

Undergraduate Thesis

Typesafe SPARQL Query Parsing mit ANTLR

Julian Mundhahs

Examiner: Prof. Dr. Hannah Bast

Advisers: Johannes Kalmbach

University of Freiburg

Faculty of Engineering

Department of Computer Science

Chair of Algorithms and Data Structures

October 2nd, 2022

Writing Period

August 14th, 2022 – October 2nd, 2022

Examiner

Prof. Dr. Hannah Bast

Advisers

Johannes Kalmbach

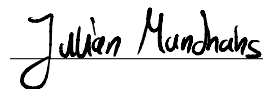
Declaration

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 hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg, October 2nd 2022

Place, Date


Signature

Abstract

Within the frame of this thesis, a parser for the SPARQL 1.1 Query Language (SPARQL QL) was finished. This parser uses the parser generator ANTLR v4. With the new parser, QLever is able to recognize the complete SPARQL QL. An improved version of the visitor pattern provided by ANTLR v4 is presented. The improved visitor makes the visitor type safe, prevents code duplication and provides metadata for error messages.

Zusammenfassung

Im Rahmen dieser Arbeit wurde ein Parser für die SPARQL 1.1 Query Language (SPARQL QL) fertiggestellt. Dieser Parser verwendet den Parser Generator ANTLR v4. Mit dem neuen Parser kann QLever die ganze SPARQL QL erkennen. Es wird eine weiterentwickelte Version des ANTLR v4 Besuchers vorgestellt. Die Weiterentwicklung zeichnet sich dadurch aus, dass sie typsicher ist, Codeduplizierungen vermeidet und Metadaten für Fehlermeldungen bereitstellt.

Contents

1	Introduction	1
2	Background	3
2.1	RDF	3
2.2	SPARQL 1.1	5
2.3	QLever	6
2.4	ANTLR	7
3	Related Work	13
3.1	Blazegraph	14
3.2	Eclipse RDF4J	14
3.3	Apache Jena	14
3.4	Virtuoso	15
4	Approach	17
4.1	Old Parser	17
4.2	New Parser	18
5	Technical Contributions	21
5.1	Type safety of the Parser	22
5.2	Meaningful Error Messages	24
5.3	Testing	26
5.3.1	Old Parser	27

5.3.2	Preexisting Tests for the New Parser	28
5.3.3	Improvements to the Tests	30
6	Evaluation	33
6.1	Speed	35
6.2	Standard Compliance	36
6.3	Code Quality	37
7	Further Tasks	41
8	Acknowledgments	45
	Bibliography	48

List of Figures

1	Simple RDF Graph	5
2	Representation of the Parsed Data for $1+2*3$	8
3	Structure of the Old Parser	17
4	Structure of the New Parser	19
5	Error highlighting	25

List of Tables

1	Share of Query Parsing of the Total Query Execution Time	34
2	Standard Compliance of Select Aspects of the Old and New Parser .	37
3	Coverage Old Test Suite	39
4	Coverage New Test Suite on the New Parser	39

List of Listings

2.1	Example ANTLR Grammar	7
2.2	Call Sequence Listener	9
2.3	Call Sequence Visitor	10
5.1	Additive Parse Tree Visitor Interface	22
5.2	Modified Additive Parse Tree Visitor Interface	23
5.3	Old Test	27
5.4	New Test	28
5.5	Macro Based Matchers	29
5.6	New and Improved Test	30
5.7	New Matchers	30

1 Introduction

QLever is a SPARQL engine written in C++. The various parts of SPARQL are defined in a total of 10 specifications [1]. These include among others a specification for a query language [2] - the SPARQL 1.1 Query Language (SPARQL QL). As a SPARQL engine, QLever also must be able to parse SPARQL QL queries. It is expected by the user that QLever conforms to the standards. Furthermore, users expect meaningful error messages. The concrete implementation and extent of error messages are not specified as part of the SPARQL Protocol Specification [3]. Providing meaningful error messages for parsing errors still must be considered when designing a new parser. Error messages must give an exact description of what went wrong and where exactly it went wrong to be useful for the end user.

The QLever project aims to have a high code quality. Achieving a high code quality requires that new code be easily maintainable and thoroughly tested. Its architecture must be chosen such that code duplication and the chance that new code introduces new error are both minimized. To achieve high maintainability the code must have low coupling.

To achieve this goal of high code quality and a standard compliant parser it was decided to switch to an ANTLR v4 [4] based parser. ANTLR v4 is a parser generator. This means that the lexer and parser are generated by ANTLR given the grammar of a language.

The resulting parse tree is then parsed using an improved version of the visitor pattern provided by ANTLR. The visitor pattern was chosen for QLever as it highly encourages low coupling of the code.

The contributions of this thesis consist of migrating most of the parsing from the old parser to the ANTLR based parser. This migration for the most part completes the switch to the new parser for Queries. Only filters are still parsed with the old parser. It is planned to also migrate this remaining code soon once the required changes in other areas have been done. The newly written code is thoroughly tested. In the migration process we improved the visitor pattern from ANTLR. The improvements result in a type safe visitor and reduce code duplication. The improvements also reduce the risk of introducing errors, while also making the code more readable. Finally, we extended the parser to provide metadata in addition to text-based error messages. This metadata enables front ends using QLever to provide richer information in error cases. The metadata consists of the exact location of the error in the query. This information might be used by a front end to highlight the error directly in the query.

2 Background

In this Section, we will give a broad overview about underlying technologies or concepts. These are required to understand what QLever aims to achieve and how this thesis improves the parser of QLever. This overview will be bottom-up. In Section 2.1 and Section 2.2, we will introduce RDF and SPARQL. These are the concepts that are required to understand what QLever does on an abstract level (Section 2.3). The overview will conclude with Section 2.4 which gives an overview over ANTLR. ANTLR is the tool used to write QLever’s SPARQL QL parser.

2.1 RDF

The specification states that the “Resource Description Framework (RDF) is a framework for representing information in the Web” [5]. RDF is standardized by the W3C in series of documents such as [5, 6, 7]. RDF is used among others by Wikidata as a format for data dumps and querying the data through the Wikidata Query Service [8].

Formally, an RDF Graph is a set of RDF triples. RDF triples are made up of these data types:

IRIs IRIs are Internationalized Resource Identifier as defined in RFC 3987 [9].

Literals Literals are Unicode strings. They can additionally have a datatype IRI and in some cases, also a tag denoting the language.

Blank nodes Blank nodes are defined to be disjunct from IRIs and literals. The meaning of blank nodes is to denote that something exists that satisfies the relationship without specifying what exactly this is.

IRIs and literals represent objects or attributes of objects of the data that is being described. An RDF triple consists of three components:

Subject This is an IRI or blank node.

Predicate This is an IRI.

Object This is an IRI, literal or blank node.

“An Olympic gold medal winner exists” can be represented in this triple form. This sentence could be represented as an RDF triple with a blank node as subject, the IRI `has_won_a` as predicate and the IRI `olympic_gold_medal` as object.

The basic idea of RDF is to represent data as an RDF Graph. The meaning of a triple is that the subject satisfies some predicate with regard to an object. In the following, we will illustrate this concept with two examples. They will show how natural statements could be translated into triples.

