions are more complex.

I implemented the “*SPARQL* Autocompletion via *SPARQL*” strategy as described in [4].

That means that a completion query is used to find matching IRIs in the knowledge graph.

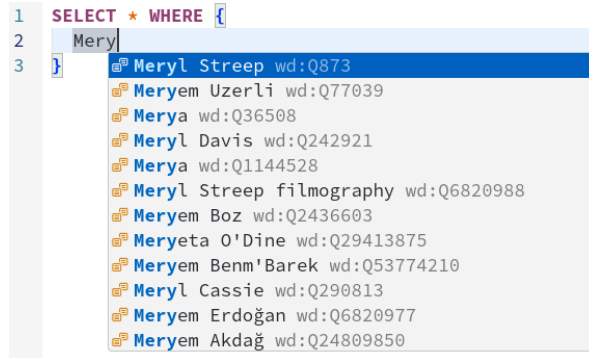


Figure 52: Suggestions for the *syntactic location* Subject after typing “Meryl”



Figure 53: After accepting the Meryl Streep suggestion

Here is the completion query used in Figure 52:

```

1  PREFIX wd:      <http://www.wikidata.org/entity/>
2  PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
3  PREFIX wikibase: <http://wikiba.se/ontology#>
4  PREFIX skos:    <http://www.w3.org/2004/02/skos/core#>
5  PREFIX schema:  <http://schema.org/>
6  SELECT ?qlue_ls_entity (SAMPLE(?name) AS ?qlue_ls_label) (SAMPLE(?alias) AS ?
   qlue_ls_alias) (SAMPLE(?sitelinks) AS ?qlue_ls_count) WHERE {
7    {
8      SELECT ?qlue_ls_entity ?name ?alias WHERE {
9        ?qlue_ls_entity rdfs:label ?name FILTER (LANG(?name) = "en")
10       ?qlue_ls_entity skos:altLabel ?alias FILTER (LANG(?alias) = "en")
11       FILTER (REGEX(STR(?name),"^Meryl") || REGEX(STR(?alias),"^Meryl"))
12     }
13   }
14   ?qlue_ls_entity ^schema:about/wikibase:sitelinks ?sitelinks
15 }
16 GROUP BY ?qlue_ls_entity
17 ORDER BY DESC(?qlue_ls_count)
18 LIMIT 101
19 OFFSET 0

```

The *search term* is used in line 11 to filter the results using the *search term*.

### 6.1.2.3 Predicate & Object Completions

At the *syntactic location* Predicate variables and IRIs are allowed.

Variable completions are computed the same way as before.

The other completions are again computed using “*SPARQL* Autocompletion via *SPARQL*”[4].

## 6.2 Formatting

The formatting capability transforms any *SPARQL* input into a formatted standard form.

There is no style guide for *SPARQL*. So I had to decide how a formatted *SPARQL* operation looks like. The configuration and the formatting algorithm define the standard form for any *SPARQL* query.

The formatting algorithm receives a UTF-8 string and returns a sequence of textedits.

First the input sequence is parsed.

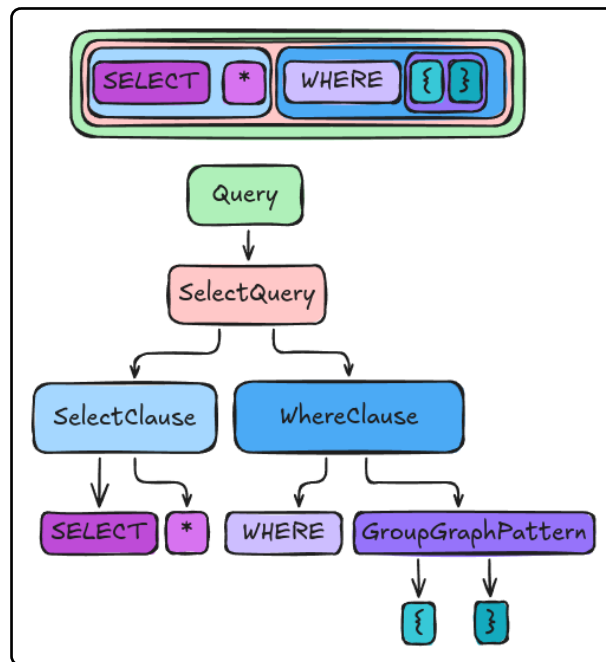


Figure 54: Parse tree for the input “SELECT \* WHERE {}”

Then the parse tree is traversed. For each node textedits are computed.

These textedits can be classified into two kinds:

- The separation edits insert textedits between the children of the node.
- The augmentation edits change the text inside the node.

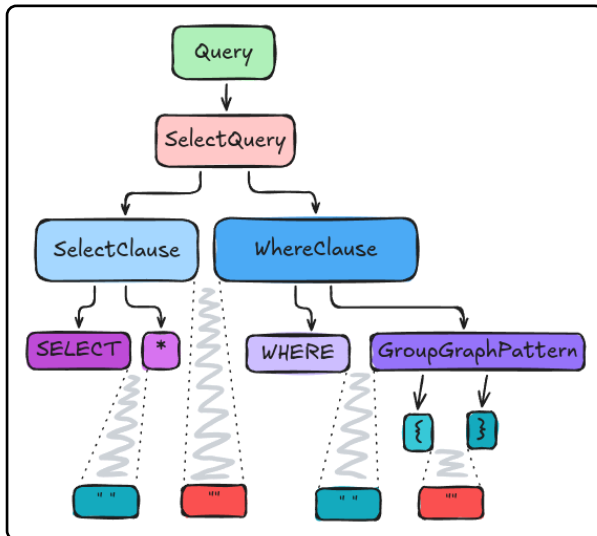


Figure 55: Separation edits for input  
“SELECT \* WHERE {}”

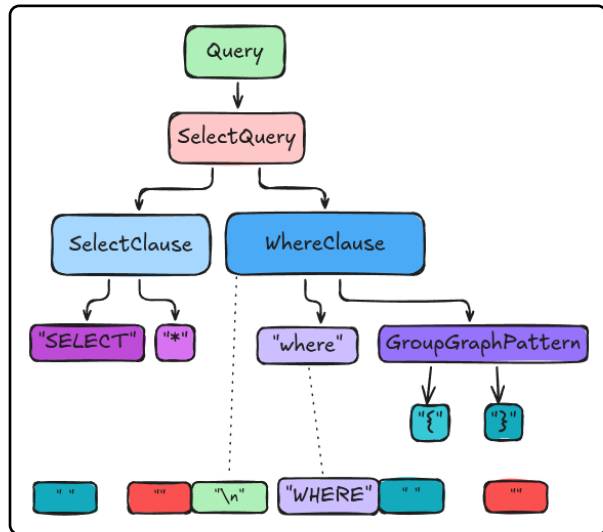


Figure 56: Augmentation edits for input  
“SELECT \* WHERE {}”

Finally the collected edits are ordered by start position.

When the edits are consecutive – the end and start overlap – edits are consolidated.

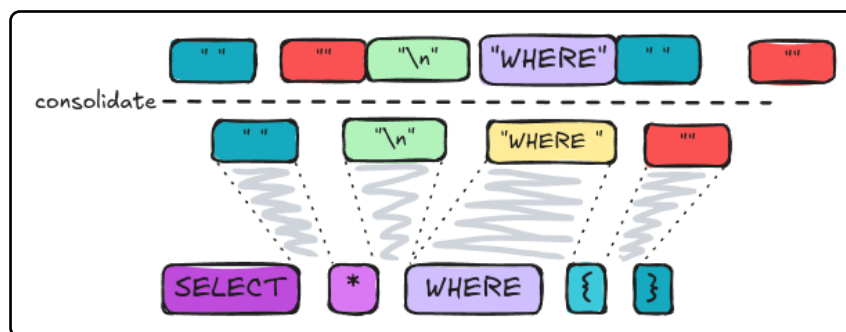


Figure 57: Consolidation and application of text edits.

Here is a collection of formatted examples:

<pre>SELECT * WHERE {   ?s ?p ?o .   ?a ?b ?c }</pre>	<pre>SELECT * WHERE {   {     SELECT * WHERE {}     GROUP BY (2 AS ?a)     HAVING (2 &gt; 2) (1 &gt; 2)     ORDER BY ASC(?c)     OFFSET 3     LIMIT 3   } }</pre>	<pre>SELECT * {   ?s ?p ?o   OPTIONAL {     ?a ?c ?c   } }</pre>
<pre>SELECT * {   ?s ?p ?o .   {}   ?a ?b ?c }</pre>	<pre>GROUP BY (2 AS ?a) HAVING (2 &gt; 2) (1 &gt; 2) ORDER BY ASC(?c) OFFSET 3 LIMIT 3</pre>	<pre>SELECT * {   ?a ?b ?c   {}   UNION {     {}     UNION {} .     ?a ?b ?c   } }</pre>
<pre>SELECT * WHERE {}</pre>		
<pre>SELECT * {   {}   MINUS {     {}     MINUS {}   } }</pre>		

