DOI: 10.15514/ISPRAS-2024-36(2)-2

TQL: a Case Study of Integrating DSL in a Product

A.D. Belousov, ORCID: 0009-0005-3641-1252 <*flygrounder@yandex.ru*>

Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000, Russia.

Abstract. Domain-specific languages power numerous modern applications and libraries, including but not limited to: Wolfram Alpha, Microsoft Excel, Graphviz. This work aims to share the experience gathered from developing TQL (Talisman Query Language) – a domain-specific language used in Talisman platform. Talisman platform is a set of tools to automate data processing tasks, developed by Ivannikov Institute for System Programming of RAS. TQL implementation, discussed in this article, supports error-recovery, can be run directly inside a browser as well as on a server, and it also has an interactive playground that visualizes the parse tree while typing. This article describes several techniques and technologies that were used to make these qualities possible while keeping a single, maintainable codebase.

Keywords: programming; compilers; domain-specific languages.

For citation: Belousov A.D. TQL: a case study of integrating DSL in a product. *Trudy ISP RAN/Proc. ISP RAS,* vol.36, issue 2, 2024. pp. 21-32. DOI: 10.15514/ISPRAS-2024-36(2)-2.

TQL: тематическое исследование внедрения предметно-ориентированного языка в продукт

А.Д. Белоусов, ORCID: 0009-0005-3641-1252 <*flygrounder@yandex.ru*>

Институт системного программирования РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.
Высшая школа экономики,
Россия, 101000, г. Москва, ул. Мясницкая, д. 20.

Аннотация. Предметно-ориентированные языки используются во многих современных приложениях и программных библиотеках, среди них: Wolfram Alpha, Microsoft Excel, Graphviz. Данная работа описывает опыт, полученный при разработке TQL (Talisman Query Language) – предметно-ориентированного языка, используемого в платформе Talisman. Платформа Talisman – набор инструментов для автоматизации задач обработки данных, разработанный институтом системного программирования РАН. Реализация TQL, описанная в данной статье, поддерживает восстановление после ошибок, может запущена как в браузере, так и на сервере, и имеет интерактивную среду, позволяющую визуализировать дерево разбора во время печати. Данная статья описывает подходы и технологии, которые сделали данные качества возможными при единой, поддерживаемой кодовой базе.

Ключевые слова: программирование; компиляторы; предметно-ориентированные языки.

Для цитирования: Белоусов А.Д. TQL: тематическое исследование внедрения предметно-ориентированного языка в продукт. Труды ИСП РАН, том 36, вып. 2, 2024 г., стр. 21–32 (на английском языке). DOI: 10.15514/ISPRAS–2024–36(2)–2.

1. Introduction

A domain-specific language (DSL) is a computer programming language of limited expressiveness focused on a particular domain. [1] There are multiple reasons why one might choose to integrate a DSL in their application:

- It can be used by people, who are not familiar with the traditional software engineering, but are familiar with the application domain
- It can be used to limit actions that users can perform in order to preserve security and privacy of other users
- It can be more expressive than general-purpose languages in a particular domain

This work is based on the experience, gathered from developing TQL — a query language for Talisman platform. [2] Talisman is a unified set of tools that automate typical data processing tasks, such as data retrieval, integration, analysis, storage and visualization. It is developed by Ivannikov Institute for System Programming of RAS. One of the core components of the platform is it’s knowledge base. It stores documents that can be related to each other, they can be searched by using both relational and full-text patterns. In order to allow such search queries, data is stored in two different systems: in a relational database (PostgreSQL) and a full-text search system (Elasticsearch). Both have their own query languages, which allow only one type of search. In order to combine capabilities of both languages, a new domain-specific language was developed — Talisman Query Language.