- Take the sentence “The Eiffel Tower is 330m high.” It states that the Eiffel Tower (subject) has a height in meters (predicate) of 330 (object).
- “The University of Freiburg is located in Freiburg which itself is in Germany”. Notice that this sentence contains two statements. As each triple can only represent one statement this sentence must be decomposed into two triples. The University of Freiburg (subject) is located in (predicate) Freiburg (object). Freiburg (subject) is located in (predicate) Germany (object).

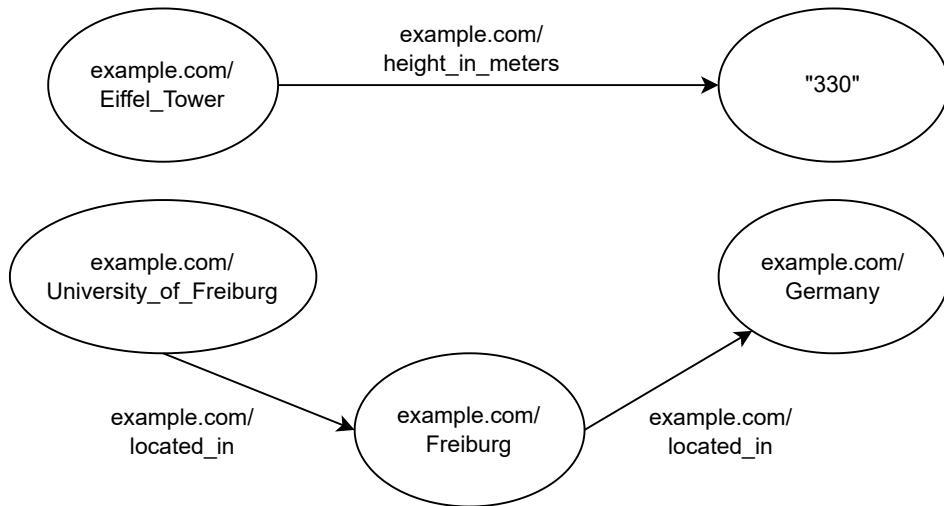


Figure 1: Simple RDF Graph

To construct the RDF graph of these three statements the corresponding triples are joined into a Graph. Different triples can refer to the same object with their subject or object. Even though an object is used multiple times it will only be represented by one node in the graph. In the example, Freiburg is used twice. It is used once as a subject and once as an object. This means that Freiburg will be joined into one node for the Graph. The RDF graph resulting from the example could look like Figure 1.

2.2 SPARQL 1.1

The SPARQL 1.1 standard is a collection of 10 standards. They define standards to query and manipulate RDF graphs. For this work the current version (1.1.) of the SPARQL specifications by the W3C from 2013 [1] is relevant. Most of this thesis will work with the SPARQL 1.1. Query Language (SPARQL QL) specifications [2]. The SPARQL QL standard defines the format of the query language used to query RDF graphs.

There is another standard that defines an extension to the query language and adds update operations. Update operations allow the modification of the data stored in an RDF Graph. QLever does not support update operations. Only the query language without update operations will be considered for this thesis.

Another standard that is of interest for this thesis is the SPARQL 1.1 Protocol [3]. The SPARQL 1.1 Protocol describes how clients should interact with servers that serve SPARQL requests. The server processing the SPARQL requests is called the SPARQL protocol service or service. The standard lays out how clients should send queries to the service and how the service should respond to those queries. The SPARQL 1.1 Protocol is a protocol defined on top of HTTP. Queries may be sent as HTTP GET query parameter or as HTTP POST. The response to a query then contains the result of the query in the response body.

Additional standards for federation, several data formats for the returned, negotiation of a result format and a mechanism for a SPARQL Protocol Service to provide metadata about the provided dataset are also available. These standards are not of interest for this thesis. Most of them are not supported by QLever.

2.3 QLever

QLever [10] is a SPARQL protocol service written in C++. QLever provides a SPARQL 1.1 endpoint for accessing the loaded knowledge graph. It provides context-sensitive autocompletion via the front end QLever-UI [11] as well as text search in text documents associated with the knowledge graph. QLever aims to differentiate itself from other SPARQL engines like Blazegraph by being faster, especially on queries with large intermediate results. A QLever-UI instance with several QLever back ends is available[12]. The QLever back ends are loaded with many datasets including the complete Wikidata, OpenStreetMap and PubChem.

2.4 ANTLR

ANTLR [4] is a parser generator. This means that given the grammar of a language, ANTLR generates a lexer and parser which can parse this language. The lexer and parser can be generated in many programming languages, including C#, C++ and Java. The generated code can then lex and parse any rule from the provided grammar.

A formal grammar G is defined as a 4-tuple (V, Σ, P, S) . V a set of nonterminal symbols and Σ a set of terminal symbols with V and Σ disjoint. S is a start symbol from V . P is a set of production rules of the form $w \rightarrow v$, with $w, v \in (\Sigma \cup V)^*$ and $w \notin \Sigma^*$. The language of a grammar is the set of strings $\in \Sigma^*$ that can be reached from the start symbol with repeated applications of the production rules.

A parser usually must answer two questions about a string $S \in \Sigma^*$. Is S in the language of the grammar? If S is in the language, which production rules can be used to obtain S from the start symbol?

We will illustrate this with a concrete example. For this example, we will use the grammar in Listing 2.1. The example grammar can parse simple expressions that add and multiply numbers.

```
1 Additive: Additive "+" Additive | Multiplicative;
2 Multiplicative: Multiplicative "*" Multiplicative | BracketedValue;
3 BracketedValue: ("0".."9")+ | "(" Additive ")";
```

Listing 2.1: Example ANTLR Grammar

Each production rule in ANTLR is terminated by a semicolon. The string on the left side of a `:` is the left-hand side of a formal production rule (w). The rest of the ANTLR production rule is the right-hand side of the formal production rule (v). `|` is used to separate sequences of symbol where the left or the right side may match. This means that `A: B | C;` is equivalent to the two production rules $A \rightarrow B, A \rightarrow C$.

Strings encased in quotes are terminal symbols. Brackets are used to group symbols. $*$ denotes the Kleene star and is applied to the previous symbol or group. $+$ denotes that preceding symbol must be present at least once. Formally, G^+ is defined as $G G^*$. $"0" \dots "9"$ denotes that any character whose codepoint is between 0 and 9 is matched.