<pre>PREFIX foaf: &lt;&gt; SELECT * WHERE {   ?P foaf:givenName ?G ;   foaf:surname ?S ;   ?p ?o ;   &lt;&gt; &lt;&gt; }</pre>	<pre>SELECT * {   {}   GRAPH ?a {     ?a ?b ?c   } }</pre>	<pre>SELECT * {   ?s ?p ?o   SERVICE &lt;iri&gt; {     ?a ?c ?c   } }</pre>
<pre>SELECT * {   FILTER (1 &gt; 0) }</pre>	<pre>PREFIX foaf: &lt;foaf/0.1/&gt; DESCRIBE ?x ?y &lt;dings&gt; WHERE {   ?x foaf:knows ?y }</pre>	<pre>INSERT {   ?v &lt;a&gt; &lt;b&gt; } WHERE {   VALUES ?v { 1 2 } }</pre>
<pre>SELECT * {   BIND (1 AS ?var) }</pre>	<pre>PREFIX foaf: &lt;foaf/0.1/&gt; ASK {   ?x foaf:name "Alice" }</pre>	<pre>DELETE DATA {   ?a ?b ?c .   GRAPH &lt;a&gt; {     ?c ?b ?a .     ?c ?b ?a   } .   ?d ?e ?f   GRAPH ?d {     ?a ?d ?c   }   ?d ?e ?f }</pre>
<pre>SELECT * {   VALUES ?a { 1 2 3 } }</pre>	<pre>PREFIX a: &lt;&gt; LOAD SILENT &lt;a&gt; INTO GRAPH &lt;c&gt; ; LOAD &lt;b&gt; ; CLEAR GRAPH &lt;b&gt; ; DROP GRAPH &lt;c&gt; ; ADD SILENT GRAPH &lt;c&gt; TO DEFAULT ; MOVE DEFAULT TO GRAPH &lt;a&gt; ; CREATE GRAPH &lt;d&gt;</pre>	<pre>SELECT * WHERE {   ?a &lt;iri&gt;/^a/(1&lt;&gt;)+   (&lt;iri&gt;   ^a   a) ?b }</pre>
<pre>PREFIX foaf: &lt;foaf/0.1/&gt; SELECT ?name ?x FROM &lt;a&gt; FROM &lt;b&gt; WHERE {   ?x foaf:name ?name }</pre>	<pre>PREFIX foaf: &lt;&gt; SELECT * WHERE {   ?P foaf:givenName ?G ;   foaf:surname ?S ;   ?p ?o ;   &lt;&gt; &lt;&gt; }</pre>	<pre># comment PREFIX test1: &lt;test&gt; # comment PREFIX test2: &lt;test&gt; # cmt SELECT ?a WHERE {   # comment   test1:a &lt;&gt; ?a . # comment   test2:b &lt;&gt; ?b .   ?b &lt;&gt; ?a .   # comment   {} # comment } # comment</pre>
<pre>CONSTRUCT {   &lt;Alice&gt; vcard:FN ?name } WHERE {   ?x foaf:name ?name } LIMIT 10</pre>	<pre>SELECT * WHERE {   &lt;subject&gt; &lt;predicate1&gt; [     &lt;predicate2&gt; &lt;object1&gt; ;     &lt;predicate3&gt; &lt;object2&gt;   ] }</pre>	
<pre>SELECT * WHERE {   &lt;s&gt; &lt;p1&gt; [ &lt;p2&gt; &lt;o&gt; ] }</pre>	<pre>SELECT * WHERE {   ?s # comment   ?p ?o }</pre>	
<pre>SELECT * WHERE {   FILTER (?a)   ?s ?p " , , " .   FILTER (?a)   ?a ?b ?c . FILTER (?a) }</pre>	<pre>SELECT * WHERE {   &lt;a&gt; &lt;b&gt; &lt;c&gt;, &lt;d&gt; }</pre>	

I wrote a deeper explanation of the formatting algorithm in my [blog post](#) [9].

## 6.3 Diagnostics

The diagnostics capability provides detailed feedback to the user. Each diagnostic consists of a range in the input, a severity level and a message.

Qlue-Is provides six diagnostics, discussed in the following sections.

### 6.3.1 Unused Prefix Declaration

The *unused-prefix-declaration* diagnostic detects prefix declarations that are unused.

Such prefix declarations make a query less readable while providing no benefit. However, because such queries are still valid, the severity of this diagnostic is: **Warning**.

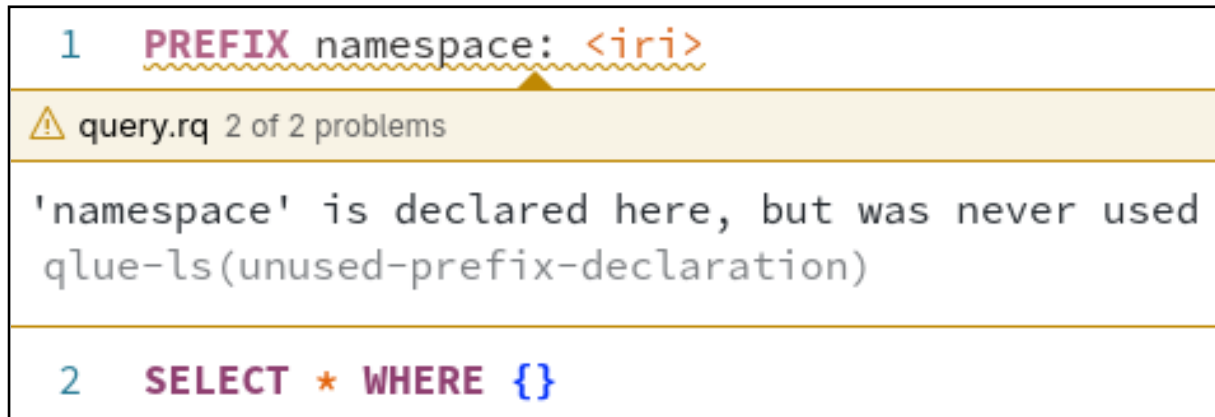


Figure 58: *SPARQL* query with an unused prefix. The diagnostic is underlining the prefix declaration and telling the user that this prefix is unused.

The algorithm to find unused prefixes starts with parsing the input.

Then it finds all *declared prefixes* and *used prefixes*, the difference of those two sets are the *unused prefixes*. To find the *declared prefixes* I traverse the subtree starting at the Prologue node in the parse tree. Every PNAME\_NS node, in this subtree, has to be a *declared prefix*.

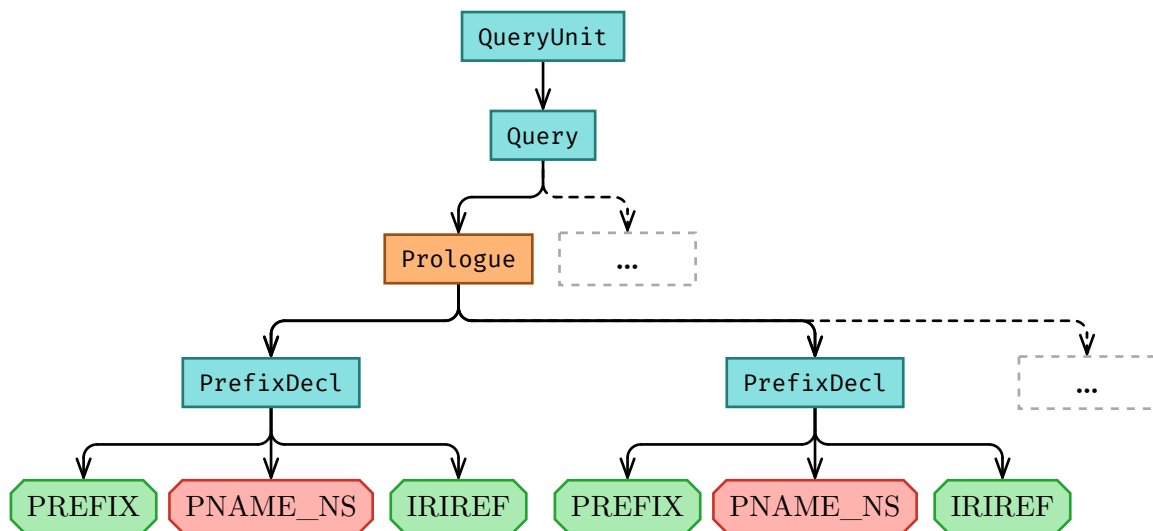


Figure 59: Parse tree of a *SPARQL* query. The Prologue node is highlighted orange, every PNAME\_NS dependent of Prologue is highlighted red.

To find the *used prefixes* the full parse tree is traversed. Every PrefixName node contains a used prefix.

### 6.3.2 Undeclared Prefix

The *undeclared-prefix* diagnostic detects prefixes that are used but not declared.

Such undeclared prefixes will result in an error. That's why the severity of this diagnostic is: **Error**.

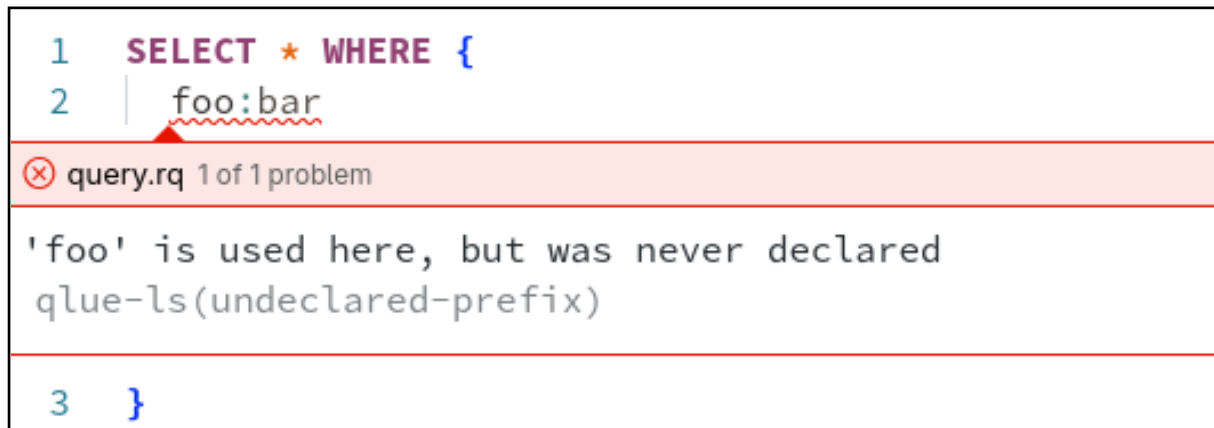


Figure 60: *SPARQL* query with an undeclared prefix. The diagnostic is underlining the undeclared prefix usage and telling the user that this prefix is undeclared.