User experience for a DSL or any programming language in general depends on the language support and tooling surrounding it. The initial implementation of TQL included only a parser for query execution, lacking language support for users to write queries. This led to the following problems while using the language:

- No syntax error reporting. When a user accidentally forgot to close a bracket or made any other type of syntax error, they would have to find this error manually. This can be improved by reporting which tokens in a query led to a syntax error.
- Difficult to navigate inside complex queries. When having a complex query with nested selectors, it becomes troublesome to identify, which part of the query is nested inside which selector. This can be improved by highlighting keywords and highlighting matching brackets with the same color.
- Having to read the manual to recall the correct keyword. While being powerful, TQL has a lot of keywords that one needs to learn in order to use it efficiently. Suggesting keyword completions can ease that learning curve for users.

A solution based on the existing TQL parser was considered, but it had the following technological limitations:

- Language parser did not include error recovery, so it could only find the first syntax error.
- Parser could only be run in the same environment as the rest of Talisman backend: on a server.

Based on these issues, it was decided to create a new TQL implementation.

The primary focus of this article is to demonstrate how a single implementation of a DSL can offer a seamless editing experience within a browser, while also being utilized on a server for processing user programs. Attempting to meet these requirements poses several challenges, which are outlined in this article along with the solutions employed by the new TQL implementation.

2. Background

2.1 Parser implementation

There are several choices when implementing a parser:

- Implementing it manually
- Using a combinator library
- Using a parser generator

A manual implementation provides developers with the highest degree of flexibility and control over parser behavior. It allows implementing custom error recovery mechanisms and choosing any parsing algorithm. However, this flexibility comes at a cost: manual parser implementations tend to be repetitive and error-prone.

Combinator libraries provide useful building blocks for building a parser. These libraries make parser code easier to read and more composable: each parsing routine is a combinator, that can be then used in other combinators. This approach is less flexible than the manual one: combinator libraries are built on some particular parsing algorithm. Also, they tend to have problems with error recovery, but more on this later.

Parser generators use the most automated approach: they accept a formal grammar as an input and generate a parser automatically. Generators usually have some ways to customize the resulting parser with user-written code, but these capabilities are limited compared to manual parsing and combinator libraries. Error recovery capabilities are usually limited with some set of predefined options, that can be customized only to some degree.

2.2 Error recovery in combinator libraries

Traditionally, combinator libraries and parser generators implemented some variations of LR parsing. This approach gives some flexibility in expressing a language grammar: for example it allows left-recursive rules. However, it makes writing custom error recovery difficult. Usually libraries and generators provide some predefined strategies for error recovery. These strategies are usually enough for error reporting, but not for providing completions. This is one of the reasons, why popular language servers implement manual recursive-descent parsers with custom recovery logic.

There are some combinator libraries, that use recursive descent algorithm and provide capabilities for custom recovery logic: megaparsec [3] for Haskell and chumsky [4] for Rust. Recursive descent imposes some limitations on grammar: for example it does not allow left-recursive rules. Other than that, this approach provides usability of combinator libraries with flexibility of manual parsing.

2.3 CST vs AST

The initial implementation of TQL parsed a query directly into an abstract syntax tree (AST). AST is a tree, representing a program’s structure. As an example, if-then-else node would have three children: a condition, a positive branch and a negative branch. This structural information is necessary to analyze and execute the program. AST is an appropriate abstraction for executing a query, but is not appropriate for providing editor support. While editing a query, most of the time, user input won’t be syntactically correct, so AST cannot be built. In such contexts, concrete syntax tree or CST is often used. Unlike AST, each token of original input is included in CST with it’s coordinates (row, column) from the original input. When a parser cannot parse the program, it might skip some tokens, presenting them as an error node. This way a partial tree can be built and analyzed.

In TQL implementation both CST and AST are used:

- Initially, a parser produces a CST, which can be partial in case of syntax errors. This tree is used to provide language support while editing.

- In case no syntax errors were discovered, CST can be transformed to an AST. It is used by Talisman backend for executing a query.

2.4 Language server protocol

Features like syntax highlighting and autocomplete are provided by code editors and IDEs, in this section we will look into problems that they face and solutions that emerged over time.

Language support has been traditionally provided for each editor individually. Usually, that would require developing a plugin for each editor, using different technologies and architecture. This leads to duplication of efforts: adding a single feature has to be done separately for each supported editor. Inconsistency is another issue with this model: each editor might have different language features available.