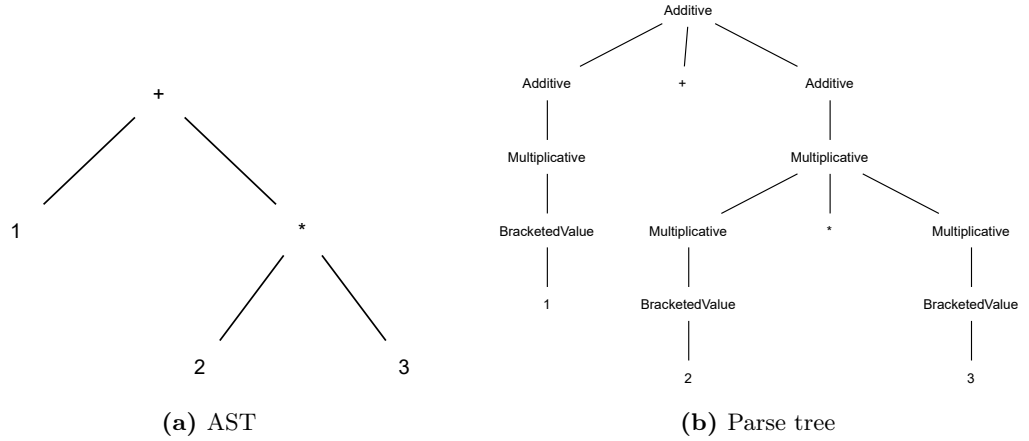


Figure 2: Representation of the Parsed Data for $1+2*3$

Assume that the string $1+2*3$ is parsed as an **Additive**. Parsing a string as an **X** means that we parse the string with the grammar and **X** is the starting symbol. It is easy to see that $1+2*3$ is in the language of the example grammar. This should become abundantly clear with the following explanations about how exactly the string can be derived. To be able to represent the derivation of a string in an easy format, only a subset of all grammars may be used. These grammars may only have one nonterminal symbol on the left side of each production rule. The grammars are also unique. This means that there is only one way to obtain a string for the start symbol. These two restrictions make it possible to represent the production rules used for the derivation of a string as a tree. Two common formats are abstract syntax trees (AST) and parse trees. Both are trees, but the parse tree is more explicit. The parse tree contains all terminal children and the whole derivation tree. It is shown in Figure 2 how the example string $1+2*3$ is displayed in both formats.

In the AST, every node corresponds to a part of the input string that was parsed according to the grammar. In parse trees each node corresponds to the left side of the production rule. The children are the components of the right side of the matched production rule. These nodes in the parse tree are called **Context** in ANTLR. ANTLR uses parse trees to work with parsed input.

The tasks up to this point are done by code provided or generated by ANTLR. The last step of converting the parse tree into the application specific model must be done manually. The application specific model is the model that the application uses for the data that is being parsed. There are many reasons to transform a parse tree into an application specific model. There might be semantic conditions that are checked, but cannot be expressed with the grammar. The application might need specific data structures or helper methods to efficiently handle the data. The syntax might not fit the model perfectly. The parsed data then must be transformed to fit this underlying model. Take SPARQL QL filters. Filters used in the body of a query do not apply to the layer in which they are defined in the grammar. Their scope is larger than their position in the grammar suggests. To handle the query efficiently, such things must be corrected.

```
1 enterAdditive(AdditiveContext);
2 enterAdditive(AdditiveContext); // Begin of Left Subtree
3 enterMultiplicative(MultiplicativeContext);
4 enterBracketedValue(BracketedValueContext);
5 exitBracketedValue(BracketedValueContext);
6 exitMultiplicative(MultiplicativeContext);
7 exitAdditive(AdditiveContext); // End of Left Subtree
8 enterAdditive(AdditiveContext); // Begin of Right Subtree
9 enterMultiplicative(MultiplicativeContext);
10 ...
```

Listing 2.2: Call Sequence Listener

ANTLR provides two ways to process these parse trees: parse tree listeners and visitors.

With a parse tree listener, the parse tree is traversed depth first. The listener has two methods each for every production rule. One method that is called when this node is discovered and one that is called when the node is finished during the traversal. The parse tree in Figure 2b would then be processed as shown in Listing 2.2.

All the methods are called sequentially by ANTLR. The listener must store the transformed data in instance variables.

Parse tree visitors on the other hand have one function each for the production rules. This function then transforms the content of this rule into the application specific model. The visitor functions then return the transformed value. The visitor functions can also recursively transform any of the children into the application specific model. This means that the visitor can also decide how, and which parts of the parse tree are traversed. The transformed values of the children can also be used to compute the value for the node itself. This may be the case if the right side of the matched rule contains nonterminal symbols. In that case, the children themselves again must be transformed.

```
1 visitAdditive(AdditiveContext); // Root
2     visitAdditive(AdditiveContext); // Left Subtree
3         visitMultiplicative(MultiplicativeContext);
4             visitBracketedValue(BracketedValueContext);
5     visitAdditive(AdditiveContext); // Right Subtree
6         visitMultiplicative(MultiplicativeContext);
7             visitMultiplicative(MultiplicativeContext);
8     ...
```

Listing 2.3: Call Sequence Visitor

Listing 2.3 show how the calls to the visitor could look like. Indented calls were made by the previous call on the next outer indentation level.

3 Related Work

There are already several projects that can parse the SPARQL 1.1 Query Language (SPARQL QL). In this chapter, we will give a brief overview of the parsers of the most notable open-source SPARQL engines.

Many projects are written using languages that are not compatible with C++. But there are structural similarities with other projects. It is common to use parser generators like JavaCC, Bison and ANTLR to write parsers. Some of the projects use a visitor pattern like QLever. A difference is that QLever uses an improved version of the visitor pattern. Other projects use more tightly integrated code. These merge the grammar and the code to transform the parse tree.

JavaCC ¹ is a parser generator written in Java. This means that JavaCC generates a parser given a formal grammar in a specific format. JavaCC can generate parsers in Java, C++ and C#. JJTree is a preprocessor for JavaCC that adds the generation of parse trees and visitor interfaces to transform those parse trees.

¹<https://github.com/javacc/javacc>

3.1 Blazegraph

Blazegraph ² is a graph database written in Java. It is currently used for the official Wikidata Query Service [8]. The open-source development has been ceased in 2018. Blazegraph’s SPARQL QL parser is written using the JavaCC parser generator. The parse tree and visitor of JavaCC/JJTree are used in Blazegraph. This parser is architecturally like QLever’s new parser but it lacks the more sophisticated features like the type safety of the visitor and exception metadata. Additionally, it is written in Java.

3.2 Eclipse RDF4J

Eclipse RDF4J ³ is a framework to handle RDF data written in Java. Like Blazegraph and Apache Jena, JavaCC uses JavaCC and JJTree for its parser. The generated parser and visitor are different components. This architecture is similar to Blazegraph’s. In contrast to QLever, Eclipse RDF4J’s parser is written in Java and lacks some of the more advanced features present in QLever’s parser.

3.3 Apache Jena

Apache Jena ⁴ is a project for writing Semantic Web applications. It supports queries in the SPARQL QL and is written in Java. JavaCC is used for generating a parser for the SPARQL QL. Jena uses JJTree and a visitor to process the data. Fundamentally Jena also uses the visitor pattern to process its parse tree, but it is different in that much of the logic is moved into the grammar file.

²<https://github.com/blazegraph/database>

³<https://github.com/eclipse/rdf4j>

⁴<https://github.com/apache/jena>

The specification what types are returned by rules and which functions should be called are all specified in the grammar file. This is a structural difference to QLever.

3.4 Virtuoso

OpenLink Virtuoso ⁵ is an engine that supports multiple paradigms. It is mostly written in C. It features a SPARQL QL parser in C written using Bison. But like Jena, much of the logic of how to handle specific rules is specified in one file with the grammar itself.

⁵<https://github.com/openlink/virtuoso-opensource>

4 Approach

We define the state of the QLever repository [10] directly before this work (commit hash `1a35ae5`¹) as the initial state. The final state for this comparison is the state directly after this work with commit hash `f9f0a5b`².