The algorithm is almost the same as for *unused-prefix-declaration*. The two sets are just subtracted the other way around. The *used prefixes* minus the *declared prefixes* are the *undeclared-prefixes*.

### 6.3.3 Uncompressed IRI

Often *SPARQL* queries contain long unreadable IRIs.

For example `<http://www.wikidata.org/entity/Q5>`. *SPARQL* has the prefix mechanism to help with this. When a prefix is declared in the Prologue, it can be used to abbreviate the IRI:

```

1 PREFIX wd: <http://www.wikidata.org/entity/>
2 ...
3 wd:Q5

```

This makes *SPARQL* much more readable.

The *uncompressed iri* diagnostic detects raw IRIs that could be abbreviated using a prefix.

Raw IRIs are bad style, this is why the severity is: **Info**.



Figure 61: *SPARQL* query with the full IRI `<http://www.wikidata.org/entity/Q5>`. The diagnostic is underlining the IRI and telling the user that this IRI could get abbreviated with `wd:Q5`.

The language server needs a list of known prefixes.

These are either configured by the user or are shipped with the server.

The uncompressed IRIs are retrieved from the parse tree by scanning the parse tree except the Prologue sub-tree.

If the IRI is listed in the Prologue, or if the IRI has a known prefix the diagnostic is created.

### 6.3.4 Ungrouped Select Variable

When a *SPARQL* query has a `GROUP BY` solution modifier, only variables in the `GroupClause` are allowed to be selected.

The *ungrouped-select-variable* diagnostic detects selected variables that are not in the `GroupClause`. Selecting variables that are not in the `GroupClause` will cause an error. That's why the severity for this diagnostic is: **Error**.

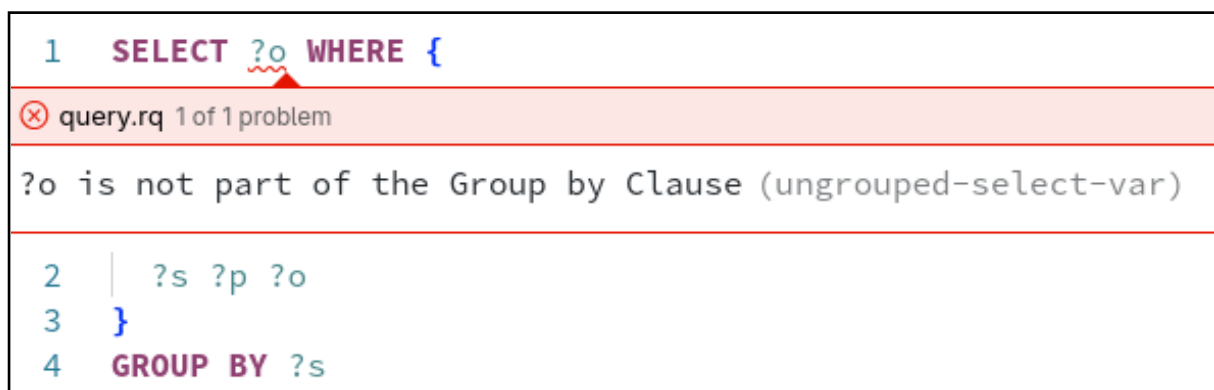


Figure 62: *SPARQL* query with a `GroupClause` containing `?s` and a `SelectClause` containing `?o`. The diagnostic is underlining the selected variable `?o` and telling the user that this selection is illegal.

The algorithm is straight forward when the parse tree is given.

For every `SelectQuery` or `SubSelect` compare the variables in the `SelectClause` and the `GroupCondition`.

The *selected variables* minus the *grouped variables* are the selected variables that are ungrouped.

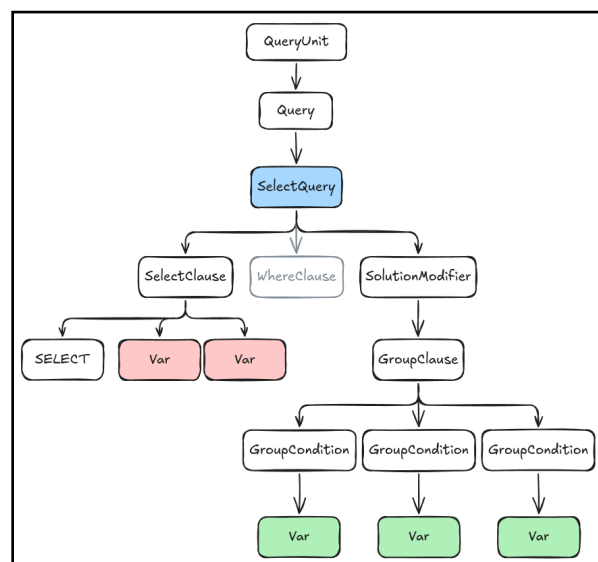


Figure 63: The parse tree for a *SPARQL* with two **selected variables** and 3 **grouped variables**.

### 6.3.5 Invalid Projection Variable

In the `SelectClause` one can bind terms to variables. For example:

```
1 SELECT (42 as ?x) WHERE {}
```

The *invalid-projection-variable* diagnostic detects if a variable that has been assigned in a `SelectClause` has already been declared in the body of the `SelectQuery`. Such a query is illegal and therefore the severity level is: **Error**.

```

1  SELECT (42 as ?s) WHERE {
2  |   ?s ?p ?o
3  | }

```

query.rq 1 of 1 problem

?s is already defined in the query body (invalid-projection-var)

Figure 64: *SPARQL* query with 42 assigned to `?s` in the `SelectClause` and also the triple `?s ?p ?o` in the query body. The diagnostic is underlining the assigned variable `?s` and telling the user that this assignment is illegal.

The algorithm also uses the parse tree to find every `SelectQuery` and `SubSelect`. For each, check if a assigned variable is in the `WhereClause`.

### 6.3.6 Same Subject

Often many triples share the same subject. *SPARQL* provides a notation that allows to only write the subject once for many triples.

Here is a example for this notation:

<pre> 1  ?s ?p1 ?o1 . 2  ?s ?p2 ?o2 . 3  ?s ?p3 ?o3 </pre>	<pre> 1  ?s ?p1 ?o1 ; 2    ?p2 ?o2 ; 3    ?p3 ?o3 </pre>
--	--

The *same-subject* diagnostic detects triples that share the same subject.

```

1  SELECT * WHERE {
2  |   ?s ?p1 ?o1 .
3  |   ?s ?p2 ?o2 .
4  | }

```

query.rq 2 of 2 problems

Triple with same subject "?s" can be contracted (same-subject)

Figure 65: The *SPARQL* query has two triples. Both start with the same subject `?s`. The diagnostic informs the user that there is a better way of writing this.



## 6.4 Code Actions

The code action *capability* receives an input string  $s$  and a range in the string  $r = (c_1, c_2)$ , where  $c_1, c_2 \in [0, |s| + 1]$  and  $c_1 \leq c_2$ .

It returns a set of code actions. A code action is a label and a set of text edits.

The deepest node in the parse tree of  $s$  that fully contains  $r$  is the *covering element* of  $r$ .

Qlue-ls provides six diagnostics:

### 6.4.1 Add Variable to Result

The *add to result* code action is returned if the covering element of  $r$  has type Var.

The text edits append the variable to the SelectClause. Thus adding the variable to the result.

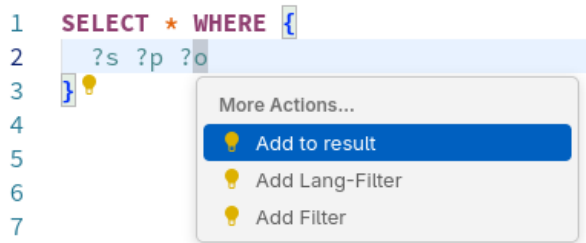


Figure 66: *SPARQL* query with `SELECT *` as `SelectClause` and the code action menu.

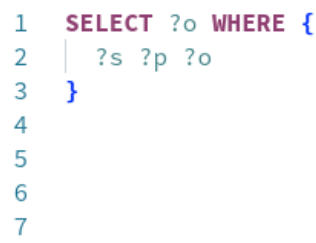


Figure 67: *SPARQL* query after the code action, with `SELECT ?o` as `SelectClause`.

The implementation for this code action uses the parse tree to find the next `SelectQuery` or `SubSelect` node in the ancestors of the covered element. If the `SelectClause` of this node does not already contain a variable that matches the covered element, a text edit is created that adds the variable to the end of the `SelectClause`.

### 6.4.2 Add Aggregate to Result

The *add aggregate to result* code action is also returned if the covering element of  $r$  has type Var.

If the `SelectQuery` or `SubSelect` that contains the node has a `GroupClause`, this code action adds an aggregate to the `SelectClause`.