These problems led to the creation of Language Server Protocol [5] or LSP for short. It was created by Microsoft for Visual Studio Code editor in 2016 and since then became an industry standard for providing language support. This protocol defines messages that a language client (an editor) and a language server use to communicate with each other. These messages are language-independent, treating a program as a text file and utilizing text coordinates (rows and columns). This protocol allowed editor developers to implement it, benefiting from all language servers that also implement it. Developers working on editor support could now implement language features once and benefit from all the editors implementing a client.

Fig. 1 shows messages defined by LSP, that client and server send each other.

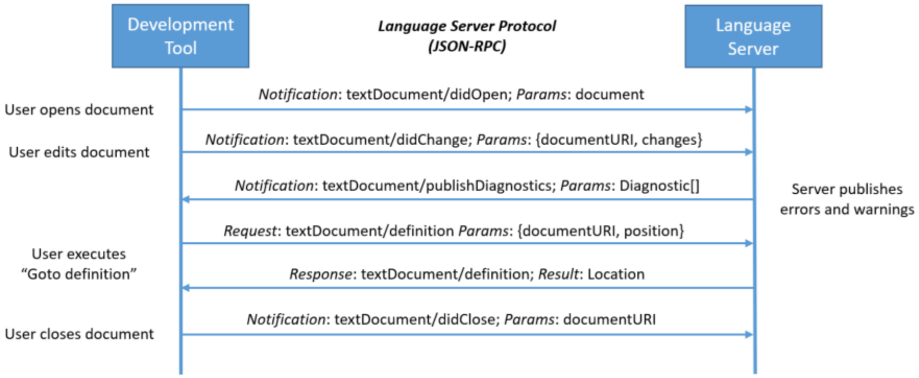

Development Tool
Language Server Protocol (JSON-RPC)
Language Server
User opens document
Notification: textDocument/didOpen; Params: document
User edits document
Notification: textDocument/didChange; Params: {documentURI, changes}
Notification: textDocument/publishDiagnostics; Params: Diagnostic[]
Server publishes errors and warnings
Request: textDocument/definition Params: {documentURI, position}
User executes "Goto definition"
Response: textDocument/definition; Result: Location
Notification: textDocument/didClose; Params: documentURI
User closes document

Fig. 1. Language Server Protocol

We've decided to use this protocol to provide language support for TQL. It allows integrating our language with any tooling that also implements this protocol.

3. Related Work

One tool that can be used to create compiler frontends is Bison. [6] In order to use it, one must describe a grammar and semantic actions that correspond to different grammar rules. Bison tool then generates a parser that uses GLR algorithm in order to process inputs. However, LR parsers tend to be less error-resilient than recursive-descent ones, as shown in this [7] article. Error-resilience is an important property when it comes to language servers, that is why this work does not use an LR parser generator.

Not all parser generators use variations of LR algorithm. One prominent example that does not use it is ANTLR [8]. Generated parsers implement LL(*) algorithm, which uses top-down approach. ANTLR has a few built-in error recovery strategies and it allows implementing custom ones. Built-in strategies use generalized algorithm, that can produce decent results, but the best error recovery can be achieved by using custom error strategies for different rules. Each rule in the generated code is wrapped in a try/catch block that, in case of an error, reports it and attempts to recover. However, recovering from low-level rules can produce undesirable syntax trees as these rules may lack some context that higher-level rules have. In order to disable this behaviour, one must explicitly rethrow an error in all such rules. This clutters the code and makes it less maintainable. In a manual parser, errors might be handled only in rules where it is relevant, avoiding clutter in rules where it is not. While ANTLR theoretically has enough flexibility to implement different error recovery strategies for different rules, it makes it quite difficult in practice, as it is not an intended use case. Xtext [9], that was specifically created to make writing language servers easier, uses ANTLR for parsing and thus suffers from the same problems.