In this Section, we will compare the old and new parser on an abstract level. Section 4.1 will give an overview over the structure of the old parser. An introduction into QLever’s new parser will follow in Section 4.2.

4.1 Old Parser

The old custom written parser from 2019 is comprised of two parts. The structure of the parser is shown in Figure 3. The first part is the `SparqlLexer` which performs the function of a lexer. Its task is to split the query string into tokens. A token is a part of the query that has a meaning itself. This might be an IRI or a Keyword in a query.

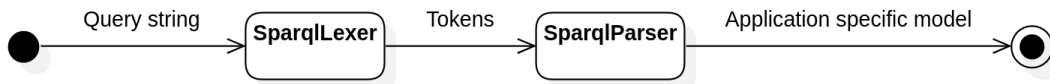


Figure 3: Structure of the Old Parser

¹<https://github.com/ad-freiburg/qlever/tree/1a35ae533123dee1d910105c174e79071a0f87f0>

²<https://github.com/ad-freiburg/qlever/tree/f9f0a5bf2ddfaf363160bc0c2fadcb44c5aadabc>

The types of tokens that exist are defined in the code of the lexer. The individual token types are detected with Regular Expressions.

The lexer provides two methods to the parser to work with the detected tokens. Both methods can match on a token with a specific type (e.g., Keyword or IRI) or a concrete string. The parser can query the next token whether the next token matches. The parser can also require that the next token matches. If such a requirement is not satisfied, an exception will be thrown. After a successful match, the token is consumed and the following token is read. The next operations will query the next token. Finally, a method to retrieve the last successfully matched token is provided. The second part of this parser architecture is the **SparqlParser**. **SparqlParser** does the parsing (syntactic analysis), as well as transforming the parsed data into the application specific model. The **SparqlParser** used the **SparqlLexer** to process the tokens and figure out which production rules were used to obtain the string. This processing was done in functions. These functions often processed not a single node in the tree, but a subtree. This design enticed the sharing of state in the function across the border of different production rules. It also did not require a clear representation for each production rule in the application's model.

4.2 New Parser

The new parser uses ANTLRv4. The structure of the new parser is shown in Figure 4. The query string is first passed through the lexer which is provided or generated by ANTLR. The tokens are then parsed by the **SparqlAutomaticParser** which also is generated by ANTLR. Lastly, the parse tree is transformed into the application specific model by the **SparqlQLeverVisitor**. The visitor is the only part of the parsing process that must be implemented manually. For QLever, the visitor pattern is used to perform this transformation. The SPARQL QL grammar [2] was already translated into a format suitable for ANTLR by Johannes Kalmbach. The visitor was also already implemented for some rules by previous contributors to the codebase.

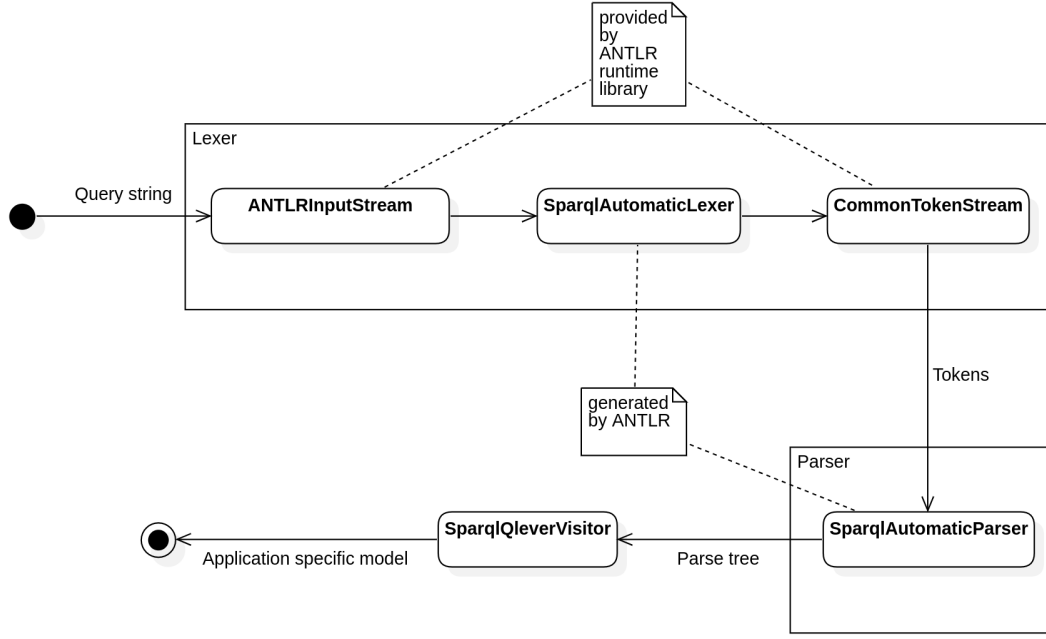


Figure 4: Structure of the New Parser

The main part of this work was to complete the visitor and to improve it. After the contributions of this thesis, it is possible to parse complete queries using the new ANTLR based parser. The visitor can now transform most rules. Only filters are still parsed separately. Furthermore, this thesis contributes improvements to the visitor pattern for Qlever. These improvements make the visitor type safe. Finally, code to make testing easier and provide better error messages was also implemented.

The structure of ANTLR parse tree visitor is quite different to the structure of the old parser. As described in Section 2.4, the visitor has one method each for the rules of the grammar. These methods transform the rule (node in the parse tree) into the application specific model. It can recursively call the visitor to transform the child rules into the application specific model. The method only uses the already transformed children to compute its result. It does not access the raw parsed data of any children. This structure enforces a strict separation between the transformation of different rules. It also requires a clear model of what each rule in the grammar represents with respect to the application's model.

Lastly, it also makes the individual functions of the visitor shorter. The parser had functions which had more than 200 lines. Most of the visitor's functions are no longer than 25 lines. Many functions have five lines or less. Although the single functions of the visitor can not directly share information, the visitor itself can be stateful. QLever only uses this sparsely. This state is used e.g., to generate integer identifiers that must be increasing and without jumps. A less complex state is good because it reduces complexity and makes testing easier.

This new structure has several benefits:

- The small independent functions make the parser much easier to understand and maintain.
- They enable a deeper level of testing that was not possible before.
- It is directly visible which parts of the grammar are supported or implemented and which are not.

5 Technical Contributions

The main part of this thesis was to finish the visitor implementation. In this Section, we will give a detailed insight into the contributions made by this thesis. In Section 5.1, we will describe how we made the visitor type safe. Section 5.2 contains the improvements that we made to error messages emitted by the parser. Section 5.3 describes the improvements we made to the tests for the parser.

All contributions of this thesis are all merged into the respective codebase. Most of the contributions are made to QLever [10]. Some contributions were also made to QLever-UI [11]. Guides to utilize the contributions of this thesis can be found in their respective projects. The contributions can also be tested with the publicly available QLever endpoint [12]. The endpoint provides QLever-UI as a front end to several QLever instances.

