Your Thesis Title Here

Rafael Tanzer

Cyber-Physical-Systems, ..., ...
TU Vienna

Supervisor: Michele Chiaria, Simone , Francesco Pontigia and Ezzio Bartocci

May 2025

This thesis is submitted in partial fulfillment of the requirements for the degree of *Bachelor of Science*.

Abstract

Contents

Part I

Foundations

Introduction

1.1 Motivation and Problem Statement

CPS team - waht do they need it for code growing user base - interconnecticty os , other qualities speaking for the usage of vs code - already in possesion of Haskell Checker but

1.2 Objectives and Scope

The main goal of the thesis was to create a *Language Extension* ?NAME? for *MiniProb*. Language extensions are a big part of VS Code's extension ecosystem and enable the editor to utilize custom language tooling. Such extensions support the user during the implementation process while writing code, by providing language assistance like syntax validation or code completion. While this implementation targets VS Code, the underlying logic and functionality are not limited to it — through the use of the *Language Server Protocol (LSP)*, the developed tooling can be reused across various editors and IDEs that support the protocol. The core functionality of these extensions stems from *parsers* which, against a formal language definition, convert written text into abstracted parts, validating the code in the process. These parts can then be used to extend the capabilities of the tooling to encompass referential validation, type checking, code completion, diagnostics, and other language services that enhance the development experience.

In order for parsers to function accordingly, they require formal language definitions or models. These definitions range from various types of grammars to abstract machines such as automata, which together provide the structural and syntactic rules necessary for correct interpretation and validation of source code. Based on these definitions, parsers are able to systematically identify the hierarchical structure of a program by recognizing patterns, token sequences, and nested constructs, ensuring that the code adheres to the expected form before any further processing or analysis occurs. In this thesis, a generative context-free grammar is used to formally describe *MiniProb*, a domain-specific language (DSL) designed for authoring *POMC* files. Based on this grammar, a parser - serving as the foundation - enables the implementation of a fully functional language extension to aid developers writing POMC files.
Ultimately, the resulting extension, titled ?NAME?, is intended to provide comprehensive language support for MiniProc. This includes syntactic analysis through syntax highlighting and validation, accurate resolution of symbol references and code completion as well as the implementation of type checking mechanisms to ensure semantic correctness during development. //maybe basic code completion for expressions

An appropriate set of regression tests was developed to ensure the continued correctness and stability of the language tooling. These tests include parsing tests, which verify that valid input is correctly recognized and structured according to the grammar; validation tests, which ensure that semantic rules are properly enforced and errors are accurately reported; and linking tests, which check that references between symbols or declarations are correctly resolved across different parts of a program. Together, these test categories help maintain the integrity of the parser and language services as the implementation evolves.

Lastly, this thesis includes an evaluation of the newly implemented parser, focusing on its correctness and performance in practical usage scenarios. The evaluation is based on metrics collected from representative input samples, measuring factors such as parsing speed, memory usage, and error handling. Where relevant, comparisons are also drawn against the existing *Haskell*-based parser for *.pomc* files, offering a point of reference to assess improvements or trade-offs introduced by the new implementation.

Background on Languages & Grammars

2.1 Formal Language Theory

Formal language theory deals with the study of languages – sets of strings constructed from alphabets – and the formal grammars that determine and generate them. In contrast to natural languages, which have evolved over centuries under the influence of diverse cultural, historical, and environmental factors, formal languages do not inherently relate to any perceived constructs of our environments and are generally not intuitively understood. Additionally, formal languages do not share the rich evolutionary progression — a lengthy process of gradual adaptations and refinements spanning generations — of their natural counterparts. Instead, they employ sets of axiomatic *production rules* that describe each language individually. The field of formal language theory sprung from linguist Noam Chomsky's attempts during the 1950s to definitively characterize the structure of natural languages using mathematical rules.[Jia+09] This analytical approach led to development of the *Chomsky hierarchy*, which proved to be a vital theoretical foundation for later discoveries and applications, as it was found that all information(photos, videos, numbers, axioms) can be represented as finite strings.

The *Chomsky Hierarchy* is a hierarchical classification of formal grammars, that labels them to four groups. The grammars are ranked based on the individual *expressive power* of the languages they produce, with each class including the less expressive ones. The whole hierarchy consists of four groups of grammars (and corresponding language classes), that are identified by inspecting the production rules, which get progressively less restrictive.