4. Implementation

4.1 Requirements

The new implementation of TQL, that we will call TQLS (Talisman Query Language Server), had to meet the following requirements:

- It had to provide an HTTP API to be used programmatically and a frontend library to be used directly in user's browser
- Language implementation should support:
    - Error reporting
    - Syntax highlighting
    - Autocomplete
- It had to be integrated with the main Talisman backend

These requirements determined technologies and techniques used to implement the language.

4.2 Introduction to TQL

TQL is used to search for entities in Talisman knowledge base. Most common entity types include:

- Document – a piece of information that was retrieved from some source.
- Concept – an entity, that is known to the knowledge base. Concepts can be mentioned in documents.
- Link – represents a connection between two concepts.

Concepts and links form a typed labeled property graph. Each concept and each link has a type, that determines their properties. Consider the following example: document might mention two people, that are known to the knowledge base as concepts. These concepts would have type “person”. Two people might have a link of type “friendship”, that links them together. Concepts of type “person” might have properties like name, birth date, gender, etc. Links of type “friendship” might have a property that says when this friendship was established.

TQL operators can be divided into the following groups:

- Full-text operators are used to search in the text representation of documents.
- Relational operators are used to find entities based on their properties and their relations to other entities.
- Universal operators can be applied both to full-text and relational expressions.

Queries below illustrate some full-text operators:

- *apple* – searches for documents that contain word “apple”.
- *-apple* – searches for documents that do not contain word “apple”.
- *apple~* – searches for documents that contain “apple” using fuzzy search.

- *“apple pie”~1* – searches for documents that contain phrase “apple pie” with at most one word between them.

The following queries demonstrate relational operators:

- *concept(apple)* – searches for documents mentioning a concept named “apple”.
- *concept.fruit(apple)* – the same as above, but this concept should have a type “fruit”.
- *identifier=OK-1* – searches for a document with the identifier “OK-1"”.
- *concept(document(identifier=OK-1))* – searches for documents that mention some concept that is also mentioned in a document with the identifier OK-1. Selectors can be nested inside each other.

Queries below demonstrate universal operators that can be used with both full-text and relational expressions:

- *concept(apple) fruit* – searches for documents mentioning a concept named “apple”, while also containing a word “fruit” in them. Logical “and” is represented as a space.
- *concept(apple),fruit* – searches for documents mentioning concept named “apple”, or containing a word “fruit” in them. Logical “or” is represented as a comma.
- *concept(apple) (fruit,food)* – searches for documents mentioning a concept named “apple” and containing a word “fruit” or “food” in them. Parentheses change evaluation order. By default, “and” has a higher priority than “or”.

Evaluation of TQL queries is performed by the Talisman backend and is beyond the scope of this work. One important consideration is that TQL queries are not compiled ahead-of-time to SQL and Query DSL. Instead, the resulting AST is evaluated by Talisman backend in a bottom-up manner, using the appropriate search mechanism at each step.

4.3 Language as a Service

It might seem natural to include a DSL implementation into the host application’s codebase. However, treating a DSL as a separate product provides several benefits:

- One might choose a more appropriate tech stack for implementing a DSL.
- It allows using a DSL in different environments: i.e., inside a native app, on a website and on a server.
- Makes it easier for a different team to work on the project: they can use a different release schedule, version control and tooling.

For the remaining of this work, we will be calling this approach “Language as a Service” or LaaS. LaaS allows one to design and implement a DSL separately from designing and developing the host product. Modern applications tend to be multi-platform: there can be a native application, a website and an HTTP API. By abstracting away platform-specific details, one might focus on making a better DSL. API for integrating with platform-specific code tends to be a thin wrapper of this core abstract API. Later we will discuss possible technology choices, that allow such integrations.

4.4 Running TQLS inside a browser

Editors, providing language features like autocomplete, that were examined during our research made requests to a remote server for providing completions using either HTTP or WebSocket protocol. This approach leads to more delay before providing completions due to network requests, requires a constant Internet connection and creates more load on servers. This is why one of the design goals for TQLS was being able to run it inside a browser.