5.1 Type safety of the Parser

```
1 class AdditiveAutomaticVisitor : public antlr4::tree::  
    AbstractParseTreeVisitor {  
2     antlrcpp::Any visitAdditive(AdditiveContext* context);  
3     antlrcpp::Any visitMultiplicative(Multiplicative* context);  
4     antlrcpp::Any visitBracketedValue(BracketedValueContext* context);  
5 }
```

Listing 5.1: Additive Parse Tree Visitor Interface

The visitor pattern provided by ANTLR has one function each for the different contexts. These functions have unique names, e.g., `visitAdditive` for the `AdditiveContext`. They take the respective context as their only parameter. The return value is a container type that can contain any value. This container type is named `antlrcpp::Any`. The visitor interface that ANTLR would generate for the example in Section 2.4 is shown in Listing 5.1.

Notice that all visitor functions have the return type `antlrcpp::Any`. The actual return values are packed into `antlrcpp::Any` when a function returns. They are unpacked again when the value is used. This unpacking can cause exceptions when trying to unpack a value of a type that is not the type of value in the container. Trying to unpack a float from a container that contains an integer will result in an error being thrown. This means that handling these values is not type safe and may result in runtime errors. The advantage of this approach is that it enables ANTLR to provide some default behavior. The user may choose to only provide an implementation for certain contexts. For all cases that are not defined by the user, the result of the last subrule is returned. This approach also allows to return values of different types.

However, this manifestation of the visitor pattern is not well suited for QLever. The `antlrcpp::Any` type requires unpacking of the actual result every time a sub rule is processed. This adds code overhead and, even worse, code duplication. Contexts are often visited from more than one place. The type that is contained must be duplicated in all the places where it is visited from. The type also must match the type that is returned by the function. This allows errors to happen while programming or refactoring code.

Instead, we propose a modified visitor interface to solve these problems. There is still one function each for the different contexts. They still take the context as the only parameter. But these functions are now all called `visit`. The resolution which function is called is decided by the type of the context which is the only parameter. The return values are not packed into a container but instead returned directly. This is also reflected in the functions return type. If required, `std::variant` is used to achieve a union of types in a safe way. The modified visitor does not inherit from the `AbstractParseTreeVisitor` and therefore does not have any defaults. All rules must be implemented. An example for this modified pattern for the example grammar in Section 2.4 is shown in Listing 5.2.

```
1 class AdditiveVisitor {
2     Expression visit(AdditiveContext* context);
3     Expression visit(Multiplicative* context);
4     Expression visit(BracketedValueContext* context);
5 }
```

Listing 5.2: Modified Additive Parse Tree Visitor Interface

With the modified pattern, unpacking is no longer required. This removes the need for boilerplate code and code duplication associated with unboxing. Errors related to unboxing are now compile time errors instead of runtime errors. This improves or removes the described problems.

5.2 Meaningful Error Messages

Meaningful error messages are useful while developing because they allow a quicker diagnosis when a problem occurs. End users are not familiar with the code or the implementation details. Meaningful error messages are required for end users to diagnose errors. If the error is too technical, it is of little use. On the other hand, clear error messages that pinpoint what exactly went wrong enable users to fix the problems on their own.

QLever strives to output useful error messages. During this work, the error messages output by the parsing subsystem of ANTLR were improved. Previously, the error messages only contained text. The text was made up of a description of the problem that occurred and the whole query. But this is only sufficient in some cases to quickly identify the location of the error.

Take “GROUP BY is not allowed when all variables are selected via SELECT *” as an example. The source is identifiable from the text because there is only one select and group in the query.

“Variable ?foo is used but not aggregated despite the query not being grouped by ?foo.” requires the naming of the variable to be able to pinpoint the problem. The messages would be unclear without the concrete variable, as there may be many variables selected in a query. This is the current state of the error messages from the parser.

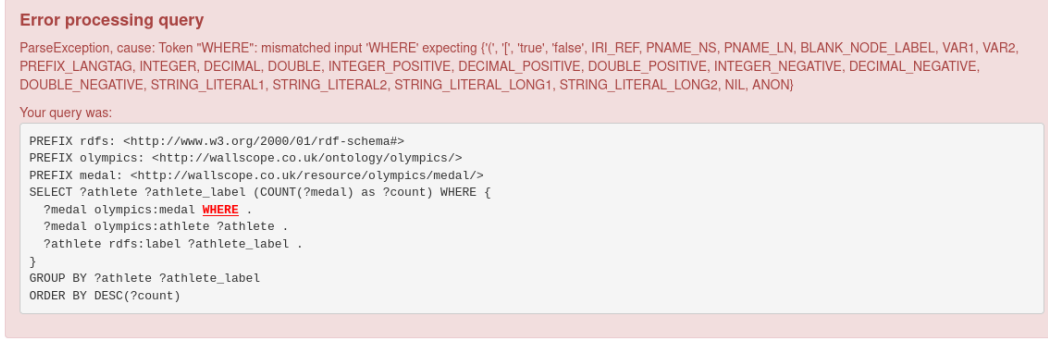
However, this is still not sufficient. Assume that a query contains a misplaced WHERE. The error message could look something like Expected a token of type IRI but got a token of type KEYWORD (WHERE) in the input at pos 120 : [...]. Finding the location of the error by its location is hard. Doing this would require counting the characters.


```

2022-09-11 19:34:51.558 - ERROR: where {
    ?medal olympics:medal WHERE .
    ?medal olympics:athlete ?athlete .
    ?athlete rdfs:label ?athlete_label .
}
GROUP BY ?athlete ?athlete_label
ORDER BY DESC(?count)

```

(a) QLever Server



(b) QLever UI

Figure 5: Usage of Exception Metadata for Highlighting of Errors

To solve the problem of finding the location of an error, metadata on exceptions have been introduced. The metadata contains the query that caused the exception and the exact location that caused the error. The location is provided as a position in the query string, as well as line number and position in the line. All positions are in Unicode code points and not bytes. The metadata is returned by QLever when an error occurs. Clients using QLever can then use this information to highlight the error. Usages of the metadata in QLever and QLever-UI are shown in Figure 5. In both cases, the metadata is used to highlight the position of the clause that caused the error. This highlighting makes it easy to find the position of the error in the query.

The metadata is automatically added to errors that are raised by ANTLR. The errors raised by ANTLR are syntactic errors. While transforming the parse tree to an application specific model, semantic errors can occur. We have implemented utilities to raise such errors with metadata.

5.3 Testing

In this Section, we will give a more detailed explanation on the test written for the new parser and how the testing code evolved. Section 5.3.1 will describe the style of the tests written for the old parser. Section 5.3.2 continues with improved tests that were written for the methods of the visitor. These tests were implemented by other QLever contributors. This is the starting point for this thesis with regards to tests. In Section 5.3.3 we will outline the improvements we made to the tests.

Good and thorough testing should be part of any professional software project. Testing is even more important for QLever, because untrusted user input is processed and errors could have security implications. The rewriting of the parser is also motivated by the desire to improve the code quality in general. The most important reason was to make the parser more compliant with the standard. Conveniently, such a rewrite with a cleaner architecture also is a very good opportunity to improve the quality of the tests. Firstly, tests must check whether the tested code behaves as expected. Apart from that, we formulated three key points that characterize good tests for QLever:

1. The test must be easy and fast to write new tests.
2. The tests must be easy to maintain.
3. The tests must be easy to read and understand.