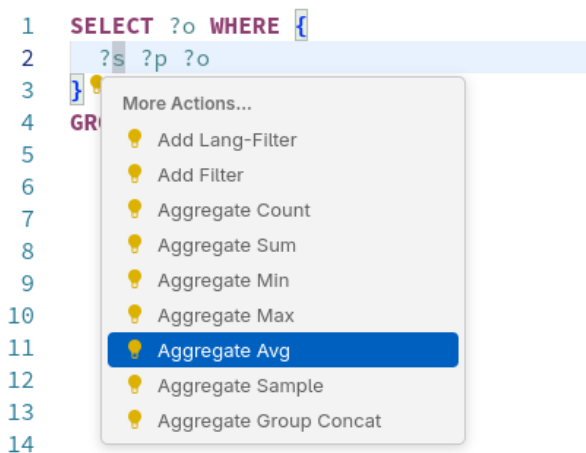


Figure 68: *SPARQL* query with `GroupClause` and no aggregate in the `SelectClause`.



Figure 69: *SPARQL* query after the code action, with new aggregate in the `SelectClause`

The implementation also finds the next `SelectQuery` or `SubSelect` ancestor and creates a text edit that adds the aggregate functions to the end of the `SelectClause`.

### 6.4.3 Transform Into Sub-Select

The *transform into sub-select* code action transforms a `SelectQuery` into a `SubSelect`.

The covering element of *r* needs to be a `SelectQuery` or `SubSelect`.

```
1 SELECT ?s (42 as ?x) WHERE {
2
3
4
5
6
7
```

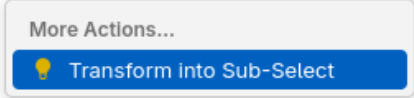


Figure 70: *SPARQL* query with `?s as (42 as ?x)` in the `SelectClause`.

```
1 SELECT ?s ?x WHERE {
2 {
3   SELECT ?s (42 as ?x) WHERE {
4     ?s ?p ?o
5   }
6 }
7 }
```

Figure 71: *SPARQL* query after the code action, with new aggregate in the `SelectClause`.

The implementation is straight forward. The text range of the `SelectQuery` or `SubSelect` is contained in the parse tree. There are 2 text edits before and after this range that wrap the range in another `SelectQuery`. My implementation also indents the nested `SubSelect` properly. It also copies the selected variables of the inner `SubSelect` into the outer, as shown in Figure 71.

### 6.4.4 Add Filter

The *add filter* code action is triggered if the covering element of *r* is a `Var` node. If the variable is part of a triple, this code action adds a filter statement to the end of that triple.

```
1 SELECT * WHERE {
2   ?s ?p ?o
3 }
4
5
6
7
```

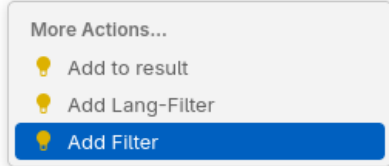


Figure 72: *SPARQL* query with no `Filter`.

```
1 SELECT * WHERE {
2   ?s ?p ?o FILTER (?o)
3 }
4
5
6
7
```

Figure 73: *SPARQL* query after the code action, with a `Filter` for the variable `?o`.

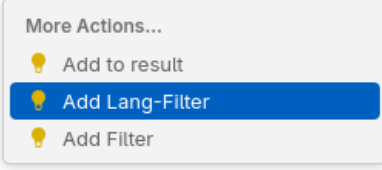
### 6.4.5 Add Language Filter

The *add language filter* code action is very similar to the *add filter* code action. The only difference is that it pre-fills the filter with a constraint on the language.

```

1 SELECT * WHERE {
2   ?s ?p ?o
3 }

```


Figure 74: *SPARQL* query with no Filter.

```

1 SELECT * WHERE {
2   ?s ?p ?o FILTER (LANG(?o) = "en")
3 }

```

Figure 75: *SPARQL* query after the code action, with a Filter on the language of ?o.

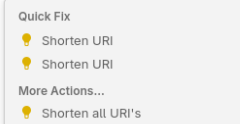
### 6.4.6 Compress IRI

The *compress IRI* code action is returned if the covering element of  $r$  is a *IRIREF* node. It transforms raw *IRIREF*s into *PrefixedNames*.

```

1 SELECT * WHERE {
2   ?s <http://www.w3.org/2000/01/rdf-schema#label> ?
3 }

```


Figure 76: *SPARQL* query with a raw *IRIREF*.

```

1 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2 SELECT * WHERE {
3   ?s rdfs:label ?label
4 }

```

Figure 77: *SPARQL* query after the code action, with the *IRIREF* replaced with a *PrefixedName*

This code action is tied to the *uncompressed iri* diagnostics, discussed in Section 6.3.3.

When this code action is applied, the diagnostic is resolved. Such a code action is called *quickfix*.

### 6.4.7 Declare Prefix

The *declare prefix* code action is the *quickfix* for the *undeclared prefix* diagnostic, discussed in Section 6.3.2. If an undeclared prefix is used, and the language server knows this prefix, this code action adds the prefix declaration.

```

1 SELECT * WHERE {
2   ?s rdfs:label ?label
3 }

```


Figure 78: *SPARQL* query with an undeclared prefix.

```

1 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2 SELECT * WHERE {
3   ?s rdfs:label ?label
4 }

```

Figure 79: *SPARQL* query after the code action, with the missing prefix declaration

### 6.4.8 Contract Triples

The *contract triples* code action is a *quickfix* for the *same subject* diagnostic, discussed in Section 6.3.6. It contracts triples with the same subject.

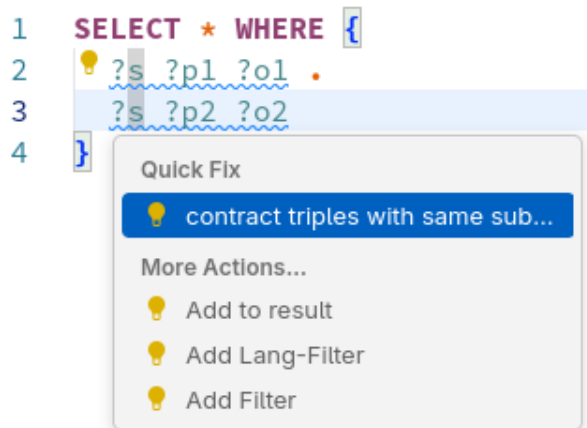


Figure 80: *SPARQL* query with an undeclared prefix.

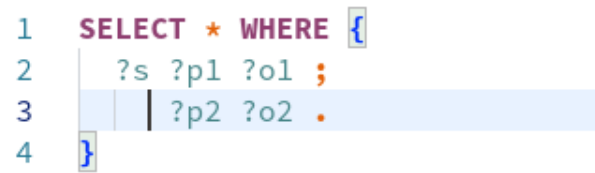


Figure 81: *SPARQL* query after the code action, with the missing prefix declaration.

## 6.5 Hover

The hover capability is triggered when the user hovers a lexeme. This capability returns text, to be displayed in the editor. The text can be raw text, markdown or HTML

Qlue-ls provides hover information in two cases:

1. When hovering a keyword, like `FILTER`
2. when hovering an IRI, like `wd:Q1`

The first case is quite simple.

If there is documentation of the hovered keyword, this text is returned.

Currently only the keywords `FILTER` and `PREFIX` are supported.

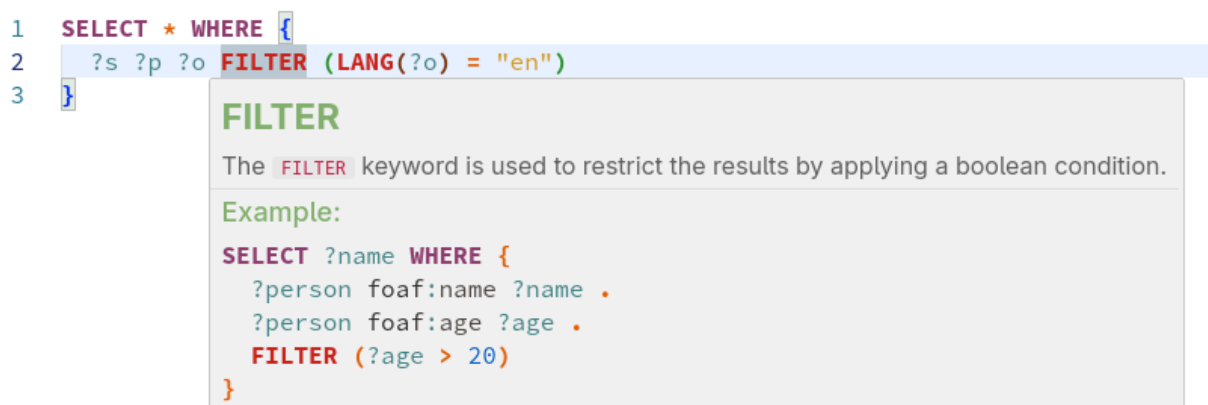


Figure 82: *SPARQL* query with hover information on the keyword `FILTER`.

```

1 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2
3 PREFIX
4 The PREFIX keyword defines a namespace prefix to simplify the use of URIs in the query.
Example:
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name
WHERE {
  ?person foaf:name ?name .
}

```

Figure 83: *SPARQL* query with hover information on the keyword PREFIX.

The second case is more complex.

When a user hovers an IRI, the capability should display text that contains additional information on this IRI. To obtain this additional information a *SPARQL* query is sent to the endpoint.

This query is configurable.