Traditionally, writing something for the browser assumed a particular tech stack: HTML, CSS and JavaScript. While these technologies still power the majority of the web, there are some new ones that allow using a different stack. One of the most prominent ones is WebAssembly. It defines a special binary format that is supported in modern browsers, that can be executed safely inside a sandbox. Different languages support compilation in WebAssembly which allows a wider tech stack choice. These binaries can be distributed using conventional npm packages which makes it easy to use for frontend developers.

4.5 Running TQLS on a server

While language features like providing completions can be run inside a browser, there still was a need in providing a server access to parser. Talisman backend had an old implementation of TQL parser. Supporting two different parsers written in two different programming languages would impose a huge burden on the developers and would lead to inconsistencies between language support for the user and for executing queries. TQLS uses GraphQL [10] API for exposing a backend API.

GraphQL has several benefits over traditional REST APIs:

- API clients can be generated from the schema
- Schema can be automatically validated against having breaking changes
- There is no need to write a separate endpoint for each possible query. Server exposes all the data it has available and client selects the exact subset it needs

Tree is represented as a flat list, where each element has a unique identifier. Different element types have different GraphQL types, all of which implement Element interface.

```
type Tree {
    rootId: String
    nodes: [Element!]!
}
interface Element {
    data: Metadata!
}
type Metadata {
    id: String!
    start: Int!
    end: Int!
}
```

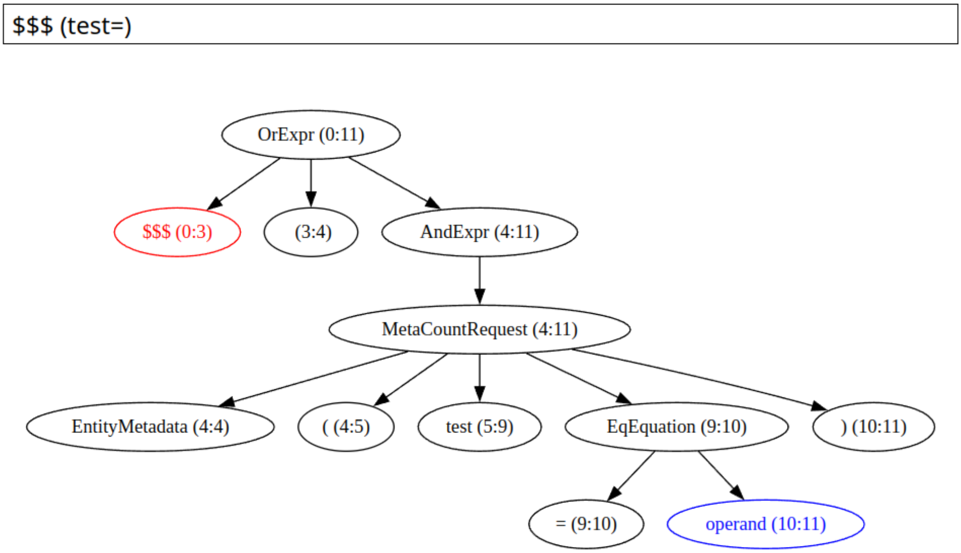
4.6 Error recovery

Error recovery is implemented as a set of heuristics, that are based on common usage patterns for TQL. Error recovery may add two additional node types to the resulting CST:

- Skipped tokens – tokens that were skipped by parser. These tokens are still preserved in a tree, providing more context for code completion.
- Recovered tokens – these tokens are not present in the query, yet they were expected and were added in the tree by a parser, preserving a correct tree structure.

The following is an example of a heuristic, used by TQLS: whenever a parser encounters a closing parenthesis after the left part of the equation (an identifier and an equation sign), it does not skip it, but instead adds a recovered operand in a tree and continues parsing as normal. This way, less input is skipped and the resulting tree is more useful for code completions generation.