GTest ¹ is used as a unit testing framework for QLever. GTest provides the `TEST` macro to write tests. The first parameter is the name of the test suite and the second parameter the name of the test. Test suites can group tests together. In QLever, this is used to group the tests for Components into one test suite. The single tests then test the individual features of the code.

¹<https://github.com/google/googletest>

GTest provides a series of assertions to check conditions in the tests. They start with `ASSERT_` or `EXPECT_`. The only difference between them is that `ASSERT_` asserts abort on a failed condition. `EXPECT_` asserts continue after failed conditions. GTest also introduces the concept of Matchers. A constructed Matcher can be used to check whether a value fulfills a condition with an `EXPECT_THAT`. Some basic Matchers are provided by GTest. GTest for example provides the equality Matcher. To construct an equality Matcher, the Matchers constructor is passed a value that is then used for the equality check. Matchers can be combined to represent more complex conditions.

5.3.1 Old Parser

```
1 TEST(ParserTest, testParse) {
2     [...]
3     auto pq = SparqlParser("SELECT ?x WHERE {?x ?y ?z}").parse();
4     ASSERT_TRUE(pq.hasSelectClause());
5     const auto& selectClause = pq.selectClause();
6     ASSERT_GT(pq.asString().size(), 0u);
7     ASSERT_EQ(0u, pq._prefixes.size());
8     ASSERT_EQ(1u, selectClause._varsOrAsterisk.getSelectedVariables()
9         .size());
10    ASSERT_EQ(1u, pq._rootGraphPattern._children.size());
11    ASSERT_EQ(1u, pq._rootGraphPattern._children[0]
12        .getBasic()
13        ._whereClauseTriples.size());
14    [...]
15 }
```

Listing 5.3: Old Test

The old parser does not have components that can easily be tested independently. These tests therefore only test the whole parser at once. The general procedure is always the same. A test query is parsed (line 3) and the result is then asserted (line 4 onwards). The structure of the resulting object is traversed manually and various fields and properties are asserted for their expected values. While it is also important to test large components, this form of tests has several drawbacks.

The testing of large components and the manual assertion process produces code that is sparse with regards to content. Manual assertions also mean that the assertions are duplicated between tests. This makes it possible to assert a property differently in different tests by accident. These properties of tests make them not very good regarding the three criteria. Long tests are neither fast to write nor easy to understand. They are also not easy to maintain because of their length. The sparsity of the tests makes maintenance even harder.

5.3.2 Preexisting Tests for the New Parser

```
1 TEST(SparqlParser, LimitOffsetClause) {  
2     string input = "LIMIT 10";  
3     ParserAndVisitor p{input};  
4  
5     auto limitOffset = p.parser.limitOffsetClauses()->accept(&p.visitor).  
        as<LimitOffsetClause>();  
6     EXPECT_THAT(limitOffset, IsLimitOffset(10, 1, 0));  
7 }
```

Listing 5.4: New Test

```
1 MATCHER_P3(IsLimitOffset, limit, textLimit, offset, "") {  
2     return (arg._limit == limit) && (arg._textLimit == textLimit) &&  
3         (arg._offset == offset);  
4 }
```

Listing 5.5: Macro Based Matchers

The tests for the new parser are already a substantial improvement to the tests described in Section 5.3.1. The tests for the new parser test individual functions of the visitor instead of the whole component. GTest Matchers [13] are primarily used for asserting. These tests are more focused, more readable and shorter than the tests shown in Section 5.3.1. However, there are still aspects left that can be improved. The tests still have boilerplate code related to parsing (lines 2-5). The Matchers are defined using macros from GTest. A Matcher is then used to assert the parsed value (line 6). Listing 5.5 is an example for a Matcher defined with a GTest macro. Line 1 specifies the number of parameters, the name of the Matcher and the names of the parameters. Lines 2 and 3 contain the condition that this Matcher asserts. Constructing the Matcher with this macro has the disadvantages that the parameters to construct the Matcher (here: `limit`, `textLimit` and `offset`) are polymorphic. Polymorphic parameters make it impossible to use initializer lists or omit types to shorten the code. Matchers defined using macros by default only provided a message that the composite condition failed. This usually requires stepping through the test case with a debugger to identify the cause of the test failure.

5.3.3 Improvements to the Tests

```
1 TEST(SparqlParser, LimitOffsetClause) {
2     auto expectLimitOffset = ExpectCompleteParse<&Parser::
        limitOffsetClauses>{};
3     [...]
4     expectLimitOffset("LIMIT 10", m::LimitOffset(10, 1, 0));
5     expectLimitOffset("OFFSET 31 LIMIT 12 TEXTLIMIT 14",
6         m::LimitOffset(12, 14, 31));
7     [...]
8 }
```

Listing 5.6: New and Improved Test

```
1 auto LimitOffset = [](uint64_t limit, uint64_t textLimit,
2     uint64_t offset) -> Matcher<const
        LimitOffsetClause&> {
3     return testing::AllOf(
4         AD_FIELD(LimitOffsetClause, _limit, testing::Eq(limit)),
5         AD_FIELD(LimitOffsetClause, _textLimit, testing::Eq(textLimit)),
6         AD_FIELD(LimitOffsetClause, _offset, testing::Eq(offset)));
7 };
```

Listing 5.7: New Matchers

To solve the problems that are present in the tests of Section 5.3.2, we introduced two main improvements to the testing code of the parser. An example of an improved test case is in Listing 5.6. The first improvement is `ExpectCompleteParse` (line 2). `ExpectCompleteParse` encapsulates the process of parsing a string as a specific rule and then asserting the parsed data with a `Matcher`. A template parameter allows to specify as which type of clause the queries should be parsed.

The `ExpectCompleteParse` object can then be called with a string and a `Matcher`. The string is parsed as the specified clause and matched against the `Matcher`. Notice that in Section 5.3.2, checking an input required lines 2 to 6. With the improved tests for the new parser, the same test only requires line 4. In the improved version, a different case is checked for lines 4 and 5.

Listing 5.7 shows an improved `Matcher`. This `Matcher` is made up of three `Matchers` that each check a field of the `LimitOffsetClause` (lines 4-6). The `Matcher` only succeeds if every field matches (line 3). QLever's `Matchers` are now in the `matcher` namespace which is abbreviated to `m` in the tests. The `Matcher` is now being constructed of different `Matchers` that are provided by `GTest` in the `testing` namespace. `AD_FIELD` is a helper for the `testing::Field` `Matcher`. The construction of the composite `Matcher` is wrapped in a `lambda`. This way of constructing `Matchers` has the benefit that the error messages are very detailed. The message now also contains information on the exact condition which has failed. These errors are precise enough to find the cause from only the message in most cases. The `lambdas` also explicitly specify the parameter and return types. This makes it possible to omit types in the tests in some cases. E.g., for `vector` initializer lists can be used.

These tests provide an improvement with regard to the three criteria for good tests. The improved way of constructing `Matchers` and the easy creation of `ExpectCompleteParse` make it fast and easy to write new tests. The abstraction of creating the `Matchers` through `lambdas` also are a central place for changes if the application model changes. This makes the tests easy to maintain for most smaller changes like renaming of fields. Because the `Matcher` mimics the actual structure of the data, they are a natural, compact and easy to understand way to represent the expectations.

