Towards tackling SPARQL heterogeneity through modular parsing

Jitse De Smet [0009-0002-6513-5013], Ruben Taelman [0000-0001-5118-256X]

IDLab, Department of Electronics and Information Systems, Ghent University – imec.

Abstract

The SPARQL ecosystem has become increasingly fragmented as engines introduce valuable but incompatible language extensions. This growing diversity undermines query portability, tooling reliability, and the pace of innovation. To address this, we designed a modular parser architecture that supports dynamic extension and modular grammar definitions. This paper presents a builder-based, TypeScript-native parser framework inspired by Chevrotain and the modular principles of Comunica. Our prototype demonstrates that key SPARQL extensions can be integrated, altered, or removed with minimal effort and strong type safety. These results suggest that modular, declarative parsing is not only feasible but essential for keeping pace with evolving SPARQL standards. Looking forward, we identify the need for round-trippable ASTs, Babel-inspired generators, and transformer pipelines to enable a complete, future-proof SPARQL toolchain.

KEYWORDS: SPARQL, SPARQL TOOLING, MODULAR PARSER

Source Code: https://github.com/comunica/traqula
Demo: https://modular-parsing.demo.jitsedesmet.be/
Canonical version: https://traqula-demo-semantics-2025.jitsedesmet.be/

1. Introduction

The SPARQL query language [1], a cornerstone of the Semantic Web stack, has evolved through both standardisation and real-world innovation. While SPARQL 1.1 defines a clear and extensible foundation, the ecosystem has gradually diverged as implementers introduced powerful but engine-specific extensions. For example, Virtuoso offers full-text search capabilities [2], Apache Jena supports CONSTRUCT QUAD queries [3], and Oxigraph provides extended date-time-functionality including the ADJUST function [4]. These features are often highly valuable, but also incompatible, creating a heterogeneous landscape where queries that run on one engine may fail on another.

This diversity presents a serious challenge for SPARQL portability, tooling and federated querying. With the finalisation of the SPARQL 1.2 specification [5], the gap between supported language features is likely to widen further, since migration to SPARQL 1.2 is not trivial, requiring substantial updates to the datasets representation and underlying RDF store [6]. The RDF1.1 to RDF1.2 update is substantial mainly because of the introduction of a new triple term, specifically the object of a triple can now be a triple itself, allowing for the recursive definition of triples since the triple contained in the object can again have a triple in the object spot.

Moreover, the working group has announced that after SPARQL 1.2 finalisation, they plan to move toward a more agile “maintenance and new features” mode, which hints at even faster iteration cycles in the future. As a result, there is a growing need for tooling that embraces extensibility and modularity by design.

In this work, we show the need for a modular parser and what such a parser could look like. Unlike traditional parser generators such as ANTLR [7] or Bison [8], which rely on Domain Specific Languages (DSLs) and generate static parsing code — our parser should be defined entirely within a host programming language. This would eliminate the compile step, enable programmatic extension, and leverage strong typing to provide a safer, more developer-friendly API. The parser should not be a handwritten parser either, instead it should use declarative rules such as present in the Typescript-based Chevrotain parser toolkit [9].

A modular parser, that allows you to add, override, or swap grammar fragments at runtime, would empower both researchers and practitioners to create a new generation of language-aware SPARQL tools. This opens the door to use cases such as heterogeneous query tooling (e.g., adapting editors like YASGUI [10] to custom SPARQL dialects), while keeping maintainability in check. Additionally, it would allow SPARQL version translation, and rapid experimentation with new language features. In an ecosystem where SPARQL flavors are growing rather than converging, we believe modularity is not just a nicety—it's a necessity.

The next section touches lightly on the related work, while Section 3 describes the system architecture. Section 4 sketches the demonstration that we will provide to the workshop. In Section 5 we conclude the future work and desired impact of this research.

2. Related Work

In this section, we examine prominent software packages in the SPARQL ecosystem that implement parsing capabilities. Our findings are summarized in Fig. 1.

