BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

TOOLS GENERATOR FOR DOMAIN-SPECIFIC LANGUAGES

GENERÁTOR NÁSTROJŮ PRO DOMÉNOVĚ SPECIFICKÉ JAZYKY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **DANIEL KOSÍK**
AUTOR PRÁCE

SUPERVISOR **Ing. ZBYNĚK KŘIVKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

156284

Institut:	Department of Information Systems (DIFS)
Student:	**Kosík Daniel**
Programme:	Information Technology
Title:	**Tools Generator for Domain-Specific Languages**
Category:	Compiler Construction
Academic year:	2023/24

Assignment:

1. Familiarize yourself with ANTLR, the topic of domain-specific languages, and development environments with a focus on domain-specific languages.
2. Analyze the existing tools for generating tools for programming languages.
3. Based on the consultations with the supervisor, design a tool capable of generating the tools necessary to use the language in the Visual Studio Code environment (or equivalent). Support various domain-specific languages.
4. Implement the tool and test it on at least two domain-specific languages (as agreed with the supervisor).
5. Evaluate the achieved results and compare them with existing tools.

Literature:

- Parr, T.: *The Definitive ANTLR 4 Reference*. 2nd Edition, Pragmatic Bookshelf, 2013.
- Parr, T: *Language Implementation Patterns - Techniques for Implementing Domain-Specific Languages*. Pragmatic Bookshelf, 2009.
- Fowler, M.: *Domain-Specific Languages*. Addison-Wesley Professional, 2010.
- Voelter, M.: *DSL engineering – designing, implementing and using domain-specific languages*. CreateSpace Independent Publishing Platform, 2013. Available at: dslbook.org.

Requirements for the semestral defence:

- First three items.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Křivka Zbyněk, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	30.10.2023

Abstract

DSL Tools Generator is a tool for simplifying the development of domain-specific languages by generating parts of their implementation (e.g. abstract syntax tree) in C# and tools for using the language in a code editor. Based on an ANTLR4 grammar and a DSL configuration file, it generates a server implementation for the Language Server Protocol and a language support extension for Visual Studio Code that provides syntax highlighting, error reporting, and basic code completion functionality. The developed tool can significantly reduce the time and effort required for building a DSL with editor support.

Abstrakt

DSL Tools Generator je nástroj pro zjednodušení vývoje doménově specifických jazyků generováním částí jejich implementace (např. abstraktního syntaktického stromu) v jazyce C# a nástrojů pro použití daného jazyka v editorech kódu. Podle zadané gramatiky a konfiguračního souboru vygeneruje implementaci serveru pro Language Server Protocol a rozšíření pro Visual Studio Code, které poskytuje zvýrazňování syntaxe a syntaktických chyb a základní doplňování kódu. Výsledkem práce je nástroj, který dokáže podstatně zkrátit čas a snížit úsilí potřebné k vytvoření doménově specifického jazyka s podporou v editorech kódu.

Keywords

domain-specific language, code generation, Visual Studio Code, C#, Language Server Protocol, code editor, syntax highlighting, regular expression, TextMate grammar

Klíčová slova

doménově specifický jazyk, generování kódu, Visual Studio Code, C#, Language Server Protocol, editor kódu, zvýrazňování syntaxe, regulární výraz, TextMate gramatika

Reference

KOSÍK, Daniel. *Tools Generator for Domain-Specific Languages*. Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Zbyněk Křivka, Ph.D.

Rozšířený abstrakt

Doménově specifické jazyky (DSL) jsou jazyky navržené s úzkým zaměřením na určitou oblast, ve které pak nabízí čitelnou a stručnou syntaxi, více možností pro doménově specifickou validaci a analýzu a další výhody. Běžnými příklady DSL jsou jazyky pro rozvržení a stylování dokumentů (HTML, CSS), dotazování (SQL, GraphQL) a serializaci dat (Protobuf), programování grafických shaderů (GLSL, HLSL) a tvorbu diagramů (DOT, Mermaid).

Úspěch takových jazyků je do velké míry ovlivněn kvalitou jeho podpory v editorech kódu. To zahrnuje například zvýrazňování syntaxe (a sémantiky), hlášení chyb pomocí podtrhávání, doplňování (našeptávání) kódu a zobrazení dokumentace při najetí myši (*hover*). Language Server Protocol (LSP) umožňuje využití jednoho programu s touto funkcionalitou v mnoha editorech kódu.

Zatímco pro ostatní programovací jazyky existuje několik nástrojů pro tvorbu DSL (např. Xtext, JetBrains MPS, Spoofax, Langium, textX), pro C# (.NET) jich v současné době existuje jen velmi málo.

Cílem této práce je vytvořit nástroj pro zjednodušení vývoje doménově specifických jazyků generováním částí jejich implementace (např. abstraktního syntaktického stromu) v jazyce C# a nástrojů pro použití daného jazyka v editorech kódu.

Jako generátor syntakticého analyzátoru byl zvolen nástroj ANTLR, který je dostatečně flexibilní a navíc zpřístupňuje informace o gramatice i za běhu, což lze využít pro funkcionalitu doplňování kódu. Také bylo nutné vyřešit problém stínování (*shadowing*) způsobeného rozdíly v chování ANTLR lexerů a TextMate gramatik používaných pro zvýrazňování syntaxe. Nakonec byl vyvinutý nástroj otestován vytvořením nástrojů pro dva doménově specifické jazyky: Avro IDL a CSS.

Výsledkem práce je nástroj, který podle zadané gramatiky a konfiguračního souboru vygeneruje implementaci serveru pro Language Server Protocol a rozšíření pro editor Visual Studio Code, které poskytuje zvýrazňování syntaxe a syntaktických chyb a základní doplňování kódu. Vytvořený nástroj dokáže podstatně zkrátit čas a snížit úsilí potřebné k vytvoření doménově specifického jazyka s podporou v editorech kódu.

Tools Generator for Domain-Specific Languages

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Zbyněk Křivka, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Daniel Kosík
May 9, 2024

Acknowledgements

I would like to thank my supervisor Ing. Zbyněk Křivka, Ph.D. for his advice and guidance.

Contents

List of Acronyms

ALL(*)	Adaptive LL(*). 16
ANTLR	ANother Tool for Language Recognition. 3, 5, 14, 24, 26, 28, 37
AST	Abstract Syntax Tree. 6, 9, 20, 22, 23, 25, 33, 37
ATN	Augmented Transition Network. 16, 17, 19, 21, 24, 37
BSP	Build Server Protocol. 13, 38
DAP	Debug Adapter Protocol. 7, 13, 38
DFA	Deterministic Finite Automaton. 16
DSL	Domain-Specific Language. 4, 9, 18, 21, 37, 38
EBNF	Extended Backus-Naur Form. 14
GLSP	Graphical Language Server Protocol. 13
GPL	General-purpose Programming Language. 5
IDE	Integrated Development Environment. 6, 17
IDL	Interface Description Language. 31, 37
IPC	Inter-Process Communication. 12
JSON	JavaScript Object Notation. 2, 38
JSON-RPC	JSON Remote Procedure Call. 11, 13
LSIF	Language Server Index Format. 13
LSP	Language Server Protocol. 7, 10, 11, 13, 18, 24, 32, 37, 38
MPS	(JetBrains) Meta Programming System. 9, 18
NPM	(originally) Node Package Manager. 25
RTN	Recursive Transition Network. 16
SDK	Software Development Kit. 11
TCP	Transmission Control Protocol. 24, 25
VS Code	Visual Studio Code. 3, 19, 22, 24–26, 31–35, 37, 38

Chapter 1

Introduction

Domain-specific languages (DSLs) are languages that have been designed and optimized for a specific narrow purpose, meaning the set of programs or models that they can describe is constrained. However, their limited scope yields many benefits, such as readable and concise syntax, more opportunities for domain-specific validation and analysis, and more. Common examples of DSLs are languages for document layout and styling (HTML, CSS), data querying (SQL, GraphQL) and serialization (Protobuf), graphics shader programming (GLSL, HLSL), but also diagramming (DOT, Mermaid), tax or wage calculations, and healthcare.

Language support in a code editor is crucial to enable a fast feedback loop, increase productivity, learnability, and *discoverability*. Without it, users lack confirmation and feedback about the syntactic and semantic validity of their code and have to resort to looking up documentation about the available features in external documentation.

Multiple DSL creation toolkits exist for Java and JavaScript; however, there are currently very few tools for developing DSLs in the C# (.NET) ecosystem, aside from parser generators and parsing libraries like ANTLR, Sprache and Pidgin.

The goal of this thesis is to design and implement a tool that can streamline the development of domain-specific languages by generating tools for a given language, including language support integration for existing code editors. Specifically, this involves analyzing a description of the given language in the form of a formal grammar and producing the code for a language server and its associated VS Code extension (communicating over the Language Server Protocol). In most cases, the user will have to provide code that obtains the results of semantic analysis for a given document and specify how to utilize its results in editor features such as diagnostics (errors, warnings), semantic highlighting, and code completion.

The developed tool should be lightweight, not locked into specific editor or IDE (Eclipse, IntelliJ, etc.), quick to install (as a command-line program) and easy to integrate into existing codebases.

Chapter 2

Domain-Specific Languages

This chapter describes the basics of domain-specific languages, when they are useful, and explores the approaches used to implement them.

2.1 What is a DSL?

A domain-specific language (DSL) is a formal language designed and optimized for a particular domain, based on abstractions and features that make it easy to express ideas in that domain. Examples of such domains include:

- web design and development – HTML, CSS, SCSS, Wasp
- data querying and manipulation – SQL, GraphQL, Cypher, Gremlin, SPARQL, KQL, CQL, OQL, TypeQL, BIMQL, PathQuery, PromQL, LogQL, HQL
- data storage and modeling – LookML, ERML, DBML
- data serialization and configuration – XML, JSON, CSON, YAML, TOML, Jsonnet, GCL
- shell scripting – Bash, Fish, ZSH, PowerShell
- build configuration – Makefile, CMake, Gradle, Cocoa Podfile
- hardware description – VHDL, Verilog, SystemVerilog, Lola, AHDL
- networking and interface description – P4, YANG, Protobuf schema, Avro IDL, Thrift
- GUI description languages – QML, Quid, Slint
- graphics programming – GLSL, HLSL, MSL, WGSL, Slang
- game development – GSC (scripting language used in Call of Duty games), Papyrus (scripting language used in Bethesda games), PuzzleScript
- diagramming – DOT (GraphViz), GrGen, PlantUML, Mermaid, D2, blockdiag
- typesetting and markup – TEX, BibTeX, Typst, Markdown, Texy
- model checking, formal verification, simulation – PRISM, GAL, Promela, P, Modelica
- tax or wage calculations – RegelSpraak [2], a payroll DSL at DATEV [12]
- healthcare – ALPH

Compared to general-purpose programming languages (GPLs), DSLs are more limited in scope but offer greater productivity and conciseness within their domain. This makes them easier to use for non-programmers, since they can work in an environment that uses concepts from their area of expertise, including error messages, analysis results, etc. [11]

Separating the knowledge of the subject-matter expert from the technical aspects of the system has many benefits – one of them being the ability to perform arbitrary static analysis and validation on the code. Implementing domain-specific analysis on GPL code is usually not viable[1], except in cases where a compiler-as-a-service (like *Roslyn* for C#, *Clang* for C++) or a sufficiently extensible compiler or linter (e.g. *clippy* for Rust or *TSLint* for TypeScript) is available.

Domain-specific languages can be divided into **external** (a language with its own syntax, requiring a separate parser) and **internal** (embedded into a host language and implemented with its abstractions [11]; distinguished from a normal library API by having a language-like flow; also called *fluent interfaces*) [5]. This thesis focuses exclusively on external DSLs.