6 Evaluation

This chapter will contain an evaluation of the new parser. Section 6.1 will briefly evaluate the impact of the new parser on the total execution time of a query. After that, Section 6.2 will evaluate the old and new parser based on their compliance with the SPARQL QL standard. The quality of the parser is finally evaluated in Section 6.3.

The main criterion for the new parser is quality. For QLever, four aspects define the quality of the parser code:

- correctness of the code
- resilience against programming errors
- resilience against user errors
- maintainability

Speed is a less important factor. The actual execution of the query takes much longer than the parsing of the query (see Section 6.1).

Query parsing (ms)	Total execution time (ms)	Share query parsing of total execution time
1	116	0.86%
1	114	0.88%
2	98	2.04%
1	98	1.02%
2	31	6.45%
1	18	5.56%

(a) Olympics (2 mio. Triples)

Query parsing (ms)	Total execution time (ms)	Share query parsing of total execution time
16	24351	0.07%
1	24710	0.00%
2	362	0.55%
1	353	0.28%
78	1205	6.47%
3	1123	0.27%
1	8541	0.01%
1	8638	0.01%
2	399	0.50%
1	454	0.22%
1	1471	0.07%
1	1420	0.07%

(b) DBLP (350 mio. Triples)

Table 1: Share of Query Parsing of the Total Query Execution Time

6.1 Speed

Table 1 shows the execution time of select queries without caching, the time required for parsing the query and the fraction of the parse time in the total execution time. The queries used for testing are the example queries provided by QLever-UI for the respective dataset. The cache of QLever was cleared before each measurement. The times were measured by extending the code that is already used to measure the total execution time of the query. Commit `c6376e4`¹ was used as a base for the measurements. The QLever instance was running on a Linux system with an Intel i7-11700. The index was stored on an NVME SSD. Table 1a contains the result with the “Olympics” dataset. This dataset contains information on athletes at the Olympic games and has about 2 mio. triples. Table 1b shows the measurements with the “DBLP” dataset. The DBLP dataset contains bibliographic information on computer science journals. It has about 350 mio. triples. QLever can handle datasets that contain billions of triples like the complete Wikidata (17 billion triples). The tested queries are the provided examples when using an unmodified QLever-UI instance. Most of the queries are between 200 and 400 characters long. Queries on more complex datasets can also reach a length of around 1500 characters.

We can observe that the parsing of the query only is an insignificant part of the total execution time for a query in all the measured cases. The parsing only took more than 1% of total time in 5 out of 18 measurements. But in 4 of these 5 cases, the total execution time was smaller than 100ms. This has the effect that even queries that are parsed in 1ms are over this threshold.

¹<https://github.com/ad-freiburg/qllever/tree/c6376e46aefc7c15bbbce0eb84eb6422845d2367>

6.2 Standard Compliance

These comparisons will reference Rules of the SPARQL Query Language’s grammar. The grammar of the SPARQL QL can be found in Section “19.8 Grammar” of the SPARQL QL [2].

There are two dimensions along which the parsers will be compared. These are Recognition and Support. Recognition shall be the ability of the parser to correctly identify the input that it is presented. Recognition of the current rule is in most cases required to be able to provide useful error messages. Lacking recognition is always a deficit of the parser. Support shall be the ability of the parser to correctly processes the input and transform it into the application specific model. Only in some cases can missing support for a feature be fixed by changing the parser. In most cases the underlying problem is that the engine code does not support the feature that is being parsed.

The cell values **yes** or **no** note that the specific aspect for this feature is implemented according to the SPARQL QL specifications or not. **partially** describes that a certain aspect is only supported for a subset of the cases specified by the standard. With the new parser, this means that the backing code only supports a subset of the features specified by the standard. **not standard compliant** denotes that a feature is implemented in a way that is not conform to the standard. Like **partially**, this is also a deviation from the standard. In this case, the cause for the deviation is that the parser is more lenient than the standard. Inputs that should have been rejected according to the standard are still parsed successfully. The reason for this may be errors in the parser or an extension of the grammar by QLever in the case of **LimitOffsetClause**.

Feature	Before		After	
	Recognition	Support	Recognition	Support
a as alias for <code>rdf:type</code>	no	no	yes	yes
LimitOffsetClause	not standard compliant	yes	not standard compliant	yes
OrderCondition	partially	partially	yes	yes
GroupCondition	partially	partially	yes	yes
PropertyPath	partially	partially	yes	partially
DataBlock	partially	partially	yes	partially
SelectClause	not standard compliant	partially	yes	partially
TriplesSameSubjectPath	partially	partially	yes	partially
PrefixDecl	not standard compliant	yes	yes	yes
WhereClause	partially	partially	yes	partially

Table 2: Standard Compliance of Select Aspects of the Old and New Parser

6.3 Code Quality

The code quality is of great importance for the parser. For the parser, the quality criteria are correctness of the code, resilience against programming errors, resilience against user errors and maintainability. In contrast to speed, these criteria are much more fuzzy and therefore harder to evaluate in a scientific manner.

The switch to ANTLR (Section 4.2) has the effect that much of the parser code is generated. Code that is not written reduces the risk to introduce programming errors. The modification of the visitor pattern to make it type safe (Section 5.1) also makes it significantly harder to introduce programming errors when handling the return values of recursive visitor calls. Rigid testing (Section 5.3) of the written code also makes it harder to introduce errors in the existing code.

Providing the user with improved error messages (Section 5.2) improves the ability of the user to detect and correct errors in the query.

The switch to ANTLR and the visitor pattern also has the added benefit that it encourages low coupling. Compared to the old parser, the new parser has much lower code coupling. The old parser would parse and transform many rules in one function. The old parser has also shared state in its functions. This means that there were shared variables between the parsing and transformation of different rules. In the new parser the transformation into the application specific model is a separate function for each rule. Separate functions also enforce clear decision on what each rule should be transformed into. This architecture improves maintainability and decreases the coupling. Low coupling is a sign for high code quality.

For this evaluation, we will use the code coverage of the tests as an approximation for the correctness. One disadvantage of code coverage is that it does not take the quality of the tests into account. There are different levels of precision. Some criteria may only measure on the level of functions while others measure the execution of single statements. If we have good tests, it can be assumed that the code is probably correct. Despite these methodical flaws, the test coverage is a good approximation to measure for which fraction of the code we can assume that the code is probably correct.

Table 3a and Table 3b show the coverage of the old tests on the old and new parser. Table 4 shows the code coverage for the new tests on the new parser. The old tests test the parser as one unit. They can therefore be run against both the old and the new parser. The new tests specifically test single functions of the visitor. Therefore, these tests can only be run against the new parser.

After analyzing the old tests in Table 3, we can see that the tests achieve relatively high coverage of the lexer. Weighting the coverage by the number of lines, we get a line coverage of approximately 81% and a branch coverage of approximately 33% for the lexer.