#### Example:

Given the query:

```

1 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2 PREFIX wd: <http://www.wikidata.org/entity/>
3 SELECT * WHERE {
4   wd:Q2 rdfs:label ?label
5 }

```

Listing 6: *SPARQL* query with the lexeme “wd:Q2” hovered. wd:Q2 is the resource for the earth.

The default query to get hover information for an IRI is:

```

1 PREFIX wd: <http://www.wikidata.org/entity/>
2 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3 SELECT ?qlue_ls_label WHERE {
4   wd:Q2 rdfs:label ?qlue_ls_label FILTER(LANG(?qlue_ls_label) = "en")
5 }

```

Listing 7: *SPARQL* query to get hover information on the IRI wd:Q2.

The result of this query on the wikidata backend should be:

?qlue_ls_label
Earth

This result is then returned as hover content, as shown in Figure 84.

```

1 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2 PREFIX wd: <http://www.wikidata.org/entity/>
3 SELECT Earth WHERE {
4   wd:Q2 rdfs:label ?label
5 }

```

Figure 84: *SPARQL* query with hover information “Earth” on IRI `wd:Q2`.

## 7 Theoretical Analysis

In this section I discuss the time complexity of the presented algorithms.

### 7.1 Parsing Algorithm

When running the automata presented in Section 5.5 each step does the following:

1. Compare the *lookahead* token to the outgoing edges of the current state.
2. Switch to the next state and update the *lookahead*.

The *FIRST* sets are precomputed. Checking if the *lookahead* token is in the *FIRST* set of a nonterminal can be implemented with Hash-Sets. This lookup has a runtime complexity of  $O(1)$ . Comparing a the *lookahead* token to another token has trivially a time complexity of  $O(1)$ .

Therefore each comparison takes some constant time  $T$ . How many outgoing edges a state has at most varies from grammar to grammar.

When building these automata for the *SPARQL* grammar, the state with the most outgoing edges is in the start state of the automaton for `BuiltInCall` with 55 edges.

So comparing the *lookahead* to all edges takes at most  $55 \cdot T$ .

Switching to the next state and updating the *lookahead* if necessary is trivial and takes some constant time  $B$ .

So any step requires at most  $55 \cdot T + B$ .

How many steps are done to reach the end of the input?

This also depends on the grammar.

Some grammars create automata s.t. the parser can get stuck in a loop:

```
1 A → (Aa)?
```

Listing 8: Grammar with left-recursion.

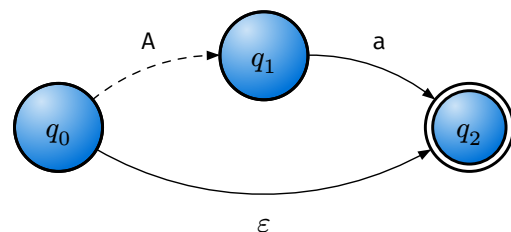


Figure 85: Automaton for grammar in Listing 8.

For the input “a” the parser will use these edges, starting from  $q_0$ :

$A \Rightarrow A \Rightarrow A \Rightarrow \dots$

The parser will never read from the input and never terminate.

To check that the created automaton for the *SPARQL* grammar does not create such loops, I looked for the longest possible walk through the automaton starting from every possible state.

On this walk only edges are used that are **not** annotated with a token.

If the longest path is finite, the parser can't get stuck in a loop.

This also gives us the maximum amount of steps the parser can do without reading from the input.

Turns out the longest possible walk without reading from the input starts in the automaton for *ArgList*, shown in Figure 36, and is 15 steps long.

Here are the edges used on this walk:

$\varepsilon \Rightarrow \text{Expression} \Rightarrow \text{ConditionalOrExpression} \Rightarrow \text{ConditionalAndExpression} \Rightarrow \text{ValueLogical} \Rightarrow \text{RelationalExpression} \Rightarrow \text{NumericExpression} \Rightarrow \text{AdditiveExpression} \Rightarrow \text{MultiplicativeExpression} \Rightarrow \text{UnaryExpression} \Rightarrow \text{PrimaryExpression} \Rightarrow \text{iriOrFunction} \Rightarrow \text{iri} \Rightarrow \text{PrefixedName}$

I conclude that for  $n$  input tokens the parser takes at most  $n \cdot 15$  steps until the input is read.

Therefore the total time parsing takes is at most:  $n \cdot 15 \cdot \underbrace{(55 \cdot T + B)}_{\text{constant}}$ .

The runtime complexity of the parsing algorithm is in  $O(n)$ .

In the best case scenario the parser reads a token every step.

This would take at least  $n \cdot T$ . Therefore the runtime complexity is also in  $\Omega(n)$ .

I conclude that the runtime complexity for parsing is in  $\Theta(n)$

## 8 Empirical Analysis

The purpose of this section is to assess the usefulness of *Qlue-ls* in comparison to existing tools.

I want to compare the feature completeness of each tool and the quality of the results.

I compared *Qlue-ls* to the following tools:

1. *Semantic web language server*, the language server presented in [7].
2. *QLever-UI*, the *SPARQL* UI for the *QLever* engine.
3. *YASGUI*, the SIB version of the *YASGUI SPARQL* editor.
4. *sparql formatter*, the formatter by sparkling.

Table 3 shows the capabilities each tool provides. Note that this table shows nothing about the quality of the provided capability.

*Qlue-ls* is the only tool that provides all of the 5 capabilities.

	Qlue-ls	semantic-web lsp	QLever-UI	YASGUI	sparqling formatter
Completion	✓	✓	✓	✓	✗
Formatting	✓	✗	✓	✓	✓
Diagnostics	✓	✓	✓	✗	✗
Code actions	✓	✗	✗	✗	✗
Hover	✓	✗	✓	✗	✗

Table 3: This table shows the capabilities each tool provides.

## 8.1 Completion

To evaluate the quality of the completions capability of *Qlue-ls* I evaluate 2 dimensions:

1. Coverage (How many different completions are available)
2. Performance (How much time does a completion take)

### 8.1.1 Coverage

To evaluate coverage I put together a list of different completions categories.

This list is the first column of Table 4. The other columns show what tools provide this category of completions. The completion categories are explained in Section 10.1

In Table 4 the first column lists all completion I tested, the next four columns show if the tools provide such completions or not.

Table 4 clearly shows that *Qlue-ls* has the best coverage across different completions.

*QLever-UI* is very close, just missing the *Predicate - Path* and *Service Aware* completions. This makes sense since the completion capability of *Qlue-ls* takes the *QLever-UI* implementation as blue print and improves on it.



	Qlue-ls	QLever-UI	SIB - YASGUI	semantic-web lsp
<a href="#">Variable</a>	✓	✓	✓	✓
<a href="#">Keyword</a>	✓	✓	✗	✓
<a href="#">Valid Keyword</a>	✓	✓	✗	✗
<a href="#">Snippet</a>	✓	✓	✗	✗
<a href="#">Select Clause - Variable</a>	✓	✓	✗	✗
<a href="#">Select Clause - Aggregate</a>	✓	✓	✗	✗
<a href="#">Subject</a>	✓	✓	✗	✗
<a href="#">Predicate - prefix</a>	✓	✓	✓	✓
<a href="#">Predicate - IRI</a>	✓	✓	✗	✗
<a href="#">Predicate - Path</a>	✓	✗	✗	✗
<a href="#">Object - prefix</a>	✓	✓	✓	✓ (for, classes)
<a href="#">Object - IRI or Literal</a>	✓	✓	✗	✗
<a href="#">Context Aware</a>	✓	✓	✓	✗
<a href="#">Service Aware</a>	✓	✗	✗	✗

Table 4: Comparison of the completion capability across tools

Table 5 compares the *Snippet* completions *Qlue-ls* and *QLever-UI* provide. This table also shows that *Qlue-ls* provides more *Snippet* completions.

	<b>Qlue-ls</b>	<b>QLever-UI</b>
Prologue - Prefix	✓	✓
Prologue - Base	✓	✗
Query - Select	✓	✓
Query - Construct	✓	✗
Query - Describe	✓	✗
Query - Ask	✓	✗
Solution Modifier	✓	✓
Update - Graph-management	✓	✗
Update - Delete	✓	✓
Update - Insert	✓	✓
UNION	✓	✓
OPTION	✓	✓
MINUS	✓	✓
FILTER	✓	✓
BIND	✓	✗
InlineData	✓	✗

Table 5: Comparison of the *Snippet* completions between *Qlue-ls* and *QLever-UI*

### 8.1.2 Performance

To evaluate the performance of the provided completions I group the completions into 3 categories.

- Offline completions (Completions that don't use the *SPARQL* endpoint).
- Schema-driven online completions (Completions that use the schema described in the knowledge graph).
- Data-driven online completions (Completions that use the data in the knowledge graph).

The time offline completions take is negligible.

Schema-driven completions usually fire one query in the beginning to retrieve the schema.

After this initial query the completions also basically instant.

Data-driven completions fire a query for each completion. These completions can take a lot of time. This depends on the size of the knowledge graph and the speed of the triple store.

*Qlue-ls* and *QLever-UI* use data-driven completions for the *Subject*, *Predicate* and *Object* completions. They tradeoff the time delay against quality of the results.

## 8.2 Formatting

There is currently no style guide or standard form for *SPARQL*, making an objective comparison difficult. Since *SPARQL* queries are relatively small, the time formatting takes is negligibly small. So to compare the formatting capability I tested each tool manually to find the key differences.

The *QLever-UI* uses *Qlue-ls* for formatting, so a comparison is not necessary.