DSLs can be used to specify **behavior** (via code generation or interpretation), **requirements/tests**, or to produce an output artifact like **diagrams** or **documents**. They might also describe a model of some part of a system (e.g. to verify some of its properties).

Although many DSLs are targeted at programmers, a significant portion of them are intended to be used by non-programmers, often *subject-matter experts* who have a lot of experience in their domain.

Generic configuration languages like JSON, XML, YAML, when paired with schemas, can, in some cases, be used as a lightweight alternative to DSLs – editors like VS Code are often able to provide some amount of in-editor validation, code completion, hover documentation, etc., based on the provided schema.

At least some amount of editor support is essential for most DSLs to be able to succeed and provides great value to their users. The most common type of editor support includes real-time error checking, code completion, signature (parameter) help, and hover tooltips to explore and explain syntax and semantics. Some types of editor support are not common for programming languages, but are very useful for DSLs, like live preview (of the resulting diagram or document) or an alternative view of the model (state machine, class hierarchy, railroad diagram of a grammar, etc.).

2.2 Creating a DSL

The traditional approach to implementing a DSL or programming language is to create a pipeline consisting of four basic stages:

1. lexical and syntax analysis (**parsing**) that transforms the input document (sequence of characters) into a tree representation (**parse tree**). The parser can be either hand-written, possibly using a parser combinator library, or generated by a parser generator tool such as ANTLR, lex+yacc, or flex+bison.

[1] it would require not just a reliable parser for the programming language, but also an accurate semantic analyzer (e.g. name binding, type-checking)

```
abstract record AstNode;
abstract record Statement : AstNode;
record AssignStatement(string VarName, Expression Value) : Statement;
abstract record Expression : AstNode;
record StringLiteralExpression(string Value) : Expression;
```

Figure 2.1: Example of a definition of a simple AST class hierarchy in C# (using the Records feature from C# 9.0)

2. transforming the parse tree into some kind of intermediate representation (IR) better suited for further processing; often an **abstract syntax tree** (AST) or **semantic model**
3. performing analysis of the **semantics**: name binding (resolving references to defined symbols), validation (checking that the code is valid according to the language's specification), etc.
4. evaluating the code – either by **interpreting** it or by **generating** some kind of output (e.g. code, graphics, sound)

The pipeline is often thought of in terms of two parts: a *frontend* (analysis) and a *backend* (synthesis). This thesis focuses solely on the *frontend*.

Abstract Syntax Tree

The AST is a tree-like[2] data structure commonly represented (in object-oriented languages) using a class hierarchy where each class represents one type of node, and the attributes and child nodes are stored in typed properties. In functional languages, the AST is usually represented using *algebraic data types*, specifically *discriminated unions*.

Operations on an AST:

- traverse the AST – visit each child or descendant node
- attach data from semantic analysis to specific nodes – resolved/bound references, expression types, diagnostics (errors, warnings, suggestions), etc.
- find a node at a given position in the document (needed for editor integration)
- edit and "unparse" (serialize back to source code) without losing formatting and comments

2.3 Language Support

To get language support in an editor or IDE, the author would have to either build their own development environment tailored to their language, or build an extension (plugin)

[2] when taking into account cross-references to other nodes, it becomes a graph, but even then, the primary relation is still the containment relation ("contains" / "is child of").

for an existing code editor. This involves learning the language the editor is written in, e.g. JavaScript or TypeScript in the case of VS Code, and reimplementing some of the logic from the compiler – parsing, analysis, and validation.

Thanks to the Language Server Protocol (LSP), editor extensions themselves do not have to perform any parsing or analysis of the documents being edited. Instead, they can send the text to a **language server** (over a socket or standard input/output), which is responsible for parsing and analyzing the document and provides the editor extension with information such as diagnostics (errors and warnings), code completion suggestions (*IntelliSense*), signature help, symbol documentation, and more.

Integration of the editor features with LSP can be largely delegated to libraries that handle most of the LSP communication. However, if the language server offers custom request or notification types, the extension author must implement them manually for the corresponding editor.

One of the major benefits of LSP is that the server does not need to be implemented in the same language as the editor extension (the client); therefore, it can reuse logic from (or even be part of) the main language implementation.

Not all features of LSP can be used for all DSLs, as some of them rely on specific language concepts or paradigms. The suitability of LSP (and the Debug Adapter Protocol) for domain-specific languages was investigated in [4].

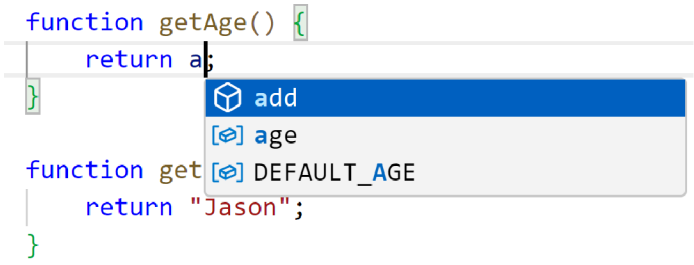
There are multiple design concerns for the concrete syntax of a language that influence what kind of editor services might be beneficial to the user [11]:

Learnability – to make it possible for the user to learn about new concepts as they need them (or as they encounter them in existing code), the editor can provide:

- code completion (to answer the question “what can I type here?”)
- real-time error checking (“can I do this?”)
- snippets (“what is the syntax for . . . ?”)
- hover/tooltips (“what is this?”)
- high-quality error messages (“why is this wrong?”)
- user-executable code fixes (“how can I correct this code?”)

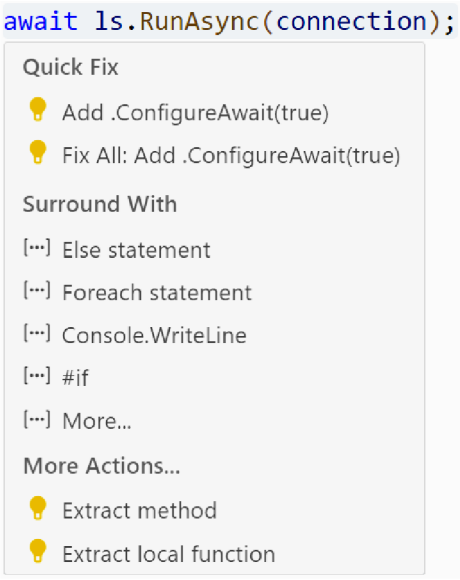
A major factor influencing the learnability (and teachability) of a language is having access to an online editor environment (often called a **playground**) that allows users to experiment, try out new features, and share snippets of code with others, all without having to install any software on their computer.

One could also imagine more involved forms of learning assistance (**context-sensitive help**), such as a side panel with an overview of the syntax (perhaps using syntax diagrams, also known as *railroad diagrams*) and/or semantics of the language that responds to the user’s current position or selection in the document. Alternatively, a **visualization** (e.g. a class diagram, finite-state machine) or **preview** (e.g. rendering result, an interactive simulation) can also help users understand whether their expectations regarding the structure or meaning of the code are correct.

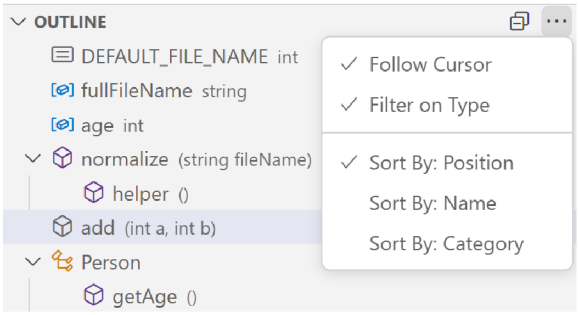


(a) Code Completion

(b) Inline Diagnostics (displayed by the *Error Lens* extension)



(c) Code Actions



(d) Document Outline

Figure 2.2: Examples of editor services provided by VS Code and LSP

Writability – code completion and code snippets help accelerate common tasks when writing code and decrease the need for looking up external documentation. Code Actions can automate refactoring operations and fix errors or detected "code smells".

Readability – documentation or other information can be displayed on hover (using tooltips) or inline (using *inlay hints* – virtual text or other elements that are only a visual decoration inside the editor area). Code folding can temporarily hide parts of the source code that are not currently relevant. Code navigation can be facilitated with commands like *Go To Definition* and *Find All References*.

2.3.1 Syntax Highlighting

For the purposes of this thesis, *syntax highlighting* means:

1. pattern matching on characters in a document, line by line, based on a list of **rules** defined using regular expressions – transforming each line into a sequence of **tokens**
2. assigning each token the corresponding color or font style (bold, italic, strike-through)

Notably, the TextMate editor for Mac used this simple (but relatively powerful) regex-based highlighting system – its TextMate grammar format (`.tmLanguage`) is supported by other editors (e.g. VS Code, Sublime Text).

Tokens in the TextMate system can have multiple *scopes*, which are symbols similar to class names in CSS – themes contain selectors that target scope names and assign colors or font styles to them.

A TextMate grammar consists of a list of rules that apply from top to bottom. Each rule attempts to match characters using its regular expression from the current position. If a rule successfully matches any characters, a token is produced with the rule's configured scope name and the matching starts again from the new position.

The regular expressions of rules are constrained to a single line. To produce multi-line tokens, a different type of rule with separate `begin` and `end` patterns is needed.

There are other grammar formats for syntax highlighting, e.g. `.sublime-syntax` (also regex-based, but more powerful, with an explicit stack of contexts), Monarch, etc.

Syntax highlighting is also used in places other than code editors, usually to highlight code snippets in documents or websites. Popular libraries and tools for this purpose include *highlight.js*, *prism.js*, *Shiki*, and *Pygments* (used by the *minted* package for LaTeX.

2.4 DSL Development Frameworks and Language Workbenches

There are multiple frameworks and DSL development environments called *language workbenches* that can be used to create a DSL relatively quickly, at least after having learned how to use the tool, which often takes a nontrivial amount of time.

The Java ecosystem contains many projects that support the creation of DSLs with editor support:

- **Xtext** (based on the *Eclipse Modeling Framework* and *Ecore*)
- **Spoofax** (also Eclipse-based)
- **JetBrains MPS** – a complete language workbench with a *projectional*[3] editor

textX is a language and a tool for creating DSLs in Python [3]. From a grammar of the language, it constructs a parser and a meta-model as a set of Python classes. It has support for automatic resolving of references, error-reporting, debugging, and even visualization of models using GraphViz. It has a playground page available online[4] where users can try out the textX language.

[3] projectional editing means the user directly manipulates the model/AST, which is then *projected* to text or graphics

[4] https://textx.github.io/textx-playground/

The **textX-LS** project that provides basic editor support for languages built with textX was inactive for several years, but a new maintainer (Milan Šović, `@AirmiX`) was announced in April 2024.

A similar project called **Langium** was started in 2021 by TypeFox and became part of the Eclipse Foundation in 2023. It is a language engineering tool for TypeScript (Node.js) with first-class support for the Language Server Protocol [10]. It uses the *Chevrotain* parser library.

There are also programming languages with support for *language-oriented programming*: Rascal, Racket, and Raku. These have dedicated support for defining languages and their behavior.

Chapter 3

Language Server Protocol

The language server protocol (LSP) standardizes the communication between code editors and language servers, which provide language-specific analysis and provide data for editor services such as auto-completion, diagnostics, finding references, etc. This allows a single language server to be reused in multiple editors with minimal effort[1]. [6]

As of January 2024, there are more than 240 language server implementations listed on the official LSP website [6].

A **language server** is a program launched (and terminated) by the client (code editor) and communicating only with that one client, usually over standard input/output or network sockets (`localhost` only), although this is not defined by the LSP specification.

The server and client exchange JSON-RPC messages (*requests*, *responses*, and *notifications*) described by the LSP specification [6]. There are many libraries (SDKs) that implement the basic protocol handling logic available for most mainstream programming environments.