Notably, all discussed major open-source SPARQL parsers rely on either parser generators or parser-building toolkits to define their grammars. In compiled languages such as Rust or Java, the parser generation step can be integrated directly into the main build step—e.g., Oxigraph uses rust-peg for this purpose. Interestingly, in our survey only Stardog's Millan does not use a parser builder. Instead, it uses Chevrotain without constructing an Abstract Syntax Tree (AST); it appears to focus solely on validation rather than full syntactic analysis.

This highlights a broader pattern: while parser generators dominate SPARQL tooling, few systems are designed with modularity or extensibility as a first-class concern. In particular, full modularity—including the ability to remove grammar rules—is not supported in current public implementations, making adaptation or evolution of these parsers difficult.

Software Package	Parsing Software	Parser Generator
Comunica	SPARQL.JS	Jison
Yasgui		SWI Prolog
Apache Jena		JavaCC
Oxigraph		rust-peg
Stardog - Millan		Chevrotain
Virtuoso		Bison
Blazegraph		JavaCC
GraphDB	RDF4J	JavaCC

Fig. 1: Each row lists a widely used software package, its associated parsing library, and the parser generator employed. When the parsing software is omitted, the parser is implemented directly within the project. For each usage claim, we provide a link to back up the claim.

3. Software Architecture

Parsers are typically implemented in one of three ways:

1. **Hand-built parsers**: These are manually implemented parsers tailored to a specific grammar. While they can be highly performant due to low-level optimizations and language-specific design, they are often difficult to maintain, extend, or modularize.
2. **Parser generators**: Tools such as ANTLR [7] and Bison [8] use a Domain Specific Language (DSL), typically based on Extended Backus–Naur Form (EBNF), to define a grammar. These grammars are then compiled into standalone parser code. While powerful, such approaches introduce a compile step and tend to be rigid, making modular extensions cumbersome.
3. **Parser building toolkits**: Libraries such as Chevrotain [9] offer a hybrid approach, enabling declarative grammar specification within a host programming language. These toolkits eliminate the compile step and allow for flexible, programmatic grammar definitions with fine-grained control over behavior and integration.

To support modularity while keeping the mental model approachable, a modular parser should be build using a parser building toolkit. Parsing itself is typically divided into multiple phases [11], of which the following are relevant to this work:

1. **Lexical Analysis (*scanning*)**: A lexer transforms a character stream into a token stream.
2. **Syntax Analysis (*parsing*)**: A parser transforms the token stream into an abstract syntax tree (AST).
3. **Semantic Analysis**: Performed during or after parsing, this phase validates constraints not enforced by grammar alone. For instance, SPARQL forbids binding to a variable which is already in scope.

Inspired by the Comunica modular query engine [12] codebase, the codebase of a modular parser should not be a big monolith but instead use many smaller packages that can be tied together to serve a larger purpose. To facilitate the maintainability of many small packages a monorepo *(https://monorepo.tools/)* structure could be considered. Within the Comunica codebase, the usage of small packages allows it to define many different builds (eg. a minimal built for the web, and a general built with and without file system access). Similar benefits can be expected in the adoption of such a structure within the modular parser:

1. **Engines**: These are prebuilt, ready-to-use components such as SPARQL 1.1 and 1.2 parsers or generators.
2. **Non-engine packages**: These expose modular building blocks used to construct engines, such as grammar fragments or core construction utilities.

However, unlike Comunica which uses Components.js, a dependency injection framework using RDF based config files, the modular query engine can be configured within the host language itself since components share a similar interface. We propose that a parser be build using a builder pattern and that parser packages export the builder used, so other may extend upon it. Using a builder pattern for the parser allows you to take a builder that is used to build one parser and manipulate the grammar rules to construct a new parser.

Concretely, we propose a builder which allows rules to be registered by name into a rule map, thereby creating a loose coupling between registered rules. Each rule is defined as a `ParserRule` object, containing both a rule name and a rule implementation. Rule implementations can be expressed declaratively using Chevrotain's grammar definition functions like:

1. `SUBRULE`: invokes another rule, registered under some name in the current parser,
2. `MANY`: matches zero or more occurrences of a pattern,
3. `OR`: matches one of several alternatives.