Type-3 grammars, or *regular* grammars, generate exactly the class of regular languages. Their productions are restricted to the two equivalent styles:

$$right-reg.: \quad A \rightarrow aB \quad or \quad A \rightarrow a \qquad and \qquad left-reg.: \quad A \rightarrow Ba \quad or \quad A \rightarrow a,$$

where A, B are nonterminals and a is a terminal. Regular expressions provide an alternative, declarative notation for these languages — each expression can be mechanically transformed into a regular grammar, and vice versa. More broadly, every grammar in the Chomsky hierarchy admits an equivalent acceptor automaton; in the case of Type-3 grammars, this correspondence yields deterministic or nondeterministic finite automata. Leveraging these conversions makes regular grammars invaluable in practice (for example, in lexical analysis, pattern matching, and protocol verification), even though their expressive power is the most limited. Type-2 grammars, or *context-free* grammars (CFGs), produce all context-free languages. Their rules take the general form

$$A \rightarrow \alpha$$

where A is a single nonterminal and α is any string of terminals and nonterminals. This additional flexibility captures nested, hierarchical structures—like balanced parentheses or most programming-language syntaxes — that regular grammars cannot. Each CFG is accepted by a corresponding pushdown automaton: the grammar expansions map naturally onto push and pop operations on the PDA's stack, making the grammar-automaton equivalence at this level another cornerstone of formal language theory.

The thesis focuses on the above mentioned classes, as these are the ones applicable in the implementation part of the language service?NAME?. The last two classifications Type-1 and Type-0 grammars, can create *context-sensitive* and *recursively enumerable* languages respectively, and are the most expressive of all with Type-0 grammars placing absolutely no constraints on the production, making them only acceptable by Turing-Machine.

2.1.1 Backus–Naur Form (BNF) and Variants

Even though production rules, alphabet specifications, and automaton definitions suffice to unambiguously define the set of valid strings/sentences in a language, their notation tends to be opaque and cumbersome in practice, prompting interest in forming intuitively comprehensible grammar definitions. The Backus-Naur Form (BNF), created by John Backus with contributions by Peter Naur and released in their Algol-60 report. [Bac+60], was a pivotal introduction furnishing a concise, human-readable syntax for language generation. By expressing each rule as

$$\langle \text{nonterminal} \rangle \ ::= \ expansion_1 \mid expansion_2 \mid \dots$$

BNF allows both sequencing and alternation of multiple expansions, enabling language designers to articulate complex, nested structures without exposing the underlying automaton states or transition tables. This declarative approach not only supports rigorous specification but also facilitates mechanical parsing, thereby advancing the practice of compiler development and language tooling.

While BNF provides a clear, formal way to describe language syntax, it suffers from verbosity and redundancy — pure BNF grammars often become bloated when encoding common patterns like optionals or repetitions requiring auxiliary nonterminals for a sufficient description and, over time, has been extended and modified for various use-cases spawning a new family of grammar notations. The **Extended Backus-Naur Form** extends BNF by adding more expressive meta-syntax, introducing operators similar to *Regular Expressions* allowing the use of optional or repeatable expressions, commonly indicated by brackets but not limited to '[...]' and '{...}', and the use of comments. Such extensions make grammars more compact and easier to read without changing the fundamental class of languages they describe.[] Multiple distinct instances of EBNF's have been development, all with minor syntactic differences, yet there really is no on-for-all EBNF used in practice, although a standardized ISO/IEC 14977 version exists. [Jinb; Jina]
Within the scope of my thesis, *?NAME?*, Extended Backus–Naur Form (EBNF) is particularly significant, since Langium employs its own custom EBNF dialect to define the grammars underpinning its language-assistance features. 8

2.1.2 Grammar Formalisms and Parsing Models

..programming languages

2.2 Language Structure and Paradigms

2.2.1 Abstract vs. Concrete Syntax

Depednming on subsection Grammar Formalisms, mentino the importance of grammars in connectino with programming langs.

The terms *abstract* and *concrete* syntax are primarily encountered in the context of programming language design. While the two concepts follow the common relationship of abstract and concrete instances — where one embodies a generalized, meta-level blueprint, while the other encapsulates the concrete, instance-level details — and the term "syntax" broadly applies to all languages, the idea of pinning meta-information onto language constructs, somewhat implies further calculations based on the language itself.