A **language client** is a part of a code editor or other development tool (either built-in or in the form of an *extension* or *plugin*) that manages the lifetime of its corresponding language server and handles the communication over LSP, including calling the appropriate APIs for manipulating elements of the user interface of the editor, for instance, it might:

- show suggestions in a drop-down menu
- color or decorate a span of text or an entire line
- apply edits to a document
- display contextual help (e.g. signature help when the cursor is inside a function call)

3.1 Base Protocol

LSP is based on JSON-RPC version 2.0, which is a simple Remote Procedure Call protocol where the server and the client send each other messages serialized as JSON documents.

[1] unless the language server uses an extended version of the protocol with custom message types that are not handled by LSP libraries

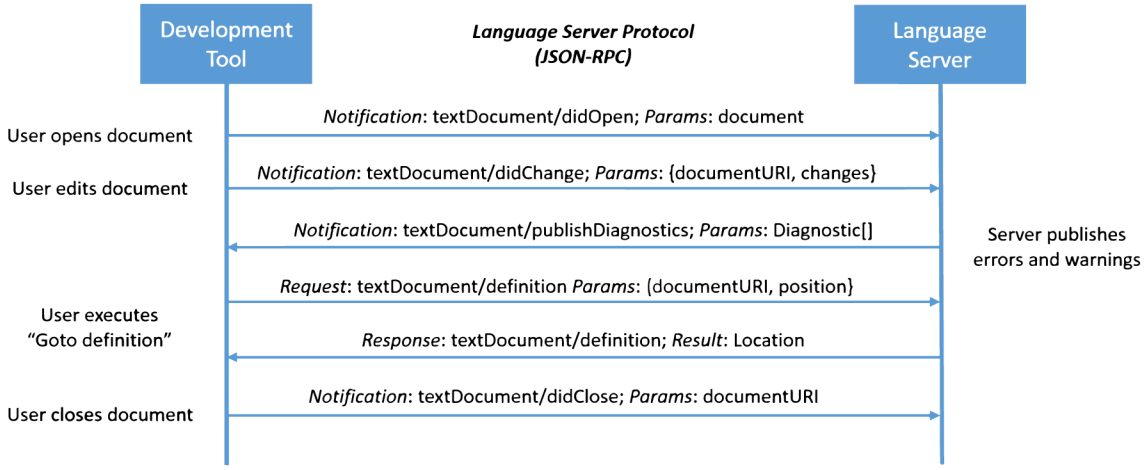


Figure 3.1: LSP communication example from [6]

The underlying transport mechanism can be anything, as long as both sides agree on it. The specification requires LSP implementations to support communication over standard input/output. Other options include network sockets and Inter-Process Communication (IPC). [6]

3.2 Messages

The LSP contains the following categories of messages [6]:

- **Lifecycle Messages** – initialization, registration of capabilities, shutdown, tracing
- **Document Synchronization** – open/change/save/close notifications from the client
- **Workspace Features** – (file) create/rename/delete notifications from the client, workspace (multi-document) edits (like automated refactorings), fetching configuration, project-wide symbol search
- **Language Features** – Completions, Diagnostics, Folding, Formatting, Inlay Hints, Code Lens, Semantic Highlighting, Code Actions, Finding/Highlighting/Resolving/Renaming Symbol References and Definitions, Document Links

Since not all types of messages and LSP features are mandatory, the client and the server send their **capabilities** during initialization to communicate which features they support [6]. Alternatively, if the client supports *dynamic capability registration*, the server can choose to delay the announcement of a capability if it takes a significant amount of time to load (and would block the server from being able to respond to requests for services that would otherwise be available immediately).

3.3 Related Protocols and Formats

Multiple other protocols were created that follow the same principles as the Language Server Protocol and even use the same underlying *base protocol* (the subset of LSP that is not specific to language servers and can be reused in other protocols).

One of these is the Debug Adapter Protocol (DAP) that allows a single debugger to be reused from any editor that supports this protocol. However, for historical reasons, DAP does not use JSON-RPC like LSP does – it was based on the now-obsolete *V8 debugging protocol*.

Other protocols with an LSP-compatible base protocol include Build Server Protocol (BSP), Graphical Language Server Protocol (GLSP) and MSTest Runner Protocol. In theory, a single server could even support multiple of these protocols simultaneously.

LSIF is "a standard format for persisted code analyzer output"[2] – a data format for storing the data computed by LSP servers (or separate programs called *indexers*) for later retrieval without access to the LSP server, e.g. for resolving references when viewing repository code on the web. The web server probably does not know how to correctly configure and launch the LSP server. However, the development environment or Continuous Integration do, so they can persist ("dump") all the information necessary for answering LSP requests on the codebase into a file that is then read by the web server.

Sourcegraph has developed an alternative indexing format called *SCIP*.

[2] https://lsif.dev/

Chapter 4

ANTLR

This chapter focuses on the ANTLR parser generator: why it is useful, how it is used, and its strengths and weaknesses.

ANTLR (*ANother Tool for Language Recognition*) is a popular parser generator created by Terence Parr. It is a Java program that accepts a grammar file (in an EBNF-like format) and generates code for a lexer and parser for the described language (plus a listener and visitor class for custom processing of the produced parse trees) in one of several supported target programming languages (Java, C#, C++, JavaScript/TypeScript, Python, Dart, Go, PHP, Swift) [8].

Version 4, released in 2013, brought some interesting improvements that are very helpful when building editor support (mainly for code completion) and error handling. [7]

Users are not required to have extensive knowledge of traditional parsing theory (LL(1), etc.) to write a grammar, since ANTLR4 supports unbounded look-ahead (thanks to the adaptive prediction mechanism described in section 4.2 on page 16) and automatic elimination of direct left recursion (parse trees are transformed back to match with the grammar). However, indirect (two or more mutually left-recursive rules) or hidden left-recursion is not supported (e.g. a rule like `expr : 'not'? expr 'and' expr` would have to be split into two separate rules, one with `'not'` and one without).

4.1 Usage

After invoking the ANTLR tool with the grammar file name as an argument, the following files are generated: Lexer, Parser, Listener, Visitor (if the `-visitor` option is given), a `.tokens` file listing the defined token types and their numeric values, and an `.interp` with data necessary for simulating the parser using an interpreter.

To be able to instantiate the generated parser class, it is necessary to first construct a `CharStream`, use it to create a `Lexer`, use it as the token source for a `TokenStream`, and finally create a `Parser` with the `TokenStream` as input. The generated parser class includes parsing methods for each parser rule defined in the grammar. These parse the previously provided token stream into a parse tree and report any errors to all configured error listeners.

```
grammar Ex; // generates class ExParser
// action defines ExParser member: enumIsKeyword
@members { bool enumIsKeyword = true; }
stat : expr '=' expr ';' #assignmentStat
     | expr ';'          #exprStat
     ;
expr : expr '*' expr     #multExpr
     | expr '+' expr     #addExpr
     | expr '(' expr ')' #funcCallExpr
     | id                #idExpr
     ;
id : ID | {!enumIsKeyword}? 'enum' ;
ID : [A-Za-z]+ ;  // match letter-only identifiers
WS : [ \t\r\n]+ -> channel(HIDDEN) ;  // ignore whitespace
```

```
void stat() {
  switch (adaptivePredict("stat", callStack)) {
    case 1:
      expr(); match('='); expr(); match(';'); break;
    case 2:
      expr(); match(';'); break;
  }
}
```

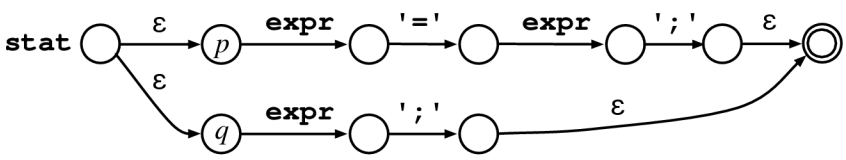


Figure 4.1: An example ANTLR4 grammar with the corresponding generated parser code (simplified) and ATN for the `stat` rule. Adapted from [9]

ANTLR is quite flexible and extensible. The generated C# classes are marked as `partial` so that the user can add members to them. Also, the templates used for code generation (using the *StringTemplate* library) are overridable – if a template for a specific codegen target is found in CLASSPATH, it replaces the default one.

4.2 Adaptive LL(*) and Augmented Transition Networks

Parsers generated by ANTLR4 use a unique parsing strategy called *Adaptive LL(*)* (ALL(*)) developed by Terence Parr, Kathleen Fisher, and Sam Harwell [9]. The major innovation compared to LL(*) (used in ANTLR3) is that the grammar analysis is performed at parse-time (*just-in-time* instead of *ahead-of-time*) and that it no longer uses any backtracking.

ALL(*) relies on a runtime representation of the grammar called the Augmented Transition Network (ATN) [9]. It is a special type of a Recursive Transition Network (RTN), which is itself a type of a *transition network* where the transitions between states might "call" another transition network. RTNs are sometimes used for parsing natural languages.

The generated parser code looks mostly like an ordinary recursive-descent parser. The biggest difference is how it predicts which decision to take when there are multiple alternative paths forward. The `adaptivePredict` method is used to select the correct alternative. It works by running an ATN interpreter from the current state, with a DFA-based cache to increase performance.

ATNs in ANTLR can have various types of states and transitions: epsilon transitions, rule transitions, atom transitions, set transitions, etc.

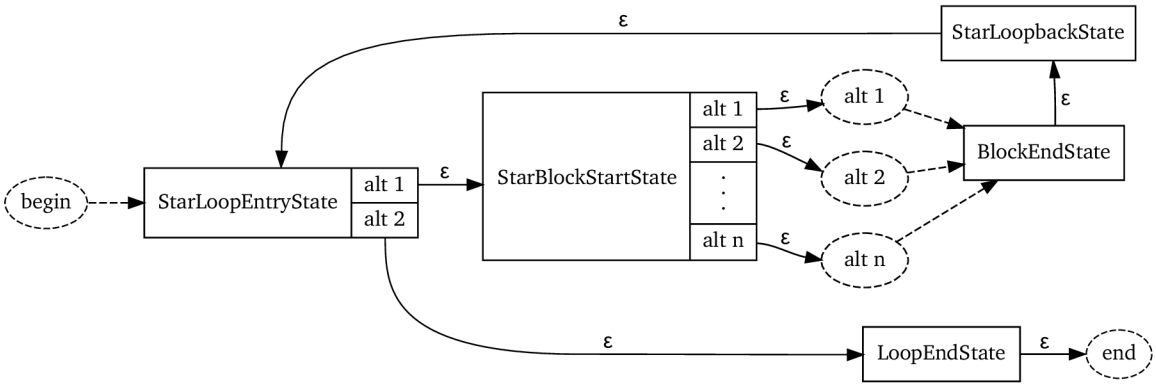


Figure 4.2: ATN of a `(1 | 2 | ... | n)*` block. Adapted from ANTLR Java API documentation[1]

The ATN is serialized as an array of integers as part of the generated code and also in a separate `*.interp` file. Deserialization occurs at runtime in the `ATNDeserializer` class.

The runtime library of ANTLR also provides the `ParserInterpreter` and `ParserATNSimulator` classes, which use the ATN to simulate an ANTLR4 parser without having to build the actual

[1] https://www.antlr.org/api/Java/org/antlr/v4/runtime/atn/ATNState.html

source code into an executable binary[2] – this is used for visualizing and debugging ANTLR grammars in editor/IDE tools and also in the *ANTLR Lab*[3] web application.

Since the ATN is just a slightly different form of the grammar, it can be used to drive autocomplete functionality. After simulating transitions until the current cursor position, we can collect all token or rule transitions available from the current state (after skipping any epsilon transitions) and suggest them as code-completion items. This functionality is available in the *antlr4-c3* library by Mike Lischke[4].