Figure 2 shows an example of a CST for a query $$$ (test=). Both types of additional nodes were added by the error recovery procedure. Unquoted dollar signs are not allowed, that is why they were skipped. After an equals sign, a parser encountered a closing parenthesis and it inserted a recovered operand in a tree.



```
$$$ (test=)
```

Fig. 2. CST with skipped and recovered tokens

In order to report errors to the user, a recursive procedure searches tree for both types of nodes and combines them in a single list. For the above example, two errors will be reported:

1. Start: 0 | End: 3 | Found: $$$ | Expected: expression

2. Start: 10 | End: 11 | Found:) | Expected: operand

4.7 Code completion

Code completion procedure takes a query and a cursor position as an input and returns a list of completions as an output. Each completion is a text range and a string, that should replace that range if the completion is accepted. Completion procedure is split into two stages:

- Determine a completion context. Completion context consists of a range inside a query and the current selector. The current selector is the innermost selector, that surrounds the cursor.
- Make a list of completions based on the provided context.

In order to find a completion context, the first step is to find a token under cursor in a tree. To do that, a recursive procedure iterates through root's children from left to right and makes a recursive call if cursor is contained within a child node's range. This way, if cursor is placed between two tokens, the left one will be chosen by the procedure. This is due to the fact, that completion is expected to finish token before the cursor, not after. Token and it's ancestors are then used to determine a completion context. An example, where the found token is not an entire completion context, is when dealing with multi-word properties, which are written in quotes. Consisting of several tokens, they should be considered a single completion context.

After completion context is found, it's text is compared with known keywords. A list of available keywords is determined by the current selector.

4.8 Programming language

TQLS uses Rust as an implementation language. Rust was chosen due to the following reasons:

- Chumsky combinator library uses recursive descent algorithm and allows custom error recovery strategies.
- Robust support for WebAssembly, allowing it to run in a browser.
- Algebraic Data Types (enums) which are useful for modelling syntax trees.
- Robust GraphQL support, which allows implementing a backend API.

Rust-analyzer [11] was used as a source of inspiration for designing TQLS.

4.9 Type-safe CST API

Parser produces a concrete syntax tree, which reflects grammar rules. Unlike AST, CST is what we call a homogeneous tree: each node uses the same structure in code. Code below demonstrates structures used to represent this tree in a program:

```
struct Node {
  kind: NodeKind,
  meta: NodeMetadata,
  children: Vec<Child>,
}
enum Child {
  Node(Node),
  Token(Token),
}
struct Token {
  kind: TokenKind,
  meta: NodeMetadata,
  text: String,
}
```

Using the same node type for all tree nodes simplifies writing general tree traversal algorithms. However, when one needs to work with a specific node type, this API can be error-prone. On top of this API, there is a type-safe API which is generated from the tree description. This description is written in Ungrammar [12] format. As an example, let’s take a look at the following nodes:

```
AndExpr = Factor (' ' Factor)*
Factor = StructOperators
  | FlatFactor
  | ElementParens
```

Custom source generator will generate the following structures:

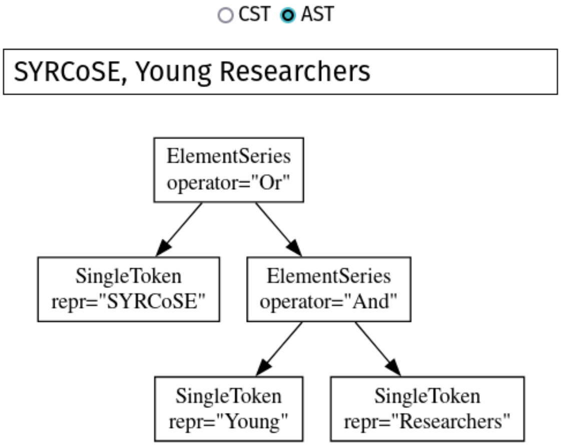
```
struct AndExpr<'a> {
  node: &'a Node,
}
impl<'a> AndExpr<'a> {
  fn get_factor_list(&self) -> Vec<Factor<'a>> {
  tree::get_children_with_type(self.node)
}
enum Factor<'a> {
        StructOperators(StructOperators<'a>),
        FlatFactor(FlatFactor<'a>),
        ElementParens(ElementParens<'a>),
    }
```

There are currently over 70 node types in an Ungrammar description and more than 3500 lines of code generated from it. Writing this code manually would be error-prone and since the tree structure can change over time, updating this file manually will be a maintenance burden.