We propose, each rule implementation returns a function that, when invoked, receives the parsing context and any parameters, and outputs part of the final syntax tree. Listing 1 shows an example parser rule definition. The ParserBuilder can then be used for compositional construction and extension through methods like `addRule`, `deleteRule`, `merge`, and typePatch. The typePatch utility would enable type updates to existing rules — particularly useful when extending or modifying a dependent rule without altering the original rule's implementation. After the construction of your parser, you can build it, as shown in Listing 2, returning a parser which allows you to start parsing a string from any of the parser rules added to the builder - a property transferred from the underlying parser builder toolkit.

```
import type { SparqlRule } from '@traqula/core';
const iriOrNil: SparqlRule<'iriOrNil', URL | null> = <const>{
  name: 'iriOrNil',
  impl: ({SUBRULE, CONSUME, OR}) => () => OR<URL | null>([
    {ALT: () => SUBRULE(iri, undefined)},
    {ALT: () => {
        CONSUME(nilToken);
        return null;
      } },
  ]),
};
```

Listing 1: The definition of a parser rule parsing either a URI of the nil token, returning the parser URI or null respectively.

```
import { ParserBuilder } from '@traqula/core';
const parser = ParserBuilder
  .create([ iriOrNil, rule1 ])
  .addRule(rule2)
  .patchRule(rule1Alternative)
  .build({
    tokenVocabulary: myLexerBuilder.tokenVocabulary,
  });
// The argument and return types of the function are known,
// ast will thus be inferred to have the type `URL | null`.
const ast = parser.iriOrNil(myString, myContext, myParameters)
```

Listing 2: The construction of a parser including the iriOrNil rule constructed in Listing 1. It also shows how to parse using the iriOrNil rule as the starting rule.

As for the lexer, a similar approach to the parser should be taken. Tokens should be coupled loosely through a name-definition map. The consumption of a token then results in the consumption of the token with that name in the used lexer. Besides that our only requirement is that the tokens can be expressed through the definition of a regex.

4. Demonstration

In the workshop demonstration, we will showcase how our proof of concept modular parser-builder enables straightforward modification and extension of the existing parsers. Starting from a prebuilt SPARQL 1.1 parser, we will incrementally evolve the grammar in four small steps using the described builder-based architecture. Each change will be demonstrated live, with code edits performed in an IDE and parser behavior verified in a browser-based UI. Specifically, we will:

1. extend SPARQL to support the ADJUST function [4],
2. add support for CONSTRUCT QUAD queries [3],

3. introduce full-text search capabilities [2], and
4. remove support for the OPTIONAL clause due to its impact on query complexity [13].

These modifications will demonstrate how the modular parser architecture—built around builders enables safe and modular grammar changes with minimal effort. The focus will be on how individual grammar components can be extended or replaced without touching unrelated parts of the parser. We will also highlight how the use of strong typing improves the developer experience by surfacing integration errors at compile time.

For each of the extensions we alter the grammar rules in accordance to the SPARQL 1.1 specification [1] (rule number shown between paratheses):

1. **ADJUST function**: We add an '`ADJUST`' token to the lexer and add a grammar rule for it, then patch the BuiltInCall (121) rule.
2. **CONSTRUCT QUAD queries**: Following Jena's approach, we patch the ConstructQuery (10) and ConstructTriples (74) rules and introduce a ConstructQuads rule.
3. **Full-text search**: We patch the objectPath (86) and object (80) rules to allow an '`OPTION`' keyword followed by a scoring clause like '`( score Expression )`'.
4. **Dropping OPTIONAL**: This involves deleting the OptionalGraphPattern (57) rule, patching the GraphPatternNotTriples (56), and removing the '`OPTIONAL`' token from the lexer.

While the demo is not interactive for attendees, all code and tooling will be made available for experimentation after the session. The demo will serve to illustrate how a modular parser builder enables a new generation of language-aware SPARQL tools with modular, declarative grammar support and a strong developer experience.