A disadvantage of this approach is that the ATN does not contain all the information from the grammar. Specifically, labels are not stored in the ATN, so it is often necessary to wrap tokens in extra rules to get more information from the code completion engine. For example, instead of suggesting all identifiers, we might want to specifically suggest package names, function names, etc. That requires adding a rule like `packageName : ID ;`.

[2] embedded actions (pieces of code in the target language) are not available without building the parser
[3] http://lab.antlr.org
[4] https://github.com/mike-lischke/antlr4-c3/

Chapter 5

Design and Implementation of the DSL Tools Generator

This chapter delves into the various aspects of the design and implementation of the DSL Tools Generator. It begins with an examination of the project's objectives and underlying principles that guided the development process. Subsequently, it provides an overview of the architecture: the inputs and outputs, how they are parsed, analyzed, and transformed, and also how the generator reacts to changes to inputs. It also includes information about the architecture of the code and how it was tested for correctness. In addition, it explores the nuances of the four main code generation modules, the challenges that arose during their development, and how their outputs fit together. The chapter also describes which parts of the DSL tools have to be implemented by the user themselves.

> A **user** is someone (likely a programmer) who uses the *DSL Tools Generator* program to generate tools for a given DSL.
> An **end-user** is someone who uses the DSL (and its tools) inside an editor. The end-user might be the same person as the user, or it might be a completely different person, not necessarily a programmer.

5.1 Goals and Principles

The goal is to implement a program that, given a grammar and a configuration file, generates the code for tools that provide support for working with the given language. As described in section 2.3 on page 6, the most straightforward way to provide editor support for a language is with a combination of a **language server** and a corresponding **editor extension** (for an existing code editor), communicating over LSP. Therefore, that is what the generator should generate – after some initial **configuration** by the user. The end-user editing experience should include syntax highlighting and semantic highlighting, real-time reporting of syntax and semantic errors, and basic code completion.

This tool aims to be more lightweight than comparable tools (especially industrial-strength language workbenches like Xtext and MPS). It targets the .NET ecosystem which lacks such a tool. It should be installable as a simple command-line tool. To enable quick prototyping

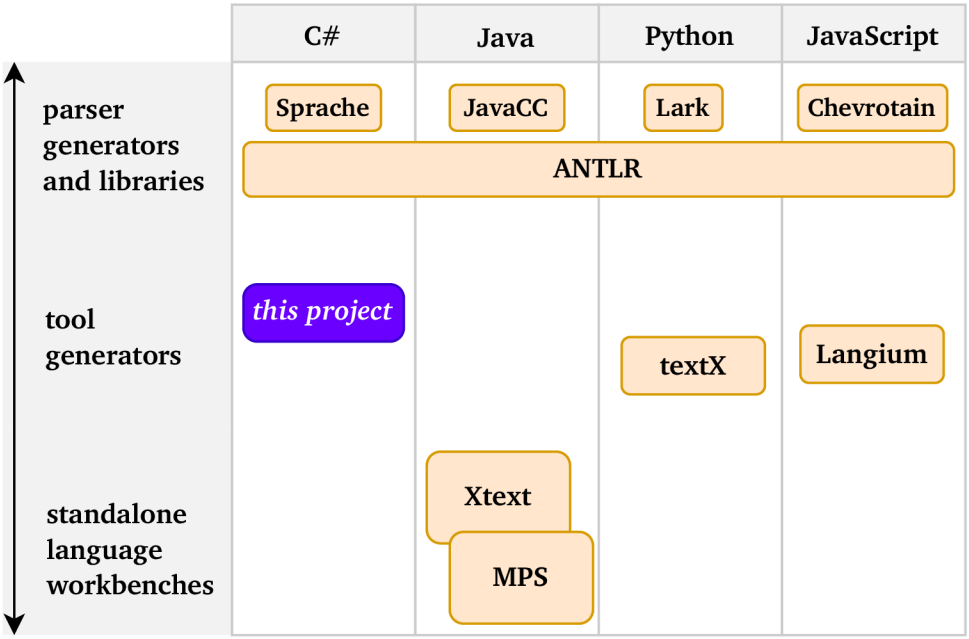


Figure 5.1: An overview of how this tool fits into the larger ecosystem of similar tools.

and fast feedback loop, it should automatically regenerate outputs when one of the inputs changes. Adding a DSL as a small component to an existing project should be straightforward. Using the tool should be possible with just basic knowledge of parsers, and should not require deep knowledge of parsing theory or lots of experience with metamodeling.

The VS Code code editor was chosen as the target for the generated editor extension, because it is not only very popular (meaning that both users and end-users are likely to be familiar with it), but also relatively easily extensible. A disadvantage of choosing VS Code is that it relies on TextMate grammars to highlight syntax. That means that the generator should be able to generate a TextMate grammar for basic syntax highlighting. TextMate grammars tokenize input in a different way than ANTLR lexers (this is described in more detail in section 5.10 on page 25). Any highlighting dependent on syntax or semantics (as opposed to just tokens) can be accomplished using the *Semantic Highlighting* feature of LSP.

5.2 Architecture

Since one of the goals of this program is to be reactive, i.e. automatically react to changes in input files, it uses *Reactive Extensions for .NET* to build a reactive pipeline. By composing (chaining) operators on `IObservable<T>` objects, a pipeline is created that re-runs all relevant generators when a new value of the input is available. The initial event emitters are based on a single `ChangeToken` (from a `PhysicalFileProvider`[1]) for each input file. There are only two input files at the moment, the grammar (`.g4`) and configuration (`dtg.json`) files, but future versions could possibly read other files, for example, the `.interp` files emitted by ANTLR that contain details about the ATN.

[1] both available in the `Microsoft.Extensions.FileProviders` package

The project consists of multiple separate generators that can be run individually or all at once. However, which ones are actually run is configured by the user. Currently, there are four generators: `LanguageServerGenerator`, `AstCodeGenerator`, `VscodeExtensionGenerator`, and `TmLanguageGenerator`.

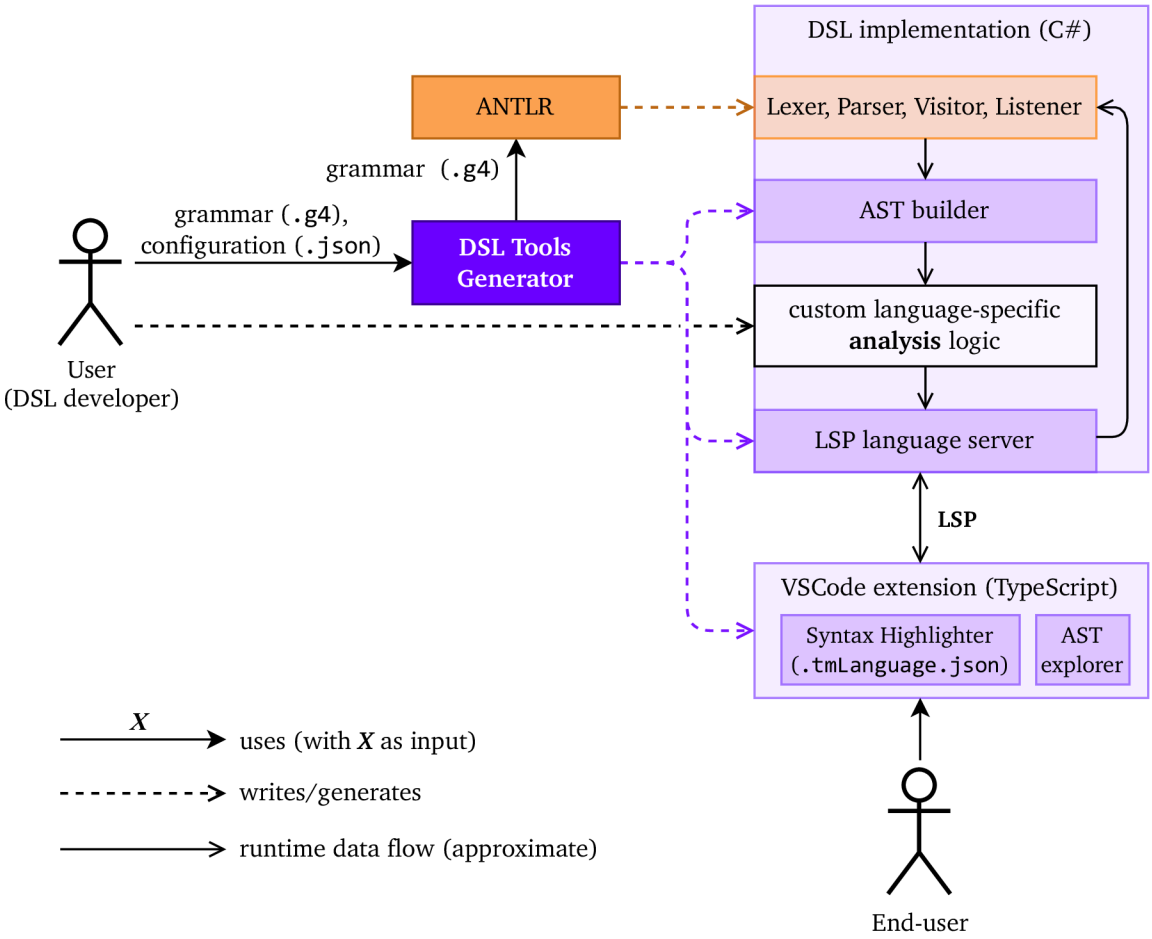


Figure 5.2: Overview of the architecture of the DSL Tools Generator.

Currently, the language server and AST code generators output code in C#, but this could later be extended to other target languages. The AST generator is split into model generation and model-to-text transformation phases, so only the second stage would have to be written for an additional target language.

The code generators internally use a custom `IndentedTextWriter` class and a C# feature called *interpolated string handlers* to generate fairly readable code with the appropriate indentation and formatting, even when inserting multiline strings through string interpolation.

Most of the functionality is tested using a test suite consisting of a total of 139 tests (including individual data rows of parameterized tests) using the *xUnit* testing framework. Most of these are tests of the TextMate grammar generation and AST code generation.

5.3 User Interface

A command-line interface was a natural choice for a tool used by programmers to generate source code. However, there is still the possibility of adding a companion VS Code extension (with a graphical user interface) in the future.

The generator needs a way of configuring the output directory paths, language/extension identifiers, and many other (mostly optional) settings. This is done in a JSON configuration file, `dtg.json`, located in the project folder. A command `dtg generate dtgConfigSchema` is available for generating a JSONSchema file used by VS Code to provide completion and documentation for the various available options. The JSON parser component used (`System.Text.Json`) is configured to allow comments and trailing commas, making editing less painful and giving users the possibility to document their configuration choices.

The `System.CommandLine` package is used to parse command-line arguments and also to provide tab-completion in terminals. The command `dtg generate` or `dtg watch` runs all configured generators (once, or in *watch* mode). Alternatively, the `dtg generate` command can be given the name of the generator to run (`ast`, `tmLanguage`, `vscodeExtension`, `languageServer`) to run only a single generator, optionally in *watch* mode specified using the `--watch` option.

5.4 Parser Generator

For the generated tools to be able to process the AST of the documents in the defined language, they need a way to parse the input into a tree representation (i.e. a *parse tree*, which can then be converted into an AST). To generate the parser, the ANTLR parser generator was chosen, including its grammar definition (meta)language. Not only is it quite popular, meaning that a significant portion of our target users already know it, it is also quite permissive in what grammars it accepts (ALL(*) vs. LL(k)). ANTLR has a runtime representation (ATN) of the grammar, which is useful for code completion and error handling. It is able to generate code in several target languages, which gives us the flexibility to target more languages in the future as well.

