# 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:

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

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

$$A \to \alpha$$

where $A$ is a single nonterminal and $\alpha$ 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

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

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 generally be slotted into two overarching paradigms, **imperative** and **declarative** languages, ... in the way of reaching the desired outcome. Declarative languages describe *what* the computation should accomplish by adhering to sets of expressions, constraints or logical statements. [9; 10] 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.

- **Imperative**: describe *how* to do things (sequential steps, mutable state, control flow).
- **Declarative**: describe *what* you want (constraints, relationships, goals), leaving execution order to the runtime or solver.
- Key trade-offs: control vs. abstraction, predictability vs. conciseness, ease of optimization, typical use-cases.

## 2.3 Domain-Specific Languages (DSLs)

- Definition and motivation: tailor syntax/semantics to a narrow problem domain.
- Classification by paradigm:
  - **Imperative DSLs** (e.g. MiniProb, CUDA, many build systems): embed domain logic in a procedural style.
  - **Declarative DSLs** (e.g. SQL, HTML, many configuration languages): express constraints or data relationships.
- Criteria for choosing one over the other: domain control, ease of reasoning, tool support, typical analysis patterns.

### 2.3.1 Imperative DSLs (Your Thesis Focus)

- Overview of their structure: explicit state, control flow, side-effects.
- Example: MiniProb (POPACheck)
  - Recursive calls, ‘observe‘, mutable grid updates.
  - How it compiles down to a probabilistic push-down automaton.
- Pros & Cons in your domain:
  - Pro: fine-grained control, easy to encode sequential reasoning.
  - Con: harder to optimize globally, side-effects can obscure analysis.
- Comparison with Declarative DSLs for the same problem (e.g. PCTL, PRISM models).

### 2.3.2 Declarative DSLs

- What makes a DSL declarative: absence of explicit control flow, emphasis on logical relations.
- Typical examples in probabilistic modeling: PCTL, PRISM, probabilistic logic programs.
- Why—despite their advantages—MiniProb remained imperative in POPACheck.

DSL or General (do api prog lang have to be haskell complete? C3, java haskell -»)

# 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

# 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.

[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.

[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.

# Full Grammar Specification

# Expanded Code Listings

## Test Data