When designing programming languages, the abstract syntax is of particular interest, as its streamlined view of the language is ideally suited for developing validation constraints or type systems and for general interpretation. This is done most commonly by encoding the syntax into a **Abstract Syntax Tree** (**AST**), a tree-like structure containing nodes for each high-level construct (such as expressions, declarations, or statements) connected according to their logical hierarchy, while deliberately omitting lexical details to yield a canonical. [SK95] The **Concrete Syntax Tree** (**CST**) is the specialized counterpart to the AST and consists of a full parse-tree, preserving every terminal and nonterminal token(keywords, delimiters, etc.) and mirrors each grammar production in its node structure, providing the complete syntactic context possibly needed for precise error reporting and tooling. [Aho+06]

2.2.2 Imperative vs. Declarative Languages

Languages used for further computation can be analyzed from multiple perspectives and classified with different qualities in mind. [Van12] For this thesis we focus on slotting languages into two overarching paradigms, **imperative** and **declarative** languages, which differentiate the by inspecting way of reaching the desired outcome.?NOLIKETHIS? Declarative languages describe *what* the computation should accomplish by adhering to sets of expressions, constraints or logical statements. [WGM08] These formulas set the rules and goals characterizing the desired result or relationship of input and output values. The actual control flow and low-level decisions of the execution is left to the specific language implementations and runtime engines.

Generally, because of the vague wording used, the paradigms themselves are not always completely and accurately describing the behaviour of a programming language, possibly leading to different concepts overlapping. This difference is especially clear in modern languages, which often implement functionality by borrowing constructs from the opposite paradigm — for example, imperative C# incorporates declarative features through its LINQ-Library, while Haskell, functional by design, supports imperative-style programming. [**netHaskellClaims**]

Imperative languages, in contrast to declarative ones, describe *how* a computation will accomplish the desired end-state by following explicit sequences of commands. These commands consist of statements that actively manipulate the computations state, which is typically managed through assignments to variables and taking charge of the control flow with provided language constructs (loops, conditionals, calls, procedures, etc.). Consequently, correct execution is no longer responsibility of the underlying system, but the implementation and therefore the programmer itself. The imperative paradigm reflect von Neuman machine — accessing memory and hardware directly. This fined grained control enables the use of low-level optimization techniques [**lowLvelOpt**]. As programmers instruct the machine how to do something, certain performance characteristics and resource constraints can be impelled depending on the context of the computation. Such techniques encompass memory and cache management, *Instruction-Set* optimization or compiler optimization.

Verbose is missing?

However, while this explicit control provides developers with great functional freedom, it also places a greater burden on them to manage every detail of execution. Because the programmer must specify each step — from how data is stored in memory to the order in which operations occur — imperative code can become verbose and cluttered obscuring the high-level intent. Moreover, mutable state and side effects introduce the risk of subtle bugs, especially in concurrent or parallel settings where unintended interactions between state changes can lead to race conditions. Also, formal reasoning and verification are more complex, since proving correctness requires tracking every state transition rather than relying on *referentially transparent* expressions.

Typical use cases for imperative programming include algorithmic or process-oriented problems where an explicit series of steps is natural. Low-level system programming, embedded development, and performance-sensitive routines frequently rely on imperative constructs to manage hardware resources directly. In scenarios that demand fine-grained stateful interactions — such as updating user interfaces incrementally, reading and writing files in a specific order, or controlling physical systems via commands — imperative languages shine by giving developers precise command over execution. Simulation engines, game logic, and scripting or “recipe”-style automation languages also favor imperatively describing each step to achieve the desired outcome.

2.3 Domain-Specific Languages (DSLs)

Domain-Specific languages are programming or specification languages which are tailored to specific application domain, providing constructs and abstractions that capture the domain concepts. [MHS05] In stark contrast to general-purpose languages (GLPs) that aim for broad applicability, a DSL waives generality in favor of higher expressive power in its domain of interest. By offering notation closely related to a domains phrasing, a DSL allows solutions to be formulated at the same level of abstraction as the domain itself, mapping language primitives onto the domain concepts. Developers, therefore, are able to create programs or specifications more productively and securely.

The close relationship of DSL and domain furnishes multiple advantages over conventional GLPs: it reduces the amount of general programming knowledge needed, expanding the pool of developers from traditional programmers to also domain experts. Moreover, DSLs declutter code, rather than describing a subproblem of the domain with multiple lines of code, fewer are needed because constructs inherently carry domain specific information, also contributing to broader comprehensibility for those familiar with the domain. DSLs solutions are also easier to formally verify, as they are restricted to a limited context, than GLPs for which proving correctness is extremely hard.

However, DSLs can suffer from limited scope and flexibility even within their target domain, forcing developers to fall back on general-purpose languages when they need functionality outside the DSL's narrow focus. Furthermore, if the domain itself is complex or rapidly evolving, creating and maintaining a DSL may still demand deep domain expertise, because the abstractions must accurately capture all necessary domain rules and nuances.