5.5 Language Definition

There are multiple options regarding the optimal way to describe the DSL so that the generator can automatically generate a syntax highlighter and an AST data structure:

a) define a custom DSL for describing DSLs (a *metalanguage*)

b) use an existing metalanguage (e.g. a grammar definition language of a parser generator like ANTLR)

c) use a data serialization or configuration format like JSON, XML, YAML, or TOML

Option a) offers a lot of flexibility, but in this case combining b) and c) is enough and brings some additional benefits. Adopting ANTLR's grammar definition language allows users to

reuse existing grammars, and projects that already use ANTLR can seamlessly upgrade to using *DSL Tools Generator* and get editor support "for free". Using JSON (more precisely, *JSON with Comments*, sometimes abbreviated as `jsonc`) for the configuration file with a custom JSONSchema allows editors such as VS Code to offer basic validation and completion support.

An ANTLR grammar can either be a *combined grammar*, containing rules for both the lexer and the parser, or be separated into a *lexer grammar* and a *parser grammar* that references the lexer's `.tokens` file via the `tokenVocab` grammar option. The lexer rules can be reused as token definitions for syntax highlighting, and the parser rules can serve as the basis from which to infer the structure of the AST. The rest is specified in the configuration file.

5.6 Grammar Analysis

To analyze the input language description, we could either build a custom parser of the ANTLR grammar definition language or use a third-party library.

An alternative approach of extracting the information from the official ANTLR tool (written in Java) in some manner could potentially bring some benefits, such as always having data from the source of truth, thus hypothetically future-proofing this project (although the ANTLR project is quite stable). However, getting the data from Java to .NET is not straightforward. The *IKVM* project which provides interoperability between Java and .NET only supports Java 8 at the moment, while ANTLR requires at least Java 11).

The *Antlr4Ast* C# library by Alexandre Mutel was chosen. It provides an AST of the grammar definition – figure 5.3 shows what types of nodes it might contain.

The AST code generator and TextMate grammar generation modules each perform some analysis on elements of the grammar in order to be able to produce correct output.

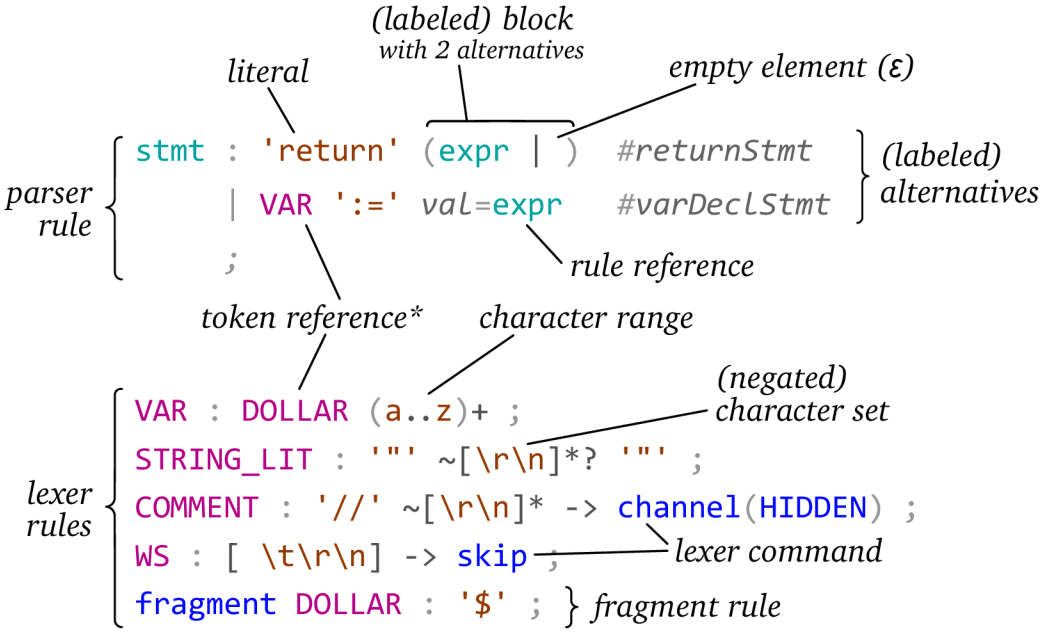


Figure 5.3: Overview of the various types of nodes in the AST of an ANTLR grammar (using terminology from the *Antlr4Ast* library.)

5.7 AST Code Generation

The developed tool automatically generates the class hierarchy for an AST based on the parser rules defined in the grammar. Each rule is analyzed using pattern matching, a target-independent code generation model is created and then transformed (serialized) into code. Currently, only the transformation of the model into C# code is supported, but other target languages can be added in the future.

The model mainly describes the semantics of the code to be generated – information about the node classes (name, base class, what rule they were derived from, etc.), their properties and subclasses, but also how to map data from the parse tree to nodes of the AST.

An `AstBuilder` class (subclass of the `BaseVisitor` class generated by ANTLR) is generated that visits each node and builds an AST node with data mapped from the parse tree. The simplest case would be retrieving the text of the only `ID` token – `context.ID().GetText()`, or of a labeled token – `context.varName.Text`.

To know how to access the right parts of the parse tree to retrieve data for child nodes without a label, the grammar is traversed using an element counting algorithm that assigns each node two indices within the parse tree (see figure 5.4): the child node index and the index among child nodes of the same type. For example, a specific `NUMBER` token might be the third child node but only the first `NUMBER` token of its parent. In some cases, one or both indices are unknown (for instance, after a repeated or optional element).

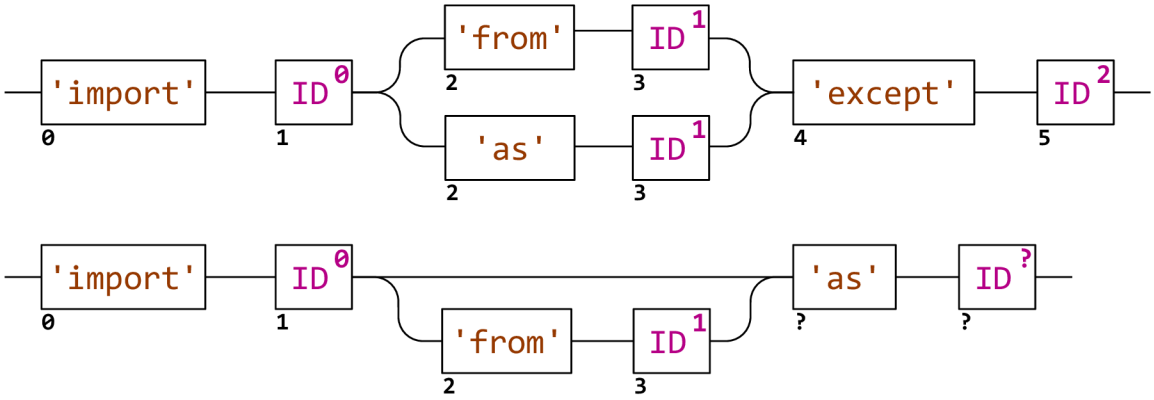


Figure 5.4: Syntax element numbering. The number outside the boxes is the index of that parse tree node within its parent's list of child nodes. The number inside the box is the the index within child nodes of the same type. A question mark indicates that the index is unknown (cannot be statically determined).

For each parser rule, one or more node classes are generated with properties inferred from the syntax elements:

- a child node (rule reference) is mapped to a property of the child node's type (e.g. `Expression Value`)
- a labeled token is mapped to a `string` property (e.g. `string VariableName`)
- an unlabeled token is mapped to a `string` property if the token name looks like something important (like *identifier*, *literal*, *value*, *name*, *value*, *type*)

- an optional literal or token reference is mapped to a `bool` property, e.g. `isAbstract='abstract'?` → `bool IsAbstract`
- a delimited list of elements (e.g. `NUM (',' NUM)*` or `expr (COMMA expr)+`) is mapped to a list property, whose type is `IList<T>` where `T` is either `string` (token text) or the node class (e.g. `Expression`)
- recurse into blocks

For a rule with labeled alternatives, a node (sub)class is generated for each alternative:

```
expr : ID #idExpr | NUM #numExpr ;
```

↓

```
public abstract partial record Expression : AstNode;
public partial record IdentifierExpression(string Identifier) : Expression;
public partial record NumberExpression(string Number) : Expression;
```

The names of properties are based on the element's label (if present) or the name of the referenced rule or token and then pluralized (using the *Humanizer* library) if needed. For instance, `NUMBER+` would be mapped to a property named `Numbers`.

The list of properties is postprocessed to rename or merge any duplicates, i.e. multiple properties with the same name. For example, `(expr '+' expr)` would generate two properties named `Expression`, so they are renamed to `LeftExpression` and `RightExpression`.

5.8 Language Server Code Generation

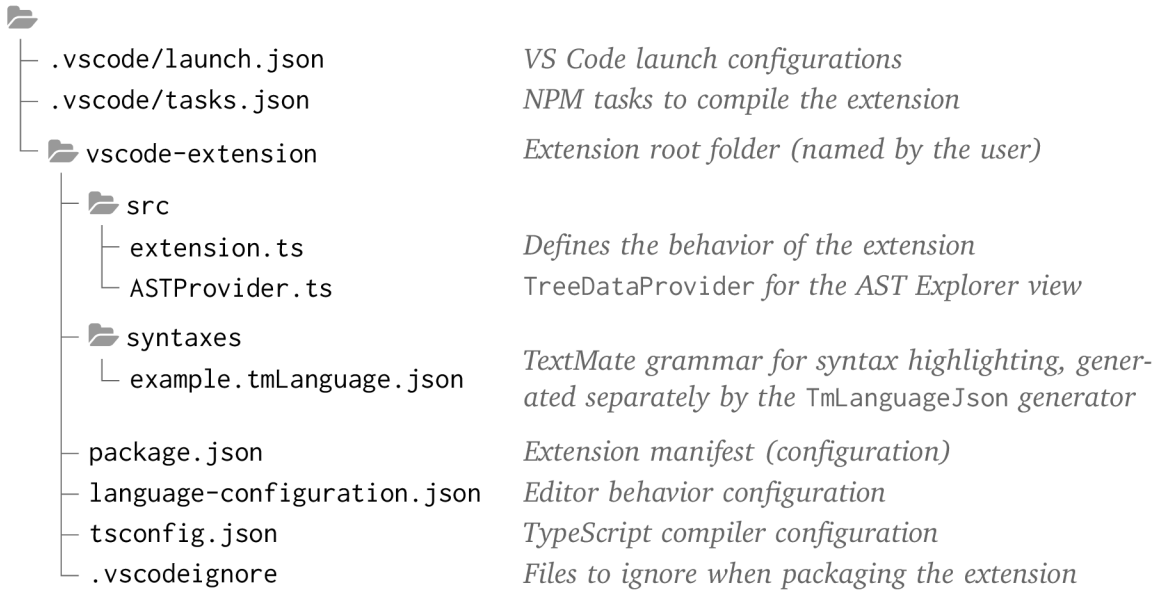
The tool can automatically generate C# code for a language server that uses the OmniSharp LSP library. The generated code includes the following functionality:

- **document synchronization handling**
- **live diagnostics** – errors discovered by the ANTLR lexer and parser are automatically collected and reported to the LSP client (and then displayed in VS Code) immediately
- custom notification type for **reporting AST data** (to be displayed in the AST Explorer view)
- base class for **semantic highlighting** – can be overridden by the user to inspect the AST node and highlight it accordingly
- default **hover** handler implementation
- **TCP server mode** for quick development with hot-reload
- a basic **code completion** handler

The generated code includes an adapted version of the code from the *antlr4-c3* library created by *Mike Lischke* and ported to C# by *Jonathan Philipps*, which provides code completion functionality for parsers generated by ANTLR. It finds possible tokens or rules that could follow the current position in the document by walking through the parser's ATN.