4.10 Interactive playground

In order to debug and test TQLS, an interactive playground application was developed. It runs inside a browser using WebAssembly. This way, it is possible not only to test our application and visually validate the resulting tree, but also to ensure that the exposed API is sufficient and is easy to use. In order to render tree in a browser, TQLS serializes it as a DOT graph and playground renders it using a graphviz library.

Figure 3 shows an AST for a query SYRCoSE, Young Researchers rendered inside a playground.



○ CST ● AST

SYRCoSE, Young Researchers

ElementSeries operator="Or"
SingleToken repr="SYRCoSE"
ElementSeries operator="And"
SingleToken repr="Young"
SingleToken repr="Researchers"

Fig. 3. Interactive playground

5. Conclusion

This work describes our experience developing and integrating a DSL with the rest of our services. Our focus was on making it maintainable and portable, here is a summary of our solutions to this problem:

- Treating a DSL as a separate application with it’s own architecture and tech stack
- Using CST for editor support and AST for executing queries
- Implementing Language Server Protocol to integrate with any tools supporting it
- Using WebAssembly to run TQLS in a browser
- Using GraphQL to run TQLS on a server
- Using Rust programming language to implement our design
- Generating type-safe CST API from Ungrammar description
- Using interactive playground to test and debug TQLS

References

[1]. M. Fowler and R. Parsons, Domain-Specific Languages. Boston, MA: Addison-Wesley Educational, 2009.

[2]. “TALISMAN – tracking and learning insights from social media analysis,” Ispras.ru. [Online]. Available: https://talisman.ispras.ru/. [Accessed: 13-May-2024].

[3]. M. Karpov, “megaparsec: Industrial-strength monadic parser combinator library.” [Online]. Available: https://github.com/mrkkrp/megaparsec. [Accessed: 31-Mar-2024].

[4]. J. Barretto, “chumsky: Write expressive, high-performance parsers with ease.” [Online]. Available: https://github.com/zesterer/chumsky. [Accessed: 31-Mar-2024].

[5]. “Language Server Protocol Specification,” Github.io. [Online]. Available: https://microsoft.github.io/language-serverprotocol/specifications/lsp/3.17/specification/. [Accessed: 31-Mar-2024].

[6]. C. Donnelly and R. Stallman, Bison. Samurai Media Limited, 2015.

[7]. A. Kladov, “Challenging LR parsing,” rustanalyzer, 16-Sep-2020. [Online]. Available: https://rustanalyzer.github.io/blog/2020/09/16/challeging-LR-parsing.html. [Accessed: 12-May-2024].

[8]. T. Parr, Definitive ANTLR 4 Reference, 2nd ed. Raleigh, NC: Pragmatic Programmers, 2013

[9]. S. Efftinge and M. Spoenemann, “Xtext - Language Engineering Made Easy!,” Eclipse.dev. [Online]. Available: https://eclipse.dev/Xtext/. [Accessed: 12-May-2024].

[10]. “GraphQL,” Graphql.org, 26-Oct-2021. [Online]. Available: https://spec.graphql.org/October2021/. [Accessed: 02-Jul-2024].

[11]. “rust-analyzer: A Rust compiler front-end for IDEs.” [Online]. Available: https://github.com/rust-lang/rust-analyzer. [Accessed: 31-Mar-2024].

[12]. “Introducing ungrammar,” Github.io, 24-Oct-2020. [Online]. Available: https://rust-analyzer.github.io//blog/2020/10/24/introducing-ungrammar.html. [Accessed: 02-Jul-2024].

Информация об авторах / Information about authors

Артём Дмитриевич БЕЛОУСОВ – старший лаборант, студент Высшей школы экономики. Сфера научных интересов: конструирование компиляторов и облачные вычисления.

Artyom Dmitrievich BELOUSOV – senior laboratory assistant, student of Higher School of Economics. Research interests: compiler design and cloud computing.