2.3.1 Imperative DSLs

Whichever paradigm a DSL follows is mostly guided by the specifics of the domain to describe. Even though DSLs usually adhere to a declarative approach [DKV00], which follows from a major motivation: wanting to describe *what* is expected of the domain in high-level abstraction, a domain may emit qualities that invoke more imperative thinking. If the nature of a domain is inherently composed of sequential actions, consists of procedures or stateful operations, the imperative paradigm is more appropriate. Imperative DSLs require the user to script solutions in step-by-step fashion using domain-specific commands and statements. Essentially,the user embeds domain concepts as a sequential programming model, where the control flow and state transitions of the domain computations have to be managed.

In domains where fine-grained control and explicit ordering matter, imperative DSLs provide maximal flexibility, because every aspect of execution is adjustable. This level of control becomes crucial when timing or order of operations yield different outcomes, or when fine-tuned performance optimizations are required. Furthermore, building an imperative DSL is often simpler than creating a declarative one: typically, an interpreters process each command in sequence, or a translator converts DSL instructions into a host general-purpose language.

Mybe just leave the below out

Imperative DSLs shine in scripting and macro contexts, where users sequence domain-specific commands to drive behavior, examples include Excel macros that automate spreadsheet tasks or scripts within a game engine that dictate character actions and game logic. They are also well suited to testing frameworks and scenario specifications, where user interfaces describe interactions by specifying a timely sequence of user inputs. In robotics and industrial automation, imperative DSLs allow practitioners to program custom action sequences—such as moving a robotic arm to pick up an object, then placing it precisely—where each step must follow a strict order to ensure safety and accuracy. More broadly, any process-control domain that relies on precise sequencing and stateful interactions—be it a complex simulation workflow, an interactive operation pipeline, or automated machinery —benefits from the clarity and control that an imperative DSL provides.

Probabilistic Programming & Cyber-Physical Systems

The *Miniprob* DSL

Language Servers, Parsers & Type Checking

https://code.visualstudio.com/api/language-extensions/overview

5.1 Language Server Protocol (LSP) Basics

5.2 Parser and Syntax Checking

5.3 Type Checking Functionality

5.4 Overview of Langium

Part II

Design & Implementation

Requirements & Technology Survey

Architectural Design

7.1 High-Level Architecture Diagram

7.2 Module Decomposition

7.3 Data Flow and Control Flow

Implementation Details

8.1 Langium Grammar Definition for *Miniprob*

8.2 Semantic Checks and Type System

8.3 VS Code Extension Points

Testing & Validation

Part III

Evaluation & Discussion

Performance & Usability

19

Discussion

11.1 Meeting the Requirements

Bibliography

[Aho+06] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd. Reading, Massachusetts: Addison–Wesley, 2006. ISBN: 978-0321486813.

[Bac+60] J. W. Backus et al. “Report on the algorithmic language ALGOL 60”. In: *Commun. ACM* 3.5 (May 1960), pp. 299–311. ISSN: 0001-0782. DOI: 10.1145/367236.367262. URL: https://doi.org/10.1145/367236.367262.

[DKV00] Arie Deursen, Paul Klint, and Joost Visser. “Domain-Specific Languages: An Annotated Bibliography”. In: *SIGPLAN Notices* 35 (Jan. 2000), pp. 26–36.

[Jia+09] Tao Jiang et al. “Formal Grammars and Languages”. In: (Nov. 2009). DOI: 10.1201/9781584888239-c20.

[Jina] Pete Jinks. *BNF/EBNF Variants*. https://www.cs.man.ac.uk/~pjj/bnf/ebnf.html. Last modified 12 March 2004. Accessed 26 May 2025.

[Jinb] Pete Jinks. *Notations for Context-Free Grammars: BNF, Syntax Diagrams, EBNF*. https://www.cs.man.ac.uk/~pjj/bnf/bnf.html. Last modified 12 March 2004. Accessed 26 May 2025.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. “When and how to develop domain-specific languages”. In: *ACM Comput. Surv.* 37.4 (Dec. 2005), pp. 316–344. ISSN: 0360-0300. DOI: 10.1145/1118890.1118892. URL: https://doi.org/10.1145/1118890.1118892.

[SK95] Kenneth Slonneger and Barry L. Kurtz. “Specifying Syntax”. In: *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Reading, Massachusetts: Addison–Wesley, 1995. Chap. 1, pp. 1–30. ISBN: 0-201-65697-3.

[Van12] Peter Van Roy. “Programming Paradigms for Dummies: What Every Programmer Should Know”. In: (Apr. 2012).

[WGM08] Hui Wu, Jeff Gray, and Marjan Mernik. “Grammar-driven generation of domain-specific language debuggers”. In: *Softw., Pract. Exper.* 38 (Aug. 2008), pp. 1073–1103. DOI: 10.1002/spe.863.

Full Grammar Specification

Expanded Code Listings

Test Data