5.9 VS Code Extension Generation

Currently only generation of (desktop) VS Code extensions is implemented; however, the same `.vsix` extension package should also work in other compatible editors like VSCodium, Eclipse Theia or Gitpod (although this was not tested). Support for *VS Code for the Web*[2] and other code editors might be added in the future.



After the extension is started via the debugger in VS Code, it connects to the language server over TCP – the user is expected to launch the server manually (and restart it if necessary). This has the benefit of easily allowing active development of the language server, including restarting it or applying changes via the *hot-reload* functionality of the .NET Runtime. As soon as the connection is lost, the language client tries to reconnect after 10 seconds, but the user can force reconnection later using the "*XYZ*: Restart Language Server" command (where *XYZ* is the extension's configured display name).

When publishing the extension, the language server project is automatically built with the output directory set to a `LanguageServer` directory within the extension. When installed and activated inside VS Code, the language client will launch the executable of the language server (and manage its lifetime) and connect to it over standard input/output.

An *AST Explorer* is included that shows the AST as an interactive tree view component that allows browsing of the child nodes and properties of individual nodes of the AST. This becomes especially useful once the user adds additional properties for semantic analysis.

5.10 Syntax Highlighting

The process of translating an ANTLR lexer grammar into a TextMate grammar for syntax highlighting consists of the following steps:

[2] https://vscode.dev/

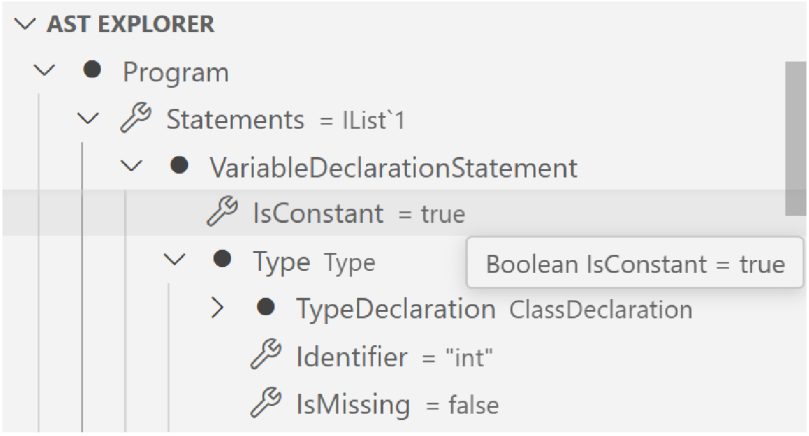


Figure 5.5: An optional *AST Explorer* view that shows the hierarchy of AST nodes in a VS Code tree view component.

1. **walk over the grammar's AST** (provided by the *Antlr4Ast* library) and **collect all rules** for translation – implicit token rules (from literals in combined grammar's parser rules) and explicit lexer rules defined in the combined grammar or in a lexer grammar "imported" via the `tokenVocab` option
2. for each lexer rule, **generate the corresponding TextMate rule** that matches the same input
3. **reorder** the generated TextMate rules (the order of rules in a TextMate grammar is important)
4. **assign a TextMate scope name** to each rule (used by selectors in color themes to set the token's color and font style)
5. **serialize** as a `.tmLanguage.json` file

Translating an ANTLR lexer rule into a TextMate rule

A set of ANTLR lexer rules can be translated into a sequence of equivalent TextMate rules by traversing the rule's ASTs and recursively generating a regular expression for each node. The rules' ASTs together can be viewed as a forest of trees with cross-references to other trees (rules) – traversing the rule references might lead to infinite recursion, so tracking the chain (stack) of visited rules is necessary.

Each *syntax element* (literal, token reference, character set, etc.) has an appropriate translation into a regular expression. These are listed in table 5.1 on the next page.

If the element was marked as optional (`?`) or repeated (`+` or `*`), the suffix is appended to the regular expression. Care must be taken to ensure that the suffix applies to the element as a whole and not just its last part. For example, to make a pattern like `abc` optional, it would be incorrect to produce a pattern like `abc?`, since that would only make `c` optional. Instead, the whole pattern must be enclosed in a group[3]: `(?:abc)?`.

[3] A non-capturing group `(?:...)` is preferred, as it does not affect the numbering of capturing groups.

element	original form	translated (regular expression)	explanation
alternative	$\alpha\ \beta\ \gamma$	$R(\alpha)R(\beta)R(\gamma)$	
	`'a' 'b' [0-9]`	`ab[0-9]`	
alternative list	$\alpha\|\beta\|\gamma$	$R(\alpha)\|R(\beta)\|R(\gamma)$	
	`'a' | 'b' | [0-9]`	`a|b|[0-9]`	
literal	`'if'`	`if`	
character set	`[ab0-9])`	`[ab0-9]`	
	`[^a]`	`[\^a]`	^ must be escaped
	`~[\r\n]`	`[^\r\n]`	negation
lexer block	`('a' | 'b' | [0-9])`	`[ab0-9]`	
	`('^' | 'a')`	`[\^a]`	^ must be escaped
	`~('\r' | '\n')`	`[^\r\n]`	negation
range	`'a'..'z'`	`[a-z]`	
wildcard	`.`	`.`	
token (rule) reference	`DIGIT`	$R(A(\texttt{DIGIT}))$	use the regex for the referenced rule
	`EOF`	`\z`	end of file

$R(e)$ denotes the regular expression for element e.
$A(r)$ denotes the alternative list for rule r.

Table 5.1: Overview of the translations of various types of syntax elements to a regular expression for the *Oniguruma* regex engine.

Special handling is given to rules that match keywords. These are additionally enclosed in *word boundary* anchors (`\b`) to prevent unwanted matches. These anchors are only placed at those sides of the rule that are determined to start/end with a *word character*, since a pattern like `\b(?:@import)\b` would only ever match right after a word.

Lexer rules in ANTLR might be marked as case insensitive, in which case the pattern is enclosed with the equivalent regex syntax:

```
SELECT_KW options { caseInsensitive=true; } : 'select' ;
                              ↓
                        (?i:select)
```

The "Shadowing" Problem

There is a side effect of the translation between ANTLR grammars and TextMate grammars. It is caused by the differences in behavior between ANTLR-generated lexers and TextMate-based syntax highlighters when multiple rules can match the same input. Unlike ANTLR lexers, TextMate highlighters move on after finding the first rule that finds a match, regardless of its length. That means that if there is a rule further down the list that could match more characters, it will not be considered, as the engine has already advanced to the next character.

This means that some rules may never match any characters because they are completely or partially *shadowed* by other rules, which causes the syntax highlighting to appear incorrect and broken.

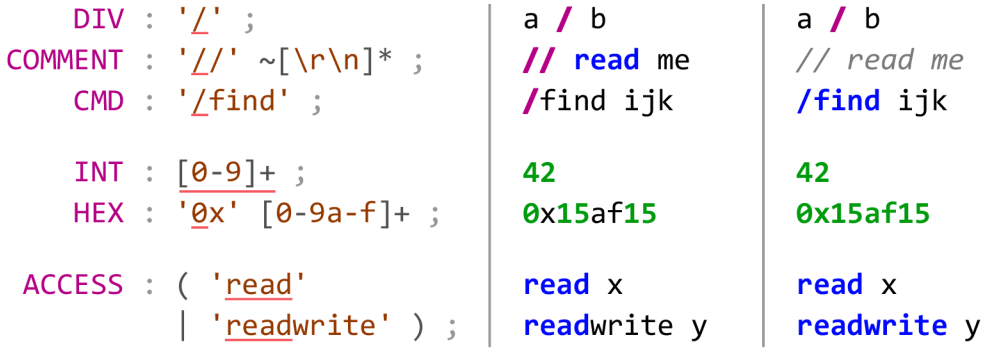


Figure 5.6: An example of a grammar that contains three instances of the shadowing problem if translated naively into TextMate grammar patterns: as long as `DIV` is always tried first, `COMMENT` and `CMD` will never match any input as the first slash character will already have been consumed by `DIV`. A similar situation, but with non-literal rules, happens with `INT` and `FLOAT`, the latter can never match anything. This problem can also occur inside a single rule, namely in blocks with multiple alternatives.

Reordering

The generated TextMate rules are sorted according to the following ordering:

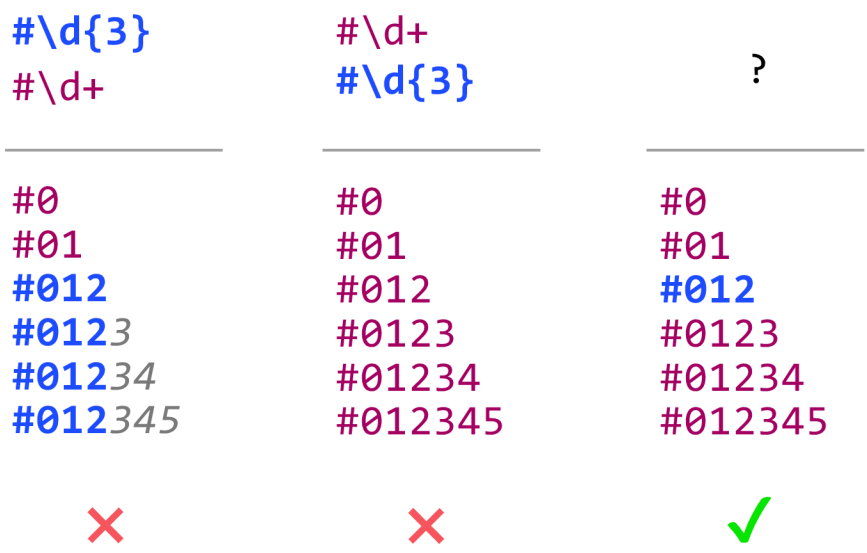


Figure 5.7: An example of two rules that partially shadow each other. This problem cannot be solved by simple reordering of the two rules.

1. shadowing mitigation patterns
2. keyword rules (since they have *word boundary* anchors and so are unlikely to produce unwanted matches; also, they are in danger of being shadowed (dominated) by more general rules like `ID : [a-zA-Z_]`)
3. all other rules, in an order that prevents unwanted shadowing

The picked approximation (complexity – number of descendants of the rule in the grammar) is not very accurate, but most cases where it would cause a problem in practice (involving keywords) will be solved by using *word boundary* anchors.

Assigning a TextMate scope name

The generated TextMate rules will not do anything at all unless they have a TextMate scope name that can be targeted by selectors in color themes to actually assign a color to the token.

The chosen solution is to consider carefully chosen heuristics and conventions to guess the correct scope name. The user can always override this choice by specifying the desired scope name for a given rule.

1. use the TextMate scope name explicitly specified by the user, if available
2. **`keyword`** if the rule is a keyword rule
3. **`variable`** if the rule name suggests an identifier or name of something
4. **`constant.numeric`** if the rule name suggests a numeric literal
5. **`comment`** if the rule name suggests a comment
6. **`string`** if the rule name suggests a string/text/characters

Shadowing Mitigation Patterns

If the automatic (approximate) rule ordering fails to prevent rule shadowing, the user can add a rule conflict item to the configuration file. The generator will add a special anti-shadowing rule to the top of the TextMate grammar. The regex pattern of this rule is constructed from the underlying ("conflicting") rules' patterns such that the one that matches the most characters is selected.

The basic idea is to try each sub-pattern and remember the position at the end of the match and then, if all sub-patterns found a match, compare which of the match-end positions is farther from the start (or equivalently closer to the end) of the string.