File	Line Coverage	Branch Coverage
SparqlLexer.cpp (218 Lines)	73%	29%
SparqlLexer.h (96 Lines)	100%	43%
SparqlParser.cpp (939 Lines, corrected for unused Code)	60%	38%

(a) Old Parser

File	Line Coverage	Branch Coverage
SparqlLexer.cpp (241 Lines)	63%	26%
SparqlLexer.h (102 Lines)	n.a.	n.a.
SparqlParser.cpp (302 Lines)	44%	26%
SparqlQleverVisitor.cpp (1734 Lines)	67%	30%
SparqlQleverVisitor.h (485 Lines)	50%	13%

(b) New Parser

Table 3: Coverage Old Test Suite

File	Line Coverage	Branch Coverage
SparqlLexer.cpp (241 Lines)	63%	20%
SparqlLexer.h (102 Lines)	n.a.	n.a.
SparqlParser.cpp (302 Lines)	38%	20%
SparqlQleverVisitor.cpp (1734 Lines)	84%	39%
SparqlQleverVisitor.h (485 Lines)	95%	14%

Table 4: Coverage New Test Suite on the New Parser

The more complex parser only has a line coverage of 60% when correcting for unused code. This is a problem because the parser is the more complex component and thus it is more likely to introduce bugs there.

A second observation is that the new tests reach a high coverage on the new parser. The coverage is 24% higher than the coverage of the old tests on the old parser before this work. `SparqlQleverVisitor.h` can be ignored for this inspection because only a small fraction of `SparqlQleverVisitor.h` (less than 50 lines) is code. The majority of `SparqlQleverVisitor.h` are function definitions.

In conclusion, it can be said that investigated metrics all suggest that this work increased the quality of the parsing code in QLever.

7 Further Tasks

- Implementation of missing built-in functions. The SPARQL QL defines around 60 built-in functions. These include hashing operations, string operations (upper/lower casing, regex matching, length), mathematical functions (absolute value, ceiling function, floor function), time operations (get components of a timestamp) and more. Most of them are currently not supported by QLever. Work has already been to make the implementation of additional functions quick and easy. Some work is currently also being done on implementing some of the more requested functions. The scope of this task should be manageable, and the work required should be on the order of several weeks. One area where we see problems arising is if data types required for a function are not at all or not sufficiently implemented. This could require changes to the data types and would result in an extended amount of work required.
- Filter parsing using ANTLR. Filters are the only parts of a query that are still parsed with the old parser. Once the necessary changes on the engine side are done it should be possible to also parse filter using ANTLR. This would make it possible to support a wider range of filters. Additionally, this would make it possible to remove the old parser and lexer entirely. The scope of this task is highly dependent on the scope of the changes on the engine side. Adding expressions that work with Boolean values is currently still required. Implementation of some built-in functions can also be required. The general idea is to replace the custom `SparqlFilter` with expressions.

A short term solution might be to bind the filter condition to an internal variable. This variable can then be used in a filter that only filters on a variable. This short term solution should be implementable in a few days. A long-term solution for filters would also have to support query optimization. This includes, among other things, heuristics for how expensive it is to evaluate the filter and how many elements the filter filters out.

- Implementation of missing features of the SPARQL QL. QLever currently does not support every feature of the SPARQL QL standard. With regard to the missing features, the built-in functions are one of the more often requested features. There are still other features of the SPARQL QL standard that QLever does not support or where QLever deviates from the standard. One example to illustrate this is the **Values** Clause. Empty values clauses are not supported. Only IRIs and RDF literals are currently allowed in values clauses. Numeric and Boolean literals as well as undefined values are not yet supported. Another example is the **Where** clause. The implementation currently requires that a **Where** clause must not be empty although it may be empty according to the standard. The features for named graphs and federation can be used inside a **Where** clause. They are also not supported.

These two examples show the big scope of this task. Single subtasks should be completable in a couple of weeks. But achieving conformity with the standard in all areas would take many months.

- More consistent use of strong types. There are two classes of types which we distinguish when talking about the modeling of data in QLever. The concept is similar to the concept of weakly and strongly typed programming languages. IRIs, variables, and literals can all be represented through a single string. But in that case, the context of the string is required to be able to know what the string represents. From a type perspective, strings that contain a variable or an IRI cannot be discerned. We call this a weak type.

Strong types on the other hand would have distinct types for IRIs, variables, and literals. These types can then enforce invariants like the property that variables names must start with “\$” or “?”. These types can also provide helper functions specific to the type. There are already strong types for many objects like IRIs and variables, but they are not used consistently. The parser uses these strong types in most places. The downstream engine only rarely uses the strong types. It is often the case that the strong types are converted to strings for their future usage at some point. Most of this work will be in the code of the engine. Applying this consistently over the whole codebase should be a matter of many weeks, to a couple of months.

- Rewriting the existing tests for the old SPARQL parser using Matchers and merge them with the tests write for the new ANTLR based parser. This task can also be extended to rewrite the tests for other components using Matchers. Only knowledge about GTest is really required to rewrite the tests. Rewriting the tests should be completable in a few days.

8 Acknowledgments

First and foremost, I would like to thank my adviser Johannes Kalmbach who spent countless hours reviewing my code to help bringing it to an acceptable quality.

I am extremely grateful to Prof. Dr. Hannah Bast for agreeing to supervise my thesis.

I am also thankful to my mother, Heike Mundhahs, and my friends Gerrit Freiwald and Mario Goltz for proofreading my thesis. They helped with making the thesis more readable and easier to understand.

Bibliography

- [1] “SPARQL 1.1 overview,” W3C recommendation, W3C, Mar. 2013. <https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/>.
- [2] A. Seaborne and S. Harris, “SPARQL 1.1 query language,” W3C recommendation, W3C, Mar. 2013. <https://www.w3.org/TR/2013/REC-sparql11-query-20130321/>.
- [3] L. Feigenbaum, K. Clark, E. Torres, and G. Williams, “SPARQL 1.1 protocol,” W3C recommendation, W3C, Mar. 2013. <https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/>.
- [4] “Antlr v4.” <https://github.com/antlr/antlr4>.
- [5] P. Hayes and P. Patel-Schneider, “RDF 1.1 semantics,” W3C recommendation, W3C, Feb. 2014. <https://www.w3.org/TR/2014/REC-rdf11-mt-20140225/>.
- [6] R. Cyganiak, D. Wood, and M. Lanthaler, “RDF 1.1 concepts and abstract syntax,” W3C recommendation, W3C, Feb. 2014. <https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/>.
- [7] G. Carothers and E. Prud’hommeaux, “RDF 1.1 turtle,” W3C recommendation, W3C, Feb. 2014. <https://www.w3.org/TR/2014/REC-turtle-20140225/>.
- [8] “Wikidata Query Service (WDQS).” <https://query.wikidata.org/>.

- [9] M. Duerst and M. Suignard, “Internationalized resource identifiers (iris),” RFC 3987, RFC Editor, January 2005. <http://www.rfc-editor.org/rfc/rfc3987.txt>.
- [10] H. Bast, B. Buchholz, J. Kalmbach, *et al.*, “Qlever.” <https://github.com/ad-freiburg/qlever>.
- [11] H. Bast, J. Bürklin, D. Kemen, *et al.*, “QLever UI.” <https://github.com/ad-freiburg/qlever-ui>.
- [12] H. Bast, B. Buchholz, J. Kalmbach, *et al.*, “QLever Demo.” <https://qlever.cs.uni-freiburg.de>.
- [13] “Matchers reference.” <https://github.com/google/googletest/blob/main/docs/reference/matchers.md>.