*YASGUI* provides simplistic formatting. Their formatting algorithm only seems to remove newlines and indents inside `{}` blocks. It does not remove whitespace inside a triple or insert newlines in the Prologue. *SPARQL* queries formatted with this algorithm can in cases be very hard to read.

Listing 9 shows a query formatted with *YASGUI*. Listing 10 shows the same query formatted with *Qlue-ls*.

```

1  PREFIX ns1: <a#>
2  PREFIX ns2: <b#> PREFIX rdfs: <c#>
3  SELECT
4  ?a ?b WHERE {
5      ?a      ?b          ?d      .
6      ?a
7          ?d
8          ?c
9  }
10 LIMIT
11 10
```

Listing 9: *SPARQL* query formatted with *YASGUI*.

```

1  PREFIX rdf: <a#>
2  PREFIX rdfs: <b#>
3  PREFIX rdfs: <c#>
4  SELECT ?a ?b WHERE {
5      ?a ?b ?d .
6      ?a ?d ?c
7  }
8  LIMIT 10
```

Listing 10: *SPARQL* query formatted with *Qlue-ls*.

The formatting algorithm of *sparqling-formatter* is more sophisticated.

I found 3 differences to my formatting algorithm.

The *sparqling-formatter* is not error resilient. If the query is not valid, the formatting algorithm will throw an error.

Sometimes the algorithm inserts dots into the query. For example when the query in Listing 11 is formatted, a dot is inserted in line 3, as shown in Listing 12.

```

1 SELECT *
2 WHERE {
3   ?a ?b ?d
4 }

```

Listing 11: *SPARQL* query before formatting.

```

1 SELECT *
2 WHERE {
3   ?a ?b ?d .
4 }

```

Listing 12: *SPARQL* query formatted with *sparqling-formatter*.

Furthermore the algorithm throws errors for some valid queries, for example for the query shown in Listing 13.

```

1 SELECT *
2 WHERE {
3   FILTER
4   (?a in 1,2 )
5 }

```

Listing 13: *SPARQL* query that causes errors in the *sparqling-formatter* formatting algorithm.

To summarize, *Qlue-ls* has a more mature formatting capability than the *sparqling-formatter* or *YASGUI*. While the *sparqling-formatter* comes close, the fact that it throws errors for valid *SPARQL* queries makes it unreliable.

### 8.3 Diagnostics

The only other tool that supports this capability is the *Semantic web language server*, but it only provides one diagnostic.

*Qlue-ls* provides 6 diagnostics in total, including the diagnostic the *Semantic web language server* provides. The diagnostics *Qlue-ls* provides are presented in detail in Section 6.3.

### 8.4 Code Actions

Here the comparison is easy, since the other tools don't support this capability at all. *Qlue-ls* provides the code actions presented in Section 6.4.

### 8.5 Hover

Only the *QLever-UI* and *Qlue-ls* provide a proper hover capability.

When hovering IRIs, they implement the same strategy: Fire a custom hover query against the configured *SPARQL* endpoint and show the result.

But *Qlue-ls* also provides hover information when hovering keywords.

This makes the Hover capability of *Qlue-ls* slightly better.

## 9 Acknowledgements

I would like to thank Prof. Dr. Hannah Bast for the guidance during this project.  
 I would also like to thank my friends and family to their support.  
 A special thanks to Julian Mundhahs who always had an open ear.

I would also like to mention the book “Compilers: Principles, Techniques, and Tools” by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman .

It was a great help in figuring out the parser.

I would also like to thank Alex Kladov for his blog post “Resilient LL Parsing Tutorial” which was of great help in designing the parser.

## 10 Appendix

### 10.1 Completion categories

Here is a set of figures that show different completion categories. These categories are used in the evaluation of the completion capability of *Qlue-ls* (Section 8.1).

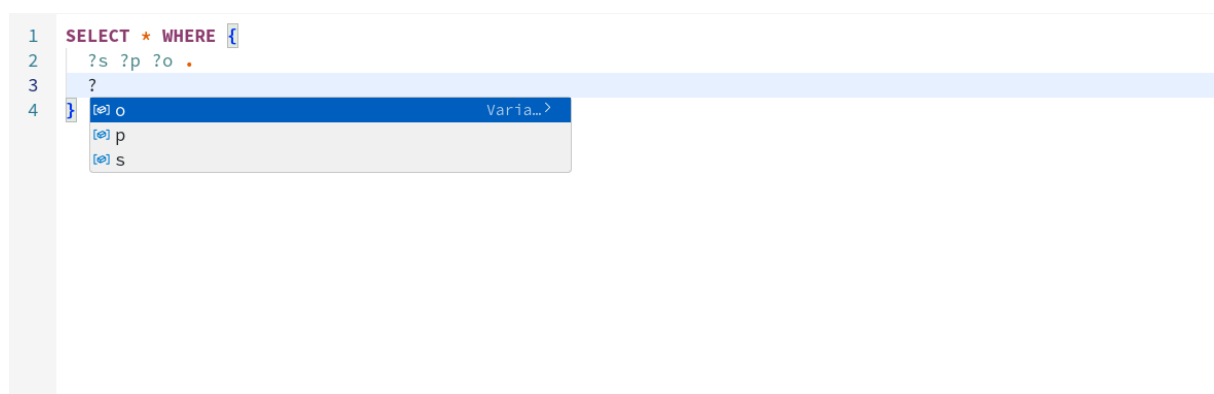


Figure 87: **Variable** completions suggest variables that are available within the scope of the triggered completion position. In the screenshot the completion is triggered in the WHERE CLAUSE of a SELECT QUERY right after a ?. The only variables available in the scope of this position are the variables in the first triple ?s ?p ?o. There variables are also the suggested completions.

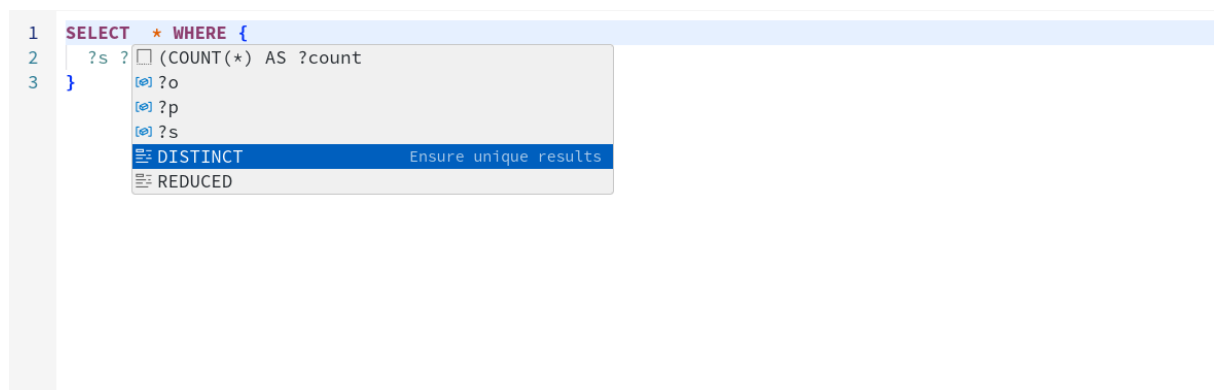


Figure 88: **Keyword** completions suggest *SPARQL* keywords.

**Valid Keyword** completions also suggest *SPARQL* keywords but they have to be valid at the completion trigger position. In the screenshot the completion is triggered in the SELECT CLAUSE in front of the star. Here the two keywords DISTINCT and REDUCED are valid and also suggested. Therefore these are **Valid Keyword** and **Keyword** completions.

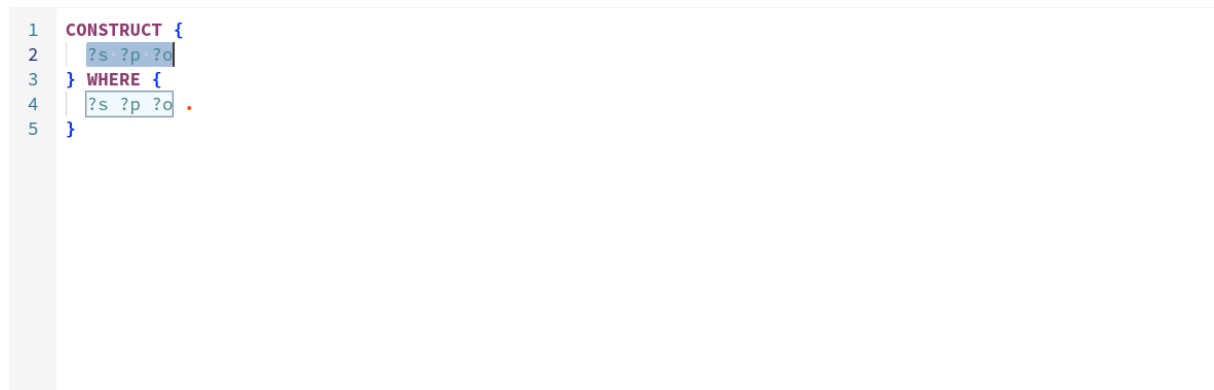


Figure 89: **Snippet** completions suggest a templates with predefined slots. The user can jump to these slots. They are useful to suggest constructs that are often used. In the screenshot the template for a CONSTRUCT query is suggested. This template contains two slots at the two triple blocks of the query.