Unfortunately, storing and later comparing an arbitrary position is not one of the operations available in the *Oniguruma* regex flavor used by TextMate grammars. However, we can instead store the rest of the line after each match. Then, if the rest of the line after the second sub-pattern's match (`rest2`) is a suffix of the rest of the line after the first sub-pattern's match (`rest1`), L_1 matched **fewer** characters.

`(?=(?<L1>\d:\d)  (?<rest1>.*)$)`	pre-match L_1
`(?=(?<L2>\d:\d:\d|\d)(?<rest2>.*)$)`	pre-match L_2
`(?:(?<L1match>\k<L1>)(?!.+?\k<rest2>$)`	match L_1 if it's not shorter than L_2
`  |(?<L2match>\k<L2>))`	otherwise match L_2

Figure 5.8: Pattern that finds the longer match of two sub-patterns (named L_1 and L_2). Whitespace added only for readability and is not part of the pattern.

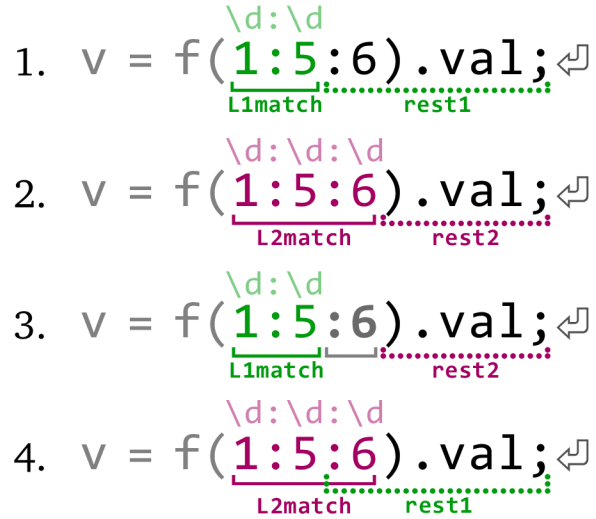


Figure 5.9: Illustration of the trick used to find the pattern that matches the most characters

Chapter 6

Testing and Evaluation

This chapter will describe how the developed tool was tested by creating two independent sets of DSL tools and will evaluate the quality of the results and the effort required to achieve them.

An ideal language for testing purposes should be complex enough to be able to test most features of the *DSL Tools Generator*. This means that it should have multiple token types, some of them being highlighted differently based on additional context from the syntax or semantics, opportunities for validation of semantics and for code completion based on previous declarations, etc.

6.1 Avro IDL

One such language is Avro IDL, an Interface Description Language with a C-like syntax created as an alternative to schemas defined in JSON for the Apache Avro data serialization system [1]. An Avro IDL file defines a single protocol and may also contain import statements and named schema declarations (records, enums, and fixed-length types). A grammar of the Avro IDL language is available in its GitHub repository[1].

After downloading the grammar file into an empty directory named `AvroIDL`, a basic C# console project was created using the command `dotnet new`. The necessary dependencies (ANTLR and the OmniSharp LSP library) were added using the following commands:

```
dotnet add package Antlr4.Runtime.Standard
dotnet add package OmniSharp.Extensions.LanguageServer --version 0.19.9
```

A configuration file was created specifying the name of the grammar file, the name of the C# project file, the output paths, the path to the ANTLR tool, the namespaces in which to place the generated files, and VS Code extension name and ID.

At this point, the generated VS Code extension was already capable of providing basic syntax highlighting of an example Avro IDL file.

Then adding a few lines of C# code was needed to start the language server if the `--ls` command-line argument is given and to initialize any options for the language server. This in-

[1] https://github.com/apache/avro/blob/main/share/idl_grammar/org/apache/avro/idl/Idl.g4

cludes registering any LSP request handlers (written by the user or as part of the generated language server code), in this case the `BasicHoverHandler` and `BasicCodeCompletionHandler`.

After manually starting the language server[2] and starting debugging of the VS Code extension, the AST explorer view and a basic hover and code completion functionality was available (see figures 6.1 and 6.2).

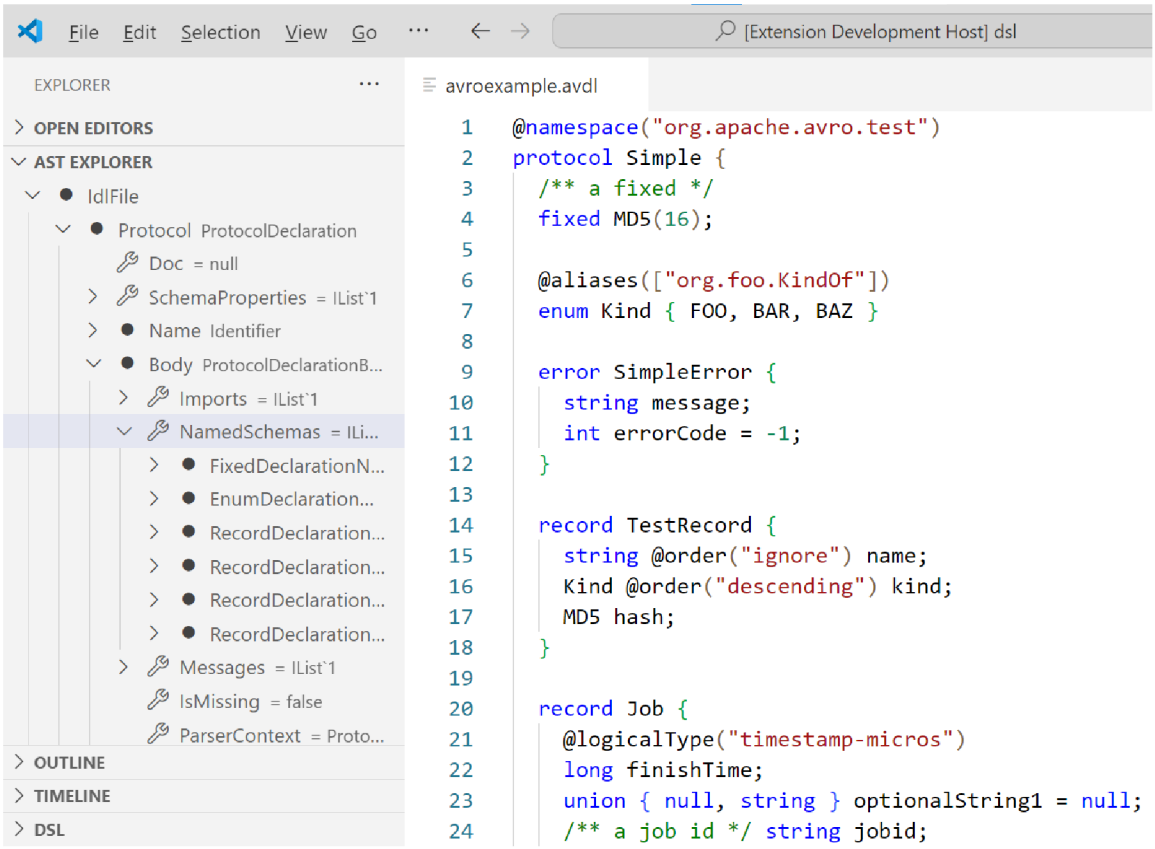


Figure 6.1: An example Avro IDL file being edited using the VS Code extension and language server generated automatically using *DSL Tools Generator* (before any manual adjustments).

Next, the following highlighting improvements were achieved by adding a custom handler derived from the generated `BasicSemanticTokensHandler`:

- classify (highlight) identifiers in declarations or references to a named schema (record, enum, etc.) as types
- classify keywords as identifiers when used as identifiers (Avro IDL allows using keywords as identifiers)
- classify schema properties (e.g. `@order`) as the *decorator* token type
- classify enum members as the *enum member* token type

[2]Manually starting the server is only needed during development, where it provides the benefit of being able to debug and restart the server arbitrarily from a development environment

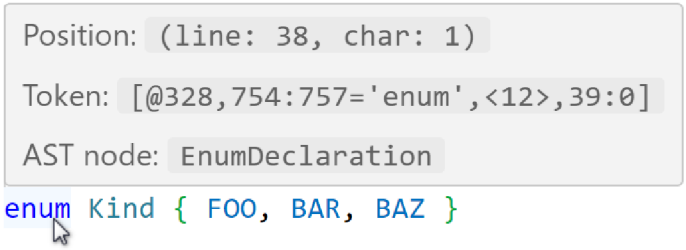


Figure 6.2: The default LSP hover request handler (`BasicHoverHandler`) – can be overridden to display arbitrary plain or formatted text based on the AST node.

A simple semantic analysis step was added that collects named schema declarations into a symbol table and displays them as suggestions using a custom `CompletionHandler` when a type is expected (see figure 6.4 on the following page).

The created editor support is comparable to and, in some aspects, exceeds the main VS Code extension for Avro IDL[3] by *StreetSideSoftware*, which only provides basic syntax highlighting.

6.2 CSS

The same approach was taken with the CSS (*Cascading Style Sheets*) language, with the following differences:

- the grammar was written manually to only contain a subset of CSS
- the language ID was arbitrarily set to `FCSS` to distinguish it from the CSS language support built into VS Code
- code completion items were sourced from a pre-defined list of element (tag) names and property names and values instead of using a symbol table

The created tools are shown in figures 6.5 and 6.6.

6.3 Evaluation

For both of the tested languages, the DSL Tools Generator was able to successfully generate a VS Code extension with syntax highlighting and a language server with an AST data structure. However, while it does streamline tool creation, several areas for improvement have been identified, suggesting many opportunities for future development.

One possible improvement would be to reduce the number of initial setup steps, for example, by creating a template for the `dotnet new` command that would automatically install the required dependencies and create a sample configuration file. Also, the AST generator could detect situations where all alternatives of a rule have an element with the same label, so the property could be lifted into the base class. Semantic highlighting could be further improved –

[3] https://marketplace.visualstudio.com/items?itemName=streetsidesoftware.avro

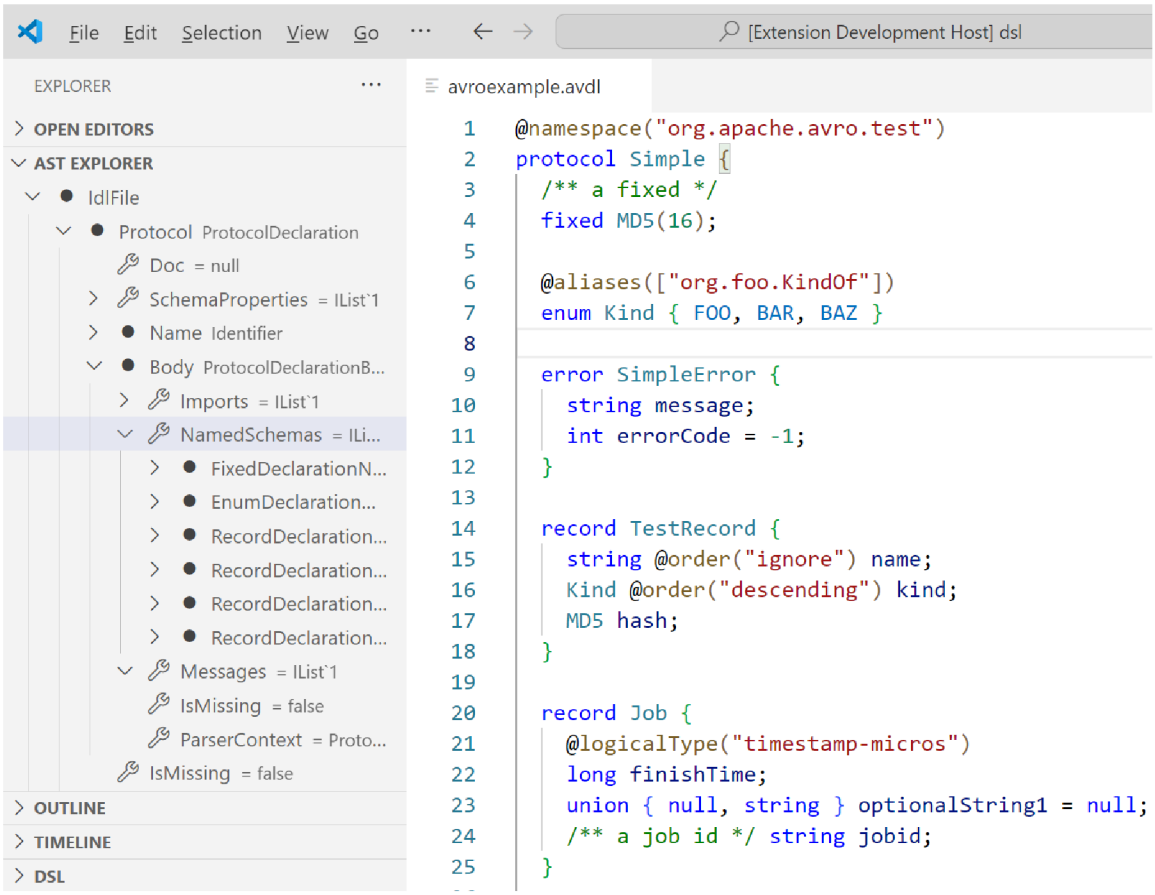


Figure 6.3: An example Avro IDL file being edited using the VS Code extension and language server created using *DSL Tools Generator*, **after** manual adjustments to semantic highlighting.