5. Conclusion

In this paper, we presented the need for a modular parser, and offered an initial prototype to cover this need. Our prototype uses a builder-based architecture for constructing extensible SPARQL parsers. By embracing runtime modularity, declarative rule definitions, and strong typing, our approach enables a new class of SPARQL tools that can evolve alongside a rapidly diversifying query ecosystem. Through our demonstration, we showed that parser modification can be performed with minimal overhead and high confidence in correctness.

Looking ahead, several important challenges remain.

1. In order to bootstrap the adoption of the modular parser, a robust, default parser with a well-defined Abstract Syntax Tree (AST) format should be created. This AST should support round-tripping—ensuring that a query parsed into the AST and then regenerated from it yields a string-identical query. This requirement on the AST will facilitate the creation of language tools such as linters.
2. To support such round-tripping, we will need to design a corresponding generator. This generator could follow architectural patterns established by the Babel JavaScript compiler [14] combined with the builder pattern described in this work.
3. We envision the need for a flexible AST transformer system that makes it easy to map the AST into alternative representations. Such a transformer will facilitate static analysis, query-optimization, and translation to other query languages.

Together, these next steps would complete a robust pipeline: from parsing, through transformation,

to code generation—all powered by modular, declarative components. We hope this work provides a foundation for building SPARQL tools that are not only adaptable to change, but actively enable it.

Acknowledgements. Jitse De Smet is a predoctoral fellow of the Research Foundation – Flanders (FWO) (1SB8525N). Ruben Taelman is a postdoctoral fellow of the Research Foundation – Flanders (FWO) (1202124N).

References

[1] Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 Query Language. W3C, https://www.w3.org/TR/2013/REC-sparql11-query-20130321/ (2013).
[2] Virtuoso: Using Full Text Search in SPARQL. https://docs.openlinksw.com/virtuoso/sparqlextensions/#rdfsparqlrulefulltext (2024).
[3] Apache Jena: ARQ - Construct Quad. https://jena.apache.org/documentation/query/construct-quad.html#Grammar (2024).
[4] Oxigraph: SEP 0002: calendar and duration operations. https://github.com/oxigraph/oxigraph/wiki/SPARQL#sep-0002-calendar-and-duration-operations (2024).
[5] Hartig, O., Seaborne, A., Taelman, R., Williams, G., Tanon, T.P.: SPARQL 1.2 Query Language. https://www.w3.org/TR/sparql12-query/ (2025).
[6] Hartig, O., Champin, P.-A., Kellogg, G., Seaborne, A.: SPARQL 1.2 Query Language. https://www.w3.org/TR/rdf12-concepts/ (2025).
[7] Parr, T.J., Quong, R.W.: ANTLR: A Predicated - LL(k) Parser Generator. Softw. Pract. Exp. 25, 789–810 (1995). doi:10.1002/SPE.4380250705
[8] GNU: GNU Bison. https://www.gnu.org/software/bison/ (2025).
[9] Chevrotain, Soel, S.: Chevrotain - Parser Building Toolkit for JavaScript. https://github.com/Chevrotain/chevrotain/ (2025).
[10] Rietveld, L., Hoekstra, R.: The YASGUI family of SPARQL clients. Semantic Web. 8, 373–383 (2017). doi:10.3233/SW-150197
[11] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, https://www.worldcat.org/oclc/12285707 (1986).
[12] Taelman, R., Herwegen, J.V., Sande, M.V., Verborgh, R.: Comunica: A Modular SPARQL Query Engine for the Web. In: Vrandecic, D., Bontcheva, K., Suárez-Figueroa, M.C., Presutti, V., Celino, I., Sabou, M., Kaffee, L.-A., and Simperl, E. (eds.) The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II. pp. 239–255. Springer (2018). doi:10.1007/978-3-030-00668-6_15
[13] Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. 34, 16:1–16:45 (2009). doi:10.1145/1567274.1567278
[14] sebmck, nicolo-ribaudo, hzoo: Babel. https://github.com/babel/babel (2025).