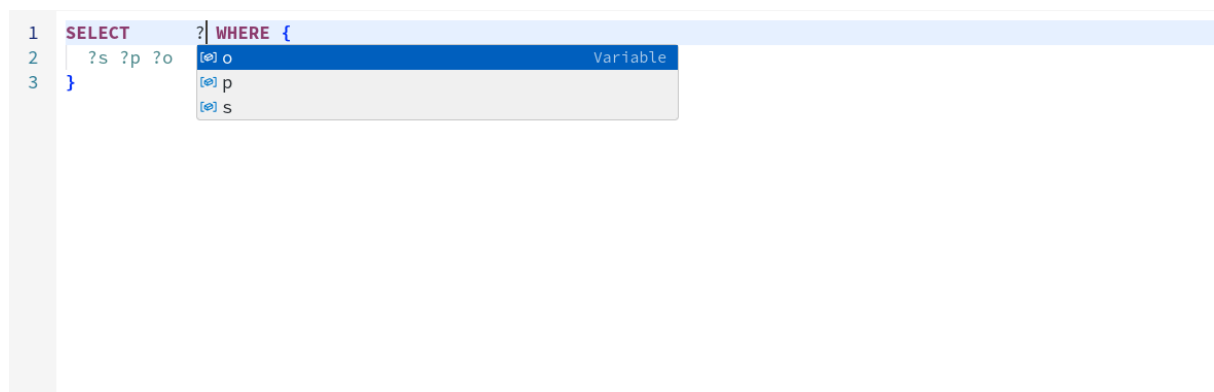


Figure 90: **Select Clause - Variable** completions are triggered in the SELECT CLAUSE. These completions suggest variables that are available in the WHERE CLAUSE and not already selected.

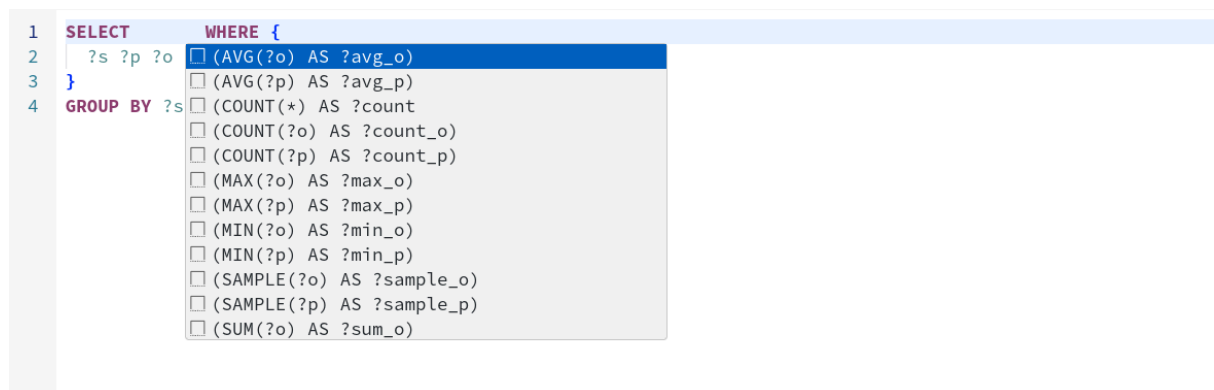


Figure 91: **Select Clause - Aggregate** completions are triggered in the SELECT CLAUSE. These completions suggest aggregate functions. A requirement is that the query has a GROUP BY solution modifier. In the screenshot the completion is triggered in the SELECT CLAUSE and there is a GROUP BY clause. The suggestions are aggregate bindings for all variables that are not in the GROUP BY clause.

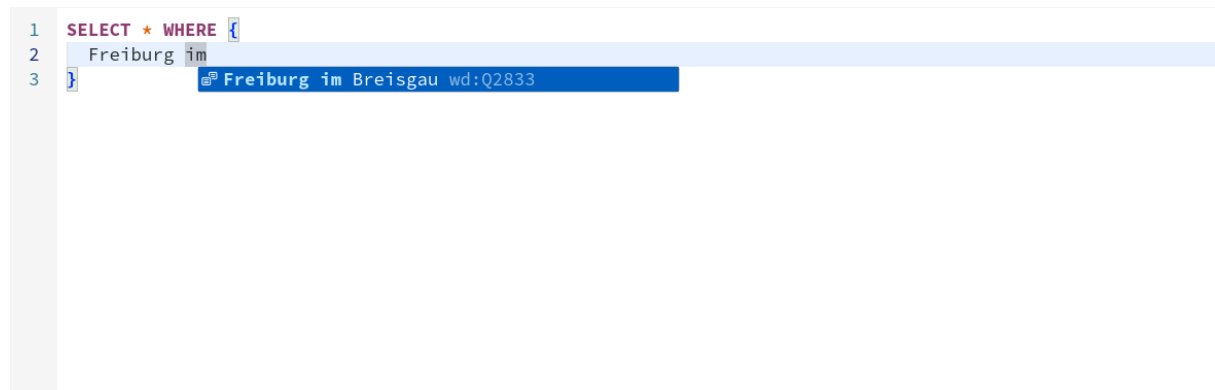


Figure 92: **Subject** completions are triggered at the first part of a triple. These completions use the *search term*, described in Section 6.1.1, and suggest entities that match this *search term*. In the screenshot the *search term* is "Freiburg im" and the suggestion is wd:Q2833. This entity has the label "Freiburg im Breisgau" and therefor matches the *search term*.

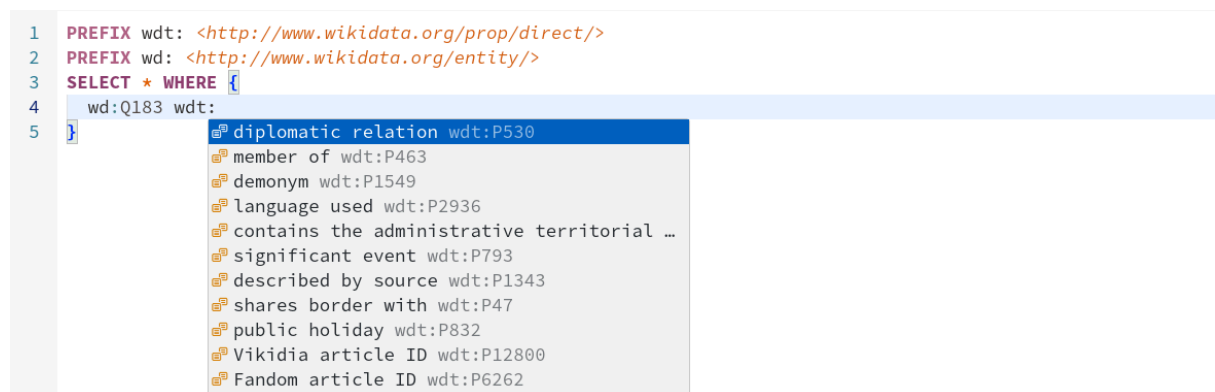


Figure 93: **Predicate - Prefix** completions are triggered at the second part of a triple. A requirement for this completion is that the *search term*, described in Section 6.1.1, is a prefix. These completions suggest entities that start with this prefix. In the screenshot the completion is triggered at the second part of a triple. The *search term* is wdt: and suggestions start with this prefix.

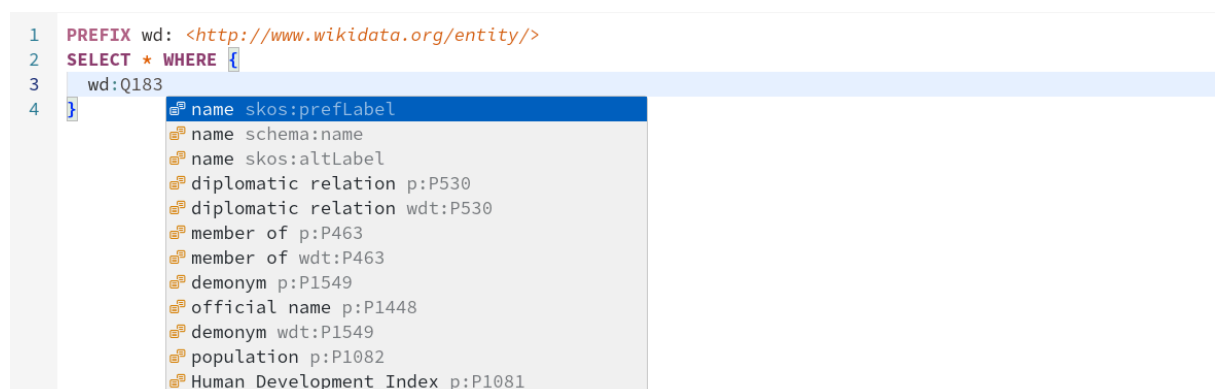


Figure 94: **Predicate - IRI** completions are triggered at the second part of a triple. The suggestions are IRIs of predicates that exist in the knowledge-graph. In the screenshot the completion is triggered after the subject wd:Q183, which describes Germany in wikidata. The suggestions are all predicates of Germany.