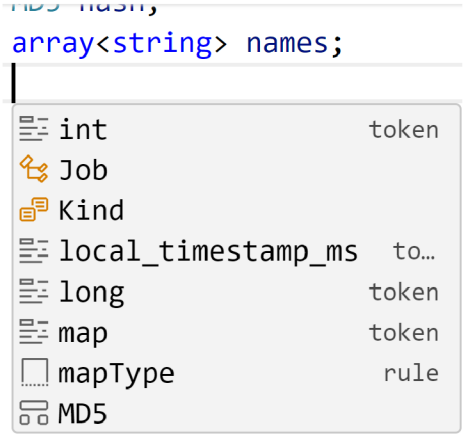
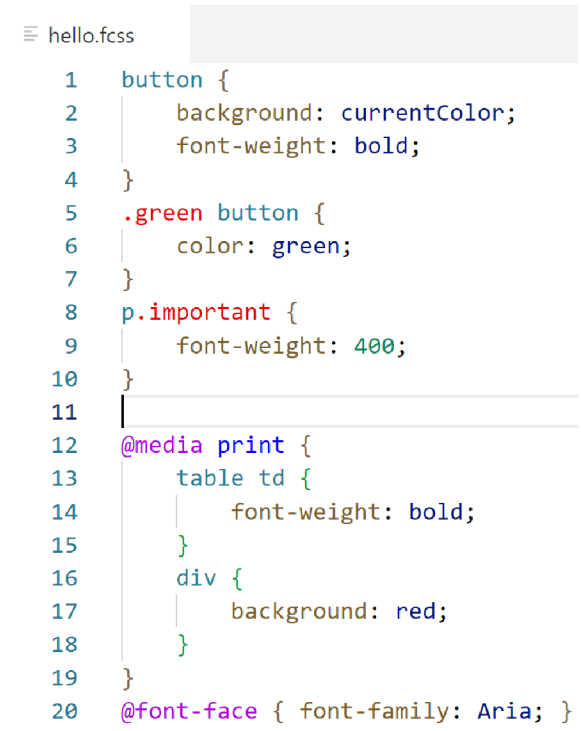
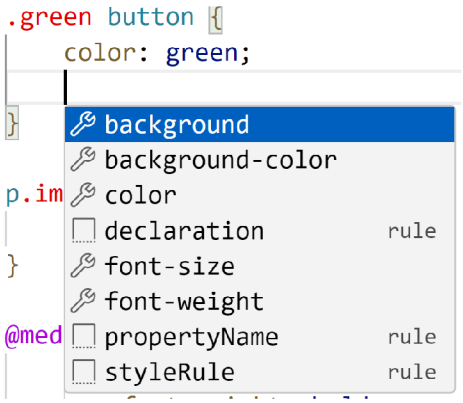


Figure 6.4: Code completion menu after adding a simple `CompletionHandler` that offers items from the symbol table as suggestions.



(a) Syntax and semantic highlighting of CSS inside VS Code (includes manual adjustments to the highlighting of class, property and tag names, and of at-rule keywords like `@media`).



(b) Code completion suggesting property names inside CSS rules.

Figure 6.5: The created language support for (a subset of) CSS

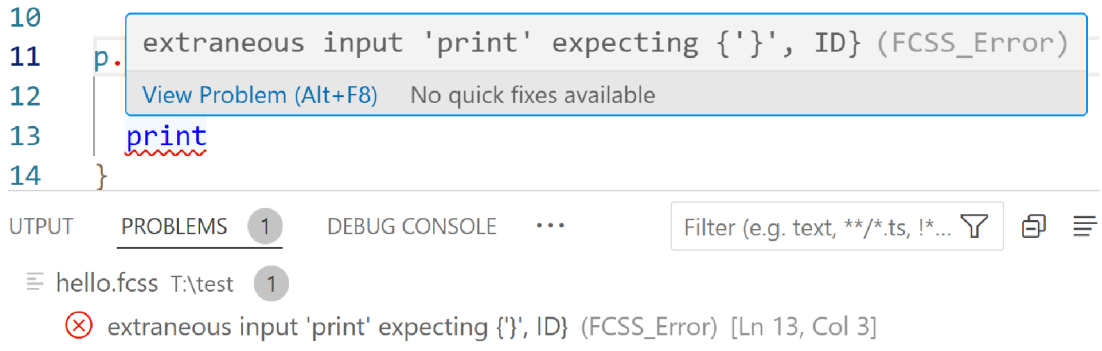


Figure 6.6: An example of a diagnostic (a syntax error in this case) sent by the language server and displayed by VS Code.

the user could specify highlighting rules in the configuration file using some kind of selectors. Code completion is a complex feature and, while currently functional, does not always return the most useful suggestions in a given moment.

Chapter 7

Conclusion

The goal of this thesis was to design and implement a program capable of generating tools for using a domain-specific language (DSL) in a code editor like VS Code. First, DSLs themselves and the options for adding editor support for them was discussed. Then, the Language Server Protocol (LSP) was described. The next chapter focused on ANTLR and how its unique ATN feature can be utilized for code completion functionality. Subsequently, the thesis explored the design choices, methodologies, and challenges encountered during the implementation – including the approach taken to resolve the *shadowing* problem caused by differences in matching behavior between ANTLR lexers and TextMate grammars used for syntax highlighting. Finally, the developed tool was tested and evaluated by using it to create editor support for two DSLs, Avro IDL and CSS.

The DSL Tools Generator is capable of saving a significant amount of time and effort when creating editor support for a DSL. Compared to manual implementation approaches that require manually defining an AST data structure and learning about TextMate grammars, the Language Server Protocol, and various technical details, the process of creating tools for DSLs was accelerated. The generator successfully generates a VS Code extension with syntax highlighting and a language server with an auto-generated AST data structure inferred from the grammar. The AST can be analyzed by user code for a semantic analysis step, for example, to provide domain-specific validation or code completion results.

The generator also supports features that further improve the developer experience, e.g. by automatically rerunning the generator when the grammar or configuration is modified, or by allowing debugging and restarting the language server with automatic reconnection from the generated VS Code extension. It also includes an AST Explorer view that helps in understanding the structure of the AST and seeing any data attached by the semantic analyzer.

7.1 Future Work

Areas where the generator could be improved and extended include:

- generating extensions/plugins for other editors than VS Code: Visual Studio 2022, IntelliJ IDEA, Vim or Neovim, Emacs, web-based editors: the web version of VS Code, Monaco, CodeMirror, Ace
- implement other features of the Language Server Protocol: Code Actions, Inlay Hints, Code Lens, Formatting…
- use the developed generator to replace the JSON-based configuration mechanism with a custom DSL for defining DSLs (a *metalanguage*)
- investigate options for adding support for other related protocols: Debug Adapter Protocol (DAP) and Build Server Protocol (BSP)
- implement more target languages for code generation
- implement native AST „un-parsing“ to make the ASTs serializable back into source code (without losing formatting and comments) so that the DSL developer can implement code actions and automated refactorings (*extract function*, *extract to local variable*, etc.)
  - this could be done by storing trivia (whitespace, comments, skipped/erroneous text) as part of token data, like Roslyn, the C# and VB.NET compiler

Bibliography

[1] *IDL Language* [online]. Apache, Oct 2023 [cit. 2024-04-30]. Available at: https://avro.apache.org/docs/1.11.1/idl-language/.

[2] Corsius, M., Hoppenbrouwers, S., Lokin, M., Baars, E., Sangers Van Cappellen, G. et al. RegelSpraak: a CNL for executable tax rules specification. In: *Proceedings of the Seventh International Workshop on Controlled Natural Language (CNL 2020/21)*. 2021.

[3] Dejanović, I., Vaderna, R., Milosavljević, G. and Vuković, Ž. TextX: A Python tool for Domain-Specific Languages implementation. *Knowledge-Based Systems*. 2017, vol. 115, p. 1–4. DOI: 10.1016/j.knosys.2016.10.023. ISSN 0950-7051. Available at: http://www.sciencedirect.com/science/article/pii/S0950705116304178.

[4] Enet, J., Bousse, E., Tisi, M. and Sunyé, G. On the Suitability of LSP and DAP for Domain-Specific Languages. In: IEEE. *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Västerås, Sweden: IEEE Computer Society, October 2023, p. 353–363. DOI: 10.1109/MODELS-C59198.2023.00066. ISBN 979-8-3503-2498-3. Available at: https://hal.science/hal-04245594.

[5] Fowler, M. and Parsons, R. *Domain-specific languages*. 1st ed. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN 978–0–321–71294–3.

[6] *Official page for Language Server Protocol* [online]. Microsoft, 2017. Last updated 2022-05-23 [cit. 2024-01-17]. Available at: https://microsoft.github.io/language-server-protocol/.

[7] Parr, T. *The Definitive ANTLR 4 Reference*. 2nd ed. Pragmatic Bookshelf, 2013. ISBN 978-1-93435-699-9.

[8] Parr, T. and Harwell, S. *ANTLR* [online]. [cit. 2024-03-25]. Available at: https://www.antlr.org/.

[9] Parr, T., Harwell, S. and Fisher, K. Adaptive LL(*) parsing: The Power of Dynamic Analysis. In: Black, A. P. and Millstein, T. D., ed. *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. New York, NY, USA: ACM, October 2014, vol. 49, p. 579–598. SPLASH. DOI: 10.1145/2660193.2660202. ISBN 9781450325851. Available at: https://www.antlr.org/papers/allstar-techreport.pdf.

[10] *Documentation | Langium*. TypeFox, 2022. Available at: https://langium.org/docs/.

[11] Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M. et al. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. Self-published, 2013. ISBN 978-1-4812-1858-0. Available at: http://www.dslbook.org.

[12] Völter, M., Koščejev, S., Riedel, M., Deitsch, A. and Hinkelmann, A. A Domain-Specific Language for Payroll Calculations: An Experience Report from DATEV. In: Bucchiarone, A., Cicchetti, A., Ciccozzi, F. and Pierantonio, A., ed. *Domain-Specific Languages in Practice: with JetBrains MPS*. Cham: Springer International Publishing, 2021, p. 93–130. DOI: 10.1007/978-3-030-73758-0. ISBN 978-3-030-73758-0. Available at: http://voelter.de/data/pub/PayrollDSL.pdf.