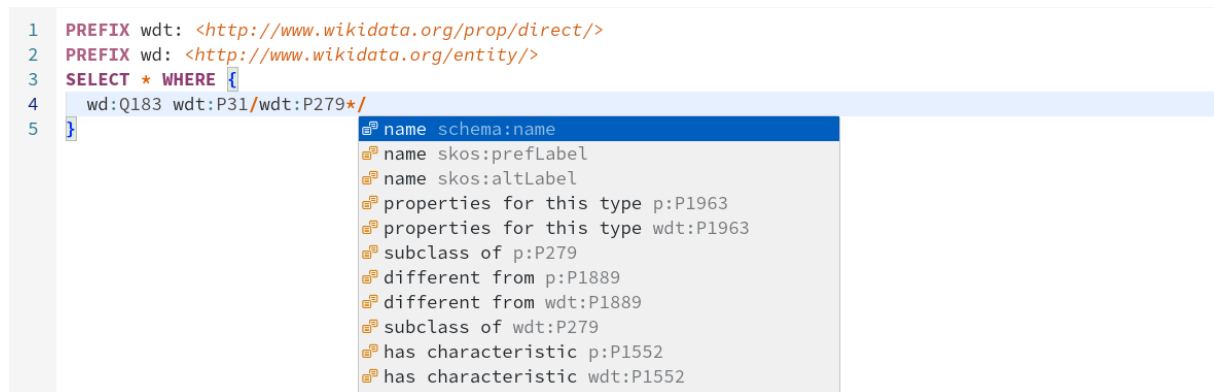


Figure 95: **Predicate - Path** completions are completions that are triggered behind a property paths. Property paths describe paths in the knowledge graph, *SPARQL* support a syntax for this. Completions at after a property paths suggest continuations of property paths. In the screen shot the completion is triggered after the property path `wdt:P31/wdt:P279*`. The suggestions are all properties that validly continue the property path.

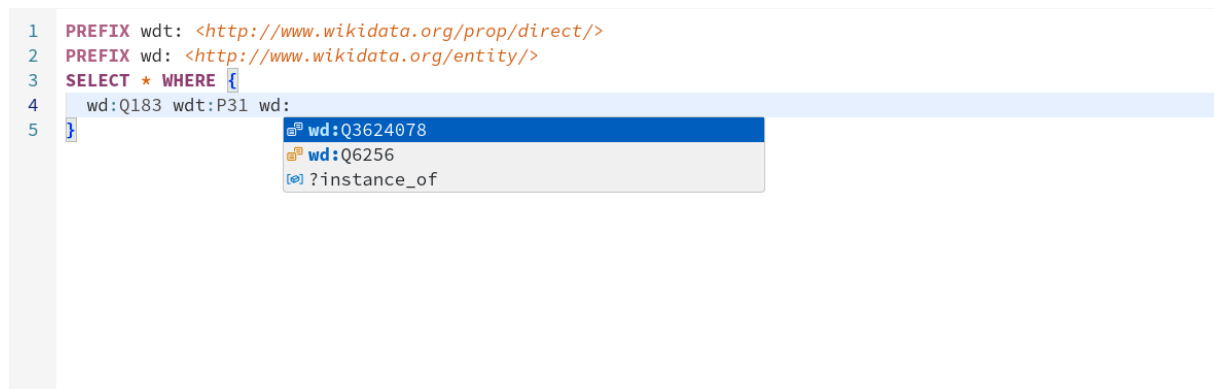


Figure 96: **Object - Prefix** completions are triggered at the third part of a triple. A requirement for this completion is that the *search term*, described in Section 6.1.1, is a prefix. These completions suggest entities that start with this prefix. In the screenshot the completion is triggered at the third part of a triple.

The *search term* is `wd:.` Both suggestions start with this prefix: `wd:Q6256` and `wd:Q3624078`.



```

1 PREFIX wdt: <http://www.wikidata.org/prop/direct/>
2 PREFIX wd: <http://www.wikidata.org/entity/>
3 SELECT * WHERE {
4   wd:Q183 wdt:P31
5 }

```

Completion suggestions for the object position of the triple `wd:Q183 wdt:P31`:

- `?instance_of`
- `sovereign state wd:Q3624078`
- `country wd:Q6256`

Figure 97: **Object - IRI or Literal** completions are completions triggered at the third part of a triple. These completions use the *search term*, described in Section 6.1.1, and suggest IRIs or Literals that match this *search term*. In this screenshot the completion is triggered at the third position of the incomplete triple: `wdt:Q183 wdt:P31`. The IRI `wdt:Q183` describes the entity of the country Germany in the wikidata dataset. The IRI `wdt:P31` is used as predicate to describe what the class of a entity is. Here `country` and `sovereign state` are suggested as object.

```

1 PREFIX wdt: <http://www.wikidata.org/prop/direct/>
2 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3 SELECT * WHERE {
4   ?freiburg rdfs:label "Freiburg im Breisgau"@en .
5   ?freiburg wdt:P17
6 }

```

Completion suggestions for the object position of the triple `?freiburg wdt:P17`:

- `?country`
- `?freiburg_country`
- `?freiburg`
- `Germany wd:Q183`

Figure 98: **Context Aware** completions are completions of subject, predicates of objects that use the context of the completion to constrain the completion. In this screenshot the first triple states that `?freiburg` has `rdfs:label "Freiburg im Breisgau"@en`. This constraints the variable to entities that have exactly this label. The completion is triggered at the object position of the triple `?freiburg wdt:P17`. The predicate `wdt:P17` is used to declare the country of a entity. The context of the completion is the first triple. A context aware completion only suggests `Germany(wd:Q183)`, since every entity with the label `"Freiburg im Breisgau"@en` is in Germany.

```
1 PREFIX osmkey: <https://www.openstreetmap.org/wiki/Key:>
2 PREFIX osmrel: <https://www.openstreetmap.org/relation/>
3 PREFIX OSM_Planet: <https://qlever.cs.uni-freiburg.de/api/osm-planet>
4 SELECT * WHERE {
5   SERVICE OSM_Planet: {
6     osmrel:62768 osmkey:name
7   }
8 }
```

The screenshot shows a SPARQL query editor with a light blue background. The query is as follows:  
1 PREFIX osmkey: <https://www.openstreetmap.org/wiki/Key:>  
2 PREFIX osmrel: <https://www.openstreetmap.org/relation/>  
3 PREFIX OSM\_Planet: <https://qlever.cs.uni-freiburg.de/api/osm-planet>  
4 SELECT \* WHERE {  
5 SERVICE OSM\_Planet: {  
6 osmrel:62768 osmkey:name  
7 }  
8 }  
A blue selection bar highlights the line 'osmrel:62768 osmkey:name'. A dropdown menu is open below this line, showing two suggestions: '?name' (highlighted in blue) and '"Freiburg im Breisgau"' (highlighted in light grey).

Figure 99: **Service Aware** completions are completions of subject, predicates or objects that use the service endpoint to provide data-driven completions. In this screenshot the endpoint is not osm-planet. But the query contains a **SERVICE** block with the osm-planet IRI. The completion suggest a object that comes from the osm-planet knowledge-graph.

## Abbreviations

<b>API</b>	Application Programming Interface
<b>HTML</b>	Hypertext Markup Language
<b>RDF</b>	Resource Description Framework
<b>LSP</b>	Language Server Protocol
<b>SPARQL</b>	SPARQL Protocol and RDF Query Language
<b>DSL</b>	Domain Specific Language
<b>IRI</b>	Internationalized Resource Identifier
<b>GUI</b>	Graphical user interface
<b>EBNF</b>	Extended Backus-Naur Form

## References

- [1] S. Harris and A. Seaborne, “SPARQL 1.1 Query Language.” [Online]. Available: <https://www.w3.org/TR/sparql11-query/>
- [2] R. Cyganiak, D. Wood, and M. Lanthaler, “RDF 1.1 Concepts and Abstract Syntax.” [Online]. Available: <https://www.w3.org/TR/rdf11-concepts/>
- [3] Microsoft and R. Hat, “Language Server Protocol.” 2016. [Online]. Available: <https://microsoft.github.io/language-server-protocol/>
- [4] H. Bast, J. Kalmbach, T. Klumpp, F. Kramer, and N. Schnelle, “Efficient and Effective SPARQL Autocompletion on Very Large Knowledge Graphs,” in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management, Atlanta, GA, USA, October 17-21, 2022*, M. A. Hasan and L. Xiong, Eds., ACM, 2022, pp. 2893–2902. doi: [10.1145/3511808.3557093](https://doi.org/10.1145/3511808.3557093).
- [5] N. Chomsky, “On certain formal properties of grammars,” *Information and Control*, vol. 2, no. 2, pp. 137–167, 1959, doi: [10.1016/S0019-9958\(59\)90362-6](https://doi.org/10.1016/S0019-9958(59)90362-6).
- [6] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao, “Describing Linked Datasets with the VoID Vocabulary.” [Online]. Available: <https://www.w3.org/TR/void/>
- [7] Vercruysse, Arthur and Rojas Melendez, Julian Andres and Colpaert, Pieter, “The semantic web language server : enhancing the developer experience for semantic web practitioners,” in *The Semantic Web : 22nd European Semantic Web Conference, ESWC 2025, Proceedings, Part II*, Curry, Edward and Acosta, Maribel and Poveda-Villalón, Maria and van Erp, Marieke and Ojo, Adegboyega and Hose, Katja and Shimizu, Cogan and Lisena, Pasquale, Ed., Portoroz, Slovenia: Springer, 2025, pp. 210–225. [Online]. Available: [http://doi.org/10.1007/978-3-031-94578-6\\_12](http://doi.org/10.1007/978-3-031-94578-6_12)
- [8] H. Chiba, “sparql-formatter.” [Online]. Available: <https://github.com/sparqling/sparql-formatter>
- [9] I. Nezis, “Qlue-ls a SPARQL language server.” [Online]. Available: <https://ad-blog.cs.uni-freiburg.de/post/qlue-ls-a-sparql-language-server/>

- 
- [10] L. Rietveld and R. Hoekstra, “YASGUI: Not Just Another SPARQL Client,” in *The Semantic Web: ESWC 2013 Satellite Events*, P. Cimiano, M. Fernández, V. Lopez, S. Schlobach, and J. Völker, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 78–86.